Department of Informatics
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava, Slovakia

Theoretical and Practical Aspects of Programming Contest Ratings

Dizertačná práca
v odbore doktorandského štúdia:
11-80-9 teoretická informatika

RNDr. Michal Forišek

Školiteľ: prof. RNDr. Branislav Rovan, Ph.D.

Bratislava, 2009

Acknowledgements

First and foremost, I would like to thank my advisor prof. Branislav Rovan for being an exceptional and inspirational teacher and advisor.

I'm also grateful to other faculty members – and students – at our university. Thanks to many of them, both studying and teaching here has been a pleasure, and that is one of the factors that significantly influenced my career choices so far.

But without any doubt my biggest thanks must go to my fiancée Jana. Without her constant love, support and understanding I would not be able to finish this Thesis.

Contents

Abstract (English)

The main results of this Thesis are related to rating systems. Currently, most of the rating systems used in practice use Bayesian reasoning about skills of the contestants. We show several disadvantages of these systems. In particular, one major disadvantage that motivated us to do this research is the inability of these systems to address scenarios in which not all tasks are equivalent in terms of difficulty.

In this Thesis, we investigate a novel approach to ratings that is based on Item Response Theory. Using this testing theory, we design a new type of rating systems that takes task difficulty into account. We would like to stress that our rating system is by no means specific to programming contests. In fact, we are even aware of sports (e.g., golf, white water slalom) where it obviously applies as well.

An important observation we make is that while it is not possible to compare rating systems directly, there is a way out. Any rating system can be used to make predictions about outcomes of future contests, and by evaluating the quality of these predictions we can evaluate the quality of the rating system. In our Thesis we give formal definitions of this approach to rating system comparison. Additionally, we show prediction algorithms for our model that are superior to prediction algorithms for existing rating systems.

We dedicated one chapter to practical tests of our rating system. We show that our model is sufficiently close to reality on real life data. We then use the real life data to make predictions and analyze them. Already our basic rating system was able to match the quality of predictions made by an existing rating system that was ad-hoc tweaked to the given specific setting by its authors. Additionally, in our model we were able to make predictions that were not possible before.

Another benefit of our approach is the ability to analyze the difficulty of past competition tasks. And not only that, we can also compute the precision of our ability estimates. We used this method to analyze data from our national competition and came to the conclusion that the regional round of this competition is too hard.

Yet another important topic we address is the possibility of attacks on rating systems. We found that this is a topic that was almost completely neglected in the past. We approached this topic using an insight from the area of information security. We identify several types of possible attacks and show that they indeed can be used against existing rating systems.

Finally, we note that for the accuracy of rating systems it is essential to ensure that our data is accurate. Hence we address two additional topics: We analyze the security of contest systems (systems that interact with the contestants and handle the evaluation of their programs), and we provide insight into black-box testing as used in most contests.

Abstrakt (Slovensky)

Hlavné výsledky tejto práce sa týkajú ratingových systémov. Moderné prakticky používané ratingové systémy takmer bez výnimky používajú Bayesovské uvažovanie na počítanie odhadov schopností súťažiacich. V práci ukazujeme viaceré nedostatky týchto systémov. Hlavným nedostatkom, ktorý nás motivoval k výskumu v tejto oblasti, bola skutočnosť, že tieto systémy nedokážu zohľadniť situácie, v ktorých majú rôzne úlohy rôznu obtiažnosť.

V tejto práci skúmame nový prístup k návrhu ratingového systému, ktorý je založený na Item Response Theory. Pomocou tejto teórie testovania sme navrhli ratingový systém, ktorý rôzne obtiažnosti úloh zohľadniť dokáže. Podotýkame, že nami navrhnutý systém nie je špecifický pre súťaže v programovaní – existuje viacero športov (napr. vodný slalom, golf), v ktorých môže byť taktiež použitý.

Ďalším prínosom našej práce je nasledovný záver: Je zjavne nemožné porovnávať ratingové systémy priamo, len na základe vypočítaných ratingov. Ale ľubovoľný ratingový systém, presnejšie model, na ktorom je postavený, sa dá použiť na robenie predpovedí o výsledkoch budúcich kôl súťaže. A práve podľa kvality týchto predpovedí vieme vyhodnotiť kvalitu ratingového algoritmu. V práci uvádzame potrebné formálne definície pre toto nami navrhnuté porovnávanie. Taktiež ukazujeme algoritmy, pomocou ktorých vieme robiť predpovede v našom systéme.

Celú jednu kapitolu venujeme testom nášho prístupu na praktických dátach. Ukazujeme, že odhady vypočítané našim modelom dostatočne dobre zodpovedajú realite. Následne v našom modeli robíme predpovede a porovnávame ich s predpoveďami robenými na základe existujúceho ratingového systému, špeciálne navrhnutého pre konkrétne prostredie, v ktorom oba prístupy testujeme. Ukázalo sa, že náš systém dokáže vypočítať porovnateľne dobré predpovede, a navyše dokáže úspešne predpovedať skutočností, ktoré sa v existujúcom modeli vôbec predpovedať nedali.

V rámci počítania ratingov dostávame ako novú dodatočnú informáciu aj parametre úloh, ktoré môžeme následne analyzovať. Náš model nám navyše umožňuje aj spočítať presnosť odhadov jednotlivých parametrov. Tieto poznatky sme využili pri analýze dát z Olympiády v informatike, kde sa podarilo ukázať privysokú náročnosť krajských kôl.

Ďalšou dôležitou témou, ktorou sa zaoberáme, je možnosť útokov na ratingové systémy. Ukazuje sa, že táto oblasť bola v minulosti takmer úplne ignorovaná. Vďaka nášmu prístupu

z pohľadu informačnej bezpečnosti sa podarilo identifikovať niekoľko typov útokov a použiť ich na existujúce ratingové systémy.

Pre fungovanie ratingového systému je esenciálne, aby pracoval s presnými vstupnými dátami. Na záver práce preto uvádzame kapitolu, v ktorej sa zaoberáme dvomi oblasťami súvisiacimi s touto požiadavkou: Rozoberáme problematiku bezpečnosti contest systémov (systémov, ktoré interagujú s riešiteľmi a zabezpečujú integritu súťaže) a ukazujeme možnosti útoku na tieto systémy. Taktiež uvádzame niekoľko postrehov z oblasti black-box testovania programov v podobe bežne používanej na súťažiach.

Chapter 1

Introduction

The main goal of this Thesis is the research of rating algorithms for programming contests, with possible applications to other areas.

In this Chapter, we give a more detailed discussion of our goals, followed by an outline of our Thesis, and a brief discussion of our motivation for this line of research.

1.1 Goals of the Thesis

The idea of a rating system has been studied for several decades. Competitiveness is a part of our human nature, and when there is competition, there is the need to rate and/or rank the competitors. Also, the need for rating is often encountered in educational systems, and many other areas.

One of the first areas to adopt a statistical rating system was chess with its Elo rating system [Elo78]. Even though subsequent research showed that each of Elő's original assumptions was flawed, the importance of the Elo model lies in bringing in the statistical approach. A similar approach was later used by Glickman in [Glib, Gli99, Glia] in developing better rating systems for pairwise comparison. Glickman's systems can also measure the accuracy and volatility of the given player's rating.

In recent years, Glickman's models were generalized to handle situations when events involve more than two subjects. Notable advances in this direction include Microsoft's recently published TrueSkill$^{\text{TM}}$ ranking system [HG06, HMG07, DHMG08a], and TopCoder's rating algorithm [Top08]. All the existing rating systems are based on the idea of Bayesian inferrence.

In our Thesis we investigate a new approach to design a rating system [For09a]. Our approach is based on Item Response Theory. Hence our rating system is also able to estimate parameters that describe items (i.e., competition tasks), and therefore it is better in all settings where the items are not interchangable. This is true in many settings, including various sports such as golf and white water slalom.

A second major goal of the Thesis is to develop an exact formal framework that can be used to make objective comparisons of different rating systems for the same setting.

Afterwards, we consider the rating systems from an information security point of view. A malevolent individual may benefit from misguiding the rating system. This is the equivalent of an attack against an ICT system. We show that it is indeed possible to attack many of the existing rating systems.

An inherent part of the Thesis is the evaluation of our theoretical results on real life data. This includes goodness-of-fit tests, task difficulty analysis [For09b], computation of ratings and predictions, evaluation of these predictions and comparison to existing rating systems.

Finally, we investigate several factors that can negatively impact the rating systems by corrupting its input data – the security of the contest systems [For06b, Mar07], and the validity of black-box evaluation [For04, For06a].

1.2 Outline of this Thesis

This Thesis is organized into several chapters. Chapter 2 contains an overview of prior research, in the remaining Chapters we present our research. A more detailed description of each Chapter follows.

- Chapter 2 contains a survey of prior research in the fields related to the main topic of this Thesis – rating systems. We also include an overview of the necessary parts of Item Response Theory.

  Additionally, in Chapter 2 we give a brief overview of other research done in the area of programming competitions. Some of this research is relevant to our work presented in Chapter 6.

- The remaining chapters of this Thesis, starting with Chapter 3, contain the results of our research.

  In Chapter 3 we present our results related to developing an IRT-based rating system: We list the assumptions that define the setting we address. We develop our new formalism that will allow us to make exact reasoning about rating systems, and show that this formalism can be used to describe an existing modern rating system. According to our research we define an exact, objective approach to comparing different rating systems via evaluating the predictions that can be made from the computed ratings. We formally prove some properties any rating system based on Item Response Theory (IRT) must have. We then define a basic IRT-based rating model and show an algorithm that can be used to make certain type of predictions in this model.

- In Chapter 4 we address several advanced topics related to rating systems. We discuss how our basic rating system can be improved in various directions, including task we-

ights, abilities that change in time and ways how additional information provided by the underlying model can be used to compute more precise predictions.

A particularly important topic we address in this Chapter is the topic of rating systems security. We analyze existing rating systems from the point of view of an attacker that wants to mislead the system. We show several types of attacks we found, and we show that existing rating systems are indeed vulnerable to these types of attacks. We also discuss whether a IRT-based rating system can resist these attacks.

- Chapter 5 is dedicated to an evaluation of our methods on real life data taken from various programming competitions. We show that using our approach we can find a sufficiently good theoretical model that matches the real life data. We will show concrete examples of predictions made by a proof-of-concept implementation of our rating system and compare them to predictions made in an existing Bayesian rating system.

  In this Chapter we also present our analysis of the difficulty of a competition, and an analysis of the time necessary to solve a task and its correlation to the subject's rating.

- For the accuracy of rating systems it is essential to ensure that our data is accurate. Hence in Chapter 6 we address two topics related to this requirement:

  We analyze the security of contest systems, give a classification of possible attack types and show concrete examples of our attacks against contest systems.

  In this Chapter we also analyze the validity of black-box testing as used in most contests to evaluate programs submitted by contestants.

1.3 Motivation

In this Section, we conclude the introduction by giving a brief essay on our motivation that lead us to this area of research.

What are programming contests, and why is it important to have programming contests?

In the past two decades, programming contests became a very important part of the current educational process in both secondary (high school) education and undergraduate study programs at universities.

The main objectives of the most famous worldwide programming contests (see [IOI05], [ACMa]) can be summarized as follows:

- To discover, encourage, bring together, challenge, and give recognition to young people who are exceptionally talented in the field of informatics.

- To help young people practice and demonstrate their problem-solving, programming and teamwork skills.
- To foster friendly international relationships among both future and present computer scientists and informatics educators.
- To bring the discipline of informatics to the attention of young people who are deciding their career options.
- To advertise informatics as an interesting area for potential investors, and to advertise the successful contestants as the next generation of excellent computing professionals.

We also add that in top-level programming contest, the main focus is on the problem solving aspect. In this way, programming contests directly influence the contestants' abilities to do scientific research and (to a lesser degree) software design.

We feel that here it is important to note that the contest itself is just a way of achieving these goals, **not a goal on its own**. (This is a common mistake made by many newcomers in this area.)

Why is a formal research of programming contests important?

Nowadays, most of the contests are organized in an ad-hoc way. For many parts of the contest organization there are known "best practices" developed in the programming contests community. But still there is a lack of proper understanding of the methods used (e.g., automatic testing of contestants' computer programs, contest evaluation methods, etc.). After a formal research in this area it would be easier to improve the organization of programming contests, thereby making it easier to fulfill the goals mentioned above.

In the past few years there is an iniciative in the international community to start a systematic research in this area. However, most of the articles published to date address only highly specific parts of this scientific area, and often there is no relation between them. We are convinced that this research shall lead to a better understanding of the procedures employed by the existing contests. However, by considering a suitable abstraction, many of the results should be applicable in a much wider range of areas. (Classroom settings / university courses may serve as an example.) Additionally, abstraction and formalism can be used as tools to bring new insights and draw parallels to existing research.

Why are rating systems relevant?

Competitiveness is a natural part of human nature, and with it comes the need to identify the best, and to compare. Often the compared abilities can not be measured directly, only by observing the outcomes of events influenced by these abilities. In this case, we can use the outcomes to gain information that allows us to estimate the unknown abilities.

Of course, comparing abilities is not the only application of rating systems. Often a more important application is the ability to predict the outcome of future events. One particular area where this is obvious is when we encounter monetary benefits tied to predicting correctly (such as betting on outcomes of sports events). Having an accurate rating system and the ability to make better predictions than the other participants can be easily turned into profit.

Chapter 2

Background

In this chapter we provide a concise background we need to present our results. The chapter is divided into several sections that address different areas. The content of these sections is described below.

In Section 2.1 we give an overview of programming contests and of related terminology. We also list publications related to this area that are relevant to the topics we address in this Thesis.

In Section 2.2 we present the existing results in the area of rating systems and rating algorithms. We define the necessary terminology and present the inner workings of modern, Bayesian rating systems.

In Section 2.3 we focus on Item Response Theory (IRT), a modern testing theory that enables us to take into account parameters of different test items, such as task difficulty. In later chapters we will use the IRT framework to design a new class of rating systems.

2.1 Programming contests landscape

In this section we shall give a brief overview of the common programming contest types, define the terminology used in this area, and give an overview of the existing research related to programming contests in some way. A reader associated with programming contests may skip straight to section 2.1.3 which contains the research overview.

2.1.1 Important programming contests

We shall now give a brief description of the most important and influential programming contests. In later sections and chapters we shall tend to refer to these contests by giving them as concrete examples when explaining concepts, model situations, etc.

Many more contests are discussed in more detail in the articles that are mentioned in section 2.1.3.

International Olympiad in Informatics

Eligibility

Secondary school students worldwide. Each participating country organizes its own national contest to determine up to four contestants that are allowed to participate in the International Olympiad in Informatics (IOI).

Conduct of contest

Contestants compete as individuals. Two competition days. On each day the contestants solve 3 hard tasks of algorithmic nature. No or little feedback during the contest. (This may be changed in the upcoming years.) After the contests contestants' programs are automatically tested on a set of test data prepared before the contest.

Scoring and ranking

Usually, each problem is worth 100 points. Partial credit is awarded for correctly solving a part of the test data.

Contestants are sorted according to their total score for all six competition tasks.

Top 50% of the contestants are awarded medals, the ratio of gold to silver to bronze medallists is approximately 1:2:3. This means that top 1/12 of all contestants receive gold medals. (The fact that there is more than one medal of each kind often causes confusion among not involved observers.)

ACM International Collegiate Programming Contest

Eligibility

Full-time college/university students are eligible to participate in ACM ICPC. Additional limits are imposed on the number of times an individual may participate in each of the rounds.

Conduct of contest

Contestants compete in teams of three, and represent a single institution. The contest is conducted in several rounds: university qualification, (possibly a subregional contest,) regional contest, and the World Finals. The top team from each region is guaranteed to advance to the Finals, and in addition the organizers award wild cards to the next teams from strong regions.

In each contest round contestants are given a set of 6-10 tasks of algorithmic nature. At any point during the contest the contestants may submit a program that tries to solve any of these tasks. The program is automatically tested, and the contestants are notified about the result of its testing. A task is only considered solved if the submitted program flawlessly solves all sets of prepared test data.

Scoring and ranking

In the ranklist teams are ordered based on the number of tasks solved. Ties are broken using *total time*, that consists of a sum of times when the team solved its tasks, and of a penalty for submits that were not accepted. A more precise formulation can be found in [ACMa].

TopCoder Algorithm Competitions

Eligibility

Everyone, except for TopCoder employees, is eligible to participate. Some special tournaments (such as TopCoder Collegiate Challenge) may impose additional limits on participation. Only members of 18 years of age or older are eligible to win cash prices.

Conduct of contest

A typical contest round consists of several phases. In the first one (*coding* phase), contestants are given three tasks of algorithmic nature, with different difficulty degree, and an appropriate point value. The score a contestant is (provisionally) awarded for each of the tasks depends on the tasks' assigned point value and on the time he needs to write and submit a program that claims to solve the task.

The second phase is the *challenge* phase, where each contestant is allowed to look at programs submitted by other contestants, and to try to find flaws in these programs. Additional points are awarded for correctly identifying flaws (and providing suitable test data as a proof). Essentially, in this phase the submitted programs pass through a *white-box* testing process.

The third phase is the *system testing* phase, where each of the submitted programs is automatically tested on a large set of test data.

Scoring and ranking

The final score of a contestant is the sum of the scores assigned to his programs that successfully passed all tests in all phases, and of the additional points he gained in the challenge phase.

Note

Google CodeJam, possibly the largest programming contest to date, has in years 2003 to 2007 used the TopCoder infrastructure. Since 2008 CodeJam uses their own infrastructure, and the competition format is modelled similarly to IPSC (see below).

Internet Problem Solving Contest

Eligibility

Everyone is eligible to participate.

Conduct of contest

The contest is organized once in a year as a worldwide on-line contest. For each task the contestants are given a set of input data, and their goal is to produce correct corresponding output data. The choice of a programming language, or even of a method of obtaining the output data is entirely upon the contestants.

Scoring and ranking

Ranklist is computed in using a method similar to the one for ACM ICPC. There are separate ranklists for teams and individuals, and a separate ranklist for participants from secondary schools.

2.1.2 Programming contests terminology

In this section we shall define basic terminology used in programming contests.

General terminology

Task, project, problem

These words are commonly used to denote the assignments the contestants try to solve during the contest. The usage varies, but there is a general notion that a project is a long-term assignment, and a task/problem is a short-term assignment.

As the word "problem" may be ambiguous – the other meaning being "trouble" – in this thesis we shall use the term *task* (or *competition task*) to denote the short-term assignments. (Exceptions may be made when addressing a concrete contest that uses the other notion.)

Solution, submission, program

The word *solution* is commonly used in two meanings: first, it can be a correct answer to the given task, or second, the word *solution* can denote the product submitted by a contestant. Note that the second possible usage is slightly misleading, as when the contestant does not solve the given task, his product is not a solution in the first sense.

To avoid confusion, we shall **not** use the word *solution* in the second sense.

The word *solution* shall be synonymous to *correct solution*. When talking about contestants' output, we shall use the word *submission* (as in "submitted for grading"). If the submission has the form of a computer program, we may also use *program* to describe it.

Task statement

The wording of the assignment as the contestants see it. Task statements usually contain a motivation / story (possibly including the background of the task), and a formal definition of the task.

Limits

For automatically evaluated contests, one of the vital parts of the formal definitions in the task statement is the definition of limits under which the contestants' programs are supposed to operate.

The most important limits are the *time limit*, the *memory limit*, and the *maximum input size*.

The time limit and the memory limit are limits enforced during the testing of the contestants' programs. The maximum input size is a limit the contestants may use when designing and implementing their programs to avoid overflow errors.

From a theoretical point of view, the presence of a maximum input size limit guarantees that the set of allowed inputs (that can be used to test the program) is finite.

In some of the existing contests, some of the limits, even though they're present and enforced, are not known to the contestants. For example, the ACM ICPC contest usually does not publish information on the time limit.

Contest system

The set of software used to conduct a contest is usually called a *contest system*. More on this topic will be given in a separate section.

Terminology used in task statements

The task statements tend to use a specific set of terminology. In addition, some contests take their eligibility criteria into consideration when deciding what terminology is appropriate in a task statement. Reader interested in these issues can find a IOI-centered discussion of this topic in [Ver04].

Contest systems

The set of software used to conduct the contest is usually called a *contest system*. Usually, the organizers of each contest implement their own contest system. Some of the more famous publicly available contest systems are PC^2 (PC squared, [ABL], used to organize ACM ICPC contests in multiple regions worldwide), SIO.NET ([Mic06], a new Polish contest system used for most programming contests in Poland), Portuguese Mooshak [Moo] and Peking University JudgeOnline [PF]. In Slovakia the CESS system [Kou04] was developed and is being used in

various contests. The most famous online contest system is the University of Valladolid online judge ([UVa], operating since 1997).

The main functionality provided by the contest system is *an automated judge*. This is a piece of software responsible for automatical evalution of the contestants' submissions. The canonical way of evaluating a submission consists of *compiling* the source code submitted by the contestant, *executing* the resulting program on a prepared set of test inputs, and checking the program's output for correctness.

When the contestant's program is being run, it is usually isolated in a logical area called a *sandbox*. The goal of the sandbox is to prevent malicious submissions from accessing the restricted areas of the contest system, and to enforce various limits set on the submitted program.

Other functionality necessary for running a contest usually includes *user management* (e.g., authentication, access control), *administration functions* (e.g., adding/editing tasks, test data, setting time limits, rejudging submissions), *contest evaluation* (computing the rankings), and a *contest portal* (displaying rankings, handling submissions and clarifications, displaying additional information).

Some of the existing contest systems offer additional functions, such as allowing the users to *backup* their data, or *print* their source codes.

2.1.3 Overview of the existing research on competitions

The past research of programming contests can be grouped into the following areas:

- Mapping and clasifying the existing contest landscape
- Research of new possibilities and approaches
- Exact formal research of existing topics

We shall adhere to this classification when giving an overview of the existing research and its published results. The main focus in our Thesis is, of course, on the publications from the last group. However, as this is a complex polyscientific topic, we opted to review significant research in the other two groups as well.

Contest landscape

Wolfgang Pohl's paper [Poh06] gives an excellent overview of the existing contest landscape, and tries to establish several different ways of classifying the contests. To name a few dimensions suggested by Pohl:

Projects vs. tasks

Tasks have a fixed, pre-determined outcome the contestants are supposed to reach, whereas projects have a more vague definition, allowing more space for creativity, and usually requiring a written or spoken presentation of the results.

Long time vs. short time rounds

Giving the contestants more time (usually several weeks) allows the organizers to use more difficult tasks, tasks requiring some research, etc. Note that this dimension is not equivalent to the previous one – long time task based contests are quite common on the national level.

Automatic vs. manual grading

Delivery: a HW/SW product vs. an answer

Cormack et al. in [CMVK06] give a more detailed description of ACM ICPC, IOI, and the TopCoder algorithm competitions than we do in section 2.1.1. The major contribution of this article lies in identifying how various aspects of the programming contests relate to the purpose of these competitions. The topics highlighted in this article include:

- The need to identify a winner
- Pressure elements felt by contestants
- Rewarding prior knowledge vs. skill
- Collaboration vs. competition
- Spectators, scoreboards, and blackouts

Several papers with a more detailed description and/or analysis of other programming contests were published, among others [Dag06] on the Beaver contest for secondary schools, and [vdV06] on the CodeCup contest.

Research of new possibilities and approaches

Opmanis in [Opm06] presents a thorough research of various trends that may influence programming contests in the future. Some of the more significant trends presented in the article include:

- Several ways to involve the contestants in the testing process
- New scoring schemas
- Several ways to improve the debugging process during the contest

The problem of attacting more girls (and women in general) to computer science has received much interest over the last decades. The topic has been researched, and there are lots of ongoing activities with this goal. As an example, see [GGT].

In the area of programming contests this topic was addressed by the following publications:

Boersen and Phillipps in [BP06] give a description of the approaches used in organizing the Young Woman's Programming Contest, a successful contest at the New Zealand. This contest is compared to existing contests, and the differences are pointed out. Among the important points we can find cumulative scoring (the task is divided into separately-scored parts), and the presence of successful role models the contestants could identify with.

Fisher and Cox in [FC06] give a detailed overview of what they denote "exclusionary practices" – the aspects of programming contests that are most likely to make the contests less appealing to women.

Known gender-based differences are reviewed and applied to programming contests. The following factors are identified as influential:

- Risk and risk handling
- Time constraints (and the resulting pressure)
- Scoring concerns
- Question topics
- Performance anxiety
- Strategies of completeness

The article identifies several major points that can help mitigate the effect of the identified exclusionary practices:

- Feedback during the contest
- Appropriate, gender neutral (or balanced) question topics
- Providing practice and motivation
- Privacy
- Allowing cooperative groups
- Appropriate (e.g., incremental) scoring model

In general, the practical observations from [BP06] agree with the theoretical results of Fisher and Cox.

However, Wang and Yin in [WY06] argue that introducing direct antagonism (i.e., direct competition between the contestants' programs, one program's success meaning other program's failure) can make the contests more visible and attractive, both to potential contestants, and to the IT industry per se.

In some sense, this claim goes in a direction opposite to the one suggested by Fisher and Cox in the article cited above, especially considering the points of risk handling, strategies of completeness, and the incremental scoring model. In other words, introducing problems with direct antagonism seems to contradict the point that girls tend to need a sense of security, and to feel confident that their efforts will lead to corresponding scores.

Currently, it is not clear whether these two sets of requirements can be satisfied at the same time. If not, contests will have to aim for a reasonable tradeoff.

Exact research of existing topics

Cormack in [Cor06] used statistical methods to analyse how much influence do random factors in test data selection have on scoring at the IOI. In the article a formal model of choosing the test data is defined as follows: The actual test cases are assumed to be drawn from an infinite population of similar cases, and the probabilities are modeled according to the actual test data used in the competition.

Based on this model, the variance in standardized rank is estimated by the bootstrap method and used to compute confidence intervals which contain the hypothetical true ranking with 95% probability.

Empirical results based on IOI 2005 data are provided. The results show that for a reasonably low influence of random factors it is crucial to have a significant proportion of both "easy" and "hard" sets of test data. But even then for too many students the confidence intervals computed for the given model crossed a medal boundary. This suggests that the influence of random factors on the current IOI scoring system is larger than desirable.

Manzoor in [Man06a] examines the data gathered in the University of Valladolid Online Judge (see [UVa]), and makes various conclusions. The most notable result is the impact of practice and experience on making different types of mistakes – with an interesting observation that while practice lowers the percentage for most of the error types, the percentage of programs giving a wrong answer remains more or less the same. Also, conclusions on popularity of different programming languages over time are made.

Based on the observations made, Manzoor suggests a new ranking model for ICPC-like contests. This model has not been tried in practice yet.

Yakovenko in [Yak06] analyses a special rule used at past IOIs, the "50% rule". Essentially, this rule states that the problem statement shall inform the contestants about additional limits on test data. The purpose of the rule was to give the contestants additional information (and confidence) about the requirements to achieve partial credit.

This rule was in effect for IOI 2004 and 2005. Yakovenko summarizes the disadvantages of this rule, argues against it, and provides suggestions on how to change the rule.

The reasons behind the problems the "50% rule" caused are closely related to a deeper, underlying problem of test case design. This issue was first addressed by the author of this thesis in [For04], and later extended and published as [For06a]. Our research of the related issue is presented in this Thesis in Section 6.2.

Similar topics were explored by van Leeuwen and Verhoeff. In [vL05], van Leeuwen presents an analysis of two IOI tasks. He develops a taxonomy of possible algorithms for solving this tasks, assigns the solutions submitted by contestants to the classes of this taxonomy, and grades them using a manual grading process. It is shown that the resulting grades significantly differ from the results of the automated testing at the respective IOI.

In addition to these results, van Leeuwen suggests visualisation as an approach to clasify algorithms in one of the tasks. The details on this task can be found in [HVa].

Verhoeff in [Ver06] builds on the results of van Leeuwen to argue the necessity of change in the testing process used at the IOI.

Several different difficulties may arise when "real" numbers are used in programming contests. (When we speak of "real" numbers, "real" is considered to mean "not necessarily integer". Almost all architectures currently use floating-point representation aiming to conform to the IEEE Standard 754 ([Ins85], see also [Kah01, IEEb, IEEa]) to store these numbers. Due to this representation, the numbers are not arbitrary real numbers but only limited-precision rational numbers.)

For a further discussion of the floating point format and related issues in the general context, see [Kah05, Gol91, Daw].

Some of the difficulties with using real numbers in programming contests were summarized by Horvath and Verhoeff in [HVb]. A more detailed discussion of these difficulties was later given by the author of this thesis in [For05]. Opmanis in [Opm06] gives new suggestions on how to overcome some of these difficulties.

2.2 Rating Systems and Rating Algorithms

In this Section we present the research related to rating systems – i.e., systems that process data from competitions and produce a numeric evaluation of the contestants' skills. We introduce the most important rating systems, and give an overview of related publications.

2.2.1 Introduction

The idea of a rating system has been studied for several decades. Competitiveness is a part of our human nature, and when there is competition, there is the need to rate and/or rank the competitors. Also, the need for rating is often encountered in educational systems, and many other areas.

In this context, *rating* means assigning a vector of properties to each subject, and *ranking* means arranging the subjects into a linear order according to some set of criteria. Usually, ranking is the goal, and rating represents the means to achieve this goal.

Note that in some contexts the term *rating system* is also used to describe systems that process individual users' preferences into summary information. An example of this type of a rating system is the movie rating system at IMDb, [IMD]. We will **not** address these rating systems. In all subsequent text, a rating system is a system that produces rating and ranking of subjects' skills, and does this solely based on their objectively measured performances.

The first rating systems were reward-based: A good performance was rewarded by granting the subject rating points. The main advantage of these rating systems was their simplicity. Due to this reason, such rating systems are still used in many popular sports, such as tennis [ATP08] and Formula 1 [FIA08].

These systems are usually designed so that their discrimination ability is highest among the top subjects, and rapidly decreases as the skill of the subjects decreases. Moreover, the reward values are usually designed ad-hoc, and the rating systems usually have little to no scientific validity.

One of the first areas to adopt a more scientific-based rating (and thus ranking) system was chess. The United States Chess Federation (USCF) was founded in 1939. Initially, USCF used the Harkness rating system [Har67]. This was a reward-based system determined by a table that listed the reward size as a simple function of the difference between the players' current ratings.

After discovering many inaccuracies caused by this system, a new system with a more solid statistical basis was designed by Arpád Elő, and first implemented in 1960. For details of this rating system see [Elo78].

The Elo rating system was based on the following set of assumptions:

- The performance of a player in a game is a normally distributed random variable.
- For a fixed player, the mean value of his performance remains constant (or varies negligibly slowly).
- All players' performances have the same standard deviation.

The importance of the Elo model lies in bringing in the statistical approach. Even though subsequent research showed that each of these assumptions was flawed, and accordingly chan-

ges to the rating system were made, the currently used rating system in chess is still called the Elo rating system in Arpád Elő's honor.

An excellent overview of various rating systems in chess can be found in [Gli95]. For a discussion of some problems of the currently used rating system, see [Son02].

The next important step in the history of rating systems was the Glicko system [Glib, Gli99] that, in addition to calculating ratings, calculated also the rating deviation for each of the players – a value that measures the accuracy of the given player's rating.

This system was later amended into the Glicko-2 system [Glia] that also computed the rating volatility – a value that measures how consistent a player's performances are.

Glickman's rating systems were designed to handle situations where each event is a comparison of a pair of subjects. For a general background on rating system models for pairwise comparison, see Glickman's PhD thesis [Gli93].

In recent years, Glickman's models were generalized to handle situations when events involve more than two subjects. Notable advances in this direction include Microsoft's recently published TrueSkill$^{\text{TM}}$ ranking system [HG06, HMG07, DHMG08a], and TopCoder's rating algorithm [Top08].

As all of the rating systems mentioned above are based on Bayesian inference (see [HHK01] for an overview), and thus they are usually known as Bayesian rating systems.

One other rating system we would like to mention is the eGenesis rating algorithm (called "eGenesis Ranking System" by its authors), presented in [TY02]. It is an ad-hoc rating system. Its importance lies in the fact that it has been developed with the awareness that the participants may cheat, and it has been designed to prevent one type of cheating.

2.2.2 Overview of the Elo rating system

In this Section we present a simple Bayesian rating systems: the currently used version of the Elo rating system. Note that this Thesis also includes a formal definition of a modern rating system that is used in TopCoder competitions. However, we postpone presenting this rating system to Section 3.4, after we define the necessary formalism.

For each of the competitors the rating system estimates his skill θ_i. If we have players A and B with ratings θ_A and θ_B playing each other, the expected score of player A can be defined using the logistic curve as follows:

$$E_A = \frac{1}{1 + 10^{(\theta_B - \theta_A)/400}} \tag{2.1}$$

(Note that the constant 400 is simply used to establish an unit of measurement.)

After an event, the ratings are updated using a simple linear formula:

$$\theta'_A = \theta_A + K(S_A - E_A) \tag{2.2}$$

where S_A is the actual score of player A. (In chess tournaments, ratings are commonly updated after the entire tournament, not after every single game. In that case, E_A is the sum of expected, and S_A the sum of actual scores for each of A's games.)

Various chess federations (and other sports) use various flavors of the Elo rating system. One obvious point where they differ is the choice of the parameter K. USCF and FIDE use different tabelated values for K, based on the player's current and/or historically maximal rating. The point of the most accurate K factor was researched in [Son02].

(Note that K expresses the maximum rating change per game. Commonly used values of K range between 10 and 32.)

Other points where implementations of the Elo rating system differ are in handling the ratings of newcomers, and dealing with ratings inflation.

2.2.3 TopCoder's event format

In this Section we describe the format of rated events in the TopCoder competition in Section 2.2.3. This information will be relevant in 5, where we use real data from TopCoder to evaluate our rating system.

Each rated event consists of the following three phases:

- coding phase
- challenge phase
- system test phase

We will now describe the phases in more detail. Parts of this description are cited from TopCoder's website.

The Coding Phase

The Coding Phase is the period during which each competitor attempts to create solutions to three problem statements. In most cases, the coding phase will last 75 minutes, which is the total time that the competitors have to submit solutions to any or all of the problems. When the coding phase begins, each competitor will have the opportunity to view the problem statements. Each problem can be opened by selecting its point value from the "Select One" drop-down box.

Each problem is pre-assigned a point value. The higher the point value assigned, the more difficult the problem will be. The competitors may open the problems in any order. As soon as a problem is selected, the submission point value for that problem will begin to decrease. The longer a problem is open, the lower the score for that problem will be.

The exact formula for the score is given in equation (2.3). MP is the maximum point value assigned to the problem, TT is the total time of the coding phase, and CT is the competitor's time between opening the problem statement and submitting the solution.

$$score = MP \cdot \left(0.3 + 0.7 \cdot \frac{TT^2}{10CT^2 + TT^2}\right) \tag{2.3}$$

Note that in Section 5.4 we show that solving times usually have log-normal distributions with varying parameters, and thus an ad-hoc fixed formula that does not reflect this fact is clearly a sub-optimal way how to convert time into points.

The Challenge Phase

The Challenge Phase generally begins five minutes after the end of the Coding Phase (the period in between is an intermission), and will last for 15 minutes.

During the Challenge Phase, the competitors have the opportunity to review the source code submissions of the other competitors in their competition room, and to challenge some of those submissions. In a challenge, the challenger provides a test case that is supposed to cause the challenged submission to return an incorrect result. A successful challenge is awarded 50 points, an unsuccessful one is penalized by 25 points.

Challenges may also be used as a way to introduce new test cases into the suite of system test cases. Any challenge during the challenge phase that is recorded as successful will be added as a system test case – to be run against all remaining submissions during the system-testing phase.

There are a few restrictions during the challenge phase:

- A competitor may only submit a challenge if he has a score of zero or greater at the time of the challenge. Once his score drops below zero, he will no longer have the option/ability to challenge.
- A competitor may only challenge submissions made in the same competition room as his own.
- A given submission may only be successfully challenged once. If any competitor has already successfully challenged a submission, it may not be further challenged.
- It is not allowed to challenge own submissions.

It is a violation of the rules to discuss any aspect of the problems or any specific submissions with anyone until the challenge phase has ended.

We note that in software testing terminology the Challenge Phase corresponds to *white-box testing* – as opposed to *black-box testing* that occurs in the subsequent phase.

The System-Testing Phase

The system-testing phase is non-interactive. Immediately following the challenge phase, the TopCoder servers will run a number of housekeeping tasks, followed by the system testing. During the system testing, every problem that is still standing will be subjected to an extensive set of test cases. The system will check to make sure that each submission returns the correct result in under 2 seconds for each of these test cases. If any submission fails any of the test cases, the submission will be marked incorrect and the points achieved for that submission would be removed from that coder's total score.

2.2.4 TrueSkill(TM) rating algorithm

TrueSkill is a modern Bayesian rating system developed by Microsoft Research mainly for the area of computer games, but it has meanwhile been used in several sports as well.

For a detailed description of this rating system see [HG06, HMG07, DHMG08a]. The main contribution of this rating system is in its ability to also handle team matches. (However, for teams the rating system is based on the assumption that the skill of the team is equal to the sum of skills of its members. In various settings this assumption can be disputed.)

2.2.5 eGenesis rating algorithm

This is an ad-hoc rating system designed to prevent cheating by repeatedly losing on purpose to help elevate the rating of a friend.

The basic idea of the rating system is the following one: Each player in the system starts with a vector of 256 bits. Half of these bits, randomly selected, are set to 1, others to 0. A player's rank is the number of bits set, that is, an integer between 0 and 256, inclusive.

When a match occurs between two players, A and B, a series of 32 hash values in the range 0 to 255 is computed based on the players' names only – the same two players always yield the same 32 values. For each value, we inspect that bit in both players' vectors. When a bit is set in the loser's vector, and clear in the winner's vector, we transfer that bit: clear it in the loser's vector, and set it in the winner's. In all other cases we do nothing. This system forces players to play a wide variety of others to boost their rank.

(A source not related to the authors of eGenesis rating algorithm claims that in a newer version only 8 random bits out of those 32 are considered. We were not able to verify this claim.)

After realizing that it is still possible to boost one's rank by playing newcomers, the system was later modified as follows: Each player starts with their bit vector set to all zeroes. They also have 128 "reserve bits" which are stored simply as a count. A players rank is equal to the number of bits set in his vector, plus the number of bits in reserve. When a match occurs, bits are transferred as above, with the following additional rule: When one of the 32 examined bits is clear both in the winner's and in the loser's vector, and the winner still has reserve

bits, one clear bit is selected in the winner's vector (based on a different hash function that again incorporates both players' names), this bit is set to 1 and the winner's reserve bit count is decreased. Note that the loser's vector and reserve are untouched.

We doubt that the ratings computed by this algorithm have a sufficient validity. For example, consider the situation where two friends play each other repeatedly. In this case, only the last game matters, the results of the previous ones are not reflected on the rating at all. Or, consider a player that loses five games in a row with different opponents. In this case, the order of those games has a significant impact on the resulting ratings.

Still, we found it important to mention this rating system for the fact that it actively fights cheating – which is a property seldom seen in the other rating systems.

2.3 Item Response Theory

In this chapter we give an overview of the areas of Item Response Theory that are relevant to our Thesis. A reader interested in a deeper background in Item Response Theory is advised to pursue this topic further in excellent monographies [BK04, vdLH96]. For a slightly more general overview of the areas related to IRT refer to [Par04].

2.3.1 Introduction and motivation

In practice we often encounter a situation when a variable we might be interested in can not be measured directly. For example, this situation is often encountered in *psychometrics*, when trying to measure aspects such as knowledge, abilities, attitudes, and personality traits.

The key approach to these situations is to model the measured attribute as a hidden, *latent variable*. In contrast to visible attributes, such as height and weight, these latent variables can not be observed or determined by direct measurement. However, these variables can be estimated from the results of appropriate tests.

Classical test theory

The first and to date most common approach is currently known as the Classical test theory (CTT). As a gross oversimplification, we may state that in the CTT the test is scored, and the subject's score is used to estimate the latent ability. The main goal of CTT is to construct the tests in such a way that the *reliability* and *validity* of the test are maximized.

Reliability of a test is the consistency of the test, in the sense that if a subject retakes the test (or is given a similar test by another administrator), the measured ability should be likely to be equal. Validity of a test is the degree to which a test measures what it was designed to measure.

Modern approaches

In recent years, CTT has been superseded by a more sophisticated approach, the Item Response Theory (IRT). The main difference is that IRT models include not only the latent variables we try to measure, but also *item parameters* (such as difficulty of a question in an IQ test). In general, IRT brings a greater flexibility and provides more sophisticated information than CTT did. We will describe the difference in more detail in Section 2.3.2, after we have some basic definitions.

The main drawback of IRT (and the reason why it's still not prevalent in mainstream research) is that IRT models usually have high computational demands, which only made them usable in the recent years.

2.3.2 Basic definitions

The basic setting for IRT models consists of a set of subjects, and a set of test items. Our goal is to estimate some latent ability of the subjects given their performance on subsets of test items.

The ability estimation is based on the assumption that each subject possesses some amount of the underlying ability, and that this amount can be expressed as a numerical value that represents the subject's position on the ability scale. We will use the greek letter theta (θ) to denote these latent ability scores.

In the subsequent text we will make the assumption that the ability scores are measured on a scale that ranges from negative to positive infinity, with a midpoint of zero, and the unit of measurement of one. (Note that this is just to fix an arbitrary measurement system, we do not assume anything about the distribution of ability scores.)

The usual way of estimating the ability scores is by constructing a test. The test usually consists of a set of questions, called items. In the ideal case, each of the items is a free-response question with binary (dichotomous) scoring. I.e., the subject is allowed to give any response, and afterwards the test evaluator decides whether the answer was correct or not.

Here we can describe the difference between the CTT and the IRT in more detail. In CTT, the subject is awarded an additive score for each correctly answered item. The total score for the test is then used to estimate the subject's latent ability score θ. IRT takes into consideration the responses to individual items of a test rather than just some aggregate of the responses. Thus, from this point of view, IRT can be seen as a superior test theory.

The ability of a subject to pass a test item will depend on the subject's ability score θ. As stated above, in IRT we take into consideration the individual test items. More precisely, we assume that each item comes with an inherent *item characteristic*. The item characteristic is a function that maps the subject's ability score θ to the probability that the subject answers the given item correctly. The graph of the item characteristic function is known as the item characteristic curve.

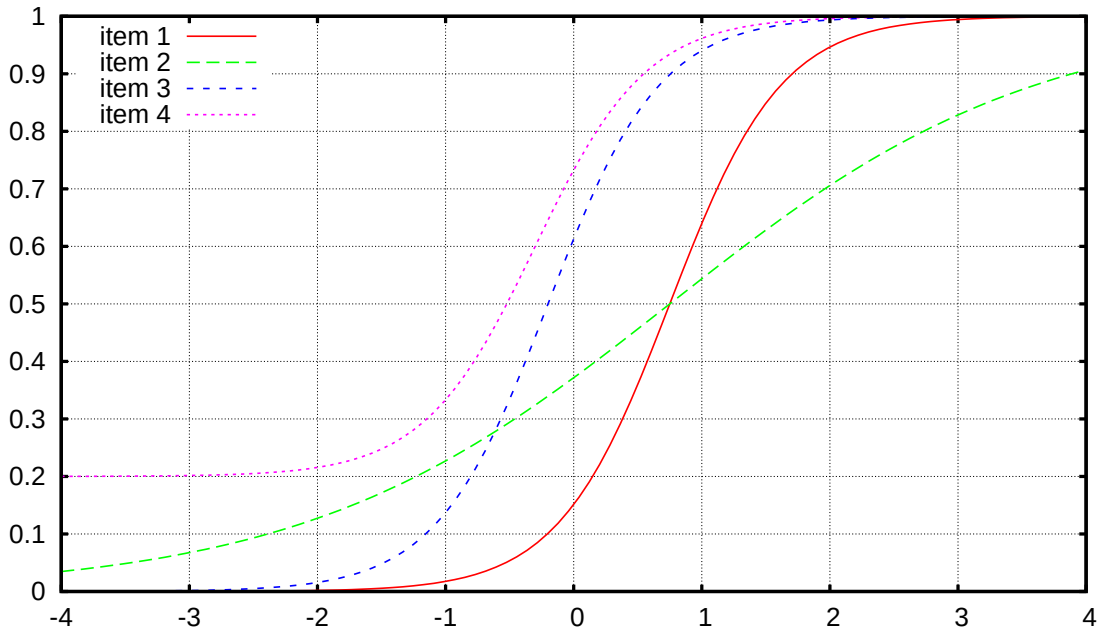


Figure 2.1: Four different item characteristic curves.

In Figure 2.1 we can see four hypothetic item characteristic curves. The property they all share is that for large values of θ the probability approaches 1. Visually, the two most important properties of the curve are its location and steepness.

Formally, the *location index* of an item characteristic curve P is the value θ where $P(\theta) = 0.5$. The location index of a curve corresponds to the difficulty of the given item – the higher the index, the more difficult the item is to solve.

We can note that in Figure 2.1 items 1 and 2 have the same location index 0.75. However, their characteristic curves are still substantially different. This can best be described by the steepness of the curve in the middle section. For historical reasons, this parameter is called the *discrimination* of the item. (It specifies the item's ability to discriminate between subjects having abilities below and above the location index.)

Furthermore, we can see that the characteristic function for item 4 does not converge to zero. Such item characteristic functions occur e.g. whenever the subjects are able to guess the correct answer. (An example of such a setting is a test where each question has a set of possible answers, and the subject has to pick the correct one.)

Of course, the item characteristic functions of real-life test items may differ from the functions plotted in Figure 2.1. There are various IRT models that attempt to approximate the true item characteristic function using different parametrized mathematical functions. We will introduce some of the basic IRT models in the next Section.

2.3.3 Logistic IRT models

One-parameter logistic model (1PL)

The one-parameter logistic model is the simplest in the family of logistic models. In this model the function used to approximate the item characteristic function has a single parameter b – usually called the *difficulty parameter* or the *location parameter*.

In this model, the probability of a subject with ability θ correctly solving a task with difficulty b is given by equation (2.4). Plots of this equation as a function of θ for different values of b are given in Figure 2.2.

$$Pr(\theta, b) = \frac{1}{1 + e^{-(\theta - b)}} \tag{2.4}$$

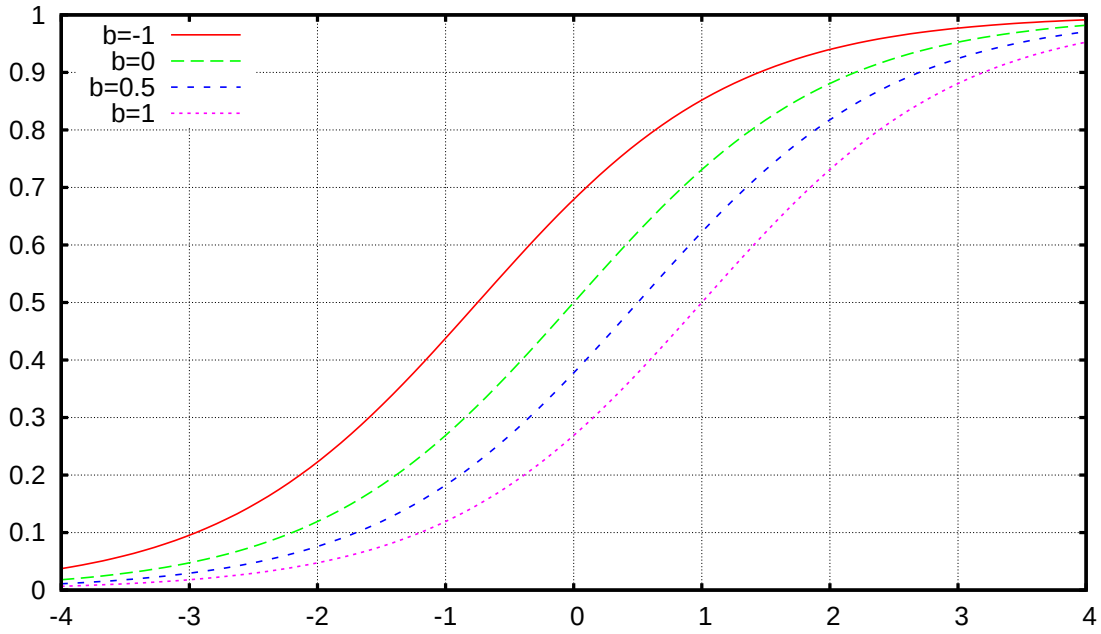


Figure 2.2: The one-parameter logistic function.

Two-parameter logistic model (2PL)

The basic disadvantage of the 1PL model is the uniformity of all its possible item response functions. In many real life situations we can not accurately describe the tasks using a single parameter only. The item response functions for two tasks with the same difficulty can often differ substantially. The main difference is the steepness of the curve. For a test item with a shallow item response curve, the probability of giving a correct answer is less influenced by the actual ability level θ.

Formally, in the 2PL model, the difficulty of a task is modelled by two parameters: the difficulty parameter b, and a new parameter a. This parameter is called the *discrimination parameter*, or the *slope parameter*.

The item response function of this model is given by equation (2.5). Plots of this equation as a function of θ for different values of b and a are given in Figure 2.3.

$$Pr(\theta, a, b) = \frac{1}{1 + e^{-a(\theta - b)}} \tag{2.5}$$

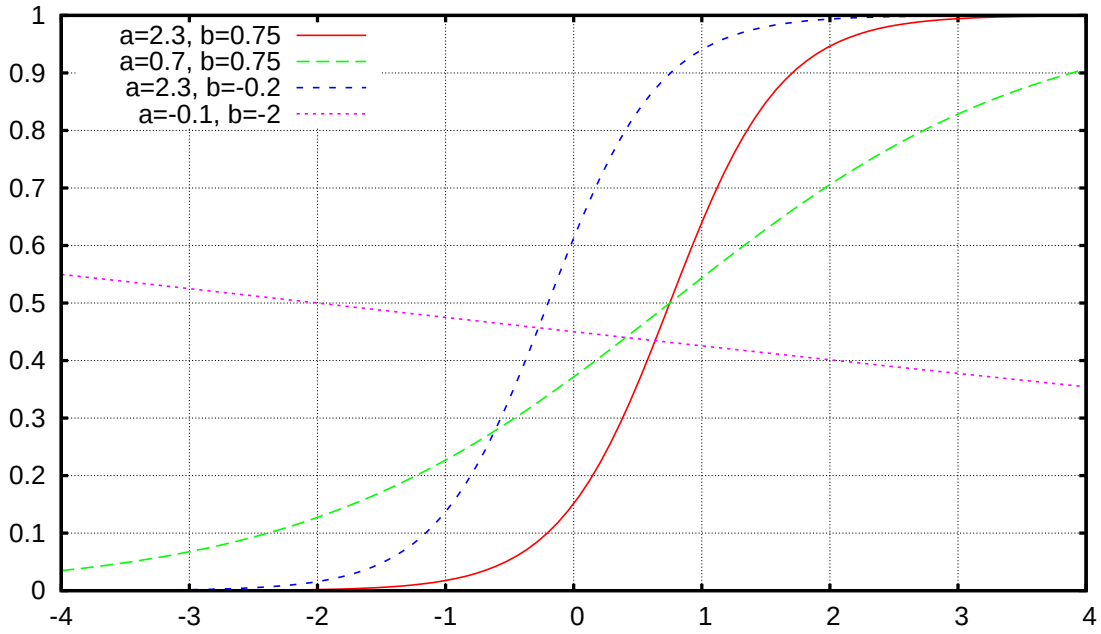


Figure 2.3: The two-parameter logistic function.

Note that in practice even negative values of a might be possible. However, negative values of a usually signify that there's something suspicious with the item. (Such as poor wording of a question that may confuse talented students.)

Three-parameter logistic model (3PL)

In the two-parameter logistic model, we always have $\lim_{\theta \to -\infty} Pr(\theta, a, b) = 0$. In real life, this is not the case in situations where the subject might be able to guess the correct answer. This is where the 3PL model differs from the 2PL model – we incorporate a third task parameter that models the probability of guessing correctly.

The item response function for this model is given by equation (2.6). We will not be using this model, as in our settings guessing does not apply. In the subsequent sections we will only consider the 2PL model.

$$Pr(\theta, a, b, c) = c + \frac{1-c}{1+e^{-a(\theta-b)}} \tag{2.6}$$

2.3.4 Ability estimation in the 2PL model

The main goal in both CTT and IRT is estimation of the latent ability parameter θ.

Assume that the subject took a test that consisted of several tasks. For each of these tasks we know its item response function Pr_i, and we also know whether the subject solved it or not.

In this scenario, we want to compute the *maximum likelihood estimate* of the unknown variable θ.

Formally, suppose that we have a random variable X with a parametrized probability distribution function f_y. We measured the random variable X and got the result x. The *likelihood* $L(y, x)$ of a given parameter value y is defined as the conditional probability $P(X = x|y)$. (Note that this is **not** the probability of y being the true parameter.)

In the 2PL model, we can then compute the maximum likelihood estimate of the unknown ability parameter θ as follows:

The subject was given n tasks. For each of these tasks we know its 2PL parameters a_i and b_i. We also know whether the subject solved each of the tasks. Formally, let $s_i = 1$ if the i-th task was solved correctly, $s_i = 0$ otherwise. The values s_i are usually called the *response pattern*.

$$L(\theta) = \prod_{i=1}^{n} Pr(\theta, a_i, b_i)^{s_i} \cdot (1 - Pr(\theta, a_i, b_i))^{1-s_i} \tag{2.7}$$

Equation (2.7) gives the likelihood of θ being the ability level, given the response pattern (s_i). Note that for correctly answered questions the multiplicand reduces to $Pr(\theta, a_i, b_i)$, whereas for incorrectly answered questions it is $(1 - Pr(\theta, a_i, b_i))$. In both cases, this is the probability of response s_i in our 2PL model.

To avoid maximizing a product of functions, we can define the log-likelihood of the parameter θ. The definition is given in equation (2.8). Obviously, the functions L and L' achieve their maxima for the same value of θ.

$$\begin{aligned} L'(\theta) &= \sum_{i=1}^{n} \ln \left(Pr(\theta, a_i, b_i)^{s_i} \cdot (1 - Pr(\theta, a_i, b_i))^{1-s_i} \right) \\ &= \sum_{i=1}^{n} s_i \cdot \ln Pr(\theta, a_i, b_i) \quad + \quad (1 - s_i) \cdot \ln(1 - Pr(\theta, a_i, b_i)) \end{aligned} \tag{2.8}$$

2.3.5 Fisher information

Whenever we observe a random variable, this observation gives us information that we can use to make a better estimate of the hidden parameters. This statistical version of information was first formalized by Sir Robert Fisher.

Intuitively, we can define information as the reciprocal of the precision with which the parameter could be estimated. We will now give a formal definition that matches this intuition.

Let X be a random variable such that its probability distribution depends on a parameter θ. Let $L(\theta, x)$ be the likelihood function. Then the *score* V is the partial derivative with respect to θ of the natural logarithm of the likelihood function. Formal definition is given in equation (2.9). We can rewrite the partial derivative of the logarithm as in (2.10).

$$V(\theta, x) = \frac{\partial}{\partial \theta} \ln L(\theta, x) \tag{2.9}$$

$$= \frac{1}{L(\theta, x)} \cdot \frac{\partial L(\theta, x)}{\partial \theta} \tag{2.10}$$

We can now define *Fisher information* as the expected variance of the score. It can easily be seen that the expectation of the score is always zero. Thus the Fisher information is the expectation of the square of the score. Formal definition is given in equation (2.11)

$$\mathcal{I}(\theta) = E\left(\frac{\partial}{\partial \theta} \ln L(\theta, x) \right)^2 \tag{2.11}$$

Note that the Fisher information is only a function of the parameter θ, the observed value x has been averaged out.

We will now informally explain the meaning of Fisher information. Let θ be the maximum likelihood estimate of the hidden parameter, as defined in Section 2.3.4. If the Fisher information for this particular value of θ is high, this means that the absolute value of the score is often high. The score is the partial derivative of the log-likelihood function, and thus it follows that in the vicinity of θ the log-likelihood function is steep, and thus the precision of this estimate is high.

It will be useful to know that Fisher information is additive (as is the case with other types of information). More precisely, if we make two independent experiments, the total Fisher information is the sum of the Fisher informations for each experiment. This trivially follows from the fact that for independent random variables variance is additive.

2.3.6 Item and test information functions

We can now apply the concept of Fisher information to IRT. More precisely, we will show how it can be used in the 2PL model, which will be the main model used in our results.

Each item the subject tried to solve provides some information about the subject's latent ability θ. This amount of information can now be formalized as the Fisher information of the item.

Substituting into the definition of Fisher information, we get:

$$\mathcal{I}(\theta) = E\left(\frac{\partial}{\partial\theta}\Big(s \cdot \ln Pr(\theta, a, b) \quad + \quad (1-s) \cdot \ln(1 - Pr(\theta, a, b))\Big)\right)^2 \tag{2.12}$$

Here, Pr is the two-parameter logistic function defined in equation (2.5), a and b are the item parameters, and $s \in \{0, 1\}$ specifies whether the subject solved the item.

In this setting, the Fisher information function of the single item can be simplified to:

$$\mathcal{I}(\theta) = a^2 Pr(\theta, a, b)(1 - Pr(\theta, a, b)) \tag{2.13}$$

The most important consequence is that the item information function attains its maximum for $\theta = b$, i.e., when the subject's ability is equal to the difficulty parameter. The value of this maximum is proportional to the square of the discrimination parameter. Thus an item with a higher discrimination parameter gives us substantially more information (for θ values close to the item's difficulty).

Figure 2.4 shows a plot of two item response functions with different discrimination parameters, and their corresponding item information functions.

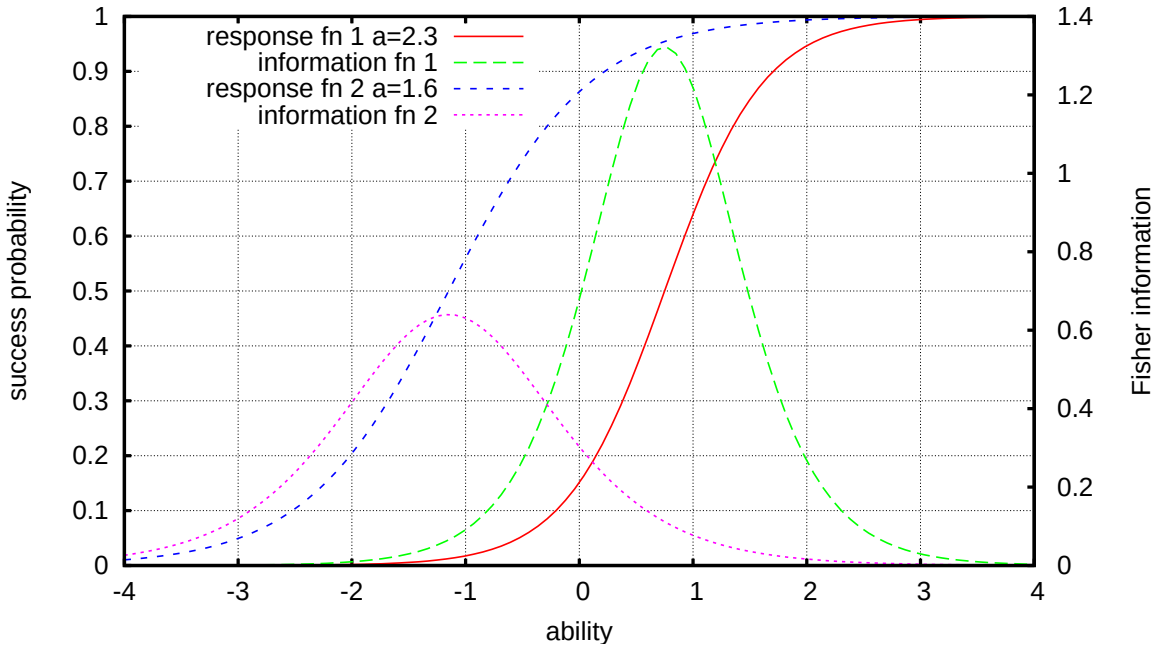


Figure 2.4: Item information functions in the 2PL model.

As we assume that the subjects' performances for different items are independent, the information function for a set of items is simply the sum of their information functions.

$$\mathcal{I}(\theta) = \sum_{i=1}^{n} a_i^2 Pr(\theta, a_i, b_i)(1 - Pr(\theta, a_i, b_i)) \tag{2.14}$$

Figure 2.5 shows a plot of the test response function and test information function for a test consisting of the two items shown in Figure 2.4.

We can now make an interesting observation: When designing a set of items to be used as a test, we would usually like the test information function to be as uniform as possible on the relevant range of θ. In the 2PL model, to achieve this we need a reasonable number of tasks, and their difficulties should be distributed along the relevant range.

However, note that this is not necessarily the case – for example, in Section 5.3 we present our analysis of the regional round of the Slovak Olympiad in Informatics. For this competition the main goal should be to identify the set of participants for the national round as reliably as possible – and to achieve this, one should maximize the test information function on a suitable interval around the expected ability threshold.

In other words, the test information function is just a powerful tool at our disposal, and it's up to us to set our goals, and then use the test information function either in the process of achieving those goals, or in the process of analysis whether the goals were achieved.

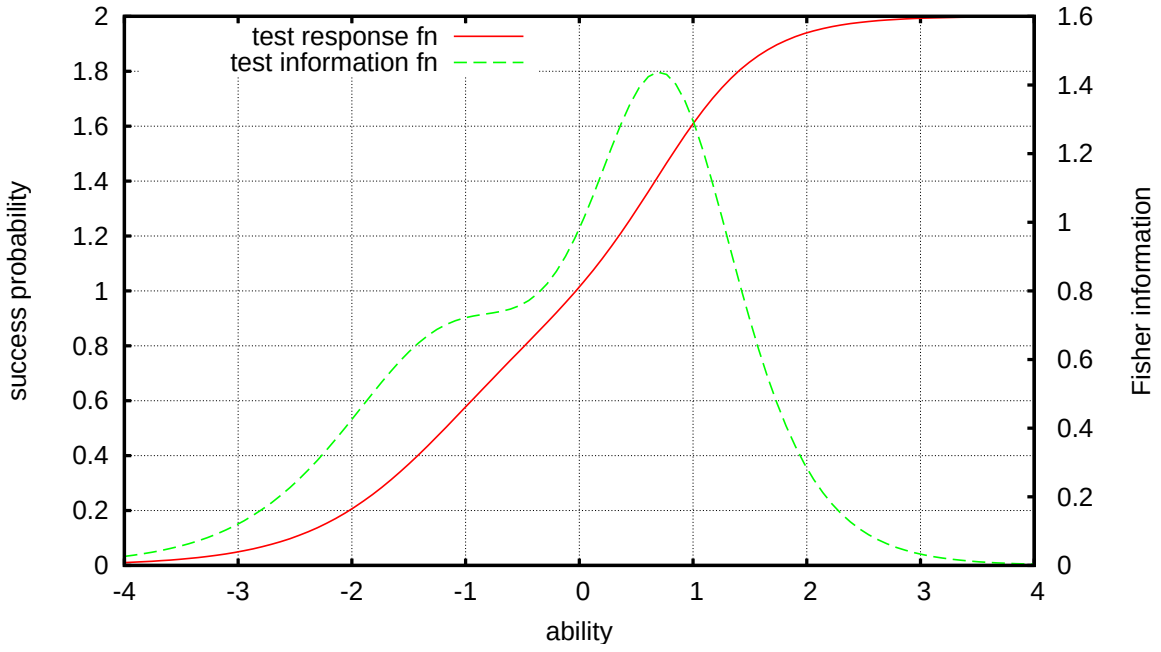


Figure 2.5: Test information function in the 2PL model.

2.3.7 Variance of the ability estimate

Given an ability estimate $\hat{\theta}$, the variance of this estimate can be estimated as the reciprocal of the test information function at that point:

$$Var(\hat{\theta}) = \frac{1}{\mathcal{I}(\hat{\theta})} \tag{2.15}$$

The standard error of measurement (SEM) is defined as the square root of the variance:

$$SEM(\hat{\theta}) = \sqrt{\frac{1}{\mathcal{I}(\hat{\theta})}} \tag{2.16}$$

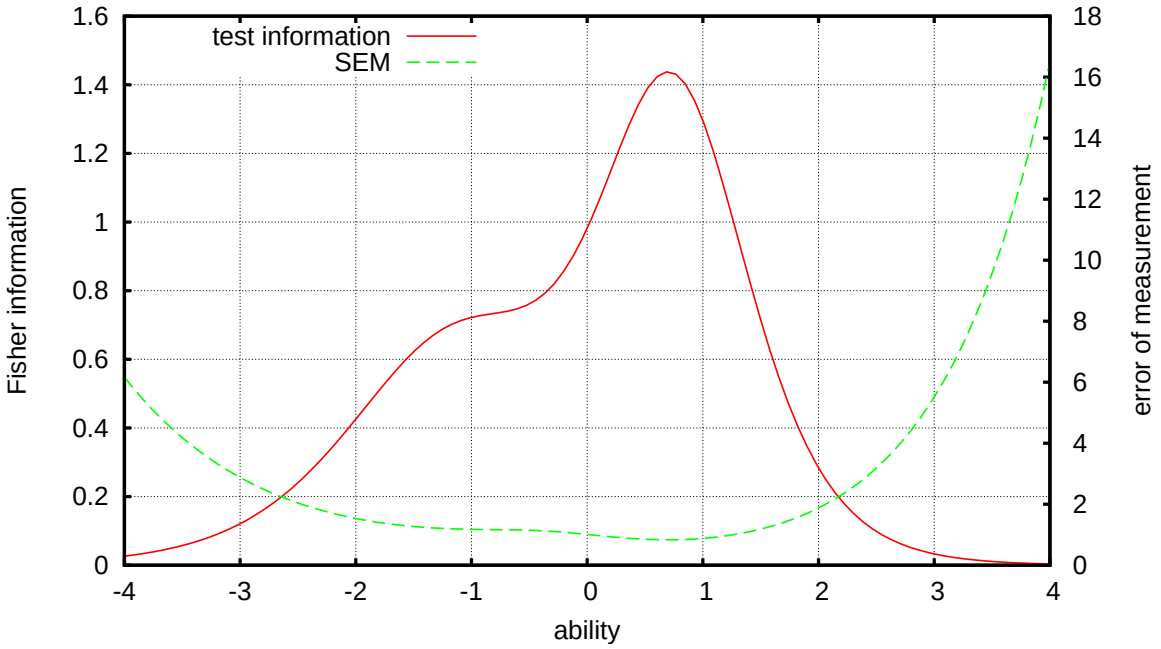


Figure 2.6: Test information function and the standard error of measurement.

In the case of the 2PL logistic model, we get

$$SEM(\hat{\theta}) = \sqrt{\frac{1}{\sum_{i=1}^{n} a_i^2 Pr(\theta, a_i, b_i)(1 - Pr(\theta, a_i, b_i))}} \tag{2.17}$$

The standard error of measurement for the sample test information function from Figure 2.5 is plotted in Figure 2.6

When designing an advanced rating algorithm, we will (in Section 4.2.3) incorporate the standard error of measurement into the process of making predictions about the future – subjects who were not tested sufficiently well (i.e., those who were tested either on a few items only, or on items of unsuitable difficulty), will have a high standard error of measurement, and this gives our rating system additional information it can use to treat them differently from subjects with the same ability estimate and a low standard error of measurement.

2.3.8 Notes on programming competitions

For a few years the International Olympiad in Informatics (IOI) community is concerned about the accuracy of the testing, scoring and ranking process. Several publications that research various aspects of this problem include [Cor06, CMVK06, For04, For06a, Opm06, vL05, Ver06, Yak06],

A publication particularly relevant to the topic of this Thesis is [KVC06] where Kemkes et al. use Item Response Theory to analyze scoring used at IOI 2005, and in particular the impact of the proportion of "easy" and "hard" test cases on the relevancy of the competition results. Based on the results of the analysis, new scoring methods with better discrimination are suggested.

(We would like to note that similar research has recently been conducted for other competitions as well, for example see [Gle08] for an analysis of two mathematical competitions.)

However, we would like to note that while these publications use IRT only passively, as a tool for analysis of tasks only, we use IRT as an active tool to make estimates and predictions.

Chapter 3

Basic IRT-based rating system

In this chapter we present our approach to the rating and ranking problem. We will design an algorithmic rating system that will be based on Item Response Theory.

Our rating system was designed with programming competitions in mind, but it can be succesfully employed in various areas. The rating systems will be particularly useful in situations where the conditions in the rated events are non-uniform. Examples of such areas include science olympiads and other exam-type competitions, computer games tournaments (matches played on different maps), and sports such as golf, white water slalom and alpine skiing (courses and tracks of different difficulty),

In order to reach this goal we define a formal setting in which we will formulate the rating algorithm. Once we have this formal setting, we develop methodology that can be used for comparing rating algorithms – the main idea of this part is that while we are not able to compare rating systems directly, we are able to compare the predictions that can be made based on the rating systems.

We then discuss and prove several important properties an IRT-based rating system must have, define the basic IRT-based rating algorithm, and describe a prediction algorithm that can use the ratings to make predictions that include but are not limited to predictions that could be made using the Bayesian rating systems.

3.1 Assumptions and setting

The setting consists of a set of subjects, and a series of events. Each event will allow us to deduce information on the latent ability of the subjects we are trying to measure.

In this section we summarize the set of assumptions we make about the setting for which we design the rating system.

Essential assumptions

scalar ability

We make the assumption that the latent ability that influences a subject's success in the events can be expressed as a scalar, i.e., a single real number.

binary evaluation of items [1]

For each item, we have an exact way to evaluate whether the subject passed (solved) the item or not.
(Note that a single event may contain multiple items.)

non-antagonistic

The subjects' performances are independent from each other and only depend on the item parameters.

(Note that this is not necessarily completely true in some settings where the subjects have access to the other subjects' performances, such as online standings. Still, if the influence of such information is determined to be negligible, we expect that it will still be possible to successfully apply our models.)

Additional assumptions

Additionally, in many real-life settings the following assumptions are true – and if so, we can exploit these in the design of a better rating system.

various items and events

The nature of the items is such that the probability of a subject's success as a function of the item is not constant – but a function of his ability and some hidden, item-dependent parameters.

long-term

The series of events occurs in time, and each event has an associated timestamp when it happened. We want the ratings to be updated after each event, and to use them to predict the outcomes of future events.

no guessing

The nature of the items is such that the probability of passing an item by chance (e.g., guessing the solution to a test question) is zero, or negligibly small.

timed/additionally evaluated items

Sometimes in addition to a pass/fail value, we also get additional information, such as

[1] Note that publications related to IRT tend to use the word "dichotomous", but we opted to use a term much more frequent in computer science.

the precise time the subject needed for that item. In such situations we assume that the secondary goal of the subject was to minimize this time.

time invariance

Time invariance of ratings has a special status: The models presented in this chapter assume that abilities are invariant in time. However, the models can be adjusted to take ability changes into account. This topic is addressed later, in Section 4.2.

Note

It is also useful to list an assumption we *do not* make.

no prior ability distribution

Except for normalizing our measurement scale, we make no assumptions on how the abilities of the subjects are distributed. Especially note that we do not assume a normal distribution of abilities.

In fact, our experiments show that for programming contests the actual distribution is significantly different from a normal one. This observation can partially be explained by the fact that for subjects of low ability the chance that they will actually participate in an event decreases – and the relation between ability and participation probability is non-linear. We address the experimental data in Chapter 5 of this thesis.

3.2 Formal definitions

In order to be able to define our own rating system, discuss and prove its properties, and compare it to existing rating systems, we found that a formal approach is necessary.

In this section, we will define a formal framework that will be able to contain both existing rating systems and our new approach. Note that this framework only covers rating systems for areas that match the assumptions listed in the previous Section – especially *binary evaluation of items* and *non-antagonistic*.

Notation 3.2.1. *For simplicity of definitions below, let $\mathscr{C}$ be the countably infinite set of potential contestants, and let $\mathscr{T}$ be the countably infinite set of potential tasks.*

Notation 3.2.2. *$A \subseteq_{fin} B$ will be used to denote that $A \subseteq B \ \wedge \ |A| < \aleph_0$. That is, $A \subseteq_{fin} B$ means that A is a finite subset of B.*

Definition 3.2.3. *Individual result.*
Individual result is an ordered pair $(result, data)$, where $result \in \{true, false\}$ and $data \in \mathbb{R}^k$ for some fixed $k \geq 0$.

Semantically, *result* specifies whether a task was solved, and *data* contains possible additional data describing properties of the solution (e.g., speed, accuracy, etc.).

Definition 3.2.4. *Round.*
A round is an ordered 4-tuple (τ, C, T, ρ)*, where* $\tau \in \mathbb{R}$*,* $C \subseteq_{fin} \mathscr{C}$*,* $T \subseteq_{fin} \mathscr{T}$*, and* $\rho : C \times T \to \big(\{true, false\} \times \mathbb{R}^k\big) \cup \{\perp\}$ *for some fixed* $k \geq 0$*.*

The number τ is a timestamp of the round (the exact date and time converted to a number on some fixed linear scale), C is the set of participating contestants, and T is the set of tasks used. The function ρ maps each pair $(contestant, task)$ either to the contestant's individual result for that task, or to $\perp$. The special value $\perp$ is used in cases when the given contestant was not tested on the given task.

In the Item Response Theory, each contestant is a subject and each task used in a round represents an item.

Notation 3.2.5. *The set of all possible rounds will be denoted* $\mathscr{R}$*.*

Definition 3.2.6. *Round sequence.*
A round sequence is a finite sequence of rounds.

Notation 3.2.7. *The set of all round sequences will be denoted* $\mathscr{S}$*. Operator* $\circ$ *will be used to denote sequence concatenation.*

Definition 3.2.8. *Contestant/task property.*
Contestant property is any function $f : C \to \mathbb{R}$*, where* $C \subseteq_{fin} \mathscr{C}$*. Similarly, task property is any function* $g : T \to \mathbb{R}$*, where* $T \subseteq_{fin} \mathscr{T}$*.*

Definition 3.2.9. *Contestant accumulator function.*
The contestant accumulator function is a homomorphism Cont from the monoid $(\mathscr{S}, \circ)$ *to the monoid* $(2^{\mathscr{C}}, \cup)$*, defined by* $Cont(\ (\tau, C, T, \rho)\ ) = C$*.*

In words, for a round sequence S the value $Cont(S)$ is the union of the contestant sets for all rounds in S. Similarly, we can define the task accumulator function.

Definition 3.2.10. *Task accumulator function.*
The task accumulator function is a homomorphism Task from the monoid $(\mathscr{S}, \circ)$ *to the monoid* $(2^{\mathscr{T}}, \cup)$*, defined by* $Task(\ (\tau, C, T, \rho)\ ) = T$*.*

Note that different rounds may contain the same task.

Definition 3.2.11. *Rating algorithm.*
Rating algorithm is an algorithm whose input is a round sequence S*. and output is an ordered triple* (θ, Γ, Δ)*, where* θ *is a contestant property,* Γ *is a finite sequence of additional contestant properties, and* Δ *is a finite sequence of task properties. Note that sequences* Γ *and* Δ *are allowed to be empty. The domain of* θ *and* Γ *must be* $Cont(S)$*, and the domain of* Δ *must be* $Task(S)$*.*

Here θ is the rating we are interested in. Additionally, the rating algorithm is allowed to compute additional information, both on contestants and tasks.

Two new definitions follow. These do not match anything related to previously known rating systems. They are the result of our research, and their meaning and use will be explained in detail in later sections.

Definition 3.2.12. *Prediction distance metric.*
Let $C \subseteq_{fin} \mathscr{C}$ be a fixed set of contestants, and let $\mathcal{P}$ be a fixed set of valid predictions. Let $\mathscr{R}_C$ be the set of all possible rounds such that $Cont(R) = C$. A prediction distance metric is a function $\mu : \mathscr{R}_C \times \mathcal{P} \to [0, \infty)$.

Semantically, given a prediction $P \in \mathcal{P}$ and a round $R \in \mathscr{R}_C$, the function μ evaluates how closely the round R matches the prediction P. We will use the value 0 to denote a perfect match.

Note that the returned value is not supposed to be an absolute measure of the goodness of prediction, just a relative one – in other words, the only use will be: "prediction P_1 was better than prediction P_2 for the actual result of the round R iff $\mu(P_1, R) < \mu(P_2, R)$".

Definition 3.2.13. *Prediction algorithm.*
Let $C \subseteq_{fin} \mathscr{C}$ be a fixed set of contestants, and let $\mathcal{P}$ be a fixed set of valid predictions. A prediction algorithm is an algorithm whose input is the output of the rating algorithm and the set C, and whose output is an element from the set $\mathcal{P}$.

The semantics of the output is that the algorithm tries to predict some attribute related to a new round that will be involving the contestants in C.

3.3 Comparing rating systems

In situations where we have multiple rating systems, it is only natural to ask which of them is *better*. The word *better* can have multiple meanings, the most natural one (but by far not the only one) being "which of them estimates the true latent ability more precisely".

It is obviously impossible to answer this question directly, as we are not able to measure the latent ability. Thus there is no way to directly compare rating systems.

However, there is a natural way out. Having a model and a set of rating estimates should enable us to predict the outcome of a future rated event. The more accurate the prediction, the more trustworthy the pair (rating system, prediction algorithm) is.

In other words, while we can not directly compare rating systems, we can compare rating systems accompanied by prediction algorithms. This is exactly the rationale behind Definition 3.2.13 (prediction algorithm).

Still, the previous paragraphs leave one open question: what exactly makes a prediction more accurate? The answer is not unique. Moreover, the answer to this question is actually

what we should start with in practice. In antagonistic situations (two-player matches) the focus is usually on predicting the winner. The non-antagonistic setting with multiple subjects allows for a much richer spectrum of possible goals. To name some: predicting placement, probability of advancing in a tournament, probability of making it to the top N, etc.

In our formalism we capture this intuition in Definition 3.2.12 (prediction distance metric).

In the rest of this section we shall define and discuss several natural prediction distance metrics.

Predicting placement

Definition 3.3.1. *Round ranking function.*
A round ranking function is a function $f : \mathscr{R} \times \mathscr{C} \to \Big(\mathbb{N} \cup \{\bot\}\Big)$ *such that* $f\big((\tau, C, T, \rho), c\big) \in \big\{1, 2, \ldots, |\mathcal{C}|\big\}$ *iff* $c \in C$*, and* $f\big((\tau, C, T, \rho), c\big) = \bot$ *otherwise.*

Semantically, the round ranking function maps each contestant to his position in the round ranklist, and this mapping depends on the actual round result ρ.

Example: A commonly used round ranking function is the function *place* defined below.

$$solved : \mathscr{R} \times \mathscr{C} \to \mathbb{N}_0 \tag{3.1}$$

$$place : \mathscr{R} \times \mathscr{C} \to \Big(\mathbb{N} \cup \{\bot\}\Big) \tag{3.2}$$

$$solved\big((\tau, C, T, \rho), c\big) = \Big|\big\{t \mid t \in T \ \wedge \ (\exists w;\ \rho(c, t) = (true, w)\big\}\Big| \tag{3.3}$$

$$place\big((\tau, C, T, \rho), c\big) = 1 + \Big|\big\{c' \mid c' \in C \ \wedge \ solved(c') > solved(c)\big\}\Big| \tag{3.4}$$

Note that we opted not to define any additional constraints on the function f. The motivation is twofold: First, in different situations different round ranking functions are used. For example, a round ranking function is not necessarily a bijection. Second, we do not need any additional properties of the function f to define the following metric.

Definition 3.3.2. *Expected placement prediction metric.*
Let $\mathcal{C}$ *be a fixed contestant set, and let* f *be a fixed round ranking function. Let* $\mathcal{P}$ *(the set of valid predictions) be the set of all functions from* $\mathcal{C}$ *to* $\big[1, |\mathcal{C}|\big]$*.*
The expected placement prediction metric function is the function μ_{EP} *defined as follows:*

$$\mu_{EP} : \mathscr{R}_C \times \mathcal{P} \to [0, 1] \tag{3.5}$$

$$\mu_{EP}(R, p) = \frac{\sum_{c \in \mathcal{C}} \big|p(c) - f(R, c)\big|}{|\mathcal{C}|^2} \tag{3.6}$$

Note that in the definition we allow non-integer predictions. More precisely, the predictions must be real numbers between 1 and the number of contestants, inclusive.

Also note that there are other possible ways to define this metric, and they may be more appropriate based on the concrete situation. As one important example one could take the quadratic mean of the errors instead of the arithmetic one – then the value returned by the metric becomes directly proportional to the Euclidean distance between two points in $\mathbb{R}^n$ that represent the predicted and actual placements.

Predicting the number of items solved

For each contestant c that takes part in the event we may want to predict the number of items he will solve. As in the previous section, a natural and simple way of evaluating such a prediction is to accumulate the absolute values of errors made in the predictions.

Definition 3.3.3. *Solved item count.*
The solved item count function is a function $solved : \mathscr{R} \times \mathscr{C} \to \Big(\mathbb{N} \cup \{\perp\}\Big)$ *such that* $solved\big(\,(\tau, C, T, \rho)\,, c\big) = \big|\{t \mid t \in T \ \wedge \ \exists data; \rho(c,t) = (true, data)\}\big|$ *iff* $c \in C$, *and* $solved\big(\,(\tau, C, T, \rho)\,, c\big) = \perp$ *otherwise.*

As in the previous case, there are now multiple ways how to define the corresponding prediction metric. In essence, any classical metric (in the topological sense) defined on $(\mathbb{R}_0^+)^n$ can be used to define a reasonable prediction metric. Below we will define the prediction metric based on the Euclidean distance.

Definition 3.3.4. *Solved item count prediction metric.*
Let $\mathcal{C}$ *be a fixed contestant set. Let* $\mathcal{P}$ *(the set of valid predictions) be the set of all functions from* $\mathcal{C}$ *to* $\mathbb{R}_0^+$. *Then the solved item count prediction metric is the function* μ_{SIC} *defined as follows:*

$$\mu_{SIC} : \mathscr{R}_C \times \mathcal{P} \to [0, 1] \tag{3.7}$$

$$\mu_{EP}\big(\,(\tau, \mathcal{C}, T, \rho)\,, p\big) = \frac{\sum_{c \in \mathcal{C}} \big(p(c) - solved(R, c)\big)^2}{|\mathcal{C}| \cdot |T|^2} \tag{3.8}$$

This metric is actually the mean square error of the prediction – in other words, the square of the Euclidean distance between the prediction and the reality, normed so that it can never exceed 1.

Discussion of use cases

Here we would like to note that in general this is **not** a reasonable measure of prediction accuracy. In our setting, the expected number of items solved depends not only on the predicted ratings, but also on item parameters of the items in the event.

However, it makes sense to use this measure, if either of these conditions is satisfied:

- The item parameters for the items used in the predicted event are known to the prediction algorithms. (In this case we should alter the definition to take this into account.)
- The rating systems and corresponding prediction algorithms are compared on a large set of events such that the distribution of item parameters for used items is close to the true distribution.

Generalizations

This prediction metric can be generalized in multiple ways.

For example, in some settings we may design prediction algorithms that predict not only the number of items solved, but also the additional data – such as score and solving time. One can cover all of these situations by the general term "score prediction", where the score is a function of the contestants' item response correctness and other parameters.

Warnings issued for the original measure are valid for this generalized measure as well. Additionally, one can easily find situations where the scoring function is significantly non-linear, and in these cases one has to define the metric accordingly. (Examples of such score functions include "for a hard task, either you score many points or zero" in TopCoder competition and "+30 seconds for missing a gate" in white water slalom.)

Another reasonable variation on this theme might be predicting the number of contestants with a positive score.

Alternative

One possible alternate way of defining the previous two metrics would be to use the Pearson product-moment correlation coefficient between the vectors of predicted and actual data.

However, there are well-known examples (for example the Anscombe's quartet [Ans73]) where high correlation does not imply linear dependence, hence this approach may not meet our expectations.

We have to reiterate the statement we made in the introduction to Section 3.3: The choice of the actual prediction distance metric to use must be made based on the practical needs – one needs to identify the type of errors that are considered most harmful, and then pick the alternative that penalizes those errors.

Estimating probability of reaching a percentile

The probability of winning a round is a natural thing to predict, along with "being in the top 3". Also, for tournaments rounds of the type "best K advance" it is natural to predict the probability that a given contestant will advance.

All of these questions can be covered by a more general task: Given a set of contestants C, a round ranking function f and a real number $d \in [0, 1]$, for each contestant $c \in C$, estimate

the probability that $f(R, c) \leq d|C|$.

Formally, the set of valid predictions $\mathcal{P}$ will be the set of functions from C to $(0, 1]$.

The most natural way of evaluating such predictions is to use the maximum likelihood approach. In words, the more likely the actual outcome is given the estimated probabilities, the better the estimate.

Formally, let p be our prediction, and let R be the round for which we were making it. Set $s_c = 1$ if $f(R, c) \leq d|C|$, and $s_c = 0$ otherwise. We will now calculate the probability $\pi(R, p)$ of the following event: "If we generate values t_c for $c \in C$ independently at random, where t_c is 1 with probability $p(c)$ and 0 otherwise, then for all c we have $t_c = s_c$."

The probability $\pi(R, p)$ can be calculated as follows: as all the t_c are generated independently, $\pi(R, p)$ is the product of individual probabilities that $t_c = s_c$. Fix a contestant c. If $s_c = 1$, the probability that $t_c = s_c$ is $p(c)$, otherwise it is $1 - p(c)$. We can put this together as follows:[2] the probability that $t_c = s_c$ is $p(c)^{s_c}(1 - p(c))^{1-s_c}$. Thus we get:

$$\pi(R, p) = \prod_{c \in C} p(c)^{s_c}(1 - p(c))^{1-s_c} \tag{3.9}$$

The larger the value $\pi(R, p)$, the better the actual result of R corresponds to the prediction p.

For large $|C|$ the value $\pi(R, p)$ gets very small, which can lead to numeric errors when evaluating it. Therefore we will define our metric in terms of $\ln \pi(R, p)$. Note that ln is increasing on $(0, 1]$, therefore $\pi(R, p_1) < \pi(R, p_2)$ iff $\ln \pi(R, p_1) < \ln \pi(R, p_2)$. We define the metric μ_{perc} as follows:

Definition 3.3.5. *Percentile reaching probability metric.*
The percentile reaching probability prediction metric is the function μ_{perc} defined as follows:

$$\mu_{perc} : \mathscr{R}_C \times \mathcal{P} \to [0, 1] \tag{3.10}$$

$$\mu_{perc}(R, p) = -\ln \pi(R, p) \tag{3.11}$$

$$= -\ln \prod_{c \in C} p(c)^{s_c}(1 - p(c))^{1-s_c} \tag{3.12}$$

$$= -\sum_{c \in C} \ln \left(p(c)^{s_c}(1 - p(c))^{1-s_c} \right) \tag{3.13}$$

$$= -\sum_{c \in C} s_c \ln p(c) + (1 - s_c) \ln(1 - p(c)) \tag{3.14}$$

An alternate way of looking at the same definition: If we divide μ_{perc} by $\ln 2$ (or, equivalently, take a base-2 logarithm of $\pi(R, p)$), we get precisely the surprise from seeing the outcomes s_c in round results generated with probability distribution given by p.

Note that a similar definition can be made for **any** scenario where we try to predict probabilities.

[2] There are other ways how to write this down formally. Here we picked one that will be beneficial later on, in equation (3.14).

3.4 TopCoder's rating algorithm

TopCoder's own description of their rating system can be found at [Top08]. In this section we present TopCoder's rating algorithm as an example of a Bayesian rating algorithm used in practice. In formulating this rating algorithm we show that our formalism is powerful enough to describe an existing, complicated rating system. Also, we will reference this rating algorithm in later sections – for example, we will compare its results with the results obtained by our new rating algorithm(s) on available data sets.

Let $S = (S_1, S_2, \ldots, S_N)$ be a round sequence – i.e., the input of the rating algorithm. The algorithm will output two contestant properties: the rating θ and the volatility vol.

The algorithm proceeds iteratively, processing increasing prefixes of S and computing intermediate estimates of the two properties. More precisely, the algorithm will successively compute the properties θ_i, vol_i and $events_i$ for each $i \in \{1, \ldots, N\}$, and for each i the properties θ_i, vol_i and $events_i$ will be computed only from θ_{i-1}, vol_{i-1} and $events_{i-1}$ (if applicable), and S_i.

We will start with $\theta_0 = vol_0 = events_0 = \emptyset$.

Now assume that we already computed the properties θ_{i-1}, vol_{i-1} and $events_{i-1}$ for some i.

Let $PrevCont = Cont((S_1, \ldots, S_{i-1}))$, and let $NewCont = Cont(S_i) - PrevCont$ be the sets of contestants with and without a previous appearance, respectively.

First of all, we can easily define $events_i$ as follows:

$$\forall c \in NewCont: \quad events_i(c) = 1 \tag{3.15}$$

$$\forall c \in Cont(S_i) - NewCont: \quad events_i(c) = events_{i-1}(c) + 1 \tag{3.16}$$

$$\forall c \in PrevCont - Cont(S_i): \quad events_i(c) = events_{i-1}(c) \tag{3.17}$$

We now extend the properties θ_{i-1} and vol_{i-1} to include the contestant in $NewCont$ as follows:

$$\forall c \in PrevCont: \quad \theta'_{i-1}(c) = \theta_{i-1}(c) \tag{3.18}$$

$$\forall c \in NewCont: \quad \theta'_{i-1}(c) = 1200 \tag{3.19}$$

$$\forall c \in PrevCont: \quad vol'_{i-1}(c) = vol_{i-1}(c) \tag{3.20}$$

$$\forall c \in NewCont: \quad vol'_{i-1}(c) = 535 \tag{3.21}$$

In words, the newcomers are all treated as contestants with ability 1200 and volatility 535.

For the round S_i we compute the average contestant rating:

$$\bar{\theta}_i = \frac{\sum_{c \in Cont(S_i)} \theta'_{i-1}(c)}{|Cont(S_i)|} \tag{3.22}$$

and the contest hardness factor

$$\begin{aligned} CHFactor_i = & \sqrt{\frac{\sum_{c \in Cont(S_i)} vol'_{i-1}(c)^2}{|Cont(S_i)|} + \cdots} \\ & \overline{\cdots + \frac{\sum_{c \in Cont(S_i)} \left(\bar{\theta}_i - \theta'_{i-1}(c)\right)^2}{|Cont(S_i)| - 1}} \end{aligned} \tag{3.23}$$

In the next definition we shall use the standard error function Erf:

$$Erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt \tag{3.24}$$

The rating system assumes that each contestant's performance can be represented as a normally distributed random variable with some mean and variance. The contestant properties θ_i and vol_i aim to approximate the mean and standard deviation of the contestant c's random variable.

Accordingly to this model, the probability that in the round S_i the contestant c_1 finishes better than the contestant c_2 can be expressed as follows:

$$WP_i(c_1, c_2) = 0.5 \left(Erf \left(\frac{\theta'_{i-1}(c_1) - \theta'_{i-1}(c_2)}{\sqrt{2\left(vol'_{i-1}(c_1)^2 + vol'_{i-1}(c_2)^2\right)}} \right) + 1 \right) \tag{3.25}$$

The expected placement of a contestant c in the round S_i can now be computed simply as 1 plus the sum of probabilities that each of the other contestants performs better than c.

$$ExpRank_i(c_1) = 0.5 + \sum_{c_2 \in Cont(S_i)} WP_i(c_2, c_1) \tag{3.26}$$

Note that we sum over all contestants including c_1, and $WP_i(c_1, c_1) = 0.5$, hence we only add 0.5 to the sum instead of 1.

The actual placement of the contestant is defined using the *place* function (3.4) as follows:

$$\begin{aligned} ActRank_i(c) = 0.5 & + \left| \{d \mid d \in Cont(S_i) \ \wedge \ place(S_i, d) < place(S_i, c)\} \right| \\ & + 0.5 \cdot \left| \{d \mid d \in Cont(S_i) \ \wedge \ place(S_i, d) = place(S_i, c)\} \right| \end{aligned} \tag{3.27}$$

Now, we calculate the expected and actual performance of the contestant c using the cumulative normal distribution Φ.

$$ExpPerf_i(c) = -\Phi \left(\frac{ExpRank_i(c) - 0.5}{|Cont(S_i)|} \right) \tag{3.28}$$

$$ActPerf_i(c) = -\Phi\left(\frac{ActRank_i(c) - 0.5}{|Cont(S_i)|}\right) \tag{3.29}$$

Of course, the actual performance may differ from the expected one. This means that the rating and volatility of the contestant c did not predict the result correctly, and they shall be updated. From the actual performance we may compute the rating that would correspond to this performance:

$$PerfAs_i(c) = \theta'_{i-1}(c) + CHFactor_i\Big(ActPerf_i(c) - ExpPerf_i(c)\Big) \tag{3.30}$$

We now want to update the ratings and volatilities to reflect these differences. For each contestant c we compute the weight of this event for him:

$$coef_i(c) = 0.8 + 0.1[\theta'_{i-1}(c) < 2000] + 0.1[\theta'_{i-1}(c) \leq 2500] \tag{3.31}$$

$$Wt_i(c) = coef_i(c) \cdot \frac{1}{\left(1 - \left(\frac{0.42}{events_i(c)} + 0.18\right)\right)} - 1 \tag{3.32}$$

The equation (3.31) uses Iverson notation: $[p]$ is 1 iff p is true, and 0 otherwise.

The weight of the event determines the influence it has on the rating change. The new rating and volatility for contestant c are calculated as follows:

$$RCap_i(c) = 150 + \frac{1500}{events_i(c)} \tag{3.33}$$

$$\theta_i(c) = \text{median}\left\{\theta'_{i-1}(c) \pm cap,\ \frac{\theta'_{i-1}(c) + Wt_i(c) \cdot PerfAs_i(c)}{1 + Wt_i(c)}\right\} \tag{3.34}$$

$$vol_i(c) = \sqrt{\frac{\big(\theta_i(c) - \theta'_{i-1}(c)\big)^2}{Wt_i(c)} + \frac{vol'^2_{i-1}(c)}{Wt_i(c) + 1}} \tag{3.35}$$

For $c \in NewCont$, the above formula for $vol_i(c)$ is replaced by $vol_i(c) = 385$. This is another one of the multiple ad-hoc parts of this rating system. Such ad-hoc parts make the scientific validity and reliability of this rating system unclear. Additionally, in Section 4.3 we show that such ad-hoc parts can often be used by an attacker.

3.5 Monte Carlo prediction algorithm for Bayesian ratings

The TopCoder rating system is described in detail in Section 3.4. For each contestant c, the rating system provides two values $\theta(c)$ and $vol(c)$. These values estimate the mean and the standard deviation of the random variable that gives the contestant's performance in an event.

In this model, it is assumed that the performances have normal distributions and that they are independent. Thus an event can be simulated simply by generating normal random variables with proper parameters, and selecting the A largest ones.

Note that the scores and item difficulties are completely ignored by this model, only performances are compared.

See Algorithm 1 for a pseudocode implementation. The algorithm uses the Box-Muller transform [BM58] to generate the random variables, and a linear-time selection process (see e.g. [CLRS01], chapter 9). A single iteration of this algorithm is linear in the number of contestants.

(This algorithm was previously known in the public domain, however, we were not able to assign authorship to a particular author.)

We would like to stress two major disadvantages of this prediction algorithm:

- Its convergence is slow – obviously the number of simulation steps necessary to achieve precision ε (with high probability) is at least linear in $1/\varepsilon$.
- It does not easily generalize for multiple round tournaments. It is easily seen that the time complexity necessary to predict the outcome of d consecutive rounds within some fixed ε precision grows exponentially with d.

3.6 Towards an IRT-based rating algorithm

Informally, the ideal ratings would be the maximum likelihood estimates for all the properties – a set of values that maximizes the conditional probability of seeing the actually observed results in the probabilistic model described by the properties. However, we will need to address several technical issues in order to (at least approximately) achieve this goal.

The first obvious problem are all the extremes: contestants that did not solve any task, those who did solve all presented tasks, and tasks that were solved either by no one or by anyone. Clearly we can not deduce anything about these extremes, except for the fact that they are outside the range where deductions can be made.

However, if we have such an extreme, there will automatically be no maximum likelihood estimate – pushing the corresponding parameter towards $+\infty$ or $-\infty$ will always increase the likelihood. For example, if we have a contestant c that solved each task he was given, it would be optimal to set $\theta(c) = \infty$. We will formally prove this below, after the appropriate definitions.

In accordance with (2.8) we can define the global log-likelihood function as given below in Definition 3.6.2. Note that as before we directly define the log-likelihood function instead of the likelihood function, as their maxima coincide and the log-likelihood function is additive, which (among other benefits) is making it more numerically stable.

Algorithm 1: MonteCarloTournamentPrediction

Input: A vector of contestants $\mathcal{C}$
Input: A vector of contestant ratings $rating$
Input: A vector of contestant volatilities vol
Input: Advancer count A
Input: Number of rounds to simulate N
Output: Vector of advancement probabilities P

begin
  foreach $c \in \mathcal{C}$ **do** $P_c \leftarrow 0$
  for $i = 1$ to N **do**
    $X \leftarrow \emptyset$
    foreach $c \in \mathcal{C}$ **do**
      $u \leftarrow uniform\_random(0, 1)$
      $v \leftarrow uniform\_random(0, 1)$
      $n \leftarrow \sqrt{-2 \ln u} \cdot \cos(2\pi v)$
      $x.ability, x.contestant = (n \cdot vol_c + rating_c), c$
      $X \leftarrow X \cup \{x\}$
    end
    $Y \leftarrow$ the set of A elements from X with largest abilities
    foreach $y \in Y$ **do** $P_{y.contestant} \leftarrow P_{y.contestant} + 1$
  end
  foreach $c \in \mathcal{C}$ **do** $P_c \leftarrow P_c / N$
  return P
end

Definition 3.6.1. *Parameter likelihood for a single item.*
Let $R = (\tau, C, T, \rho)$ be a round. We can now define the following helper parameter likelihood function:

$$H'_R : C \times T \times \mathbb{R}^3 \to \mathbb{R} \tag{3.36}$$

$$H'_R(c, t, \theta_c, a_t, b_t) = \begin{cases} 0 & \text{if } \rho(c,t) = \bot \\ \ln Pr(\theta_c, a_t, b_t) & \text{if } \exists d;\ \rho(c,t) = (true, d) \\ \ln\left(1 - Pr(\theta_c, a_t, b_t)\right) & \text{otherwise} \end{cases} \tag{3.37}$$

In words, the function H'_R treats the last three parameters as the ability of the contestant c and the 2PL model parameters of the task t, and returns the likelihood of getting the result $\rho(c,t)$ given these parameters.

Using the above helper function we can define the log-likelihood of the entire sequence of rounds as follows:

Definition 3.6.2. *Parameter likelihood for a sequence of rounds.*
Let S be a sequence of rounds, let CP be the set of all contestant properties defined on $Cont(S)$, and let TP be the set of all task properties defined on $Task(S)$. Then the log-likelihood function L'_S is the following function:

$$L'_S : CP \times TP \times TP \to \mathbb{R} \tag{3.38}$$

$$L'_S(\theta, a, b) = \sum_{S_i \in S} \sum_{c \in Cont(S_i)} \sum_{t \in Task(S_i)} H'_{S_i}(c, t, \theta(c), a(t), b(t)) \tag{3.39}$$

Theorem 3.6.1. *If $\forall t;\ a(t) > 0$ and there is a contestant $c \in Cont(S)$ such that*

$$\begin{aligned} &\forall S_i \in S, \quad S_i = (\tau_i, C_i, T_i, \rho_i) : \\ &c \notin C_i \quad \vee \quad \forall t \in T_i;\ (\rho_i(c,t) = \bot \ \vee\ \exists d;\ \rho_i(c,t) = (true, d)) \end{aligned} \tag{3.40}$$

and for some round $(\tau_i, C_i, T_i, \rho_i)$ and task $t \in T_i$ we have $\rho_i(c,t) \neq \bot$, then the function L'_S has no global maximum.

Proof. Assume the contrary, let θ, a and b be properties for which L'_S has its global maximum. Let $\Psi(\theta, x)$ be the function θ_x defined as follows:

$$\theta_x(c') = \begin{cases} \theta(c') & \text{if } c \neq c' \\ x & \text{otherwise} \end{cases} \tag{3.41}$$

Now define the function $f(x) = L'_S(\Psi(\theta, x), a, b)$. The function $f(x)$ now returns the log-likelihood of the parameters if we change $\theta(c)$ to x.

Now let Q be the part of L'_S invariant to the change in θ:

$$Q = \sum_{S_i \in S} \sum_{\substack{c' \in Cont(S_i) \\ c' \neq c}} \sum_{t \in Task(S_i)} H'_{S_i}(c', t, \theta(c'), a(t), b(t)) \tag{3.42}$$

We can now write f in terms of Q and the variable part:

$$f(x) = Q \quad + \sum_{S_i \in S} \sum_{t \in Task(S_i)} H'_{S_i}(c, t, x, a(t), b(t)) \tag{3.43}$$

For those rounds $(\tau_i, C_i, T_i, \rho_i)$ and tasks $t \in T_i$ where $\rho_i(c,t) = \perp$ we have $H'_{S_i}(c, t, x, a(t), b(t)) = 0$, hence we can ignore these terms and only sum over the tasks that c attempted. We know that there is at least one such task, and that for each such task $t \in T_i$ there is a d such that $\rho_i(c,t) = (true, d)$.

This means that the non-constant part of $f(x)$ is a sum of a finite (but non-zero) number of functions of the form $\ln Pr(x, a_t, b_t)$. But as all $a_t > 0$, each of these functions is increasing in x. Hence $f(x)$ is increasing in x. Therefore the value $\theta(c)$ is not the global maximum of f, thus θ, a, b can not be a global maximum of L'_S. □

The cases where the extreme is a contestant that has failed to solve a task, or a task for which everybody had the same pass/fail status are proved in an analogous way.

Note 3.6.2. *The condition $\forall t;\ a(t) > 0$ in Theorem 3.6.1 is unnecessarily strict. One could easily prove a weakened version of this statement. However, some restrictions on the property a are necessary for the statement to hold. As a trivial counterexample consider two tasks with opposite values of a and b.*

As we chose to state the Theorem with the requirement $\forall t;\ a(t) > 0$, we feel the need to argue its reasonability. We remind you that the property a are the task discrimination parameters. A task with a negative discrimination parameter is a task for which a subject with a lower *ability score has a* higher *probability of solving.*

Such tasks do occassionally occur in practice, but in any reasonable scenario their percentage is negligible. In other words, if our scenario involves too many tasks with significantly negative discrimination parameters, we are doing something wrong. Most probably the skills necessary to solve the tasks in such a scenario are not related to the measured latent ability at all.

Anyway, Theorem 3.6.1 as stated covers a sufficient majority of expected real life scenarios.

We could be tempted to allow special values $\pm\infty$ for extreme cases like the ones described above. This would fix the issue with the non-existent global maximum. However, as we show in Theorem 3.6.3, this would introduce other pathological scenarions we wish to avoid.

Theorem 3.6.3. *Assume that we allow θ, a and b to return $\pm\infty$, and define*

$$H'_{S_i}(c, t, \theta^*_c, a^*_t, b^*_t) = \lim_{x \to \theta^*_c} \lim_{y \to a^*_t} \lim_{z \to b^*_t} H'_{S_i}(c, t, x, y, z) \tag{3.44}$$

in cases where some of θ^*_c, a^*_t, b^*_t are equal to $\pm\infty$, Then the following patological case occurs: There is a round sequence S and a maximum likelihood estimate of parameters θ, a, b such

that for some contestant c we have $\theta(c) = \infty$ even though there is a task c attempted and did not solve.
Formally, the contestant c satisfies the following condition:

$$\exists S_i \in S,\ S_i = (\tau_i, C_i, T_i, \rho_i) : \quad \exists t, d : \quad \rho_i(c, t) = (false, d) \tag{3.45}$$

Proof. Let $S = (S_1)$ be a round sequence consisting of a single round. Let $S_1 = (\tau, C, T, \rho)$, where $T = \{t_1, \ldots, t_n\}$ and $C = \{c_1, \ldots, c_{n+1}\}$. Define ρ as follows: pick an arbitrary d and set $\forall i, j :\ \rho(c_i, t_j) = (i > j, d)$. In words, each contestant attempted each task, and solved precisely those with a lower number that his own.

Consider any global maximum θ, a, b of the function L'_S. WLOG we can assume that $\sum a \geq 0$ – otherwise we can multiply all a, b and θ by -1 without changing the value of L'_S.

Now note that c_{n+1} is a contestant that has solved all tasks. The proof of Theorem 3.6.1 can easily be adopted to our current situation to show that $\theta(c_{n+1}) = \infty$.

Now t_n is a task that was only solved by a contestant with an infinite ability. Using the same technique we can show that in our maximum likelihood estimate we can have $a_n > 0$ and $b_n = \infty$.

And for the third time, now c_n is a contestant that solved all tasks except for one that requires infinite ability. The same argument as above can be used to show that we can find a global maximum of L'_S where $\theta(c_n) = \infty$. □

As before, similar pathological cases can be formulated for tasks and opposite ends of the ability/difficulty spectra.

Note 3.6.4. *For the situation presented in the proof of Theorem 3.6.3 there are multiple global maxima of the likelihood function. More precisely, for any integer x we can set $\theta(c_y) = -\infty$ for $y \leq x$ and $\theta(c_y) = \infty$ otherwise, set all $a(t_y) = 1$, set $b(t_y) = -\infty$ for $y < x$, $b(t_y) = \infty$ for $y > x$ and set $b(t_x)$ arbitrarily.*

In each of these cases, the global maximum fails to reflect the obvious linear order of the subjects.

From Theorems 3.6.1 and 3.6.3 it follows that if we want to base a rating algorithm on IRT and maximum likelihood estimates, we will have to restrict the contestant and task properties to bounded intervals.

The actual value of the bound will not matter, as it just corresponds to setting the scale of the properties.

We also noted another issue: We will need to avoid (significantly) negative values of the task discrimination parameter a. We can do this because the entire setting is symmetric: properties $-\theta$, $-a$, $-b$ have the same likelihood as θ, a, b. Be enforcing a to be (mostly) positive we are not biasing the system, we are merely deciding that the positive values should represent a higher level of ability/discrimination/difficulty.

Note 3.6.5. *For the situation presented in the proof of Theorem 3.6.3, if we take $N = 10$, bound the absolute values of all properties by 10, and further require that $\forall t : \ a(t) \geq -1$, then the maximum likelihood estimate of the properties looks as follows:*

$$a = (10, 10, 10, 10, 10, 10, 10, 10, 10, 10)$$
$$b = (-9, -7, -5, -3, -1, 1, 3, 5, 7, 9)$$
$$\theta = (-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10)$$

In words, all tasks discriminate as strongly as possible, and both their difficulties and the contestants' abilites are spread evenly along the allowed spectrum – which is precisely what we need.

3.7 IRT rating model

In this section we present our first rating algorithm that is based on Item Response Theory.

According to the observations proved in Theorems 3.6.1 and 3.6.3 we need to pick a finite range for the properties θ, a and b.

We are free to pick the bound, as this only affects the scale of the computed values. In all our practical tests we used the bound $\forall c, t : \ \theta(c), a(t), b(t) \in [-10, 10]$. We additionally enforce $\forall t : a(t) \geq -1$ in order to force the positive values of b and θ to represent a higher level of difficulty/ability. The rationale for this step is included in the proof of Theorem 3.6.3.

As in Section 3.4 the input to our algorithm will be a round sequence S. Our rating algorithm will output two contestant properties θ and SEM, and two task properties a and b – the 2PL model item parameters for each of the tasks. (Note that the standard error of measurement SEM is computed after θ, a and b are established, and therefore its values need not be bounded by design.)

We define that the output properties θ, a and b are those where the parameter log-likelihood function for S, as defined in Definition 3.6.2, has its global maximum. (If there are multiple global maxima, we are free to pick any of them.) The property SEM is then computed using Equation (2.17).

(Note that it would be possible to compute standard errors of measurement for the values a and b as well. We opted not to include this, as in the main setting of programming competitions we do not find this necessary. The presence of many contestants that have only solved a few tasks is a significant issue in this setting, whereas each task is attempted by many contestants, giving us usually enough information to estimate its parameters sufficiently. Still, in different settings, this information can be computed and used in predictions.)

Possible implementations of this model will be discussed in Section 4.1.

3.8 Some predictions for IRT ratings

We now want to note the superiority of our approach in comparison to Bayesian rating systems. We will now show that we can algorithmically compute predictions that could not be made by the Bayesian model.

In the Bayesian model the only basic prediction we were able to make was to compute the probability of one contestant outperforming another one – i.e., the model could only predict their relative performance. In our model we are able to use the additional data to also predict absolute performances, such as the expected number of tasks solved by each participant, and the probability of placing in the top K.

In this section we will show our algorithm that can efficiently compute such predictions in the case where the both the set of participants and the set of used tasks are known.

(If the set of tasks to be used is unknown, this prediction can be computed as an average over the previously used task sets, or even better by random sampling the pool of known tasks.)

Predicting solved task count

Assuming that we trust our estimated θ values perfectly, we can simply predict the number of solved tasks as the sum of the individual probabilities of solving each of them, i.e., $\sum_{t \in T} Pr(\theta(c), a(t), b(t))$. (Later, in Section 4.2.3, we will show how to take into account the fact that our estimates are not precise.)

Predicting probability to reach the top K

Let C be a set of contestants that will take part in the next round, with contestant c having ability θ_c. Let T be a set of tasks that will be used, and let the task parameters a_t, b_t of each $t \in T$ be known as well.

For each $c \in C$ we want to predict a probability p_c of the event that in the next round c will finish in one of the top K spots.

To do this, we will use binary search to compute the expected number of tasks solved that will correspond to the threshold. Inside the search, we will use dynamic programming to compute the required probabilities.

More precisely, given a potential threshold t, we have to answer the question: what is the expected number of people that will solve at least t tasks? If this number is higher than K, the real expected threshold is higher than t, and vice versa. The expected number of people that will solve at least t tasks can be computed as the sum of the individual probabilities. The probability that a given contestant solves at least t tasks can be computed by dynamic programming using the recurrence relation described below.

Let the tasks be numbered 1 to $M = |T|$, and let θ be the ability of our contestant c. By $P_{x,y}$ we will denote the probability that c solves exactly y out of the tasks 1 to x. We have:

- $P_{0,0} = 1$
- $P_{0,y} = 0$ for $y > 0$
- $P_{x,0} = P_{x-1,0}(1 - Pr(\theta, a_x, b_x))$ for $x > 0$
- $P_{x,y} = P_{x-1,y-1}Pr(\theta, a_x, b_x) + P_{x-1,y}(1 - Pr(\theta, a_x, b_x))$ for $x, y > 0$.

Explanation for the last recurrence relation: When computing the probability $P_{x,y}$, consider two cases. With probability $Pr(\theta, a_x, b_x)$ contestant c solves the task x. Then he needs to solve exactly $y-1$ other tasks, which happens with probability $P_{x-1,y-1}$. And with probability $1 - Pr(\theta, a_x, b_x)$ contestant c will not solve the task x. Then he needs to solve exactly y other tasks, which happens with probability $P_{x-1,y}$.

Using the above recurrence, computing all values $P_{x,y}$ can be done in $O(M^2)$. In the simplest setting with binary evaluated tasks, we will need $O(\log M)$ iterations of the binary search, hence the total time complexity of this algorithm will be $O(M^2 \log M)$.

Chapter 4

Advanced topics on rating systems

In the previous Chapter, we gave a detailed description of the first IRT-based rating system, and defined a way how to compare rating systems.

In the first part of this Chapter (Section 4.2) we will discuss various ways how the initial rating system can be further improved. We also list several open questions for further research.

In Section 4.3 we consider the rating systems from a security point of view. We show that the current state of the art is seriously negligent to this point of view. We provide a list of various methods how the existing rating systems can be compromised by a malevolent individual or a group of individuals.

We present two concrete attacks in Sections 4.4 and 4.5.

Finally, in Section 4.6 we address an important concept related to one class of possible attacks – the fact that an attack-resistent rating system should never award a purposefully bad performance. We postpone the presentation of one more attack (related to this Section) until Section 5.6.

4.1 Implementation of the basic IRT ratings

In this section we briefly describe the numeric approach used in our proof-of-concept implementation (shown in Appendix A) of the rating algorithm.

We had to take into account that we are trying to optimize a huge function. For real life data we have, the objective function can easily have more than ten thousand parameters. Naive methods such as gradient descent and Newton's method (see e.g. [Sny05]) fail miserably in such setting.

Instead of directly adapting a known optimization method, we were able to make the following observation about our objective function (as defined in Definition 3.6.2): Suppose that we fix all task-related parameters as constants. Then the function can be broken down to a sum of simpler functions, where each simpler function value only depends on one contestant's ability. Hence if we knew the exact task-related parameters, we could optimize the ability

estimates independently.

The same observation can be made in the opposite direction as well: If we knew the exact abilities, estimating the task-related parameters can be broken down into a set of two-dimensional optimization problems.

While we do not know either set of parameters, these observations are still useful, as they allowed us to implement a bootstrapping optimization algorithm in which we alternately compute the best ability estimates given the current task properties and vice versa.

With a good implementation, the time complexity of a single bootstrapping round is linear in the input size (which can be bounded from above as the number of contestants times the number of tasks).

4.2 Possible improvements

In this Section we discuss several possible ways in which our basic rating system can be improved. We introduce the concept of task weights and show that it can be used to define a simple way how to make our rating system time-sensitive. We discuss the newcomer problem, and show that in our model we have a systematic and scientifically founded solution to this problem. Finally, we briefly mention the idea of incremental calibration as a way to speed up the computation of ratings.

4.2.1 Task weights

In this section we will introduce the concept of task weights – i.e., different tasks having a different impact on the ability estimates. One possible motivation for this is discussed in the next section.

Note that we *should not* formally define the task weight as a task property. This is due to the fact that we might (and actually will) want to assign different weights to the same task when used in different rounds. Therefore we define the task weight as follows:

Definition 4.2.1. *Task weight function.*
Task weight function is a function $wt : \mathscr{R} \times \mathscr{T} \to R_0^+$ *with the property that* $wt(R,t) > 0$ *iff* $t \in Task(R)$ *and* $wt(R,t) = 0$ *otherwise.*

What we want to achieve is for a task weight w to have an equal impact on the ratings as w tasks of weight 1 each. Formally, this can be achieved by changing the parameter log-likelihood function we are maximizing to:

$$L'_S : CP \times TP \times TP \to \mathbb{R} \tag{4.1}$$

$$L'_S(\theta, a, b) = \sum_{S_i \in S} \sum_{c \in Cont(S_i)} \sum_{t \in Task(S_i)} wt(S_i, t) \cdot H'_{S_i}(c, t, \theta(c), a(t), b(t)) \tag{4.2}$$

4.2.2 Abilities that change in time

So far our models were based on the assumption that we are trying to estimate abilities that remain constant in time. As this does not have to be the case in each setting, in this section we discuss a possible addition to our model that will make it sensitive to ability change over time.

Now the natural way how to introduce the concept of time is to give the highest weight to the results that occurred closest in the past. (Note that our definition of a round already included a timestamp τ that will now be used for this purpose.)

Our proposal here is to set the task weight as an exponentially decreasing function of time elapsed since the particular round. Formally:

Definition 4.2.2. *Time-sensitive task weight function.*
We say that the task weight function wt is time-sensitive if there is a constant $c \in (0, 1]$ *and a constant* $\tau_{now} > 0$ *such that*

$$\forall R \in \mathscr{R}, R = (\tau, C, T, \rho) : \quad \forall t \in T : \quad wt(R, t) = c^{\tau_{now} - \tau} \tag{4.3}$$

This proposal has not been researched yet, and we see this as one possible topic for further research in this area.

4.2.3 Standard errors and the newcomer problem

Many existing rating systems have trouble with rating newcomers. One common "solution" is to assign some provisional ratings to newcomers (this is done, e.g., in the TopCoder rating system [Top08]). However, these provisional ratings can easily influence the accuracy of predictions in a negative sense.

In this section we show that our rating system can be naturally extended to handle the newcomer problem without any need for special treatment.

For a fixed set of tasks T, the predicted solved task count for a contestant c would simply be $\sum_{t \in T} Pr(\theta(c), a(t), b(t))$ – if we did trust our estimated θ perfectly, which we don't.

But we can be more precise here. Based on the computed ability estimate and the set of tasks the contestant attempted, our rating algorithm can also compute the standard error of measurement for this ability estimate – in other words, we can quantify our trust in each of the computed estimates.

We will now show how to take this into account to make more accurate predictions.

Definition 4.2.3. *Predicted solved task count when taking SEMs into account.*
Let c *be a contestant with parameters* $\theta(c)$ *and* $SEM(c)$ *computed by our rating algorithm, and let* T *be a set of tasks in a round, where task* $t \in T$ *has 2PL parameters* $a(t)$ *and* $b(t)$.

Let φ_{μ,σ^2} be the probability density function of the normal distribution $N(\mu, \sigma^2)$. Then the predicted number of tasks c will solve is:

$$\begin{aligned}
PNT(c,T) &= \sum_{t\in T} Prob(c \text{ solves } t) \\
&= \sum_{t\in T} \int_{-\infty}^{\infty} \varphi_{\theta(c),SEM(c)^2}(x) \cdot Pr(x, a(t), b(t))\, dx \\
&= \sum_{t\in T} \int_{-\infty}^{\infty} \frac{1}{SEM(c)\sqrt{2\pi}} \cdot \exp\left(-\frac{(x-\theta(c))^2}{2SEM(c)^2}\right) \cdot \frac{1}{1+e^{-a(x-b)}}\, dx \\
&= \frac{1}{SEM(c)\sqrt{2\pi}} \cdot \sum_{t\in T} \int_{-\infty}^{\infty} \exp\left(-\frac{(x-\theta(c))^2}{2SEM(c)^2}\right) \cdot \frac{1}{1+e^{-a(x-b)}}\, dx \\
&\simeq \frac{1}{SEM(c)\sqrt{2\pi}} \cdot \sum_{t\in T} \int_{lo}^{hi} \exp\left(-\frac{(x-\theta(c))^2}{2SEM(c)^2}\right) \cdot \frac{1}{1+e^{-a(x-b)}}\, dx
\end{aligned}$$

for $(hi, lo) = \theta(c) \pm 4SEM(c)$.

In our definition we consider a random variable with the distribution $N(\theta(c), SEM(c)^2)$ (a normal distribution with mean $\theta(c)$ and standard deviation $SEM(c)$) and define the predicted solved task count as the expected solved task count for a contestant c whose ability is given by this random variable. Note that this is in accord with the statistical interpretation of $SEM(c)$.

The effect is that for contestants with high value of SEM (which are all contestants with a low number of tasks attempted, and some contestants with highly unpredictable performance) the interval where $\varphi_{\theta(c),SEM(c)^2}$ is significantly non-zero will be larger.

Based on this approach, we will later (in Section 5.5) be able to algorithmically predict the expected score in a settings that does provide one.

4.3 Attacks on rating systems

The existing rating systems were designed using a "behind the glass" paradigm – it is silently assumed that the subjects are unaware of the ratings. This is far from being true in practice. First of all, there are psychological aspects to knowing one's rating (and even to knowing that one is being rated). Topics like performance anxiety are well researched, and we will not address this topic here.

However, we will focus on a much more important aspect – one that becomes immediately obvious to a person with a computer security mindset. The rating system is a collection of formulas and algorithms, and as such, it can be viewed as an IT system. And the security of this system plays a role as important as in cryptography. Whereas in cryptography the main goal of the IT system is to preserve the integrity, authenticity and confidentiality of the stored data, in a rating system what we need to preserve is the validity of the ratings. And while

exploiting this analogy, note that there is a direct correspondence between finding a flaw in an encryption algorithm and finding one in the rating system.

More precisely, an ideal rating system has to anticipate that malevolent individuals may try to exploit its design flaws to improve their ratings beyond their actual skill levels. In practice, our experience is that the security aspect is often completely overlooked by the rating system's authors – while often happily exploited by users who are able to discover a flaw.

4.3.1 Rating conservation

The most basic method of influencing the ratings is actually mostly passive. Ot is known as rating conservation, or rating protection.

In order to protect the current rating, the player can refuse to take part in matches, select opponents more carefully (e.g., only play against weak players), etc. There is ample anecdotal evidence that this is happening in chess, see [Hei02] and [Thi06] for some examples. In particular, in [Hei02] we can find the following observation: "Ratings can, and do, enormously discourage participation among many potential players."

Also, see [Nun05] where the author proposes a system for the Chess World Championship, including a proposal to qualify players based on their Elo ratings. The author realizes these problems and tries to address them – albeit in a non-systematic ad-hoc way, by introducing an "activity bonus".

4.3.2 Opponent selection and player collaboration

In settings where one his able to choose his opponents, the rating system has to anticipate this and it must not be possible to influence the ratings by choosing one's opponents.

In particular, in such settings there is an attack type that is particularly difficult to defend against – a group of players that decides to collaborate.

(Of course, this can only be classified as an attack if the group, while significantly smaller than the entire population, can still influence the ratings in a significant way.)

The simplest way how a group can collaborate is when they try to increase the rating of one of the group's members at the expense of the others' ratings.

We were able to make such an attack on the eGenesis system, and we present it in Section 4.5.

Settings in which one is able to select their opponent may also have disadvantages that can not be described as attacks, but are still harmful.

For example, the ad-hoc rating system used by the online mathematical competition AoPS For The Win! (see [oPSI08]) has the property that once the rating difference between two players exceeds a certain threshold, we get to a situation where if the higher rated player

wins, their rating increase is rounded down to zero. In such situation there is no point for the higher rated player to agree to a game – he has nothing to gain and much to lose.

4.3.3 Worse performance on purpose

This attack is closely related to a concept that we treat in more detail in the next Section 4.6 – the concept of monotonicity of a rating system. It is desirable for a good rating system to be monotonous, in the sense that if a contestant's performance on some item were worse, their rating has to be be lower.

In an incorrectly designed rating system it may be beneficial to include a deliberately bad performance to "throw off" the estimate. For example, in an incremental Bayesian rating systems an unexpectedly bad performance will significantly raise the estimated volatility of the subject, thereby causing the next performances to have a larger impact on the subject's rating.

In Section 4.6 we give a formal definition of monotonicity, in Section 4.6.2 we show that our basic rating algorithm has this property (and hence is resistant to this class of attacks), and in Section 4.6.3 we show this type of attack against the TopCoder rating algorithm.

4.4 Attacking the Elo rating system

In this section we will show that a small, constant-size group of cooperating players can significantly affect their Elo ratings – in particular, boost the rating of one of them.

We assume that the Elo rating system uses the rating update function described in Section 2.2.2, with an added constraint that rating can not be negative. Additionally, in our calculations we used the following simple rule to determine the K factor:

- Players below 2100: K factor of 32 used
- Players between 2100 and 2400: K factor of 24 used
- Players above 2400: K factor of 16 used

Note that this table of K factors is popular with online chess servers.

We assumed that we have a sufficient supply of players[1] with initial rating 1000.

Our goal was to achieve a rating over 2400 with at least one of our players[2], using a reasonably small number of games and a small number of conspiring players.

The general idea is that if we want to increase a player's rating, we should let him win against strong opponents – because then his expected score is significantly less than 1, and

[1] This is a reasonable assumption. In some settings this is simply the initial rating given to new players. Alternately, a player with real rating over 1000 can "create" as many fictional players with roughly his own rating as needed.

[2] In chess, most players with rating over 2400 hold at least the International Master title.

hence the rating change is large. Based on this observation, we devised the following heuristic algorithm:

Start with N players with rating 1000 each. Label the players 1 to N. In each match, the lower numbered player will win. Obviously, the order of matches matters. In each round, out of the possible $\binom{N}{2}$ pairs of players, we pick the pair $i < j$ for which $rating(i) - rating(j)$ is minimized.

Using this greedy approach, we found that for the optimal value $N = 7$ the first player will reach a rating over 2400 after 3054 games have been played in total. At this point, the ratings of the other six players will be close to 2000, 1600, 1200, 800, 400, and 0, respectively. (See Code listing 4 in Appendix A for our implementation of the greedy approach.)

Evaluation after multiple matches

Maybe surprisingly, our situation *improves* significantly in the setting when the ratings are only updated after multiple matches have been played – e.g., after an entire tournament, or at the end of a day. The point is that in this case the current ratings are used to compute the expected score over a series of matches. And this expectation will be way less precise than the sum of expectations in the case where ratings change after each match.

Assume that we are able to schedule our games in such a way that the ratings are updated precisely at those points in time when we want them.

In this setting, it is sufficient to do the following:

1. Player 2 wins 45 times against Player 3.
2. Ratings are recomputed, Player 2 rose significantly.
3. Player 1 wins 45 times against Player 2.
4. Ratings are recomputed. Player 1 achieves rating approx. 2417.

4.5 Attacking the eGenesis rating system

In this section we show how, in spite of the protective measures, one can successfully attack the eGenesis rating system (as described in Section 2.2.5 and the official source [TY02]).

Let P be the player whose rating we plan on maximizing. In order to do so, we will introduce $k + 1$ fake newcomers $N_1, \ldots, N_{k+1}$.

In the first phase, we can let the player P and newcomers N_1 to N_k repeatedly win against N_{k+1} in order to transfer their reserve bits into their vectors. After $128(k + 1)$ games P and each of the N_i have what we may consider to be random vectors with 128 zeroes and 128 ones.[3]

[3]Technically, we made a small omission here: It is possible that as N_i plays N_{k+1}, their common 32 bits

In the second phase, we let player P win once against each of the newcomers N_1 to N_k in order to transfer their 1 bits into his vector.

Player P has 32 bits in common with each of N_1 to N_k. In other words, for each of the 256 bits in P's vector, the probability that he has it in common with a fixed N_i is $32/256 = 1/8$. Hence for each bit the expected count of is for which P and N_i have the bit in common is $k/8$.

We are interested in the 128 bits in P's vector that are zero. Pick one of these bits. The situation we want to avoid is when for all N_i that share this bit with P the bit is zero in their vectors as well. This happens with probability $(1/2)^{k/8}$.

Hence for each missing bit the probability that P will be able to gain it from one of the N_i is $1 - (1/2)^{k/8}$. Therefore the expected rating of P after his k matches will be $128 + 128 \cdot (1 - (1/2)^{k/8}) = 256 - 128 \cdot (1/2)^{k/8}$.

To summarize, we need to play $129k + 128$ matches in order to create a player with expected rating $256 - 128 \cdot (1/2)^{k/8}$.

Note that already for $k = 10$ the expected rating is over 200, and that without the interaction with any other players we can get a player with expected rating arbitrarily close to the maximum.

This attack is resistant against multiple possible modifications of the eGenesis rating system, including:

1. *The modification where only random 8 out of the 32 common bits are transferred to the winner.*

   After 5 games P vs. N_i we can expect 31 of those 32 bits to be transferred anyway. Hence after $133k + 128$ matches we have expected rating $256 - 128 \cdot (1/2)^{31k/256}$ (which is still over 200 for $k = 10$).

2. *The modification where the transferred bits from the reserve are limited somehow when playing the same opponent.*

   We just replace N_{k+1} by a larger set of fake opponents, increasing the number of fake players but not changing the number of matches needed.

4.6 Monotonicity

In this section we address one significant property a rating system can, and in our opinion also should, have – monotonicity. The rationale behind defining this property is the notion that a worse performance should never pay off. In other words, we can describe this property as follows: if you had performed worse, your rating would now be lower.

---

will be all set to 1 before N_i manages to transfer all reserve bits into his vector. This can be handled in the same way that is used in item 2 at the end of this section.

4.6.1 Definitions

Definition 4.6.1. *Linear order on individual results.*
Let $<$ be a binary relation on the set $\{true, false\} \times \mathbb{R}^k$ for some fixed k. Then $<$ is called a linear order on individual results, iff all of the following holds: The relation $<$ is a total asymmetric transitive binary relation, and for all $x, y \in \mathbb{R}^k$ we have $(false, x) < (true, y)$.

Definition 4.6.2. *Monotonicity of a rating algorithm.*
Let $\mathcal{A}$ be a rating algorithm, and let $<$ be a linear order on individual results. This algorithm is called monotonous with respect to $<$, if it has the following property:

Let S_1, S_2 be two round sequences that only differ in one round. Moreover, let $R_1 = (\tau, C, T, \rho_1)$ be the round in S_1. Then the corresponding round in S_2 is $R_2 = (\tau, C, T, \rho_2)$, and the function ρ_1 and ρ_2 are identical, except for a single value: there is a contestant $c \in C$ and task $t \in T$ such that $\rho_1(c, t) > \rho_2(c, t)$.

Let θ_1 and θ_2 be the contestant properties output by $\mathcal{A}$ for S_1 and S_2 respectively. Then we require that $\theta_1(c) > \theta_2(c)$.

Note that there are other, less strict definitions possible, for example we may allow rating to increase and only require that the worse performance may not improve the contestant's ranking.

4.6.2 Monotonicity of our IRT-based rating system

Obviously, it is not possible to prove a general claim about the monotonicity of our rating system. The problem is that there are way too many corner cases when, in fact, our rating system is far from being monotonous.

For example, if we had a bad task t with a negative discrimination parameter a_t, then *not* solving this task actually *should* give us a better ability estimate for the given contestant.

However, under normal circumstances we can expect our rating system to be monotonous. The intuition behind this claim is that a single change (from a contestant solving a task to the contestant not solving it) will have the largest impact on the ability of the given contestant (which should decrease due to the fact that he now solved less tasks) and on the parameters of the given task (which should become slightly harder due to less people solving it).

Formulating and proving the exact conditions for monotonicity of our rating system is a complex task that is beyond the scope of this Thesis. Still, in this Section we formulate and prove a theorem that supports our claims.

Theorem 4.6.1. *Let S be a sequence of rounds, let $R = (\tau, C, T\rho)$ be a round, let $c \in C$ be a contestant, and let $t \in T$ be a task such that $\exists d : \rho(c, t) = (true, d)$. Let $\overline{S}$ be the sequence of rounds obtained from S by changing $\rho(c, t)$ into $(false, d)$.*

Let L'_S be the parameter log-likelihood function for S, as defined in 3.6.2. Let $L'_{\overline{S}}$ be the parameter log-likelihood function for $\overline{S}$.

Let θ, a and b be the task/item properties for which L'_S obtains its unique global maximum. Assume that $a(t) > 0$.

Then there is an $\varepsilon > 0$ with the following property: let $\overline{\theta}$, $\overline{a}$ and $\overline{b}$ be task/item properties for which $\forall c' \neq c : \ |\theta(c') - \overline{\theta}(c')| < \varepsilon$, $\forall t' \neq t : \ |a(t') - \overline{a}(t')| < \varepsilon$, and $\forall t' \neq t : \ |b(t') - \overline{b}(t')| < \varepsilon$. Let $\overline{\theta}(c)$, $\overline{a}(t)$, and $\overline{b}(t)$ be chosen in a way that maximizes $L'_{\overline{S}}$. Then $\overline{b}(t) > b(t)$ and $\overline{\theta}(c) < \theta(c)$.

Proof. The difference $D = L'_{\overline{S}} - L'_S$ is simply the function

$$D(\underline{\theta}, \underline{a}, \underline{b}) = \ln\left(1 - Pr(\underline{\theta}(c), \underline{a}(t), \underline{b}(t))\right) - \ln Pr(\underline{\theta}(c), \underline{a}(t), \underline{b}(t))$$

Let $f(x)$ be a single-variable function defined as follows: $f(c) = x$, and $\forall c' \neq c : \ f(c') = \theta(c)$. Let $g(x) = D(f(x), a, b)$ be a single-variable function in which we restrict all parameters except for $\theta(c)$ to the values of the original global maximum of L'_S.

From the fact that (θ, a, b) is a global maximum of L'_S we have that

$$\frac{\partial L'_S}{\partial \theta(c)} = 0$$

and therefore

$$\frac{\partial L'_{\overline{S}}}{\partial \theta(c)} = \frac{\partial (L'_S + D)}{\partial \theta(c)} = \frac{\partial D}{\partial \theta(c)} = g'$$

We will now compute g'. Let $a_t = a(t)$ and $b_t = b(t)$.

$$\begin{aligned} g'(x) &= \left(\ln(1 - Pr(x, a_t, b_t)) - \ln Pr(x, a_t, b_t)\right)' \\ &= \frac{-a_t}{1 + e^{-a_t(x - b_t)}} - \frac{a_t e^{-a_t(x-b_t)}}{1 + e^{-a_t(x-b_t)}} \\ &= -a_t \end{aligned}$$

From our assumption $a_t > 0$ it follows that $g'(x)$ is negative for all x.

In a similar way we can define a function $h(x)$ that would correspond to the partial function of D for the variable $b(t)$ and show that $h'(x)$ is positive for all x.

The rest of the proof follows from the continuity of all functions in our proof. □

In words, the theorem we just proved states that in some reasonable cases the single change in results we introduced will cause the difficulty of task t to increase, and the ability estimate of contestant c to decrease.

4.6.3 Attacking a Bayesian rating system

While it is obvious that monotonicity of a rating system is a desirable property, this property is far from being universal. For Bayesian rating systems this is often not the case. We were able to show that the TopCoder rating system is not monotonous. We were then able to use

this observation to attack the rating system and force it to give us a higher rating then before by losing on purpose. As the details of this attack use real life data, we postpone them into Section 5.6.

Chapter 5

Evaluation of real life data

In this Chapter we show experimental evidence for the soundness of our models, and evaluate the performance of a proof-of-concept implementation of our rating system and prediction algorithms.

This is the summary of the main topics covered in this chapter:

- We compute ratings and task parameter estimates for tasks used in the Slovak Olympiad in Informatics. Based on the computed data, we argue how to change the difficulty of tasks in the regional round of the Olympiad.

- We show evidence that in programming contests the solving time has a log-normal distribution. I.e., if we consider a fixed task, only look at contestants that solved the task, and measure how long it took each of them, the logarithms of their solving times will have a normal distribution.

- In our IRT-based model, we show how to predict advancement probabilities for a tournament round, generalize our algorithm to a setting where the score for a task is a function of the solving time, and then we show how this algorithm can be applied to the TopCoder setting. This approach is then used to make predictions whose quality is comparable to the Bayesian model's predictions.

- Additionally, we show that in our model we can make predictions that were impossible to make in the Bayesian model. We actually compute a few such predictions and provide evidence that these predictions were accurate.

- We show an attack scenario against the TopCoder rating system. This attack is shown on real life data, where an intentionally worse performance in one event leads to a significantly higher rating after several rounds – even exceeding the original maximum rating.

5.1 Data sets used

We evaluated our approaches on two different data sets. We will now describe these data sets in detail.

Data set 1 contained the results of two Slovak programming competitions: the Slovak Olympiad in Informatics, and the Correspondence Seminar in programming.

We analysed the data for school years 2006/07 and 2007/08. In each of these years our data set consisted of 44 tasks (4 series with 10 tasks each used in the Correspondence Seminar in Programming, and 4 tasks used in the regional round of the Slovak Olympiad in Informatics). In year 2006/07 the data set contained 177 contestants, in year 2007/08 it contained 201. In both years there was a significant overlap between the contestants in both competitions (over 70%).

All data we used is publicly available at the contest websites [OI, KSP].

Data set 2 contained the results of the programming competitions organized by TopCoder.

We used the data from all rated events up to, and including, TCO 2008 Online Round 1. This gave us a data set containing of $1,228$ tasks and $18,375$ contestants. Of course, each of the contestants only attempted a subset of tasks. The total number of pairs (c, t) such that contestant c was given the task t is slightly over $500,000$.

All data we used is publicly available in TopCoder's data feeds [Top09].

5.2 Computing IRT-based ratings for TopCoder competitions

We used an implementation of our rating algorithm to compute ratings and SEMs for all TopCoder's competitors. The set of tasks used included all tasks from all rated events between 2006-05-09 and 2008-02-16, inclusive.

Out of 12 657 contestants that participated at least once in this interval, 2 660 did not solve any task they were given, and 37 solved all tasks they were given. For these contestants, the ratings computed by our rating system were equal to the boundary values, which were chosen to be ± 5 in this case.

The distribution of computed ratings for the remaining 9 960 contestants is given in Figure 5.1. The distribution of the task difficulties (their 2PL b parameters) is given in Figure 5.2. The distributions are in accord with our experience – note that the median task is too hard for the median of the community.

In Figure 5.3 we give plots of computed logistic item response functions for several tasks of different types.

In most TopCoder competitions, the contestants are broken down into two divisions based on TopCoder's estimate of their skill, and each division is given its own set of three tasks. In

Figure 5.4 we show all item response functions grouped by this task types, and in Figure 5.5 we show the total test information function for each of these six sets of tasks.

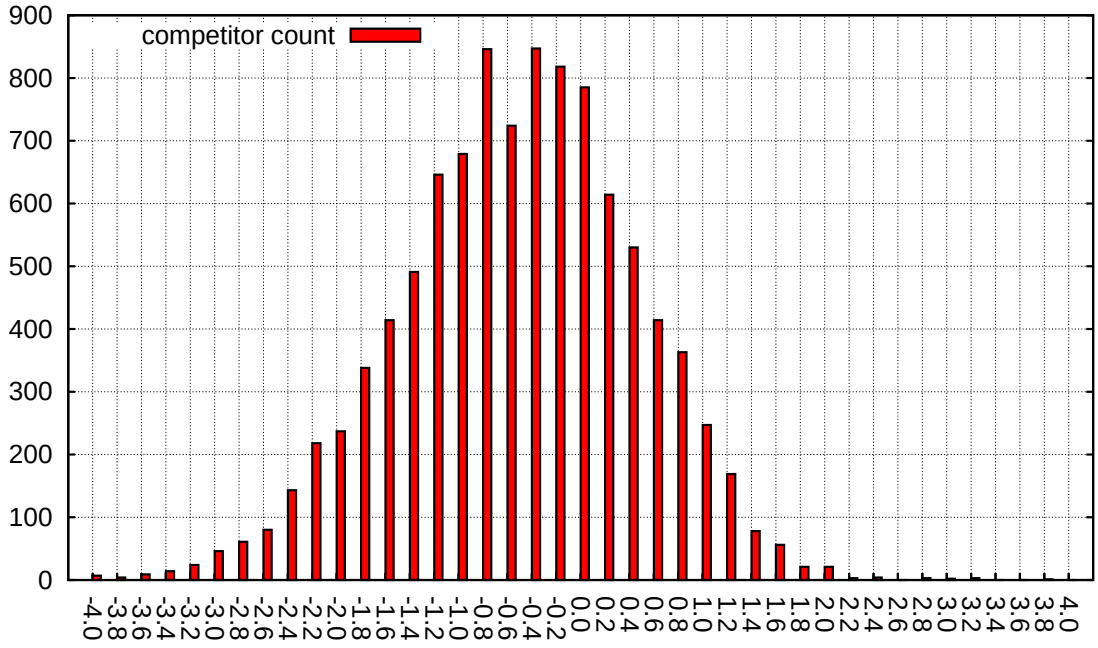


Figure 5.1: Distribution of ratings of TopCoder participants.

5.3 Analyzing task/problem set difficulty

One of the advantages of the IRT-based models over Bayesian ones is the ability to argue about task difficulty. This is precisely what we do in this section.

More precisely, in this section we present our analysis of the reliability of results of several rounds in the Slovak Olympiad in Informatics. We modify the 2PL model presented in Section 2.3.3 to better fit the particular nature of partial scoring used in this competition, apply the maximum likelihood estimate method we developed in Sections 3.6 and 3.7 to get the maximum likelihood estimates for the abilities of the contestants and parameters of the tasks, and then we analyse the test information function to show that the overall task difficulty is too high for the corresponding round.

Overview

We start with a brief introduction. Most tasks in Slovak Olympiad in Informatics (including all tasks in the regional round) are theoretical – the students solve them on paper, and the solutions are then graded by teachers. For each task the contestant is awarded 0 to 10 points, based mostly on the correctness and efficiency of the solution she found.

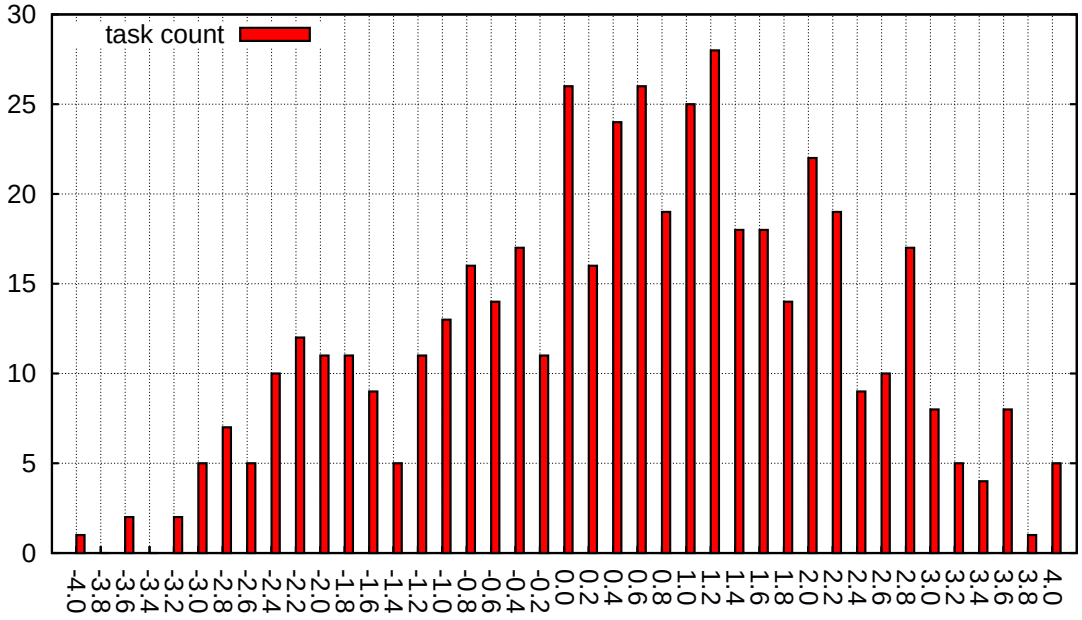
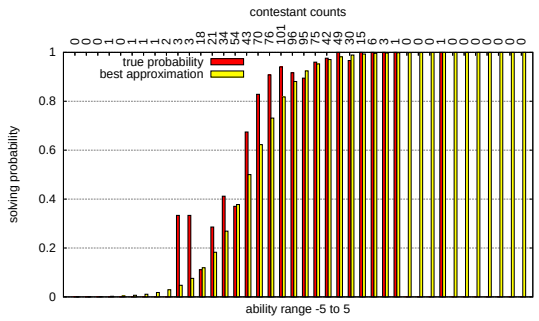
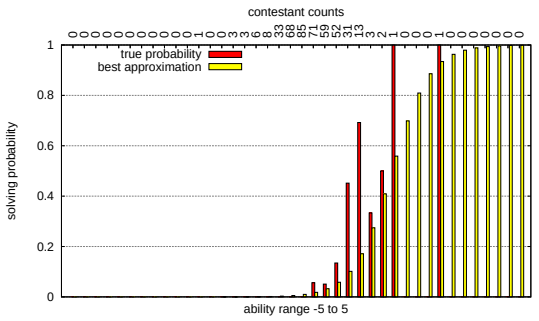


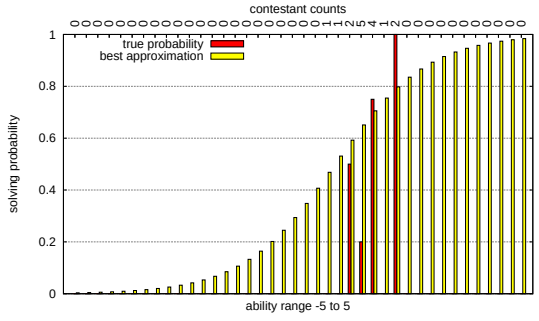
Figure 5.2: Distribution of task difficulties at TopCoder.



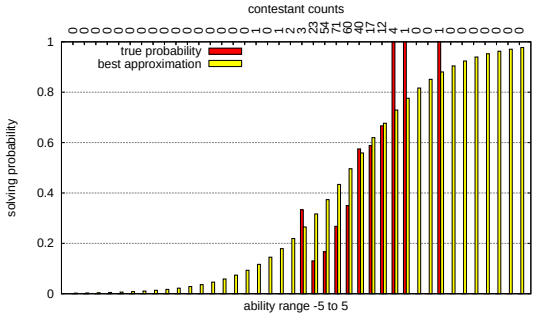
(a) Library, $a = 2.00052$, $b = -1.62554$



(b) StrangeArray, $a = 2.41793$, $b = 1.52704$

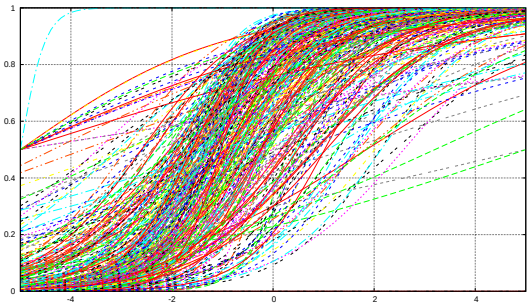


(c) BusyTime, $a = 1.00038$, $b = 0.251268$

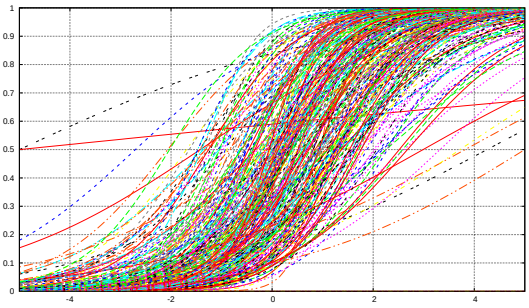


(d) Candles, $a = 1.00538$, $b = 0.63934$

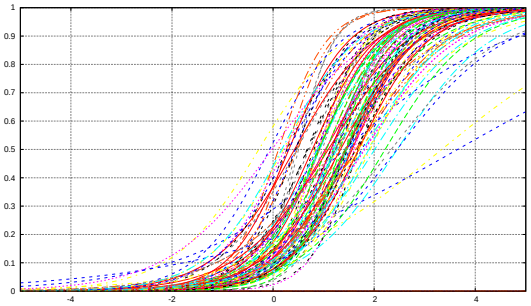
Figure 5.3: Plots of computed 2PL parameters for tasks with various difficulties and numbers of correct solutions.



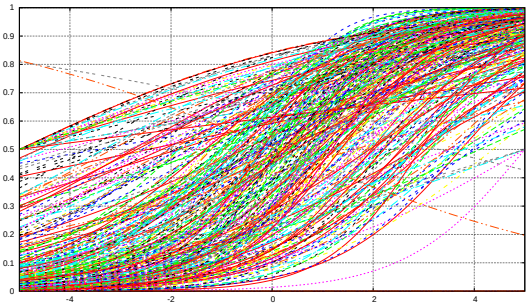
(a) Division 2, level 1 (easiest)



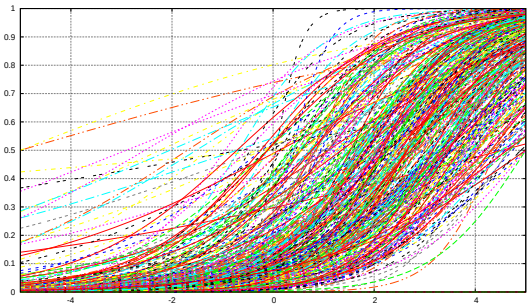
(b) Division 2, level 2



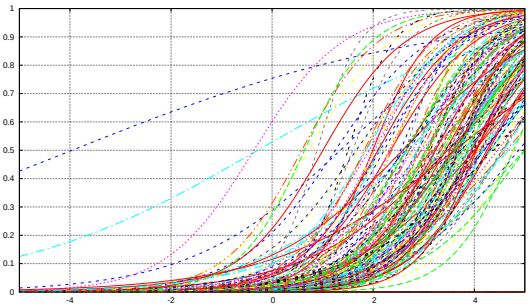
(c) Division 2, level 3



(d) Division 1, level 1

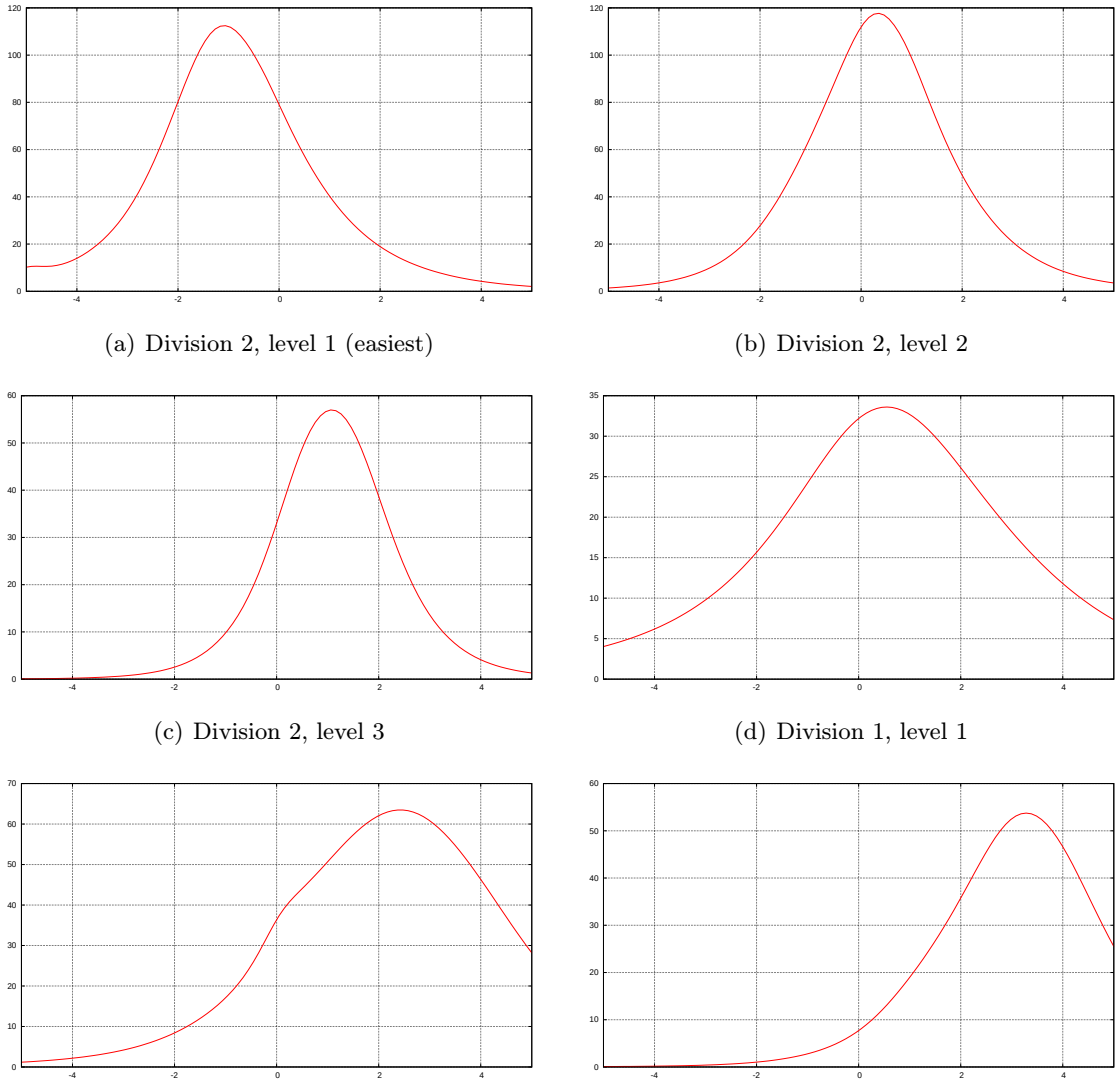


(e) Division 1, level 2



(f) Division 1, level 3 (hardest)

Figure 5.4: Logistic curves for all tasks in each of the six task groups.



(a) Division 2, level 1 (easiest)

(b) Division 2, level 2

(c) Division 2, level 3

(d) Division 1, level 1

(e) Division 1, level 2

(f) Division 1, level 3 (hardest)

Figure 5.5: Test information functions for each of the six task groups.

The general guidelines used in assigning the partial scores are:

- Any correct solution scores at least 4 points.
- Any efficient solution scores at least 7 points.
  (More precisely: If the "brute force" approach requires exponencial time, any polynomial-time solution is considered efficient. Otherwise, any solution that uses a non-trivial observation to achieve a better complexity is considered efficient.)
- Any solution that is close to the optimal asymptotic time complexity scores at least 9 points.

Discovering the optimal solution usually requires making multiple observations about the problem, and each of these observations alone is usually reflected in the efficiency of the solution.

From the regional round, approximately 30 best contestants in the country advance to the final, national round.

Based on this background, we decided that it would be suitable enough to model such a theoretical task as three separate items, each having a 2PL characteristic curve and binary evaluation. These can loosely be seen as corresponding to the three increasingly difficult steps towards getting a perfect score: finding some solution, finding some efficient solution, and finding the optimal solution.

Obviously, this model does not model the actual task difficulty precisely. However, this is outweighed by its relative simplicity, and we will show that the goodness of fit of this model to real life data is sufficient to make conclusions based on it.

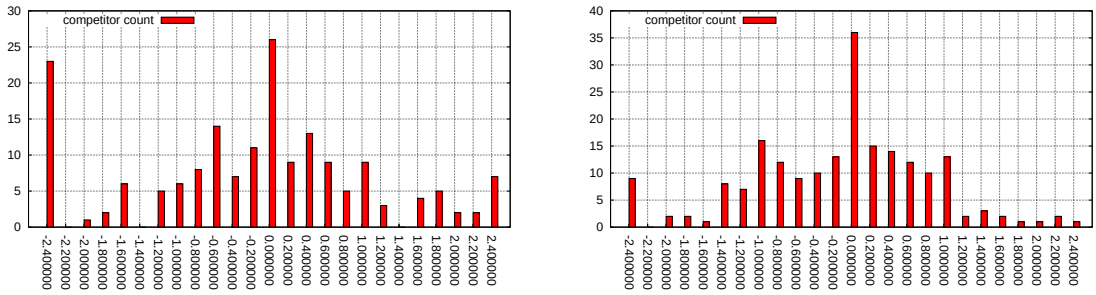
Datasets used for analysis

We used the results of these competitions to construct the response patterns as follows: For each task, we have three items. For each contestant, the first of these is set as solved iff she scored at least 3 points, the second iff she scored at least 6 points, and the third item is considered solved iff she scored at least 9 points.[1]

Based on these response patterns we found the maximum likelihood estimate of contestants' abilities and item parameters in our basic IRT-based model described in Section 3.7.

Figure 5.6 shows the distribution of ability estimates we got. Note that, as expected, the ability distribution is significantly different from a normal distribution. In particular, there is a (in 06/07 pretty significant) group of outliers at the lower end of the spectrum. This is a group of contestants that do take part in the competition (usually due to pressure or motivation from their schools), but find the tasks significantly over their skill level.

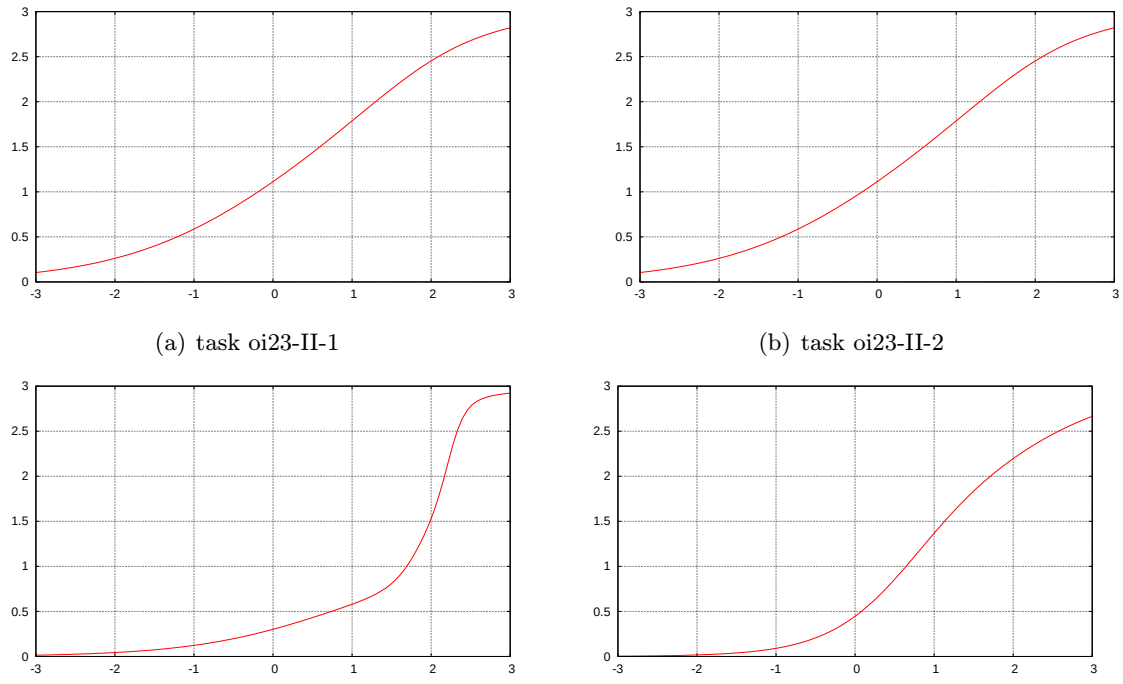
[1]The Correspondence Seminar in Programming uses a 15-point scale with similar bounds, hence we used the thresholds 4, 8, and 13 points for its results.



(a) θ distribution in 06/07

(b) θ distribution in 07/08

Figure 5.6: Maximum likelihood ability estimates for Olympiad in Informatics contestants.



(a) task oi23-II-1

(b) task oi23-II-2

(c) task oi23-II-3

(d) task oi23-II-4

Figure 5.7: Expected score based on ability for the four competition tasks in 07/08.

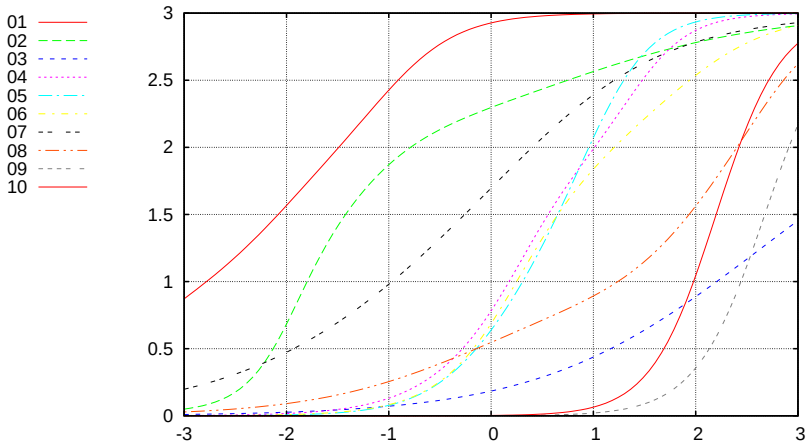


Figure 5.8: Expected scores for a series of the Correspondence Seminar

Figure 5.7 gives the graphs that show the characteristic curves (or, more precisely, expected scores mapped onto the interval $[0, 3]$) for the four tasks used in the regional round in 2007/08.

Some more details on the tasks: Tasks 1 and 2 were well-balanced tasks, both slightly hard (note that the expected score for $\theta = 0$ is below 1.5). Task 3 was a hard task, where it was really difficult to find any efficient solution. Hence the expected score exceeds 1 only at the far right end of the spectrum. The last task required the contestants to understand a theoretical model, and then to formulate the solution within this model. This task requires a significant overhead (understanding the model properly) before one can begin looking for any solutions. This overhead is nicely reflected by the fact that only around $\theta = 0$ the expected score for this task starts to be significantly positive.

For comparison, Figure 5.8 shows the expected scores for the ten tasks used in the first round of the Correspondence Seminar in Programming in the same year.

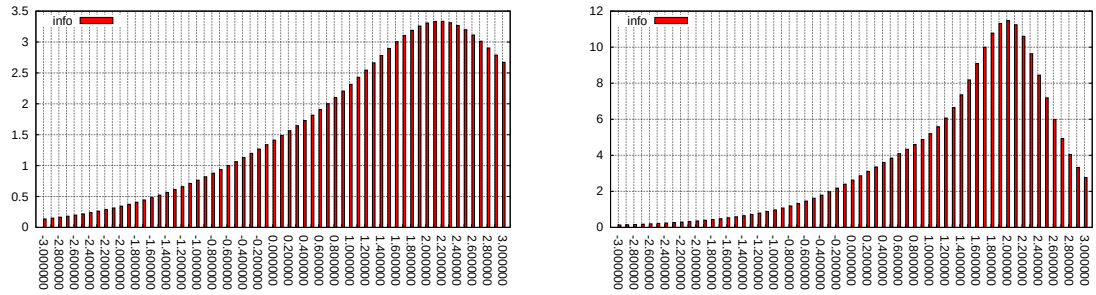
Analysing problem set difficulty

The goal of the regional round in the Olympiad is to select the participants for the national round as reliably as possible. We will now answer the question: Did we achieve this?

This question can be answered using the test information function as defined in Section 2.3.6. The test information function gives us the precision of rating estimates based on this test only. Ideally, we want the test to give us most information at and around the advancement threshold – which in our case is the thirtieth place.

Figure 5.9 gives the test information functions corresponding to the task parameters we computed.

We see that in both cases the maximum of the test information function lies in $[2, 2.2]$. If we compare this with the ability distribution plotted in Figure 5.6 we see that this interval



(a) test information fn in 06/07

(b) test information fn in 07/08

Figure 5.9: The test information functions for regionals of Olympiad in Informatics.

corresponds to the top few contestants – which is not what we wanted.

The advancement threshold in both years was around $\theta = 0.5$. We see that while the 2006/07 set was better balanced than the 2007/08 one, both have pretty low values around this threshold. Apparrently, both problem sets were on average way too hard for the main goal they are supposed to achieve. These sets would be more suitable for the national round, where the main goal is to identify the best.

Sanity check

For our results to have scientific validity we have to check whether our model was able to approximate reality sufficiently. To verify this, we did both a visual check of the data, and computed the Pearson product-moment correlation coefficients for the real and model data as follows:

For each task, we divided the contestants that solved it into groups based on the floor of their ability estimate θ. For each group i we now computed the actual total score for the task x_i, and the expected total score y_i based on their individual ability estimate. We then computed the weighted correlation coefficient of the vectors (x_i) and (y_i), where the weight of each element was the number of contestants c_i in the corresponding group.

The resulting correlation coefficients are tabulated in Table 5.1. We see that for 64 out of 88 tasks ($\sim 73\%$) the correlation coefficient exceeds 0.9, and for half of these even exceeds 0.98, which is excellent. The three most significant outliers occurred in 2007/08, and all three were difficult tasks that only 10 contestants attempted to solve, and almost nobody did.

Final notes

As we observed above, the discrimination rate of the problem sets used in the regional rounds were not sufficiently high around the threshold for advancement. As a direct consequence, the point differences around the threshold were low, and hence secondary factors came into play – the quality and clarity of presentation, minor mistakes, etc. We claim that out of the

ρ range	year 2006/07	year 2007/08
$[0.99, 1.00]$	9	12
$[0.98, 0.99)$	6	5
$[0.95, 0.98)$	9	9
$[0.90, 0.95)$	7	7
$[0.75, 0.90)$	12	8
$[0.00, 0.75)$	1	2
$[-1.00, 0.00)$	0	1

Table 5.1: Distribution of correlation coefficients between predicted and actual scores.

contestants whose abilities were close to the threshold those who advanced are more or less those who managed to write down their solutions in a better way. It is disputable whether this is a purely negative observation, but still a better problem set would make the distinction between their abilities more significant, leaving less influence to luck and these secondary factors.

Additionally, we would like to mention that there is a relation between the final scores and the motivation of contestants. A perfectly discriminating problem set can leave many contestants with scores close to zero, hence discouraging them from further competitions. On the other hand, including a very easy task can help the motivation, but at the same time increase the volatility of the final placements (e.g., if someone makes a mistake in the easy task, they may not be able to make up for this on the others). This is a complex topic that transgresses the boundaries of this thesis, but surely deserves further attention.

5.4 Analyzing the solving time

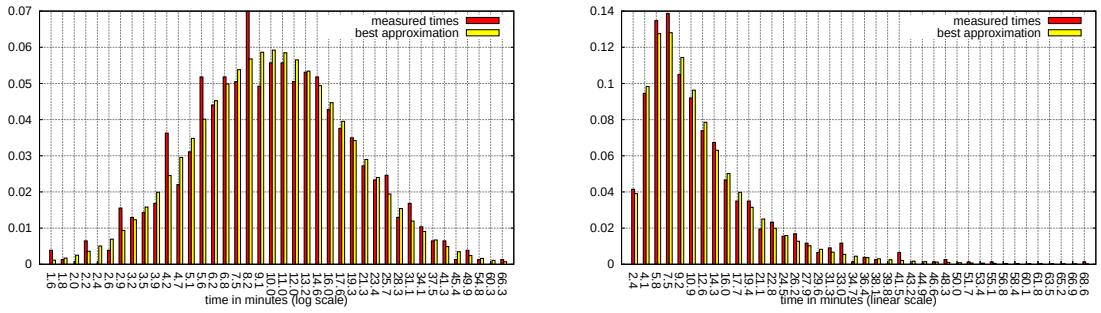
In this section we present experimental data that show that the random variable giving the solving time for a particular task usually has log-normal distribution, and investigate the corelation between our estimated ratings and solving time.

We used the Jarque-Bera normality test [BJ80, BJ81], in particular the Alglib implementation [BB], on natural logarithms of solving times for 560 tasks from past TopCoder events.

Out of these 560 tasks, for 71 the sample size was too small (at most 5 contestants succesfully solved the task).

Out of the remaining 489 tasks, for 221 the null hypotesis can be accepted at 99% confidency level, and for another 163 it can be accepted at 98% confidency level.

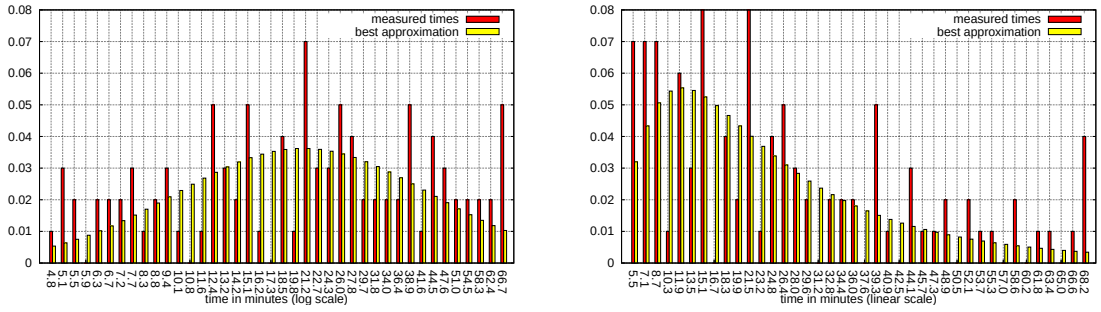
We verified a sample of the remaining 105 tasks by hand, and discovered that the main reasons for rejecting the lognormality hypotesis were mostly either small sample sizes, or too easy tasks where most of the solving times come from a narrow interval. Even for these cases



(a) log scale on x axis

(b) linear scale on x axis

Figure 5.10: Solving times for PalindromeDecoding match the lognormal distribution at 99% confidency level. Sample size $N = 772$.



(a) log scale on x axis

(b) linear scale on x axis

Figure 5.11: Solving times for Caketown match the lognormal distribution at 98% confidency level. Smaller sample size $N = 100$.

the most probable lognormal distribution provides a reasonable approximation.

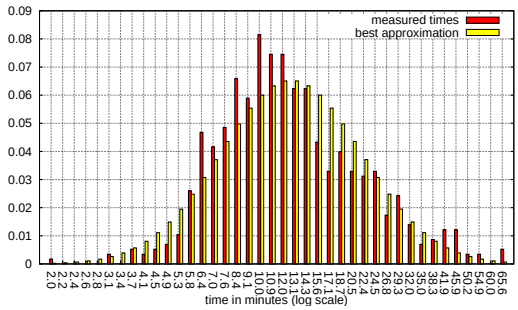
In Figures 5.10, 5.11, and 5.12 we give examples of true solving time distributions and their best lognormal approximations for three tasks coming from different sets in the above results.

5.5 Tournament advancement prediction

In this Section we describe our succesful attempt to use our rating model to predict advancement in a TopCoder tournament. We will first describe a general high-level prediction algorithm, then we describe how it was applied to the TopCoder setting, and finally we list the results our implementation achieved.

5.5.1 Predicting advancement probability in a IRT-based model

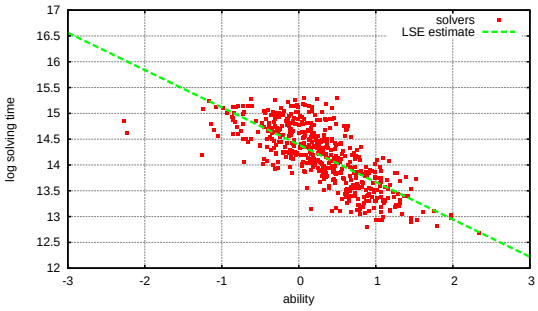
In the 2PLT model we are able to predict not only the probability that the given contestant will solve the given task, but also the time he will require in case he solves the task. We



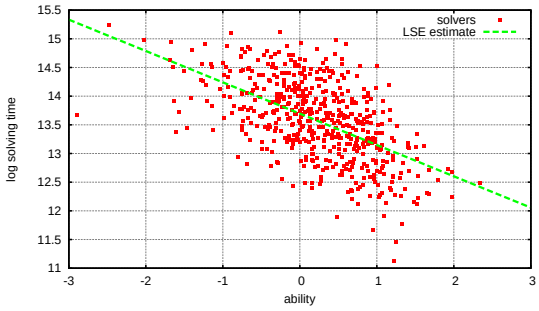
(a) log scale on x axis

(b) linear scale on x axis

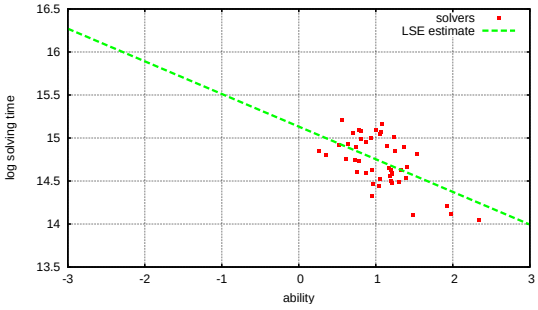
Figure 5.12: Solving times for RussianSpeedLimits do not match the lognormal distribution at 98% confidency level. Sample size $N = 577$. Lognormal distribution still offers a reasonable approximation.



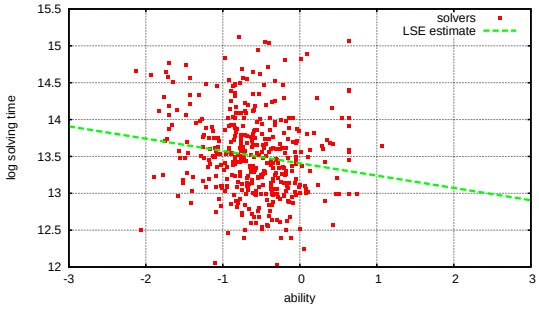
(a) CandidateKeys,$N = 514$, correlation -0.74159473



(b) PrefixFreeSets,$N = 540$, correlation -0.56548137



(c) BasketballStrategy, $N = 47$, correlation -0.62801324



(d) VoteRigging,$N = 422$, correlation -0.15639571

Figure 5.13: Correlation between abilities and solving times – and least squares linear approximation thereof – for four tasks.

will now describe our tournament advancement prediction algorithm that also predicts the expected score necessary to advance, and the expected number of advancers.

Assuming that we are given a strictly decreasing positive function *score* that maps the task solving times into additive scores (such as the TopCoder's *score* function (2.3)), we can now compute the following function: Given a threshold score α, a contestant c, and a set of tasks used in the event $\mathcal{T}'$, what is the probability that c will achieve a score higher than α?

By summing these values over all contestants, we can also answer the question: Given a threshold score α and a set of tasks used in the event $\mathcal{T}'$, what is the expected count of contestants that exceed the score α?

We can now use binary search on the interval $[0, maximum\ score]$ to find the advancement threshold α_a such that the expected number of advancers is A. The predicted advancement probability for a contestant c is then simply his probability of reaching the advancement threshold.

Note that it is possible that the predicted advancement threshold α_a will be zero. In this case, our model predicts that the actual number of advancers will be lower than A. Such a prediction was not possible in the Bayesian rating model. (We add that we were able to succesfully predict such an event in a real life situation. More on this will be given in Chapter 5.)

Of course, in our model the advancement threshold depends on the tasks used. To get an accurate estimate, we will run multiple rounds of the advancement threshold estimation, each time random sampling the tasks for the round from the set of tasks used in the past.

Pseudocode of this algorithm is given in Algorithm 2. The running time of a single iteration of the algorithm is linear in the number of competitors, and logarithmic in the required precision to which the threshold α_a is estimated.

5.5.2 Implementation for the TopCoder setting

We implemented Algorithm 2 for the TopCoder setting. The random sampling of tasks was chosen so that the set used for the estimation always consists of three past tasks: one easy, one medium, and one hard task. We uniformly use the round duration of 75 minutes, and point values 250, 500, and 1000. We deliberately ignore challenges, resubmissions and similar complications.

The key part in implementing this algorithm was the implementation of the function `ProbReach` that returns the probability of contestant c exceeding score α given tasks $\mathcal{T}'$.

We start by splitting the analysis into 8 disjoint cases based on which tasks were solved. For each of the cases we compute its probability using the logistic functions for the particular tasks.

Now, given a set of solved tasks, we have to compute the probability that the total score from them exceeds α. In the most general case with all three tasks solved, we get the following

Algorithm 2: ScoreThresholdTournamentPrediction

Input: A vector of contestants $\mathcal{C}$
Input: A vector of past tasks $\mathcal{T}$
Input: A vector of contestant ratings θ
Input: Vectors of item parameters a, b, c, d, e
Input: Advancer count A
Input: Number of rounds to simulate N
Input: Precision ε
Output: Vector of advancement probabilities P

begin
  foreach $c \in \mathcal{C}$ **do** $P_c \leftarrow 0$
  for $i = 1$ to N **do**
    $\mathcal{T}' \leftarrow$ a random sample from $\mathcal{T}$ $\alpha_{lo}, \alpha_{hi} \leftarrow 0, maximum\ score$
    while $\alpha_{hi} - \alpha_{lo} > \varepsilon$ **do**
      $\alpha \leftarrow (\alpha_{hi} + \alpha_{lo})/2$
      $advancers \leftarrow 0$
      foreach $c \in \mathcal{C}$ **do** $advancers \leftarrow advancers + \texttt{ProbReach}(c, \mathcal{T}', \alpha)$
      if $advancers \leq A$ **then** $\alpha_{hi} \leftarrow \alpha$
      else $\alpha_{lo} \leftarrow \alpha$
    end
    $\alpha_a \leftarrow \alpha_{lo}$
    foreach $c \in \mathcal{C}$ **do** $P_c \leftarrow P_c + \texttt{ProbReach}(c, \mathcal{T}', \alpha_a)$
  end
  foreach $c \in \mathcal{C}$ **do** $P_c \leftarrow P_c/N$
  return P
end

situation:

Let $A = \{[x, y, z] \mid x \leq 250 \wedge y \leq 500 \wedge z \leq 1000 \wedge x + y + z > \alpha\}$. Let p_1, p_2, and p_3 be the probability distribution functions of achieved points for each of the tasks. Then the desired value is given by the equation:

$$\texttt{ProbReach}(c, \alpha) = \int_{[x,y,z] \in A} p_1(x) \cdot p_2(y) \cdot p_3(z)\ dx\, dy\, dz \tag{5.1}$$

Sadly, we hit the curse of dimensionality. To evaluate this equation approximately, we iterate over x and y with a fixed size step, each time evaluating the remaining 1D integral (in constant time using the underlying log-normal distribution of solving times), and interpolating.

The full implementation can be found in the Appendix A.

5.5.3 Results we achieved

For the TCO 2008 Online Round 1, our implementation was correctly able to predict that the number of advancers will be less than $A = 900$. Note that the previous models were unable to make predictions of this kind. (The predicted advancer count was 872.994, the actual number of advancers was 864.)

For predicting the advancers we used the μ_{perc} metric as defined in (3.14). The Monte Carlo algorithm (Algorithm 1 in this Thesis) scored -818.85, our Score Threshold algorithm (Algorithm 2) scored -812.26, and thus narrowly won.

For the TCO 2008 Online Round 1, our implementation predicted that the advancer count will be equal to $A = 300$, and this happened.

For predicting the advancers we used the TA measure. The Monte Carlo algorithm scored -372.044, the Score Threshold algorithm scored -394.676. Thus in this case the Monte Carlo algorithm performed slightly better.

Still, in both cases, even the simplest IRT-based rating model was able to compete with a modern Bayesian rating system. We expect the more elaborate IRT-based models to outperform the current rating system easily.

5.6 Attacking TopCoder's rating system

In this section we show an example of a situation where a purposefully worse performance would be rewarded by a higher rating after a few matches. The attack will be presented on the competition data of the author of this Thesis (TopCoder handle "misof", `coder_id` 8357090).

We will start by presenting an excerpt containing several consecutive rounds of the author's competition history in Table 5.2. We remind the reader that all data used in this section is publicly available at [Top09]. The column "rank" contains our placement in the respective

round, and columns "rating" and "volatility" contain our properties as computed by the TopCoder rating system *after* the corresponding round.

date	**round**	**rank**	**rating**	**volatility**
03.25.06	SRM 294	32	3219	317
04.03.06	SRM 296	4	3227	292
04.11.06	SRM 298	4	3231	269
04.22.06	SRM 299	54	3136	332
04.27.06	SRM 300	2	3177	320
05.03.06	TCO06 Semi 2	1	3238	327
05.05.06	TCO06 Finals	5	3192	319
05.09.06	SRM 301	1	3271	347
06.14.06	SRM 307	2	3302	327
07.12.06	SRM 311	1	3363	333
07.22.06	SRM 312	19	3308	332
08.09.06	SRM 315	1	3372	340

Table 5.2: Part of the TopCoder competition history for misof.

We now considered the hypothetical scenario in which we intentionally place last (rank 430) in the round "SRM 294". It turns out that the initial rating loss is by far outweighed by the fact that the upcoming good performances give us larger rating changes. In fact, the new rating after the round "SRM 315" in this hypothetical scenario would not only be significantly higher than befort – it would even be higher than the overall highest rating ever achieved by us in real life.

Both old and new ratings and volatilities are shown in Table 5.3.

round	rank	old rat.	old vol.	new rat.	new vol.
SRM 294	32/430	3219	317	3114	488
SRM 296	4	3227	292	3147	455
SRM 298	4	3231	269	3174	423
SRM 299	54	3136	332	3098	427
SRM 300	2	3177	320	3152	412
TCO06 Semi 2	1	3238	327	3223	414
TCO06 Finals	5	3192	319	3180	394
SRM 301	1	3271	347	3268	417
SRM 307	2	3302	327	3306	394
SRM 311	1	3363	333	3374	396
SRM 312	19	3308	332	3323	383
SRM 315	1	3372	340	3393	388

Table 5.3: Ratings before and after the attack.

Chapter 6

Other aspects of programming contests

The previous three chapters focused on the main area of this Thesis – rating systems related to programming contests. In this chapter we address two other topics related to programming contests: the design – and in particular, secure design – of grading systems, and the topic of black box testing as used in programming competitions.

6.1 Grading systems

We published a part of the material in this section in [For06b]. Subsequently the article [Mar07] extends some of the addressed topics.

The goal of this section is to provide a comprehensive classification of new, specific threats that arise in the context of programming contests. This can be seen as a first step towards having a secure, trusted contest system that can be adopted by a major group of different programming contests. We argue that it is necessary to approach the design of a contest system as a design of a secure IT system, using known methods from the area of computer security. Specifically, a secure contest system should explicitly address and handle all the threats mentioned in this section.

6.1.1 Introduction

Most of the programming contests [IOI, ACMb, TC, IPS] use some kind of automatic evaluation of the contestants' submissions. Usually the contest organizers (create and) use a software package that provides this functionality, along with some additional functions, such as providing an interface for the contestants, administration functions, etc. This software package is called a *contest system*.

There are many online contest systems. One of the oldest and most famous is the University of Valladolid judge (UVa judge, see [UVa] for the site and [RML08] for a description of its

architecture), currently the largest and most modern is the Sphere Online Judge (SPOJ judge, see [Sph]). A complete description of one Slovak contest system is given (in Slovak only) in [Kou04]. Various other contest systems with sources/binaries available to the general public include [Moo], [PF], and the recently started [Mar08] (described in [Mar07]), which is currently maintained by the head of the IOI's International Technical Working Group.

This section will start by an analysis of the current situation in the area of contest systems used in various programming contests. The main focus will be on the computer security point of view.

Currently, almost each programming contest uses its own contest system. Still, one has to mention that there are some exceptions to this rule. Probably the most widely used contest system is PC^2 (PC squared, [ABL]). It is used to organize ACM ICPC contests in multiple regions worldwide. However, in personal communication with contest organizers we often got the response that they only use PC^2 because they lack both an alternative that would suit their needs better, and the resources to develop their own contest system. Moreover, PC^2 is designed for the ACM ICPC only, and thus it is completely unsuitable for any other contest type.

Clearly, the current situation has got many disadvantages to the whole community behind the programming contests. The same work (designing and developing the contest systems), and the same mistakes are repeated over and over again. There is almost no sharing of experience, and almost no reusing of the contest systems happens.

In this section we try to make first formal steps in the direction towards a contest system that could be adopted by organizers of different contests.

In Subsection 6.1.2 we give a brief overview of the internals of a contest system. In Subsection 6.1.3 we discuss the requirements a contest system should fulfill in order to be acceptable for organizers of different contests. In Subsection 6.1.4 we suggest an approach to designing a secure contest system.

Subsection 6.1.5 contains the main contribution: In this section we give an overview of specific threats to contest systems, we classify the threats into categories, and discuss ways of preventing these threats.

6.1.2 Overview of a generic contest system

In this section we give a brief overview of the features common to most contest systems. A reader familiar with some contest system may skip this section.

The main functionality provided by the contest system is *an automated judge*. This is a piece of software responsible for automatical evalution of the contestants' submissions. The canonical way of evaluating a submission consists of *compiling* the source code submitted by the contestant, *executing* the resulting program on a prepared set of test inputs, and checking the program's output for correctness.

When the contestant's program is being run, it is usually isolated in a logical area called a *sandbox*. The goal of the sandbox is to prevent malicious submissions from accessing the restricted areas of the contest system, and to enforce various limits set on the submitted program.

Other functionality necessary for running a contest usually includes *user management* (e.g., authentication, access control), *administration functions* (e.g., adding/editing tasks, test data, setting time limits, rejudging submissions), *contest evaluation* (computing the rankings), and a *contest portal* (displaying rankings, handling submissions and clarifications, displaying additional information).

Some of the existing contest systems offer additional functions, such as allowing the users to *backup* their data, or *print* their source codes.

Sandbox implementations

The most common way how to implement a sandbox environment is via monitoring system calls. Under a *NIX operating system, this can be done using the `ptrace` command. Additional restrictions can be applied, such as using `ulimit` to limit the resources of the sandboxed process. This is the approach taken by the contest systems [Kou04, Mar08].

A different approach is taken for example in USACO ([USA], contest described in [KP07]) – the contestants' submissions are in fact executed in a completely virtualized environment Vx32. The implementation details were presented in [FC08].

6.1.3 Requirements on contest systems

We will start this section by trying to identify the main requirements on a contest system that could be adopted by a major community of contest organizers. Of course, these requirements can also be read as reasons why none of the currently existing contest systems is universally adopted.

Flexibility. Each programming contest is different, and, naturally, each programming contest has other requirements on the contest system. (As a canonical example consider the part of the system that computes and displays the rankings – it has to reflect the contest's rules.)

A contest system that aims to be universally adopted has to be easy to adjust to different contest rules. The contest system has to be designed with flexibility in mind. One example of a contest system designed to be flexible is CESS2 [Kou04]. The flexibility is achieved through the fact that the parts that may need to be changed are implemented in a simple scripting language, Lua [IdFC96].

To achieve flexibility, the contest system should have a modular design, where it is easy to replace some of the modules, and reuse other ones without a change.

Robustness. First of all, a universal contest system has to be robust and reliable. There is not much hope in a contest system that tends to crash on random occassions, or when used in ways not intended by its author.

User-friendliness and support. On a similar note, the system has to be easy to use, and documented well. Note that most of the current contest systems only have the functionality they need to run the contest. They are poorly documented, or not documented at all, and they can be only used by their designers and developers.

A recent example of a contest system that aims to be robust and user-friendly is the Polish SIO.NET system [Mic06]. However, according to our best knowledge, this project has already been abandoned in favor of the SPOJ system [Sph].

Trust. The issue of trust (in the security of the contest system) has been neglected in the past. Quite on the contrary, we consider this to be one of the most important issues. We percieve the whole contest system as a secure IT system: The system has to be able to restrict access to some information, protect the integrity of stored results, and so on.

We expect that the issue of trust and security will play a more important role in the future. Especially for major contests the trust in the used contest system is really an important issue, and it really should be addressed by contest system authors.

We give an initial discussion of this matter in the next section.

6.1.4 Designing a secure contest system

The standard framework used for designing and evaluating secure IT systems is given by the Common Criteria (CC, [CC05]). A short overview of the CC is given in [CCO]. The main reasoning behind this framework is that most secure IT systems share many properties and functional requirements, and that the logical consequence is the potential of the same types of attacks.

The CC provide a framework that, given a set of functional and security requirements, helps the designer to establish a set of protective measures that have to be implemented. In other words, the output from applying the CC framework is: "if you want a system that is this secure, you have to implement these security mechanisms, otherwise it would be vulnerable to such and such types of attacks".

It is a common practice to certify secure systems against the CC framework. The certificate states that the design of a system correctly identifies all necessary functional and security requirements for the given security level.

6.1.5 Attacks on contest systems

Computer security can be viewed as a game between the attacker and the defender – both try to think of possible attacks, one tries to employ them, while the other tries to prevent them.

When designing a secure IT system, the framework given e.g. by the Common Criteria, shows us the threats that are inherent in the design of each system having the selected functional and security requirements. However, this does not suffice. We always have to consider the situation at hand, and identify and prevent the threats specific to our system.

For contest systems, there was no known list of such attacks. In the subsequent text we attempt to fill in this gap, thereby establishing a knowledge base in this area. The developers of the current contest systems should check them for vulnerabilities to the attack we describe. More importantly, future contest system designers should design the new contest systems with these attacks in mind. Note that we are aware that many of these attacks were attempted in practice, and we tried almost all of them on test installations of various contest systems.

The most intuitive way of classifying the possible attacks is to determine the time when the attack happens. In this classification we will distinguish three types of attacks: *compilation-time*, *execution-time*, and *contest-time*. (The first two types of attacks occur during the compilation and execution of the contestants' submissions, the third type contains all other attacks.)

In addition, some of the attacks fit into known categories from the area of computer security. We will now give a brief overview of these categories.

The purpose of a *denial of service* (DoS) attack is to block some users of the IT system from accessing and using a particular resource. A common example of a DoS attack is when a computer virus uses infected computers to overload the inbound or outbound internet connection of some site, thereby denying access to the site's regular visitors. A general overview of DoS attacks, preventive measures, and possible responses is given in [Cen].

The most common attack type is a *privileges escalation* attack, where the attacker exploits some flaw in the design or implementation of the system to access restricted areas, materials, etc.

The only goal of a *destructive attack* is harming the system in some way, and thus disabling some of its functions or security measures.

The term *covert channel* is used to describe a situation where the attacker uses some way not intended by the system's designer to communicate information. Covert channels are usually used to communicate classified information to unauthorized users or areas. The idea of covert channels was introduced in [Lam73].

Forcing a high compilation time

Types: *compilation-time, denial of service*
Description: A short and simple program can still require an unreasonably high compilation time. If the contest system does not enforce a time limit on the compiler, this can be used to block the compiler and thus to endanger, or even completely stop the automated testing process. An example of one such C++ program is given in Subsection 6.1.6. In fact, it is

possible to produce a legal C++ program that will never compile. We include an example of such a program as well.

Note: Here we would like to note that the IOI adopted a compilation time limit after one contestant submitted a legitimate, non-malicious program that took approximately 90 seconds to compile.

Prevention: Setting a compilation time limit is enough to handle this problem. Using more than one machine to test the contestants' programs in parallel may help to mitigate effects of this attack, but it doesn't necessarily prevent it.

Consuming resources at compilation time

Types: *compilation-time, denial of service*

Due to the language design, some compilers (especially C++ compilers) may sometimes need to make an indefinite lookahead when parsing the source code. (I.e., to understand a command it needs to examine the code that follows, and we can not bound how much code will have to be examined.)

If the compiler is given a large input file, it may need to read it whole into memory. (Some compilers even do this always, even if it is not necessary.) An especially nasty way of achieving this effect in a Unix-like environments is to have the line: `#include "/dev/urandom"` included in your source code. The compiler will read the random garbage into memory until it runs out of memory.

Note that if the operating system can be forced to run out of memory, this can have unpredictable, and even fatal results. For example, any process can be killed if it happens to request more memory when all memory is used up.

Prevention: A way to prevent this from happening is to compile the contestants's submissions inside a limited environment – the compiler will have access only to a restricted part of the system resources (e.g., memory). Probably the most simple thing to do would be to run the compiler in the same environment you use for *running* the contestants' submissions. However, note that the compiler usually needs to access various parts of the operating system (and most specifically, filesystem) that should be out of bounds for the submissions. Thus this issue clearly needs to be adressed at contest system design time.

Accessing restricted material

Types: *compilation-time/execution-time, privileges escalation*

The attacker may try to access restricted material, such as the correct outputs for the test data, author's solution, the submissions of other contestants, a log file with the evaluation results, etc. For most of these resources even read access can be abused.

The non-obvious part of this attack is that it can happen during compilation time. If the attacker knows or guesses the right location of the restricted files, he may try to

include them into his program. (E.g., imagine a submission consisting of only one line: `#include "../authors-solution.cpp"`)

Note: This attack type is most harmful in local competitions, where it is possible for the contestants to easily gain access to the contest environment. For example, in the local Slovak competition "Zenit", the grading software is shipped to schools, and on many schools the contestants get to see it after the contest (even if they are not supposed to). On several occassions it turned out that the contestants were able to use this information to be able to circumvent the grading process entirely.

Prevention: Years of computer security have shown that *security through obscurity* can never work. (If you are not familiar with this topic, read [Bla03] for an unrelated but fascinating example.) The best approach to security is transparency. The design of the system should be public, and it should be such that the contestant's program has absolutely no access to the restricted material. Ideally, when the contestant's submission is being compiled and run, most of the restricted material should not be present on the given machine at all.

Misusing the network

Types: *contest-time, privileges escalation*

If the contestant is allowed to monitor and store network traffic, especially incoming traffic for the automated judge system, he may be able to intercept other contestants' submissions and later submit them as his own.

In on-site competitions, other attacks from this category include communications between the contestants over the network, or using other contestant's password to access his submissions. (We are aware of a past situation when the latter attack was attempted in an international competition.)

Note: The first scenario above is especially dangerous in ICPC-like contests, where contestants get (almost) immediate feedback on the correctness of their solutions. The attacker can use this information (publicly available, e.g. in the form of the "current standings" web page) to select those submissions he wants to copy.

Prevention: All communication has to be done in secure ways, e.g., using the HTTPS protocol when accessing the contest web server. In the case of an on-site contest, additional preventive measures are possible, such as configuring the network hardware (switches) to allow only server $\leftrightarrow$ client communication, monitoring the network traffic by the organizers, and logging all contestants' actions (including logins into the system).

Modifying or harming the testing environment

Types: *execution-time, destructive/DoS/other*

The simplest type of a harming attack is to submit a program that tries to delete all it can. A slightly more subtle way of harming the system is generating a huge output (file),

thereby forcing the system to run out of disk space.

However, the attack does not always have to be destructive. We will give a concrete example of one attack that modified the testing environment in a clever way.

In the attacked system the contestant's program was compiled and run on several test inputs. After each run, the output of the contestant's program was compared to the correct output (using a file compare tool `fc`). If and only if for all outputs `fc` reported no differences, the submission was accepted.

The attacker's program did not produce any output. Instead, it created a new binary called `fc`. After the contestant's program finished, the automated judge software called this binary instead of the original `fc`. The only things this binary did was to copy the correct output to the contestant's output file, report that no differences were encountered, and delete itself. Note that at this moment the system was back in its original state, and all following submits were tested normally.

Another, more simple example of this type of attack is pre-computing some information and keeping it in a temporary file. In the next runs, if the temporary file is found, its contents is used to save running time.

Prevention: Even in the sandbox environment the designer of the contest system has to be very careful about what should the contestant's program be allowed to do. Almost anything that is not necessary for a correct program can be misused to harm the system.

Circumventing the time measurement

Types: *execution-time, other*

One of the more subtle attack types is trying to circumvent the automated judge's time measurement to get more computation time.

The simplest possibility is that if the system only measures the time of the process it starts, the running program can fork a child process (which now has unlimited time) and sleep until the child process finishes.

There are more subtle approaches, that can succeed even if the imposed limits do not allow the solution to fork. For example, in the currently used Linux kernels (versions 2.4 and 2.6) the process time measurement is not precise enough. A solution that always runs for a short period of time and then sleeps for a carefully determined period of time may be able to use an arbitrary amount of processor time without this showing in the process information given by the kernel. (We were able to implement this type of attack on a machine running a vanilla 2.4 series Linux kernel.)

A possible way to block the testing facility is to let the program wait for an unavailable system resource. Such a program doesn't consume processor time, and it can run for an arbitrarily long time without exceeding the processor time limit.

Prevention: In the Slovak contest system the testing machine uses a patched Linux kernel to allow exact processor time measurement for processes. The patch can be found at [Pet]. In

general, the designer of the system has to be aware of the internals of the platform he decides to use, and be sure to measure the process time correctly.

In addition, we suggest limiting the contestants' programs to one thread (disallowing `fork()`s, `clone()`s, and analogous system calls on other operating systems). A good preventive measure is to have a (sufficiently large) real time limit in addition to the processor time limit.

Exploiting covert channels

Types: *contest-time, covert channels*

The contest system often informs the contestant about the success of his submission. This information often depends on the submitted program's behavior on the (secret) test data. (ACM ICPC-like contests are a canonical example of this situation.)

The information given to the contestant can be used to get information on the secret test data. The more information is given to the contestant, the more bits of secret information can be retrieved using a single submission. The most common things to use are the termination report (finished successfully, aborted, runtime error, etc.), and the information on the time and memory consumed by the program. An outline of a simple program that sends messages using this covert channel is given in Appendix 6.1.6

Note that the information obtained in this way can be used to write an incorrect heuristic program that correctly solves the given set of test data. This attack tends to be very successful for tasks where the answer is binary (YES or NO).

Prevention: For some contest types there is no way to prevent this type of attack. The only thing that can be done is to mitigate its success. This can be done by careful choice of how the automated judge's response should look like, and by using good test data.

For example, ICPC contests penalize the contestants for incorrect submissions. (However, if the contestants manage to use the covert channel successfully, they still benefit.) In the last years IOI started to provide feedback for some of the competition tasks. This is done in a careful way – the amount of feedback is limited, and the contestants can request it only for a limited number of times.

Misusing additional services

Types: *contest-time, denial of service*

Often the additional services, such as source code printing or backup facility, are not as secured as the rest of the system, and it is possible to misuse them.

The most common form of an attack is a denial of service attack on one of the services. A simple example: a contestant submits a 500-page text file that uses up all the available paper in the printer.

Sometimes the attack may even harm other parts of the system. E.g., the attacker may try to backup inappropriate amounts of data, thus forcing the contest system to run out of disk space.

Prevention: We advice setting a hard limit on the number of printed pages, and the size of the files a contestant is allowed to print and backup. (Similar bounds should apply for other additional services.)

Note that just limiting the *size* of the file submitted for printing is not sufficient. The issue of printing is more tricky than it may seem at the first sight. Using the PostScript language one can create documents shorter than 1 kB that contain millions of pages, or take arbitrarily long to evaluate. One such example is given in the Appendix 6.1.6. We would like to stress that also many print filters (like `a2ps`) are vulnerable to this attack.

Exploiting bugs in the operating system

Types: *execution-time/contest-time, privileges escalation*

This type of attack should be partially covered by the general security requirements for secure IT systems. However, there are specific issues worth mentioning.

If the contestant is able to access the machine where the submissions are tested, he may attempt to scan this machine for known vulnerabilities, and try to exploit them. This is the general attack type.

The specific issue in this case is that the contestant's program is executed on the testing machine with the privileges of some local user. This is always considered to be a higher security risk than an outside access to the machine's services. The contestant can try to run arbitrary code on this machine, including an *exploit* – a short program that uses some bug in the operating system or one of its services to gain (usually super-user) privileges. Successful exploiting of the machine that does the testing may lead to the exposure of secret testing data or even let the contestant harm the system.

Prevention: It is advised to isolate the parts of the system the contestants should not have access to. The contestants should only be allowed to communicate with a computer that works as an interface (a gateway) to the whole contest system.

The contestants' submissions have to be executed in a secure sandbox environment that prevents them from accessing unnecessary parts of the operating system.

Obfuscation

Obfuscation is not really an attack type, it is just a tool to help other types of attacks to succeed. For example, many of the contest systems we encountered simply used `grep` (a tool to search for a string in text) to check whether the program included any forbidden system calls.

Prevention: The method described above is **not adequate** to prevent the program from

making the forbidden system calls, or other forbidden attacks. For several contest systems we were able to successfully circumvent such protection.

The best way to prevent obfuscated attacks from succeeding is, of course, checking for the real threats, not for their common symptoms. E.g., if you want to check whether a program uses the `fork()` system call, do not look for it in the source code. Instead, intercept and check all the system calls the program makes during its runtime.

6.1.6 Code snippets illustrating some of the attacks

Forcing a high compilation time

```
#include <map>
using namespace std;

typedef map<int,int> M1;
typedef map<M1,M1> M2;
typedef map<M2,M2> M3;
typedef map<M3,M3> M4;
typedef map<M4,M4> M5;
typedef map<M5,M5> M6;
typedef map<M6,M6> M7;
typedef map<M7,M7> M8;

int main() { M8 tmp; }
```

Forcing an infinite compilation time

In a non-interactive environment, the device `/dev/stdin` simply does not send anything. The compiler remains waiting indefinitely, while consuming no additional resources (neither memory, nor CPU time[1]). Wall clock time has to be used to stop the compiler in this case.

```
#include "/dev/stdin"
#include <iostream>
int main() {
  std::cout << "hello, world\n";
}
```

[1]This actually depends on the type of scheduler used, but in general it does not make sense to wake up the process before any data becomes available for it to read.

Exploiting covert channels

The following function is able to send 7 bits of information in the automated judge's response, if the response contains the termination type and the memory used (rounded to megabytes).

```
void sendInformation(int n) {
  malloc(1024 * (n&31)); // send 5 bits in the memory size
  n >>= 5;
  if (n==0) exit(0); // terminate with a wrong answer
  if (n==1) assert(0); // abort
  if (n==2) while(1); // exceed the time limit
  if (n==3) { int a=3*n; a/=9-a; } // division by zero
}
```

Masking the system call

This program intentionally uses `syscall(0,...)` instead of `read(...)`. In the same way, `fork` can be executed without ever mentioning the string `fork` within the code. Even more low-level tricks can be used for this purpose (i.e., banning the string `syscall` as well is not a correct solution of this problem).

```
#include <stdio.h>
#include <unistd.h>

char buf[100];

int main() {
  printf("%d\n",syscall(0,0,buf,10));
}
```

Misusing additional services

A small PostScript file with a huge/infinite number of pages.

```
%!PS
/Times-Roman findfont 100 scalefont setfont
/page 0 def
{
  /page page 1 add def    % increment the page counter
  page 4 gt { exit } if   % try omitting this line!
  100 550 moveto
  page 20 string cvs show % print the page number
```

```
  showpage                % ship out the page
} loop
```

6.2 Automated black-box testing

In this section we present an overview of black-box testing used in practical programming competitions. We discuss the disadvantages of this approach, show that some of them are inherent (e.g., by proving the coNP-completeness of a suitable abstraction) and provide some methods to mitigate their consequences. Parts of the material used in this section were previously published in [For06a].

6.2.1 Introduction

Competitions similar to the International Olympiad in Informatics (IOI, [IOI]) and the ACM International Collegiate Programming Contest (ACM ICPC, [ACMb]) have been going on for many years. In the ACM ICPC contest model the contestants are given feedback on the correctness of their submissions during the contest. Due to a vast amount of submitted programs this is almost always done automatically. (Often there is a human supervising the testing process.) At the IOI the submitted programs are only tested after the contest ends, and the submissions are awarded a partial score for solving each of the test inputs.

We will now describe this canonical IOI scoring model in more detail. Each of the tasks presented to the contestants is worth 100 points. Before the competition the author of the task prepares his own solution, a set of test inputs, and an output correctness checker. (The output correctness checker can be replaced by a set of correct output files, if they are unique.) The 100 points are distributed among the test inputs. After the contest ends, each of the contestants' programs is compiled and run on each test input. For each test input the program solves correctly (and without exceeding some enforced limits) the contestant is awarded points associated with that test input. This testing model is commonly known under the name *black-box testing*.

We now answered the question "How?". However, an even more important question is "Why?". What are the goals this automated evaluation procedure tries to accomplish? After many discussions with other members of the IOI community, our understanding is that the main goals are:

1. Contestants are supposed to find and implement a **correct algorithm**. I.e., their algorithm is supposed to correctly solve all instance of the given problem. A contestant that found and implemented a reasonably efficient correct algorithm should score more than a contestant that found an incorrect algorithm.

2. The number of points a contestant receives for a correct algorithm should depend on **its asymptotic time complexity**.

3. Several points should be deducted for small mistakes (e.g., not handling border cases properly).

Note that these are only general rules. There may be different task types, e.g., open data tasks or optimization tasks, where a different scoring schema has to be used. We intentionally omitted details like "worst-case vs. average-case complexity", and "complexity vs. efficiency within the given limits". However, the points mentioned above can be applied to a vast majority of IOI tasks, and to many programming tasks in general.

The canonical way to reach the above goals is a careful preparation of the test inputs. The inputs are prepared in such a way that any incorrect program should fail to solve most, if not all of them. Different sizes of test inputs are used to distinguish between differently efficient correct programs.

Well, at least that's the theory. In practice, sometimes an incorrect program scores far too many points, sometimes an asymptotically better program scores less points than a worse one, and sometimes a correct algorithm with a minor mistake (e.g., a typo) in its implementation scores zero points.

In this part of the thesis we document that these situations do indeed occur, try to identify the various reasons that can cause them and suggest steps to be taken in future to solve these problems.

In the next section we present some problematic IOI-level tasks from the recent years.

6.2.2 Investigating IOI-level tasks

To obtain some insight into the automated evaluation process we investigated all competition tasks from IOIs 2003, 2004, and 2005 (USA, Greece, and Poland). Our goal was to check whether there is an **incorrect** algorithm that's *easy to find*, *easy to implement* (in particular, easier than a correct algorithm) and *scores a significantly inappropriate amount of points* on the test inputs used in the competition.

For the tasks presented below we were able to find such algorithms. The results of this survey are presented in Table 6.1, some more details are in Appendix 6.2.11.

In most of the cases we are aware that implementations of algorithms similar and/or identical to the ones we found were indeed submitted by the contestants. Sometimes, we are also aware of inefficient but correct programs that scored much less.

In addition, both Table 6.1 and Appendix 6.2.11 contain a task "safe" that was already used in several similar contests (see the appendix for more details). The main reason for including this task in the survey results was to illustrate that sometimes an unintended (in this case: correct but theoretically inefficient) approach can achieve a full score.

As we show in Appendix 6.2.11, there are two different reasons behind these unpleasant facts. For some of these tasks the test inputs were not designed carefully, but the rest of these

task	algorithm type	points
IOI 2004: artemis	brute force, terminate on time	80
IOI 2004: farmer	greedy	90+
IOI 2004: hermes	"almost" greedy	45
IOI 2005: rectangle	symmetrical strategy	70
ACM ICPC: safe	pruned backtracking	100

Table 6.1: Task survey results.

tasks was not suitable for black-box testing at all! In other words, it was impossible to design good test inputs for these tasks.

Van Leeuwen in [vL05] carried out a more in-depth analysis of one of the tasks we investigated (Phidias, IOI 2004) and managed to show that also the test inputs for this task allowed many incorrect programs to score well, some of them even got a full score. A short overview of these results is given in [Ver06].

Together, these results show that this issue is quite significant and we have to take steps to prevent similar issues from happening in the future.

6.2.3 Tasks unsuitable for black-box testing

We would like to note that it can be formally shown that some interesting algorithmical tasks are not suitable for automatic IOI-style evaluation. In this section we present two such tasks and discuss why they are not suitable.

As a consequence, this means that while the current model of evaluation is used at the IOI, there will be some algorithmical tasks that can not be used as IOI problems. In order to broaden the set of possible tasks a different evaluation procedure would have to be employed.

Planar graph coloring

Given is a planar graph, find the smallest number of colors sufficient to color its vertices in such a way that no two neighbors have the same color.

Regardless of how the test inputs are chosen, there is a simple and wrong algorithm that's expected to solve at least half of the possible inputs: Check the trivial cases (a graph with no edges: 1 color, a bipartite graph: 2 colors). In the non-trivial case, flip a coin and output 3 or 4.

The Four-color theorem [AH97] guarantees that the answer is always at most 4. Thus when the answer is 3 or 4, our algorithm is correct with a 50% probability. Thus, for each possible set of test inputs this algorithm is expected to solve at least 50% of the test inputs correctly.

(Note that for various other reasons this task wouldn't be suitable for the IOI, we just used it because it is simple and well-known.)

Substring search

Given are two strings, *haystack* and *needle*, the task is to count the number of times *needle* occurs in *haystack* as a substring.

There are lots of known linear-time algorithms solving this task, with KMP [KMP77] being probably the most famous one. If we use n and h to denote the length of *needle* and *haystack*, the time complexity of these types of algorithms is $O(n+h)$. These algorithms are usually quite complicated and error-prone.

The problem is that there are simple but incorrect algorithms which are able to solve almost all possible inputs.

As an example, consider the Rabin-Karp algorithm [KR87].

In its correct implementation, we process all substrings of *haystack* of length n. For each of them we compute some hash value. (The common implementation uses a "running hash" that can be updated in $O(1)$ whenever we move to examine the next substring.) Each time the hash value matches the hash of *needle*, both strings are compared for equality.

The worst-case time complexity of this algorithm is $O(n(h-n))$, but its expected time complexity is $O(h+n)$.

The algorithm as stated above is correct. Hovewer, there is a "relaxation" of this algorithm that is even easier to implement, and guaranteed to be $O(h+n)$: Each time the hash value matches, count it as a match.

Note that while this algorithm is incorrect, it is very fast, and it is very highly improbable that it will fail (even once!) when doing the automated testing. Moreover, it is impossible to devise good test inputs beforehand, as the goodness of the test inputs depends on the exact hash function used. Using a known trick from the design of randomized algorithms (a random choice of a prime number used to compute the hash value, see [Hro05].) we can even implement an algorithm that will *almost* surely find the correct output for each valid input.

We would like to stress that, in practice, programs that misbehave only once in a large while are often one of the worst nightmares. Testing a program for subtle, sparsely occuring bugs, is almost impossible, and the bugs may have a critical impact when they occur after the program is released. By allowing such programs to achieve a full score in a programming contest we are encouraging students to write such programs. This may prove to be fatal not only in their future career, but also in our everyday lives (if the faulty software product touches them).

6.2.4 Heuristic methods for recognizing tasks unsuitable for black-box testing

Here we give a short summary of the guidelines informally published after IOI 2004 in [For04].

In order to make this presentation sufficiently brief we have to make a few generalizations. We will talk about an abstract problem statement along the lines "given this set of combina-

torial objects, find the best among them in some given sense". (Most programming tasks can be viewed in this way. E.g., the shortest path problem can be seen as "given the set of vertex sequences, find the one that represents a path from u to v and has the shortest length".)

We will use the term *possible answers* to denote the combinatorial objects the algorithm has to examine, and *correct answer* to denote the one it should find.

The tasks that are unsuitable for black-box testing usually exhibit one or more of the following properties:

1. The set of correct answers is large.
2. We are only required to output some value depending on the correct answer, and the value for a random answer is nearly optimal.
3. There is a known (theoretically incorrect) heuristic algorithm with a high probability of solving a random test input correctly, within the imposed limits.

The rationale behind this statement follows.

For almost all programming tasks it is possible, and not very hard, to implement a randomized brute force search that examines a subset of possible answers and outputs the best one found. If the set of correct answers is large, it is quite common that this approach will be successful and the found answer will often be optimal.

We are in a similar situation if condition 2 holds for the problem. In this situation, in addition to random search, it is possible to guess the value for the correct answer. Examples of such problems are IOI 2004 tasks Artemis and Farmer, see Appendix 6.2.11.

There are some problems that exhibit only the third property. Here it may be possible to devise test inputs that break one known heuristic algorithm, but practice shows that many contestants will implement variatons on the known heuristic algorithm(s), and that these variations will solve all (or almost all) test inputs correctly. An example is the substring search problem presented in the previous section. There it is even provably impossible to design test inputs that break all possible incorrect heuristic algorithms.

More detailed arguments can be found in [For04].

6.2.5 A theoretical result on competition testing

The general theoretical formulation of the testing problem (given are two Turing machines, do they accept the same language?) is not decidable – see [HU79]. While this result does not exactly apply to programming competitions (the set of valid test inputs is finite), it does give us insight about the hardness of our goal.

We will now show that a more exact formulation of our testing problem is coNP-complete problem. Let CompetitionTesting be the following decision problem: Given are two algorithms A and B, and three integers T, M, and N. It is guaranteed that on any input of length N or less algorithm A terminates in less than T steps and using less than M units of memory.

Is the following claim true? "On any input of length N or less algorithm B terminates in less than T steps and using less than M units of memory, and its output is the same as the output of algorithm A."

As a technicality, we require the integers T, M and N to be given in unary, hence allowing time complexities polynomial in these variables. This definition gives us a good approximation of the situation in practice.

It is obvious that CompetitionTesting is in coNP: If there is an input for which A and B differ, we can use non-determinism to guess it in $O(N)$, and then simulate both A and B for at most T steps to verify. As the given memory limit is M, we can do each step of the simulation in $O(M)$, hence the entire simulation will run in $O(MT)$, which makes the algorithm polynomial in the input size.

We will now show the coNP-hardness of CompetitionTesting by reducing 3-SAT to its complement.

It is sufficient to use a many-to-one reduction: we will map each 3-SAT formula to an instance of CompetitionTesting such that the algorithms are equivalent iff the 3-SAT formula was not satisfiable.

Consider an arbitrary 3-SAT instance $\mathcal{I}$ on variables $x_1, \ldots, x_S$. Each clause of $\mathcal{I}$ can be described using three integers from $\{-S, \ldots, -1, 1, \ldots, S\}$, where negative integers correspond to negative literals. For example, the 3-SAT instance $\mathcal{I} = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_4 \vee x_5)$ can be described as $\{\{1, 2, -3\}, \{2, -4, 5\}\}$. These numbers can easily be stored as constants in an algorithm.

Let A be an algorithm that just outputs $\top$ and terminates. We will now construct B as follows: B will read the first S bits of input. (If the input is not long enough, B will pad it with zeroes.) B then interprets these bits as truth values for the variables x_1 to x_S. Afterwards, B processes the clauses of $\mathcal{I}$ one by one, evaluating each of them and checking whether all of them are true. If it encounters a false clause, B outputs $\top$ – i.e., the same as A. On the other hand, if all clauses are satisfied, B outputs $\bot$.

Let C be the number of clauses in $\mathcal{I}$. Then we can easily construct B from $\mathcal{I}$ in $O(C)$ time. The time complexity of B is $O(S + C) = O(C)$, as it first reads S bits and then evaluates C constant-size clauses.

Let $M = S + k$, $N = S$, and $T = kC$ for some suitable constant $k > 1$. (Note that once we pick a concrete computational model for A and B, this k will be obvious from the construction of B.)

Then the tuple (A, B, T, M, N) has exactly the property we need. If the original formula $\mathcal{I}$ was not satisfiable, then on each possible input B will output the same as A, and it will do so within the limits. Otherwise, we can take the variable values that satisfy $\mathcal{I}$ and produce a valid input on which A and B differ.

Therefore CompetitionTesting is indeed coNP-complete.

Code sample

The following C++ code interprets the given parameter n as a vector of bits, and evaluates whether these bits satisfy the formula $\mathcal{I} = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_4 \vee x_5)$.

```
#define IS_SET(vector,bit) ((vector) & (1<<(bit)))
bool evaluate(int n) {
  n <<= 1; // move the least significant bit to position 1
  bool clause1 = IS_SET(n,1) || IS_SET(n,2) || (!IS_SET(n,3));
  bool clause2 = IS_SET(n,2) || (!IS_SET(n,4)) || IS_SET(n,5);
  return clause1 && clause2;
}
```

6.2.6 Other evaluation methods

In the previous sections we have shown that the currently used automated black-box testing is not suitable for some tasks the computer science offers. To be able to have contestants solve other task types it may be worth looking at other evaluation methods, discuss their advantages, disadvantages and suitability for the IOI.

Moreover, a huge disadvantage of black-box testing is that an almost correct program with a small mistake may score zero points. In our opinion the most important part of solving a task is the thought process, when the contestant discovers the idea of the solution and convinces himself that the resulting algorithm is correct and reasonably fast. We shall try to find such ways of evaluating the contestants' submissions that the contestant's score will correspond to the quality of the algorithm he found. The implementation is important, too, but the punishment for minor mistakes may be too great when using the current evaluation model. This is another reason why discussing new evaluation methods is important.

Below we will present a set of alternate evaluation methods along with our comments.

Pen-and-paper evaluation

In the Slovak Olympiad in Informatics, in the first two rounds contestants have to solve theoretical problems. Their goal is to devise a correct algorithm that's as fast as possible, and to give rationale why their algorithm works. Their works are then reviewed and scored by experienced informatics teachers, and enthusiastic university students.

While we believe that this form of a contest has got many benefits (as it forces the contestants to find correct algorithms, to be able to formulate and formally denote their ideas), we do not see it, per se, as suitable for the IOI. The main problems here are the time necessary to evaluate all the works, the language barrier, and objectivity.

Note that this model is currently being used, among others, at the International Mathematical Olympiad (IMO). In spite of the best effort of the delegation leaders and problem

coordinators at the IMO, the process is still prone to a human error, and we are aware of cases when almost identical works were awarded a different score.

Supplying a proof

For the sake of completeness, we want to state that some programming languages (e.g., see [VKK09]) allow the programmer to write not only the program itself, but also its formal proof of correctness.

While this theoretically solves all the difficulties with testing the programs for correctness, we do not see this option to be suitable for high school students. Giving a detailed formal proof is far beyond the current scope of the IOI. Moreover, a formal proof is usually far more complex than the algorithm we are proving. Thus, this approach would just turn the problem solving competition into a competition in writing formal proofs.

Code review – white-box testing

Often a much easier task than designing a universal set of test inputs is proving a given implementation wrong, i.e., finding a single test input that breaks it.

This model is currently implemented and used in the TopCoder Algorithm competitions, see [TC]. In each competition there is some allocated time when the contestants may view the programs other contestants submitted. During this phase, whenever a contestant thinks that he found a bug in some program, he may try to construct a test input that breaks it. Programs shown to be incorrect in this way score zero points. (Moreover, the contestant that found the test input is awarded some bonus points for this.)

This procedure is accompanied by black-box testing on a relatively large set of test inputs. Only programs that successfully pass through both testing phases score points.

We are aware that also in some online contests organized by the USA Computing Olympiad (USACO) the contestants are encouraged to send in difficult and/or tricky sets of test inputs. While this is not exactly white-box testing, this approach is similar in that the contestants get to design the test inputs.

Some variation of this type of evaluation could be implemented on the IOI. During the phase that's currently only used to check the results of the automated testing and to make appeals, the contestants could be able to read the submitted programs and suggest new test inputs. This is an interesting idea and we feel that it should be discussed in the IOI community.

One more note: The white-box testing still leaves the burden of proof on the wrong side of the barrier. In real life the programmer should be responsible for showing that his code is correct, but this is not the case here.

Open-data tasks

An open-data task is a fairly new notion, made famous by the Internet Problem Solving Contest (IPSC, [IPS]) and later adopted by other contests, including the IOI. The main point is that the contestants are only required to produce correct output for a given set of inputs. The only evaluated thing are the output files, the method the contestant used (within the imposed resource limits) is not important.

There is a wide spectrum of tasks that are suitable to be used as open-data tasks at the IOI. For example, tasks where different approximation algorithms exist can be used as open-data tasks and evaluated based on the value of the answer the contestant found. (See the task XOR from IOI 2002.) Large input sizes can be used to force the contestants to write efficient programs. There may be tasks where some processing of the data "by hand" can be necessary or at least useful.

In our opinion the IOI has not used the full potential of open-data tasks yet and there are many interesting problems that can be formulated as open-data IOI-level competition tasks.

More extensive black-box testing

In some contests other than the IOI (such as ACM ICPC and TopCoder) the correctness of a submitted program has a larger importance. Usually if the program fails just one of the test inputs it is considered incorrect and it scores zero points.

We do not think that this, per se, would be a good model for the IOI, as even many of the participants are just beginners in programming and they often tend to make minor mistakes. Hovewer, it is possible to move the evaluation process in this direction. (This transition could be made less painful by selecting the competition tasks from a wider difficulty spectrum.) There are interesting variations on the canonical evaluation scheme that can be used to make correctness of the implementation more important.

One particular model worth mentioning is the model commonly used in the Polish Olympiad in Informatics. (This model was also used for evaluating several tasks on IOI 2005 in Poland.) The test inputs are divided into sets and each set is assigned some points. To gain the points for a set, a program has to solve all inputs from the set correctly.

Clearly this approach makes it easier to distinguish between correct and incorrect programs (e.g., large random test inputs can be bundled with small tricky hand-crafted inputs that will ensure that most of the heuristic algorithms fail). On the other hand, its disadvantage is that the result of a minor bug in an implementation may have an even worse impact.

Requiring correctness, aiming for speed

In our opinion, the goal we want to reach is to guide the students to write correct programs only. If the current evaluation scheme shall be changed, the new scheme should be better in pushing the students towards writing correct programs.

We would like to suggest the following scheme: The author of the problem creates two sets of test inputs. The first set, called the *correctness set*, will consist of 10 relatively small inputs. A reasonably fast correct program should be able to solve each of these inputs well within the imposed limits. The second set, called the *efficiency set*, will consist of 20+ inputs of various sizes.

Programs will be **forbidden** to give a wrong answer, they are only allowed to crash/time out on larger test inputs. In particular, they will be **required** to solve all inputs in the correctness set, and they will be scored according to their performance on the efficiency set **only**.

During the actual competition the contestant will be able to submit his program at any time. For each submission he will receive a report (pass/fail, running time, etc.) for each of the (at that time unknown) inputs from the correctness set.

This proposal is still open for suggestions. (For example, we are considering to award a small amount of points to programs that fail to solve one or two inputs from the correctness set, or alternatively the contestants could be allowed to see some of these inputs for a penalty.)

New task types

Of course, one could suggest new task types with a completely different evaluation scheme. For example, the contestants could design test inputs that force a given program behave in some way. (Some possible formulations: "Find an input that will force this program give an incorrect output."; "Find a valid input for which the result is as large as possible.")

Many examples of such tasks can be found in the past years of the IPSC contest (see [IPS]). Discussing the evaluation of such tasks at the IOI is beyond the scope of this thesis.

6.2.7 Test case design

What makes the contest setting specific (and different from the most commonly researched setting) is the existence of both a formal input/output specification of the programs that are to be tested, and a fully functional implementation.

At this point we would like to mention one possible way to utilize this observation in contest test case design.

Our suggestion is based on the *refinement calculus*. Refinement calculus is a formal system invented by Back in [Bac78], [Bac80]. This system extended Dijkstra's weakest precondition calculus [Dij76] to a theory of program refinement.

In the past years the refinement calculus was shown to be suitable for many different tasks, including formal denotation of specifications, *program refinement*, and data refinement.

In our context, program refinement is a process where programs are constructed by stepwise refinement of an initially abstract contract. Each refinement step is required to preserve the correctness of the previous version of the program, while introducing a new level of detail.

The refinement calculus was further developed and investigated, and got a solid logical and algebraic foundation based on higher-order logic and lattice theory. More on the process of developing the refinement calculus can be found in [BvW90], [Mor87], and [MRG88]. Current, evolved terminology has been published in the book [BvW98], and we recommend this book as an excellent resource on refinement calculus.

6.2.8 Refinement calculus in software testing

Using the refinement calculus terminology as defined in [BvW98], software testing can be viewed as a special case of the angel-demon game: the *angel* is the environment that does the testing, and the *demon* is the functional unit that is being tested.

The contract we want them to satisfy can be informally phrased as follows: Given is a series of test cases. For each test case, the *angel* is responsible to satisfy all the assumptions the *demon* was allowed to make, then the *demon* is allowed to process the test case, and it is required to behave in an expected way (e.g., produce an expected output).

Note that the *angel* wins if the whole contract is satisfied (in which case we conclude that the tested program has passed the test), or if the *demon* is "caught cheating" – fails to satisfy the required assertions in some test case.

In the more specific area of competition testing, the *angel* can usually be reduced to (a part of) the contest system, while the *demon* is a contestant's program that is being tested.

6.2.9 Refinement calculus in test case design

In the context of program development, contract refinement (see [BvW98] for a definition) can be viewed as a process of concretization: Starting from an abstract formal specification, we produce a chain of contracts leading to a complete implementation, while preserving its correctness against the original specification.

Aichernig in [Aic01b] and [Aic01a] considers reversing the direction of the refinement process.

Let S be the original contract that serves as a formal specification for the program that shall be developed. In this case, all abstractions of S (i.e., all contracts S' such that $S' \sqsubseteq S$) can be viewed as focusing on some particular aspect of the specification S.

In particular, each fixed test case scenario is an abstraction of the given formal specification S.

The main idea of Aichernig's work is to use the refinement calculus in reverse, to calculate a suitable set of test cases by partitioning the formal specification. An example application of this approach is presented in [Aic01b].

As a possible direction for further research, we suggest trying to adapt Aichernig's approach to generating competition test data.

Refinement calculus in education

The general idea of sequential refinement was successfully applied to teaching high-school mathematics. See [BvW99] and [BvW05] for more details. We found this approach enlightening, and we heartily recommend its usage wherever appropriate.

6.2.10 Plans for the future

Clearly, the short-term solution of the problems discussed above is awareness that these problems exist. In past, it was sometimes the case that the International Scientific Committee of the IOI (ISC) was aware of some problems connected with a proposed task, but they did not find these problems important enough to reject the task.

The mid-term to long-term solution includes discussion in the IOI community. The currently used evaluation model has got many known disadvantages and if we are able to come up with a better way to do the evaluation, the whole IOI could benefit from that. To reach this goal it is imperative that members of the IOI community actively take part in discussing the alternatives, some of which were presented in this article.

6.2.11 Details on tasks investigation.

Here we present a short summary of our investigation for each of the IOI tasks where we were able to find an incorrect algorithm that, in our opinion, scored too many points. As we already stated, a detailed description of problems with the IOI 2004 tasks Artemis and Farmer was informally published after IOI 2004 in [For04].

IOI 2004: Artemis

Task summary: Find an axes-parallel rectangle containing exactly T out of N given points.

Task description: `http://olympiads.win.tue.nl/ioi/ioi2004/contest/day1/artemis.pdf`

Correct solution: $O(N^2)$ dynamic programming

Incorrect algorithm: $O(N^3)$ brute-force search, terminate before the time limit expires and output the best answer found. This algorithm requires only a few lines of code, and it scores 80 points on the test inputs used in the competition.

Reason of the problems: A bad task. It is hard to construct a test input where no valid rectangle contains exactly T points. The set of possible answers is usually very large, it is easy to find one, thus also the brute force algorithm scores well. The sad thing is that the ISC was aware of the above facts(!) and still decided to use this task in the competition.

Notes: We are aware of the fact that several contestants submitted a correct $O(N^2 \log N)$ algorithm, which timed out on the larger test inputs, thus scoring approximately 40 points. This is the immediate consequence of a "solution" the host SC and ISC applied – raising the limit for N so that **one known brute-force program** would not score well.

This problem could probably be saved by requesting that the contestants output the exact number of minimal rectangles satisfying the given criteria. This forces all brute-force programs to time out on larger test inputs.

IOI 2004: Farmer

Task summary: Given a graph with N vertices containing only cycles and lines, find the largest possible number of edges in a subgraph with Q vertices.

Task description: `http://olympiads.win.tue.nl/ioi/ioi2004/contest/day2/farmer.pdf`

Correct solution: Knapsack-style dynamic programming.

Incorrect algorithm: Greedy. Consider all the cycles. If they contain more than Q vertices, flip a coin and output Q or $Q - 1$. Otherwise, take all the cycles and several strips (in sorted order, starting from the longest ones).

Regardless of the choice of test inputs, the expected score for this algorithm is at least 50 points. For the actual test data used in the competition, the expectation rises to 90 points, as our algorithm will always solve most of the actual test inputs correctly.

Reason of the problems: The host SC and ISC were aware that some heuristic algorithms can score well. The test data was chosen in such a way that the algorithms known to them did not score too many points. Sadly, this was not taken to be a reason to reject the task, as none of the problemsetters had the insight presented in [For04]. It is impossible to create good test data for this task.

IOI 2004: Hermes

Task summary: Given a sequence of N lattice points, find a lattice path starting at $(0, 0)$ such that a postman walking the path "can see" (horizontally or vertically) each of the given points in the given order.

Task description: `http://olympiads.win.tue.nl/ioi/ioi2004/contest/day1/hermes.pdf`

Correct solution: Somewhat complicated $O(N^2)$ dynamic programming.

Incorrect algorithms: A simple $O(N)$ greedy algorithm: Out of the two possibilities for the next step choose the shorter one. In case of a tie, flip a coin. This algorithm was apparrently known to the SC and scores only 10 points.

A $O(N^2)$ greedy algorithm: Run the first greedy algorithm N times, in the k-th run do the opposite choice in the k-th step. (I.e., make exactly 1 choice that's not locally optimal.) This algorithm is clearly incorrect, and still it scores 45 points.

Reason of the problems: Bad test data. In the problem specification it was stated that 50% of the test inputs will have $N \leq 80$. The truth was that in 50% of test inputs N did not exceed 20. This number of lattice points was not enough to make sufficiently complicated inputs.

Notes: In this problem the dimension D of the grid was smaller than N. There was a correct $O(DN)$ algorithm (which is faster than the presented $O(N^2)$). No additional points were awarded for finding and implementing this algorithm.

Moreover, the size of test inputs allowed a simple backtracking algorithm to score more points than the problem statement promised.

IOI 2005: Rectangle

Task summary: Find an optimal strategy for a 2-heap NIM where an allowed move is to choose one of the heaps and remove at most half of the tokens. The player not able to make a move loses.

Task description: `http://olympiads.win.tue.nl/ioi/ioi2005/contest/day2/rec/rec.pdf`

Correct solution: An optimal move may be found by a clever observation after encoding both heap sizes as base-2 numbers. Also, the more general Sprague-Grundy theory can be applied. (An overview of this theory can be found in [BCG82].)

Incorrect algorithm: Whenever possible, try to make two equal heaps. This algorithm is incorrect and yet it scored 70 points.

Reason of the problems: Bad test data. In almost all test inputs the heap sizes were similar and thus the player was able to enforce this strategy from the first move on. Actually, our algorithm easily solves all of the large test inputs and only fails on several small hand-crafted inputs. A combination of our incorrect algorithm and a brute force search could achieve a full score.

ACM ICPC: safe/ouroboros

We are aware that this problem was presented on different contests, including ACM ICPC Mid-Central European Regionals 2000, Slovak IOI Selection Camp 2003, and ACM ICPC Murcia Local Contest 2003.

Task summary: Given a set of D digits and N, find the shortest string of digits such that it contains each of the D^N possible N-digit strings as a substring.

Task description: `http://www.acm.inf.ethz.ch/ProblemSetArchive/B_EU_MCRC/2000/problems.pdf`, problem E

Correct solution: These strings are known as de Bruijn strings/sequences (see [dB46]), they correspond to an Eulerian path in a wisely constructed directed graph.

Inefficient algorithm: Pruned backtracking is able to solve all reasonably large inputs quickly.

Reason of the problems: A bad problem. The set of all (D, N) de Bruijn sequences is large and the backtracking algorithm can find one quickly.

Chapter 7

Conclusion

Our research that resulted in this Thesis was motivated by the area of programming contests. We decided to focus on the topic of rating and ranking the contestants based on their performance.

When starting the research, we had multiple reasons to feel that the current state of the art was inadequate. The existing rating systems were not able to model the obvious fact that different competition tasks can have a different degree of difficulty. Many of the existing rating systems were designed in an ad-hoc fashion, their validity was unclear, and moreover, it was apparrent that the author did not consider the possibility of attacking the rating system. In our Thesis we attempted to rectify all of these issues.

In Chapter 3 we start by clearly identifying the assumptions that define the setting we address. We develop our new formalism that will allow us to make exact reasoning about rating systems, and show that this formalism can be used to describe an existing modern rating system. According to our research we define an exact, objective approach to comparing different rating systems via evaluating the predictions that can be made from the computed ratings.

We then set off to define our own rating system that will have multiple advantages against the existing rating systems. In contrary to the popular Bayesian approach, we based our rating system on Item Response Theory. At the end of 3 we prove several important properties that the rating system must have, and then define a basic IRT-based rating system. In Chapter 4 we then address several advanced modifications of this rating system, including task weights, abilities that change in time and ways how additional information provided by the underlying model can be used to compute more precise predictions.

Also in Chapter 4 we address one particularly important topic – the topic of rating systems security. We analyze existing rating systems from the point of view of an attacker that wants to mislead the system, usually into overestimating the ability of a given subject. We show several types of attacks we found, and we show that existing rating systems are indeed vulnerable to these types of attacks. We also show that our IRT-based rating system is naturally resistant

to some of these attacks.

We were also able to verify our theoretical results from Chapters 3 and 4 on large sets of real life data. The results of this evaluation are presented in Chapter 5. In this Chapter we also show that the data computed by our rating system can be used to argue about task difficulty, and use this approach to analyze how to change the difficulty of tasks in the regional round of the Slovak Olympiad in Informatics.

Even though this Thesis covers a large spectrum of topics related to rating systems, the research in this area is nowhere near complete. To name a few open topics:

The improvements of our basic IRT-based rating systems mentioned in Section 4.2 surely need further attention. We expect that handling abilities that change in time will prove to be quite a challenge. Modelling how abilities can change in time will require an extensive analyzis of existing real life data.

We are certain that in addition to the attack types we describe in Section 4.3 there will be other possibilities how to attack rating systems, and these have to be identified and (if possible) prevented. Also, it should be possible to continue in the topic we started in Section 4.6.2 and examine the monotonicity of IRT ratings in closer detail.

When investigating real life data, we were not able to determine the connection between our ability estimates and the time it takes the subject to solve a given task.

Appendix A

Code listings

In this Appendix we present code listings for our proof-of-concept implementation of the algorithms presented in this thesis.

Code listing 1: 2PL model calibration

```
// proof of concept implementation: OneTimeCalibration
#include <algorithm>
#include <numeric>

#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <queue>
#include <set>
#include <map>

#include <cstdio>
#include <cstdlib>
#include <cctype>
#include <cassert>

#include <cmath>
#include <complex>
using namespace std;

#define FOREACH(it,c) for(__typeof((c).begin()) it=(c).begin();it!=(c).end();++it)
#define SIZE(t) ((int)((t).size()))
#define SQR(x) ((x)*(x))

#define MIN_THETA -10
#define MAX_THETA 10
#define MIN_A -1
#define MAX_A 10
#define MIN_B -10
#define MAX_B 10

int ABFILE=0, CDFILE=0, THFILE=0;

#include <fstream>
class row { public: string coder, task; int time; bool solved; };
vector<row> INPUT;
class row2 { public: int coder, task, time; bool solved; };
vector<row2> DATA;

map<string,int> coder_to_number, task_to_number;
map<int,string> number_to_coder, number_to_task;
```

```
vector<double> theta, sem, A, B, C, D, E;
int coder_count, task_count;

void load_responses() { // {{{
  ifstream fall("generated/responses");
  row R;
  while (fall >> R.coder >> R.task >> R.time) { R.solved = R.time>0; INPUT.push_back(R); }
} // }}}

void relabel_coders_and_tasks() { // {{{
  set<string> S;
  FOREACH(it,INPUT) S.insert(it->coder);
  vector<string> C( S.begin(), S.end() );

  S.clear();
  FOREACH(it,INPUT) S.insert(it->task);
  vector<string> T( S.begin(), S.end() );

  coder_count = SIZE(C);
  task_count = SIZE(T);

  for (int i=0; i<SIZE(C); i++) { coder_to_number[C[i]]=i; number_to_coder[i]=C[i]; }
  for (int i=0; i<SIZE(T); i++) { task_to_number[T[i]]=i; number_to_task[i]=T[i]; }
  row2 r;
  FOREACH(it,INPUT) {
    r.coder=coder_to_number[it->coder];
    r.task=task_to_number[it->task];
    r.time=it->time;
    r.solved=it->solved;
    DATA.push_back(r);
  }
} // }}}

void initialize_vars() { // {{{
  for (int i=0; i<coder_count; i++) theta.push_back(0.0);
  for (int i=0; i<coder_count; i++) sem.push_back(0.0);
  for (int i=0; i<task_count; i++) A.push_back(1.0);
  for (int i=0; i<task_count; i++) B.push_back(0.0);
  for (int i=0; i<task_count; i++) C.push_back(0.0);
  for (int i=0; i<task_count; i++) D.push_back(0.0);
  for (int i=0; i<task_count; i++) E.push_back(-1.0);
} // }}}

inline double prob_correct(double theta, double a, double b) { // {{{
  double tmp = exp(a * (theta-b) );
  double pr = tmp / (1+tmp);
  if (pr < 1e-7) pr = 1e-7;
  if (pr > 1-1e-7) pr = 1-1e-7;
  return pr;
} // }}}

vector< vector<double> > task_to_thetas, coder_to_as, coder_to_bs;
vector< vector<bool> > task_to_results, coder_to_results;

double get_initial_estimate_for_b(const vector<double> &T, const vector<bool> &C) { // {{{
  int N = SIZE(C);
  double blo = MIN_B, bhi = MAX_B;
  while (bhi-blo > 1e-7) {
    double bm1=(blo+blo+bhi)/3, bm2=(blo+bhi+bhi)/3;
    double lp1=0.0, lp2=0.0;
    for (int i=0; i<N; i++) {
      if (T[i] == MIN_THETA) continue;
      if (T[i] == MAX_THETA) continue;
      double pr1 = prob_correct(T[i],1,bm1);
      lp1 += log( C[i] ? pr1 : 1-pr1 );
      double pr2 = prob_correct(T[i],1,bm2);
      lp2 += log( C[i] ? pr2 : 1-pr2 );
    }
    if (lp1 < lp2) blo=bm1; else bhi=bm2;
  }
```

```
  return blo;
} // }}}

double eval(const vector<double> &T, const vector<bool> &C, double a, double b) { // {{{
  double res = 0.0;
  int N = SIZE(C);
  for (int i=0; i<N; i++) {
    if (T[i] == MIN_THETA) continue;
    if (T[i] == MAX_THETA) continue;
    if (C[i]) {
      res += log(prob_correct(T[i],a,b));
    } else {
      res += log(1-prob_correct(T[i],a,b));
    }
  }
  return res;
} // }}}

void write_abs() { // {{{
  ABFILE++;
  char buf[100]; sprintf(buf,"all_abs.%06d",ABFILE);
  ofstream fout(buf);
  for (int t=0; t<task_count; t++)
    fout << number_to_task[t] << " " << A[t] << " " << B[t] << endl;
} // }}}

void write_thetas() { // {{{
  THFILE++;
  char buf[100]; sprintf(buf,"all_thetas.%06d",THFILE);
  ofstream fout(buf);
  for (int c=0; c<coder_count; c++)
    fout << number_to_coder[c] << " " << theta[c] << " " << sem[c] << endl;
} // }}}

void calibrate_abs() { // {{{
  task_to_thetas.clear();  task_to_thetas.resize(task_count);
  task_to_results.clear(); task_to_results.resize(task_count);

  FOREACH(it,DATA) {
    task_to_thetas[ it->task ].push_back( theta[ it->coder ] );
    task_to_results[ it->task ].push_back( it->solved );
  }

  for (int t=0; t<task_count; t++) {
    vector<double> *T = &task_to_thetas[t];
    vector<bool> *C = &task_to_results[t];
    int N = T->size();

    double curra = A[t], currb = B[t];
    if (currb==0) {
      currb = get_initial_estimate_for_b(*T,*C);
      A[t]=curra; B[t]=currb;
      continue;
    }

    double STEP = 0.1;
    for (int loop=0; loop<100; loop++) {
      double grad[2];
      grad[0]=grad[1]=0;
      for (int i=0; i<N; i++) {
        double Z = exp(curra*(theta[i]-currb));
        double Zinv = 1/Z;
        if ((*T)[i] == MIN_THETA) continue;
        if ((*T)[i] == MAX_THETA) continue;
        if ((*C)[i]) {
          grad[0] += ((*T)[i]-currb) / (Z+1);
          grad[1] += (-curra) / (Z+1);
        } else {
          grad[0] += - ((*T)[i]-currb) / Z / Zinv / (Zinv+1);
          grad[1] += curra / Z / Zinv / (Zinv+1);
```

```
        }
      }
      double step[2];
      step[0]=-grad[0]; step[1]=-grad[1];
      while (STEP > 1e-7 && eval(*T,*C,curra-STEP*step[0],currb-STEP*step[1]) < eval(*T,*C,curra,currb))
STEP /= 2;
      curra -= STEP*step[0];
      currb -= STEP*step[1];
      if (abs(STEP*step[0]) < 1e-7 && abs(STEP*step[1]) < 1e-7) break;
      if (curra < MIN_A || currb < MIN_B || curra > MAX_A || currb > MAX_B) break;
    }

    if (curra < MIN_A+1e-5) curra=MIN_A;
    if (curra > MAX_A-1e-5) curra=MAX_A;
    if (currb < MIN_B+1e-5) currb=MIN_B;
    if (currb > MAX_B-1e-5) currb=MAX_B;
    A[t] = curra;
    B[t] = currb;
  }
  write_abs();
} // }}}

double cdf(double x, double mu, double sigma) { return 0.5*(1+erf((x-mu)/sigma/sqrt(2))); }

void calibrate_thetas() { // {{{
  coder_to_as.clear(); coder_to_as.resize(coder_count);
  coder_to_bs.clear(); coder_to_bs.resize(coder_count);
  coder_to_results.clear(); coder_to_results.resize(coder_count);

  FOREACH(it,DATA) {
    coder_to_as[ it->coder ].push_back( A[ it->task ] );
    coder_to_bs[ it->coder ].push_back( B[ it->task ] );
    coder_to_results[ it->coder ].push_back( it->solved );
  }

  for (int c=0; c<coder_count; c++) {
    vector<double> *A = &coder_to_as[c];
    vector<double> *B = &coder_to_bs[c];
    vector<bool> *CORR = &coder_to_results[c];
    int N = A->size();
    double currt = theta[c];

    double tlo = MIN_THETA, thi = MAX_THETA;
    while (thi-tlo > 1e-7) {
      double tm1=(tlo+tlo+thi)/3, tm2=(tlo+thi+thi)/3;
      double lp1=0.0, lp2=0.0;
      for (int i=0; i<N; i++) {
        if ((*A)[i] == MIN_A) continue;
        if ((*A)[i] == MAX_A) continue;
        if ((*B)[i] == MIN_B) continue;
        if ((*B)[i] == MAX_B) continue;
        double pr1 = prob_correct(tm1,(*A)[i],(*B)[i]);
        lp1 += log( (*CORR)[i] ? pr1 : 1-pr1 );
        double pr2 = prob_correct(tm2,(*A)[i],(*B)[i]);
        lp2 += log( (*CORR)[i] ? pr2 : 1-pr2 );
      }
      if (lp1 < lp2) tlo=tm1; else thi=tm2;
    }
    currt=tlo;
    if (abs(currt-MIN_THETA)<1e-5) currt=MIN_THETA;
    if (abs(currt-MAX_THETA)<1e-5) currt=MAX_THETA;
    theta[c] = currt;
    sem[c] = 0;
    for (int i=0; i<N; i++) {
      if ((*A)[i] == MIN_A) continue;
      if ((*A)[i] == MAX_A) continue;
      if ((*B)[i] == MIN_B) continue;
      if ((*B)[i] == MAX_B) continue;
      sem[c] += (*A)[i] * (*A)[i] * prob_correct(currt,(*A)[i],(*B)[i]) *
(1-prob_correct(currt,(*A)[i],(*B)[i]));
```

```
    }
    sem[c] = sqrt(sem[c]);
    sem[c] = 1/sem[c];
  }
  write_thetas();
} // }}}

vector< vector<double> > coder_to_cs, coder_to_ds, coder_to_es, coder_to_times, task_to_times;
void write_cds() { // {{{
  CDFILE++;
  char buf[100]; sprintf(buf,"all_cds.%06d",CDFILE);
  ofstream fout(buf);
  for(int t=0; t<task_count; t++)
    if (!isnan(C[t]))
      fout << number_to_task[t] << " " << C[t] << " " << D[t] << " " << E[t] << endl;
} // }}}
void calibrate_cds() { // {{{
  task_to_thetas.clear();  task_to_thetas.resize(task_count);
  task_to_times.clear(); task_to_times.resize(task_count);

  FOREACH(it,DATA) if (it->solved) {
    task_to_thetas[ it->task ].push_back( theta[ it->coder ] );
    task_to_times[ it->task ].push_back( log( it->time ) );
  }

  for(int t=0; t<task_count; t++) {
    vector<double> *X = &task_to_thetas[t];
    vector<double> *Y = &task_to_times[t];
    int N = X->size();

    double sumXX = 0.0; for (int i=0; i<N; i++) sumXX += (*X)[i]*(*X)[i];
    double sumXY = 0.0; for (int i=0; i<N; i++) sumXY += (*X)[i]*(*Y)[i];
    double sumX = 0.0; for (int i=0; i<N; i++) sumX += (*X)[i];
    double sumY = 0.0; for (int i=0; i<N; i++) sumY += (*Y)[i];

    double Cnum = sumXY - sumX * sumY / N;
    double Cden = sumXX - sumX * sumX / N;
    double CC = Cnum / Cden;

    double DD = sumY / N - CC * sumX / N;

    double meanError = 0.0; for (int i=0; i<N; i++) meanError +=
((*Y)[i]-CC*(*X)[i]-DD)*((*Y)[i]-CC*(*X)[i]-DD);
    meanError /= N;

    C[t]=CC; D[t]=DD; E[t]=meanError;
    cout << number_to_task[t] << " " << CC << " " << DD << " " << meanError << endl;
  }
  write_cds();
} // }}}


int main() {
  load_responses();
  cout << "load done" << endl;
  relabel_coders_and_tasks();
  cout << "relabel done" << endl;
  initialize_vars();
  cout << "init done" << endl;
  int loop=0;
  while (1) {
    loop++; if (loop>20) break;
    calibrate_abs();
    cout << "END AB CALIBRATION" << endl;
    calibrate_thetas();
    cout << "END TH CALIBRATION" << endl;
    calibrate_cds();
    cout << "END CD CALIBRATION" << endl;
  }
  return 0;
```

```
}
```

Code listing 2: Score Threshold algorithm implementation without using SEM

```
#include <algorithm>
#include <numeric>

#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <queue>
#include <set>
#include <map>

#include <cstdio>
#include <cstdlib>
#include <cctype>
#include <cassert>

#include <cmath>
#include <complex>
using namespace std;

#define FOREACH(it,c) for(__typeof((c).begin()) it=(c).begin();it!=(c).end();++it)
#define SIZE(t) ((int)((t).size()))
typedef pair<int,int> PII;

#define MAX_CODERS 2000
int max_score[3] = {250,500,1000};

#include <fstream>
vector<string> coder_id, handle;
map<string,int> local_id;
vector<double> A,B,C,D,E,theta,sem,task_weight;
vector<string> taskname;
vector<int> tasktype;
int easy_count, medium_count, hard_count;
vector<int> participants;

void load() {
  string s1,s2,s3; double d1,d2,d3,d4; int i1;

  map<string,int> tried, solved;
  ifstream fres("generated/responses");
  while (fres >> s1 >> s2 >> d1) { tried[s1]++; solved[s1]+=(d1>0); }

  ifstream fha("generated/coder_list"); while (fha >> s1 >> s2) { coder_id.push_back(s1);
handle.push_back(s2); local_id[s1] = SIZE(coder_id)-1; }
  cerr << "handles " << SIZE(handle) << endl;

  theta.resize(SIZE(handle));
  sem.resize(SIZE(handle));
  ifstream fth("generated/all_thetas"); while (fth >> s1 >> d1 >> d2) {
    if (local_id.count(s1)) {
      theta[local_id[s1]]=d1;
      sem[local_id[s1]]=d2;
    } else {
      cerr << "ERROR: adding coder " << s1 << " with theta " << d1 << " sem " << d2 << " -- not present
in coder_list" << endl;
      coder_id.push_back(s1); handle.push_back("UNKNOWN!"); local_id[s1] = SIZE(coder_id)-1;
      theta.push_back(d1);
      sem.push_back(d2);
    }
  }
  cerr << "thetas " << SIZE(theta) << endl;
  cerr << "handles again " << SIZE(handle) << endl;

  map<string,double> mapA, mapB;
  ifstream fab("generated/all_abs"); while (fab >> s1 >> d1 >> d2) { mapA[s1]=d1; mapB[s1]=d2; }
```

```
  cerr << "ab " << SIZE(mapA) << endl;

  map<string,double> mapC, mapD, mapE;
  ifstream fcd("generated/all_cds"); while (fcd >> s1 >> d1 >> d2 >> d3) { mapC[s1]=d1; mapD[s1]=d2;
mapE[s1]=d3; }
  cerr << "cd " << SIZE(mapC) << endl;

  easy_count = medium_count = hard_count = 0;
  ifstream fea("generated/easy_tasks");
  while (fea >> s1) {
    if (!mapA.count(s1)) { cerr << "ERROR: dropping easy task " << s1 << " -- no a,b" << endl; continue;
}
    if (!mapC.count(s1)) { cerr << "ERROR: dropping easy task " << s1 << " -- no c,d" << endl; continue;
}
    if (abs(mapA[s1])==10 && abs(mapB[s1])==10) { cerr << "ERROR: dropping easy task " << s1 << " with
a,b " << mapA[s1] << " " << mapB[s1] << " -- parameters too off" << endl; continue; }
    taskname.push_back(s1); tasktype.push_back(0); easy_count++;
  }
  cerr << "easy " << easy_count << endl;
  ifstream fme("generated/medium_tasks");
  while (fme >> s1) {
    if (!mapA.count(s1)) { cerr << "ERROR: dropping medium task " << s1 << " -- no a,b" << endl;
continue; }
    if (!mapC.count(s1)) { cerr << "ERROR: dropping medium task " << s1 << " -- no c,d" << endl;
continue; }
    if (abs(mapA[s1])==10 && abs(mapB[s1])==10) { cerr << "ERROR: dropping medium task " << s1 << " with
a,b " << mapA[s1] << " " << mapB[s1] << " -- parameters too off" << endl; continue; }
    taskname.push_back(s1); tasktype.push_back(1); medium_count++;
  }
  cerr << "medium " << medium_count << endl;
  ifstream fhr("generated/hard_tasks");
  while (fhr >> s1) {
    if (!mapA.count(s1)) { cerr << "ERROR: dropping hard task " << s1 << " -- no a,b" << endl; continue;
}
    if (!mapC.count(s1)) { cerr << "ERROR: dropping hard task " << s1 << " -- no c,d" << endl; continue;
}
    if (abs(mapA[s1])==10 && abs(mapB[s1])==10) { cerr << "ERROR: dropping hard task " << s1 << " with
a,b " << mapA[s1] << " " << mapB[s1] << " -- parameters too off" << endl; continue; }
    taskname.push_back(s1); tasktype.push_back(2); hard_count++;
  }
  cerr << "hard " << hard_count << endl;

  FOREACH(it,taskname) { A.push_back( mapA[*it] ); B.push_back( mapB[*it] ); }
  cerr << "ab again " << SIZE(A) << endl;

  FOREACH(it,taskname) { C.push_back( mapC[*it] ); D.push_back( mapD[*it] ); E.push_back( mapE[*it] ); }
  cerr << "cd again " << SIZE(C) << endl;

  ifstream fpa("human_input/participants_round_2");
  while (fpa >> s1) {
    if (local_id.count(s1)) {
      participants.push_back(local_id[s1]);
    } else cerr << "ERROR: dropping participant " << s1 << " handle " << s2 << " -- not present in
coder_list" << endl;
  }
  cerr << "participants " << SIZE(participants) << endl;

  for (int i=0; i<SIZE(participants); i++) {
    s1 = coder_id[participants[i]];
    int rounds = (2+tried[s1])/3;
    cerr << "INFO: coder " << handle[participants[i]] << " theta " << theta[participants[i]] << " sem "
<< sem[participants[i]] << " rounds" << rounds << endl;
    if (rounds < 5) {
      cerr << "WARNING: coder " << handle[participants[i]] << " with theta " << theta[participants[i]] <<
" only had " << rounds << " rounds" << endl;
    }
  }

  int current_timestamp;
  ifstream fcurr("generated/current_timestamp");
```

```
  fcurr >> current_timestamp;
  ifstream ftim("generated/task_timestamps");

  task_weight.clear(); task_weight.resize( SIZE(taskname) );
  for (int i=0; i<SIZE(taskname); i++) task_weight[i] = 1;
  cerr << "task weights done" << endl;

} // }}}

inline double get_probability(int coder, int task) {
  double th = theta[coder], a = A[task], b = B[task];
  double tmp = exp(a * (th-b) );
  double pr = tmp / (1+tmp);
  if (pr < 1e-7) pr = 1e-7;
  if (pr > 1-1e-7) pr = 1-1e-7;
  return pr;
}

inline double get_chance_of_reaching(double score, int coder, int task) {
  double mp = max_score[tasktype[task]];
  if (score < 0.3*mp + 1e-7) return 1.0;
  if (score > (1-1e-7)*mp) return 0.0;
  double ttime = 75 * 60 * 1000;
  double num = ttime*ttime*(mp - score);
  double den = 10*(score - 0.3*mp);
  double ctime = sqrt(num/den);
  double logctime = log(ctime);
  double mu = C[task]*theta[coder] + D[task];
  double sigma = E[task];
  return 0.5*(1 + erf( (logctime-mu) / (sigma*sqrt(2.)) ) );
}

double scoring_chance[3][MAX_CODERS][1024];
double prior_prob[MAX_CODERS];

void precompute_scoring_chances(int index, int coder, int task1, int task2, int task3) {
  for (int p=0; p<=250; p++) scoring_chance[0][index][p] = get_chance_of_reaching(p,coder,task1);
  for (int p=0; p<=500; p++) scoring_chance[1][index][p] = get_chance_of_reaching(p,coder,task2);
  for (int p=0; p<=1000; p++)scoring_chance[2][index][p] = get_chance_of_reaching(p,coder,task3);
}

double get_chance_of_reaching(int threshold, int index, int coder, int task1, int task2, int task3) {
  double result = 0.0;
  int t[3]; double p[3];

  t[0]=task1; p[0]=get_probability(coder,task1);
  t[1]=task2; p[1]=get_probability(coder,task2);
  t[2]=task3; p[2]=get_probability(coder,task3);

  for (int i=0; i<3; i++)
    if (threshold <= max_score[i]) {
      double prob = 1.0;
      for (int j=0; j<3; j++) if (j==i) prob *= p[j]; else prob *= 1-p[j];
      result += prob * scoring_chance[i][index][threshold];
    }
  for (int i=0; i<3; i++)
    for (int j=i+1; j<3; j++)
      if (threshold <= max_score[i]+max_score[j]) {
        double prob = 1.0;
        for (int k=0; k<3; k++) if (k==i || k==j) prob *= p[k]; else prob *= 1-p[k];
        for (int e=int(0.3*max_score[i]); e<=max_score[i]; e++) {
          double range = scoring_chance[i][index][e-1] - scoring_chance[i][index][e];
          if (threshold-e > max_score[j]) { result += 0; continue; }
          if (threshold-e < 0) { result += prob*range; continue; }
          result += prob * range * scoring_chance[j][index][threshold-e];
        }
      }
  {
    double toto = p[0]*p[1]*p[2];
    for (int e=75; e<=250; e++) {
```

```
      double range1 = scoring_chance[0][index][e-1] - scoring_chance[0][index][e];
      if (threshold-e < 0) result += toto*range1;
      for (int m=150; m<=500; m++) {
        double range2 = scoring_chance[1][index][m-1] - scoring_chance[1][index][m];

        if (threshold-e-m > 1000) { result += 0; continue; }
        if (threshold-e-m < 0) { result += toto*range1*range2; continue; }

        result += toto * range1 * range2 * scoring_chance[2][index][threshold];
      }
    }
  }
  if (result > 1) result = 1;
  return result;
}

double expected_advancer_count(int threshold, const vector<int> &round_participants, int task1, int
task2, int task3) {
  double sum = 0.0;
  for (int i=0; i<SIZE(round_participants); i++)
    sum += prior_prob[i] * get_chance_of_reaching(threshold,i,round_participants[i],task1,task2,task3);
  return sum;
}

vector<double> simulate_round(vector<int> round_participants, vector<double> prior_probabilities, int
advancer_count, int iterations=100) {
  vector<double> advancing_probability( SIZE(round_participants) );

  int counter = 0;

  double total_weight = 0;

  while (1) {
    int eee = rand() % easy_count;
    int mmm = easy_count + (rand() % medium_count);
    int hhh = easy_count + medium_count + (rand() % hard_count);

    if (counter == iterations) break;
    counter++;

    cerr << string(100,'-') << endl << "ROUND " << counter << endl;
    cerr << "easy: " << taskname[eee] << " difficulty " << A[eee] << " " << B[eee] << endl;
    cerr << "medium: " << taskname[mmm] << " difficulty " << A[mmm] << " " << B[mmm] << endl;
    cerr << "hard: " << taskname[hhh] << " difficulty " << A[hhh] << " " << B[hhh] << endl;

    for (int i=0; i<SIZE(round_participants); i++) precompute_scoring_chances( i, round_participants[i],
eee, mmm, hhh );
    cerr << "precomputed scoring chances" << endl;
    for (int i=0; i<SIZE(round_participants); i++) prior_prob[i] = prior_probabilities[i];

    int lo = 1, hi = 100;
    if (expected_advancer_count(lo,round_participants,eee,mmm,hhh) < advancer_count) {
      hi=2;
    } else {
      while (expected_advancer_count(hi,round_participants,eee,mmm,hhh) >= advancer_count) hi+=100;
    }
    while (hi-lo>1) {
      int med=(hi+lo)/2;
      if (expected_advancer_count(med,round_participants,eee,mmm,hhh) < advancer_count)
        hi=med; else lo=med;
    }
    cerr << "EXPECTED THRESHOLD TO ADVANCE: " << lo << endl;

    double this_weight = task_weight[eee] + task_weight[mmm] + task_weight[hhh];
    this_weight = 1;
    cerr << "RESULT WEIGHT: " << this_weight << endl;
    total_weight += this_weight;
    for (int i=0; i<SIZE(round_participants); i++)
      advancing_probability[i] += this_weight * prior_probabilities[i] *
get_chance_of_reaching(lo,i,round_participants[i],eee,mmm,hhh);
```

```
  }

  for (int i=0; i<SIZE(round_participants); i++) {
    advancing_probability[i] /= total_weight;
    cout << handle[round_participants[i]] << " " << advancing_probability[i] << endl;
  }
  return advancing_probability;

}

int main() {
  srand(time(NULL));
  load();
  cerr << "load done" << endl;

  int ROUNDS = 500;

  vector<double> alive( SIZE(participants) );
  for (int i=0; i<SIZE(participants); i++) alive[i] = 1.0;

  vector<int> secpart, key;
  vector<double> prior, post;

  post = simulate_round(participants,alive,300,ROUNDS);
  return 0;
}
```

Code listing 3: Score Threshold algorithm implementation using SEM

```
#include <algorithm>
#include <numeric>

#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <queue>
#include <set>
#include <map>

#include <cstdio>
#include <cstdlib>
#include <cctype>
#include <cassert>

#include <cmath>
#include <complex>
using namespace std;

#define FOREACH(it,c) for(__typeof((c).begin()) it=(c).begin();it!=(c).end();++it)
#define SIZE(t) ((int)((t).size()))
typedef pair<int,int> PII;

#define MAX_CODERS 2000
int max_score[3] = {250,500,1000};

#include <fstream>
vector<string> coder_id, handle;
map<string,int> local_id;
vector<double> A,B,C,D,E,theta,sem,task_weight;
vector<string> taskname;
vector<int> tasktype;
int easy_count, medium_count, hard_count;
vector<int> participants;

void load() {
  string s1,s2,s3; double d1,d2,d3;

  map<string,int> tried, solved;
```

```
  ifstream fres("generated/responses");
  while (fres >> s1 >> s2 >> d1) { tried[s1]++; solved[s1]+=(d1>0); }

  ifstream fha("generated/coder_list"); while (fha >> s1 >> s2) { coder_id.push_back(s1);
handle.push_back(s2); local_id[s1] = SIZE(coder_id)-1; }
  cerr << "handles " << SIZE(handle) << endl;

  theta.resize(SIZE(handle));
  sem.resize(SIZE(handle));
  ifstream fth("generated/all_thetas"); while (fth >> s1 >> d1 >> d2) {
    if (local_id.count(s1)) {
      theta[local_id[s1]]=d1;
      sem[local_id[s1]]=d2;
    } else {
      cerr << "ERROR: adding coder " << s1 << " with theta " << d1 << " sem " << d2 << " -- not present
in coder_list" << endl;
      coder_id.push_back(s1); handle.push_back("UNKNOWN!"); local_id[s1] = SIZE(coder_id)-1;
      theta.push_back(d1);
      sem.push_back(d2);
    }
  }
  cerr << "thetas " << SIZE(theta) << endl;
  cerr << "handles again " << SIZE(handle) << endl;

  map<string,double> mapA, mapB;
  ifstream fab("generated/all_abs"); while (fab >> s1 >> d1 >> d2) { mapA[s1]=d1; mapB[s1]=d2; }
  cerr << "ab " << SIZE(mapA) << endl;

  map<string,double> mapC, mapD, mapE;
  ifstream fcd("generated/all_cds"); while (fcd >> s1 >> d1 >> d2 >> d3) { mapC[s1]=d1; mapD[s1]=d2;
mapE[s1]=d3; }
  cerr << "cd " << SIZE(mapC) << endl;

  easy_count = medium_count = hard_count = 0;
  ifstream fea("generated/easy_tasks");
  while (fea >> s1) {
    if (!mapA.count(s1)) { cerr << "ERROR: dropping easy task " << s1 << " -- no a,b" << endl; continue;
}
    if (!mapC.count(s1)) { cerr << "ERROR: dropping easy task " << s1 << " -- no c,d" << endl; continue;
}
    if (abs(mapA[s1])==10 && abs(mapB[s1])==10) { cerr << "ERROR: dropping easy task " << s1 << " with
a,b " << mapA[s1] << " " << mapB[s1] << " -- parameters too off" << endl; continue; }
    taskname.push_back(s1); tasktype.push_back(0); easy_count++;
  }
  cerr << "easy " << easy_count << endl;
  ifstream fme("generated/medium_tasks");
  while (fme >> s1) {
    if (!mapA.count(s1)) { cerr << "ERROR: dropping medium task " << s1 << " -- no a,b" << endl;
continue; }
    if (!mapC.count(s1)) { cerr << "ERROR: dropping medium task " << s1 << " -- no c,d" << endl;
continue; }
    if (abs(mapA[s1])==10 && abs(mapB[s1])==10) { cerr << "ERROR: dropping medium task " << s1 << " with
a,b " << mapA[s1] << " " << mapB[s1] << " -- parameters too off" << endl; continue; }
    taskname.push_back(s1); tasktype.push_back(1); medium_count++;
  }
  cerr << "medium " << medium_count << endl;
  ifstream fhr("generated/hard_tasks");
  while (fhr >> s1) {
    if (!mapA.count(s1)) { cerr << "ERROR: dropping hard task " << s1 << " -- no a,b" << endl; continue;
}
    if (!mapC.count(s1)) { cerr << "ERROR: dropping hard task " << s1 << " -- no c,d" << endl; continue;
}
    if (abs(mapA[s1])==10 && abs(mapB[s1])==10) { cerr << "ERROR: dropping hard task " << s1 << " with
a,b " << mapA[s1] << " " << mapB[s1] << " -- parameters too off" << endl; continue; }
    taskname.push_back(s1); tasktype.push_back(2); hard_count++;
  }
  cerr << "hard " << hard_count << endl;

  FOREACH(it,taskname) { A.push_back( mapA[*it] ); B.push_back( mapB[*it] ); }
  cerr << "ab again " << SIZE(A) << endl;
```

```
  FOREACH(it,taskname) { C.push_back( mapC[*it] ); D.push_back( mapD[*it] ); E.push_back( mapE[*it] ); }
  cerr << "cd again " << SIZE(C) << endl;

  ifstream fpa("human_input/participants_round_2");
  while (fpa >> s1) {
    if (local_id.count(s1)) {
      participants.push_back(local_id[s1]);
    } else cerr << "ERROR: dropping participant " << s1 << " handle " << s2 << " -- not present in
coder_list" << endl;
  }
  cerr << "participants " << SIZE(participants) << endl;

  for (int i=0; i<SIZE(participants); i++) {
    s1 = coder_id[participants[i]];
    int rounds = (2+tried[s1])/3;
    cerr << "INFO: coder " << handle[participants[i]] << " theta " << theta[participants[i]] << " sem "
<< sem[participants[i]] << " rounds " << rounds << endl;
    if (rounds < 5) {
      cerr << "WARNING: coder " << handle[participants[i]] << " with theta " << theta[participants[i]] <<
" sem " << sem[participants[i]] << " only had " << rounds << " rounds " << endl;
    }
    if (theta[participants[i]]<-4) {
      theta[participants[i]]=-4; sem[participants[i]]=1.5;
      cerr << "WARNING: coder " << handle[participants[i]] << " IS A LOSER and got a new theta " <<
theta[participants[i]] << " sem " << sem[participants[i]] << endl;
    }
  }

  int current_timestamp;
  ifstream fcurr("generated/current_timestamp");
  fcurr >> current_timestamp;
  ifstream ftim("generated/task_timestamps");

  task_weight.clear(); task_weight.resize( SIZE(taskname) );
  for (int i=0; i<SIZE(taskname); i++) task_weight[i] = 1;
  cerr << "task weights done" << endl;

} // }}}

inline double get_probability(int coder, int task) {
  double th = theta[coder], a = A[task], b = B[task];
  double tmp = exp(a * (th-b) );
  double pr = tmp / (1+tmp);
  if (pr < 1e-7) pr = 1e-7;
  if (pr > 1-1e-7) pr = 1-1e-7;
  return pr;
}

inline double get_probability_theta(double th, int task) {
  double a = A[task], b = B[task];
  double tmp = exp(a * (th-b) );
  double pr = tmp / (1+tmp);
  if (pr < 1e-7) pr = 1e-7;
  if (pr > 1-1e-7) pr = 1-1e-7;
  return pr;
}

inline double get_chance_of_reaching(double score, double th, int task) {
  double mp = max_score[tasktype[task]];
  if (score < 0.3*mp + 1e-7) return 1.0;
  if (score > (1-1e-7)*mp) return 0.0;
  double ttime = 75 * 60 * 1000;
  double num = ttime*ttime*(mp - score);
  double den = 10*(score - 0.3*mp);
  double ctime = sqrt(num/den);
  double logctime = log(ctime);
  double mu = C[task]*th + D[task];
  double sigma = E[task];
  return 0.5*(1 + erf( (logctime-mu) / (sigma*sqrt(2.)) ) );
```

```
}

double prior_prob[MAX_CODERS];

#define PREC_RATINGMIN -5
#define PREC_RATINGSTEP 0.01
#define PREC_RATINGSTEPCOUNT 1001
double prec_scoring_chance[3][PREC_RATINGSTEPCOUNT][1001];

void precompute_scoring_chances(int task1, int task2, int task3) {
  double th=PREC_RATINGMIN;
  for (int step=0; step<PREC_RATINGSTEPCOUNT; step++) {
    for (int p=0; p<=250; p++)  prec_scoring_chance[0][step][p] = get_chance_of_reaching(p,th,task1);
    for (int p=0; p<=500; p++)  prec_scoring_chance[1][step][p] = get_chance_of_reaching(p,th,task2);
    for (int p=0; p<=1000; p++) prec_scoring_chance[2][step][p] = get_chance_of_reaching(p,th,task3);
    th += PREC_RATINGSTEP;
  }
}

double get_chance_of_reaching(int threshold, int coder, int task1, int task2, int task3) {
  int t[3]; double p[3];

  t[0]=task1;
  t[1]=task2;
  t[2]=task3;

  double coder_mu = theta[coder], coder_sigma = sem[coder];

  double global_result = 0.0;

  double th=PREC_RATINGMIN;
  for (int step=0; step<PREC_RATINGSTEPCOUNT; step++) {
    double result = 0.0;

    // compute the probability that the coder's rating is in the current interval
    double left_bound = th - 0.5*PREC_RATINGSTEP;
    double right_bound = th + 0.5*PREC_RATINGSTEP;
    if (step==0) left_bound = -20;
    if (step==PREC_RATINGSTEPCOUNT-1) right_bound = 20;

    double prob_here
              = 0.5*(1 + erf( (right_bound-coder_mu) / (coder_sigma*sqrt(2.)) ) )
              - 0.5*(1 + erf( (left_bound-coder_mu) / (coder_sigma*sqrt(2.)) ) );

    if (prob_here < 1e-6) { th+=PREC_RATINGSTEP; continue; }
    int SCORE_STEP = 5;
    if (prob_here < 0.0001) SCORE_STEP = 5;
    if (prob_here < 0.00001) SCORE_STEP = 25;

    p[0]=get_probability_theta(th,task1);
    p[1]=get_probability_theta(th,task2);
    p[2]=get_probability_theta(th,task3);

    for (int i=0; i<3; i++)
      if (threshold <= max_score[i]) {
        double prob = 1.0;
        for (int j=0; j<3; j++) if (j==i) prob *= p[j]; else prob *= 1-p[j];
        result += prob * prec_scoring_chance[i][step][threshold];
      }
    for (int i=0; i<3; i++)
      for (int j=i+1; j<3; j++)
        if (threshold <= max_score[i]+max_score[j]) {
          double prob = 1.0;
          for (int k=0; k<3; k++) if (k==i || k==j) prob *= p[k]; else prob *= 1-p[k];
          for (int e=int(0.3*max_score[i]); e<=max_score[i]; e+=SCORE_STEP) {
            double range = prec_scoring_chance[i][step][e-SCORE_STEP] - prec_scoring_chance[i][step][e];
            if (threshold-e > max_score[j]) { result += 0; continue; }
            if (threshold-e < 0) {
              result += prob*range;
              continue;
```

```
            }
            result += prob * range * prec_scoring_chance[j][step][threshold-e];
          }
        }
    {
      double toto = p[0]*p[1]*p[2];
      for (int e=75; e<=250; e+=SCORE_STEP) {
        double range1 = prec_scoring_chance[0][step][e-SCORE_STEP] - prec_scoring_chance[0][step][e];
        if (threshold-e < 0) {
          result += toto*range1;
          continue;
        }
        for (int m=150; m<=500; m+=SCORE_STEP) {
          double range2 = prec_scoring_chance[1][step][m-SCORE_STEP] - prec_scoring_chance[1][step][m];

          if (threshold-e-m > 1000) { result += 0; continue; }
          if (threshold-e-m < 0) { result += toto*range1*range2; continue; }

          result += toto * range1 * range2 * prec_scoring_chance[2][step][threshold];
        }
      }
    }
    if (result > 1) result = 1;
    global_result += prob_here * result;


    th += PREC_RATINGSTEP;
  }

  if (global_result > 1) global_result = 1;
  return global_result;
}

double expected_advancer_count(int threshold, const vector<int> &round_participants, int task1, int
task2, int task3) {
  cerr << "trying threshold: " << threshold << endl;
  double sum = 0.0;
  for (int i=0; i<SIZE(round_participants); i++)
    sum += prior_prob[i] * get_chance_of_reaching(threshold,round_participants[i],task1,task2,task3);
  cerr << "threshold: " << threshold << " returns " << sum << endl;
  return sum;
}

vector<double> simulate_round(vector<int> round_participants, vector<double> prior_probabilities, int
advancer_count, int iterations=100) {
  vector<double> advancing_probability( SIZE(round_participants) );

  int counter = 0;

  double total_weight = 0;

  while (1) {
    int eee = rand() % easy_count;
    int mmm = easy_count + (rand() % medium_count);
    int hhh = easy_count + medium_count + (rand() % hard_count);

    if (counter == iterations) break;
    counter++;

    cerr << string(100,'-') << endl << "ROUND " << counter << endl;
    cerr << "easy: " << taskname[eee] << " difficulty " << A[eee] << " " << B[eee] << endl;
    cerr << "medium: " << taskname[mmm] << " difficulty " << A[mmm] << " " << B[mmm] << endl;
    cerr << "hard: " << taskname[hhh] << " difficulty " << A[hhh] << " " << B[hhh] << endl;

    precompute_scoring_chances(eee,mmm,hhh);
    cerr << "precomputed scoring chances" << endl;

    for (int i=0; i<SIZE(round_participants); i++) prior_prob[i] = prior_probabilities[i];

    int lo = 1, hi = 100;
```

```
    if (expected_advancer_count(lo,round_participants,eee,mmm,hhh) < advancer_count) {
      hi=2;
    } else {
      while (expected_advancer_count(hi,round_participants,eee,mmm,hhh) >= advancer_count) hi+=100;
    }
    while (hi-lo>1) {
      int med=(hi+lo)/2;
      if (expected_advancer_count(med,round_participants,eee,mmm,hhh) < advancer_count)
        hi=med; else lo=med;
    }
    cerr << "EXPECTED THRESHOLD TO ADVANCE: " << lo << endl;

    double sum_prob = 0;
    for (int i=0; i<SIZE(round_participants); i++) sum_prob +=
get_chance_of_reaching(lo,round_participants[i],eee,mmm,hhh);
    cerr << "sum_prob = " << sum_prob << endl;
    double factor = 1;
    if (lo>1) factor = advancer_count / sum_prob;

    double this_weight = task_weight[eee] + task_weight[mmm] + task_weight[hhh];
    this_weight = 1;
    cerr << "RESULT WEIGHT: " << this_weight << endl;
    total_weight += this_weight;
    for (int i=0; i<SIZE(round_participants); i++)
      advancing_probability[i] +=
        factor * this_weight * prior_probabilities[i] *
get_chance_of_reaching(lo,round_participants[i],eee,mmm,hhh);
  }

  for (int i=0; i<SIZE(round_participants); i++) {
    advancing_probability[i] /= total_weight;
    cout << handle[round_participants[i]] << " " << advancing_probability[i] << endl;
  }
  return advancing_probability;

}

int main() {
  srand(42);
  load();
  cerr << "load done" << endl;

  int ROUNDS = 3;

  vector<double> alive( SIZE(participants) );
  for (int i=0; i<SIZE(participants); i++) alive[i] = 1.0;

  vector<int> secpart, key;
  vector<double> prior, post;

  post = simulate_round(participants,alive,300,ROUNDS);
  return 0;
}
```

Code listing #4: Greedy attack on the Elo rating system

```
#!/usr/bin/python
import sys

def get_k(rating):
  k = 32
  if rating >= 2100: k = 24
  if rating >= 2400: k = 16
  return k

rating = {}
```

```
N = 7
for i in range(N): rating[i]=1000
GAMES = 0

def win(winner,loser,times):
  global GAMES
  GAMES += times
  e_winner = times * 1. / ( 1 + 10**( (rating[loser]-rating[winner])/400 ) )
  e_loser = times * 1. / ( 1 + 10**( (rating[winner]-rating[loser])/400 ) )
  r_winner = times
  r_loser = 0
  k_a, k_b = get_k(rating[winner]), get_k(rating[loser])
  rating[winner] += k_a * ( r_winner - e_winner )
  rating[loser] += k_b * ( r_loser - e_loser )
  if rating[loser]<0: rating[loser]=0

while rating[0]<2400:
  best, bi, bj = 987654321, -1, -1
  for j in range(N):
    for i in range(j):
      if rating[i]-rating[j] < best:
        best, bi, bj = rating[i]-rating[j], i, j
  print "bi %d bj %d best %d" % (bi,bj,best)
  win(bi,bj,1)

print GAMES,rating
```

Bibliography

[ABL] S. E. Ashoo, T. Boudreau, and D. A. Lane, *Programming Contest Control System PC^2.*, `http://www.ecs.csus.edu/pc2/`.

[ACMa] *ACM ICPC Rules*, `http://icpc.baylor.edu/icpc/info/default.htm`.

[ACMb] *ACM International Collegiate Programming Contest*, `http://icpc.baylor.edu/`.

[AH97] K. Appel and W. Haken, *Solution of the Four Color Map Problem*, Scientific American **237** (1997), 108–121.

[Aic] Bernhard K. Aichernig, *Test-case generation as a refinement problem.*

[Aic01a] ———, *Systematic black-box testing of computer-based systems through formal abstraction techniques*, Ph.D. thesis, Institute for Software Technology, TU Graz, Austria, January 2001, Supervisor: Peter Lucas.

[Aic01b] ———, *Test-case calculation through abstraction*, Lecture Notes in Computer Science **2021** (2001), 571+.

[AM07] Ricardo de Oliveira Anido and Raphael Marcos Menderico, *Brazilian Olympiad in Informatics*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Ans73] F. J. Anscombe, *Graphs in Statistical Analysis*, American Statistician (1973), 17–21.

[ATP08] ATP – Association of Tennis Professionals, *ATP Ranking and Race Frequently Asked Questions*, 2008, `http://www.atptennis.com/en/players/information/rankfaq.asp`.

[Bac78] Ralph-Johan Back, *On the correctness of refinement steps in program development*, Ph.D. thesis Report A-1978-4, Department of Computer Science, University of Helsinki, Helsinki, Finland, Oct 1978.

[Bac80] ———, *Correctness preserving program refinements: Proof theory and applications*, Mathematical Center Tracts, vol. 131, Mathematical Centre, Amsterdam, The Netherlands, 1980.

[BB] Sergey Bochkanov and Vladimir Bystritsky, *Alglib.net project.*

[BCG82] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for your mathematical plays*, Academic Press, New York, 1982.

[Bei90] Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1990.

[Bei95] ———, *Black-box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, 1995.

[BJ80] Anil K. Bera and Carlos M. Jarque, *Efficient tests for normality, homoscedasticity and serial independence of regression residuals*, Economics Letters **6** (1980), 255–259.

[BJ81] ———, *Efficient tests for normality, homoscedasticity and serial independence of regression residuals: Monte Carlo evidence*, Economics Letters **7** (1981), 313–318.

[BK04] Frank B. Baker and Seock-Ho Kim, *Item Response Theory: Parameter Estimation Techniques*, 2 ed., CRC, 2004.

[Bla03] M. Blaze, *Cryptology and Physical Security: Rights Amplification in Master-Keyed Mechanical Locks.*, IEEE Security and Privacy (2003).

[BM58] G. E. P. Box and Mervin E. Muller, *A Note on the Generation of Random Normal Deviates*, The Annals of Mathematical Statistics **29** (1958), 610–611.

[BP06] Raewyn Boersen and Margot Phillipps, *Programming Contests: Two innovative models from New Zealand*, Presented at Perspectives on Computer Science Competitions for (High School) Students, 2006. `http://bwinf.de/competition-workshop/RevisedPapers/1_BoersenPhillipps_rev.pdf`.

[Bro07] Predrag Brodanac, *Regular competitions in Croatia*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[BvW90] Ralph-Johan Back and Joakim von Wright, *Refinement calculus i: Sequential nondeterministic programs*, Stepwise Refinement of Distributed Systems (Mook, The Netherlands) (J. W. deBakker, W. P. deRoever, and G. Rozenberg, eds.), Lecture Notes in Computer Science, vol. 430, Springer-Verlag, May 1990, pp. 42–66.

[BvW98] ———, *Refinement calculus: A systematic introduction*, Springer-Verlag, 1998, Graduate Texts in Computer Science.

[BvW99] ———, *Structured derivations: a method for doing high-school mathematics carefully*, TUCS Technical Report 246, TUCS - Turku Centre for Computer Science, Turku, Finland, Mar 1999.

[BvW05] ———, *Mathematics with a little bit of logic: Structured derivations in high-school mathematics*, Manuscript, 2005.

[CC05] *Common Criteria for Information Technology Security Evaluation*, 2005, Version 2.3. ISO-IEC standard 15408. `http://www.commoncriteriaportal.org/`.

[CCa05] *Common Criteria Part 1: Introduction and General Model*, 2005, Version 2.3. `http://www.commoncriteria.org/docs/PDF/CCPART1V21.PDF`.

[CCb05] *Common Criteria Part 2: Security Functional Requirements*, 2005, Version 2.3. `http://www.commoncriteria.org/docs/PDF/CCPART2V21.PDF`.

[CCc05] *Common Criteria Part 3: Security Assurance Requirements*, 2005, Version 2.3. `http://www.commoncriteria.org/docs/PDF/CCPART3V21.PDF`.

[CCO] *Common Criteria – An Introduction.*, `http://www.commoncriteriaportal.org/public/files/ccintroduction.pdf`.

[Cen] CERT Coordination Center, *Denial of Service Attacks*, `http://www.cert.org/tech_tips/denial_of_service.html`.

[CFdV07] Giorgio Casadei, Bruno Fadini, and Marta Genoviè de Vita, *Italian Olympiad in Informatics*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[CLR89] Thomas Cormen, Charles Leiserson, and Ronald Rivest, *Introduction to algorithms*, MIT Press, 1989.

[CLRS01] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Introduction to algorithms*, MIT Press, 2001.

[CMVK06] Gordon Cormack, Ian Munro, Troy Vasiga, and Graeme Kemkes, *Structure, Scoring and Purpose of Computing Competitions*, Informatics in Education **5** (2006), 15–36.

[Cor06] Gordon Cormack, *Random Factors in IOI 2005 Test Case Scoring*, Informatics in Education **5** (2006), 5–14.

[CUD07] Lhaichin Choijoovanchig, Sambuu Uyanga, and Mendee Dashnyam, *The Informatics Olympiad in Mongolia*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Dag06] Valentina Dagiene, *Information Technology Contests - Introduction to Computer Science in an Attractive Way*, Informatics in Education **5** (2006), 37–46.

[Daw] Bruce Dawson, *Comparing floating point numbers*, `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`.

[dB46] N. G. de Bruijn, *A Combinatorial Problem*, Koninklijke Nederlandse Akademie v. Wetenschappen **49** (1946), 758–764.

[DHMG08a] Pierre Dangauthier, Ralf Herbrich, Tom Minka, and Thore Graepel, *TrueSkill Through Time: Revisiting the History of Chess*, Advances in Neural Information Processing Systems, vol. 20, 2008, pp. 931–938.

[DHMG08b] ———, *TrueSkill Through Time: Revisiting the History of Chess*, Advances in Neural Information Processing Systems 20 (J.C. Platt, D. Koller, Y. Singer, and S. Roweis, eds.), MIT Press, Cambridge, MA, 2008, pp. 337–344.

[Dij76] Edsger Wybe Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[DKS07] Krzysztof Diks, Marcin Kubica, and Krzysztof Stencel, *Polish Olympiad in Informatics – 14 Years of Experience*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[DS07] Valentina Dagiene and Jurate Skupiene, *Contests in Programming: Quarter Century of Lithuanian Experience*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Elo78] Arpad Elo, *The rating of chessplayers, past and present*, Arco Publishing, 1978.

[FC06] Maryanne Fisher and Anthony Cox, *Gender and Programming Contests: Mitigating Exclusionary Practices*, Informatics in Education **5** (2006), 47–62.

[FC08] Bryan Ford and Russ Cox, *Vx32: Lightweight User-level Sandboxing on the x86*, Proceedings of the 2008 USENIX Annual Technical Conference, 2008, `http://pdos.csail.mit.edu/~baford/vm/`, p. 293–306.

[FIA08] FIA – Fédération Internationale de l'Automobile, *Formula One Regulations*, 2008, `http://www.fia.com/sport/Regulations/f1regs.html`.

[FLS07] Pere J. Ferrando and Urbano Lorenzo-Seva, *An Item Response Theory Model for Incorporating Response Time Data in Binary Personality Items*, Applied Psychological Measurement **31** (2007), 525–543.

[For04] Michal Forišek, *On suitability of tasks for the IOI competition*, Personal communication to the IOI General Assembly.

[For05] ———, *Representation of Integers and Reals*, Published by TopCoder, Inc. `http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=integersReals`.

[For06a] ———, *On the Suitability of Programming Tasks for Automated Evaluation*, Informatics in Education **5** (2006), 63–76.

[For06b] ———, *Security of Programming Contest Systems*, Information Technologies at School (Valentina Dagiene and Roland Mittermeir, eds.), 2006, pp. 553–563.

[For07] ———, *Slovak IOI 2007 Team Selection and Preparation*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[For09a] ———, *Using Item Response Theory To Rate (Not Only) Programmers*, Olympiads in Informatics (to be published) (Valentina Dagiene, ed.), vol. 1, 2009.

[For09b] ———, *Vyhodnotenie reliability hodnotenia Olympiády v Informatike*, Proceedings of conference DidInfo 2009 (Branislav Rovan, ed.), 2009.

[FW06] Michal Forišek and Michal Winczer, *Non-formal Activities as Scaffolding to Informatics Achievement*, Information Technologies at School (Valentina Dagiene and Roland Mittermeir, eds.), 2006, pp. 529–534.

[GGT] *Girls Go Tech*, `http://www.girlsgotech.org/`.

[Gle08] Jim Gleason, *An Evaluation of Mathematics Competitions Using Item Response Theory*, Notices of the ACM **55** (2008).

[Glia] Mark E. Glickman, *Example of the Glicko-2 system.*

[Glib] ———, *The Glicko system.*

[Gli93] ———, *Paired comparison models with time varying parameters*, Ph.D. thesis, Harvard University Dept of Statistics, 1993.

[Gli95] ———, *A Comprehensive Guide to Chess Ratings*, American Chess Journal **3** (1995), 59–102.

[Gli99] ———, *Parameter estimation in large dynamic paired comparison experiments*, Applied Statistics **48** (1999), 377–394.

[Gol91] David Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comput. Surv. **23** (1991), no. 1, 5–48.

[Har67] Kenneth Harkness, *Official chess handbook*, David McKay Company, 1967.

[Hei02] Dan Heisman, *A Parent's Guide to Chess*, `http://www.chesscafe.com/text/skittles176.pdf`.

[HG06] Ralf Herbrich and Thore Graepel, *TrueSkill^{TM}: A Bayesian Skill Rating System*, Tech. report, 2006, MSR-TR-2006-80.

[HHK01] Eric Horvitz, David Hovel, and Carl Kadie, *MSBNx: A Component-Centric Toolkit for Modeling and Inference with Bayesian Networks*, MSR-TR-2001-67.

[HMG07] Ralf Herbrich, Tom Minka, and Thore Graepel, *TrueSkill(TM): A Bayesian Skill Rating System*, Advances in Neural Information Processing Systems, vol. 20, 2007, pp. 569–576.

[Hro05] Juraj Hromkovič, *Design And Analysis of Randomized Algorithms*, Springer-Verlag, 2005.

[HU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[HVa] Gyula Horvath and Tom Verhoeff, *Finding the Median under IOI Conditions*, Informatics in Education **1**, 73–92.

[HVb] ———, *Numerical Difficulties in Pre-University Informatics Education and Competitions*, Informatics in Education **2**, 21–38.

[IdFC96] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, *Lua – an extensible extension language*, Software: Practice & Experience **26** (1996), 635–652.

[IEEa] *Wikipedia entry: IEEE 754r*, `http://en.wikipedia.org/wiki/IEEE_754r`.

[IEEb] *Wikipedia entry: IEEE floating-point standard*, `http://en.wikipedia.org/wiki/IEEE_floating-point_standard`.

[IMD] *The Internet Movie Database*, `http://www.imdb.com/`.

[Ins85] Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)*, `http://grouper.ieee.org/groups/754/`.

[IOI] *International Olympiad in Informatics*, `http://ioinformatics.org/`.

[IOI05] IOI Organizing Committee, *IOI Regulations*, 2005, `http://ioinformatics.org/regulations.htm`.

[IPS] *Internet Problem Solving Contest*, `http://ipsc.ksp.sk/`.

[JP07] Metodija Janceski and Veno Pacovski, *Olympiads in Informatics: Macedonian Experience, Needs, Challenges*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Kah01] William Kahan, *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*, `http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`.

[Kah05] ———, *How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?*, Presented at Householder Symposium XVI, 2005. `http://www.cs.berkeley.edu/~wkahan/Mindless.pdf`.

[KCMV07] Graeme Kemkes, Gordon Cormack, Ian Munro, and Troy Vasiga, *New Task Types at the Canadian Computing Competition*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Kir07] Vladimir M. Kiryukhin, *The Modern Contents of the Russian Olympiad in Informatics*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt, *Fast pattern matching in strings*, SIAM Journal on Computing **6** (1977), 323–350.

[Kou04] Vladimír Koutný, *Testovací systém pre programátorské sútaže (in Slovak, English title: A testing system for programming contests).*, Master's thesis, 2004, `http://people.ksp.sk/~vlado/cessd2/CESSd2.tar.gz`.

[KP07] Rob Kolstad and Don Piele, *USA Computing Olympiad (USACO)*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[KR87] R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development **31** (1987), 249–260.

[KSP] *Correspondence Seminar in Programming*, `http://ksp.sk/`.

[KVC06] Graeme Kemkes, Troy Vasiga, and Gordon V. Cormack, *Objective Scoring for Computing Competition Tasks*, Information Technologies at School (Valentina Dagiene and Roland Mittermeir, eds.), 2006.

[Lab] UIUC IRT Modeling Lab, *Tutorial on item response theory.*

[Lam73] B. W. Lampson, *A Note on the Confinement Problem*, Communications of the ACM **16** (1973), 613–615.

[Man06a] Shahriar Manzoor, *Analyzing Programming Contest Statistics*, Presented at Perspectives on Computer Science Competitions for (High School) Students, 2006. `http://bwinf.de/competition-workshop/RevisedPapers/13_Manzoor_rev-better.pdf`.

[Man06b] ———, *Analyzing Programming Contest Statistics*, Presented at Perspectives on Computer Science Competitions for (High School) Students.

[Mar07] Martin Mares, *Perspectives on Grading Systems*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Mar08] ———, *MO-Eval: The MO Contest Environment*, 2008, http://mj.ucw.cz/mo-eval/.

[Mic06] Marcin Michalski et al., *SIO.NET Plug&Play Contest System.*, Presented at Perspectives on Computer Science Competitions for (High School) Students, 2006. http://bwinf.de/competition-workshop/RevisedPapers/16_Michalski+_rev.pdf.

[MKK07] Krasimir Manev, Emil Kelevediev, and Stoyan Kapralov, *Programming Contests for School Students in Bulgaria*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Moo] *Mooshak – a system for managing programming contests on the Web*, http://www.dcc.fc.up.pt/projectos/mooshak/.

[Mor87] J. Morris, *A Theoretical Basis for Stepwise Refinement and the Programming Calculus*, Science of Computer Programming **9** (1987), 187–306.

[MRG88] C. C. Morgan, K. A. Robinson, and P. H. B. Gardiner, *On the Refinement Calculus*, Tech. report, Programming Research Group, 1988, Technical report PRG-70.

[Mye79] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.

[Nun05] John Nunn, *The Nunn Plan for the World Chess Championship* , 2005, http://www.chessbase.com/newsdetail.asp?newsid=2440.

[OI] *Slovak Olympiad in Informatics*, http://oi.sk/.

[Opm06] Martins Opmanis, *Some Ways to Improve Olympiads in Informatics*, Informatics in Education **5** (2006), 113–124.

[oPSI08] Art of Problem Solving Incorporated, *AoPS For The Win!*, 2008, http://www.artofproblemsolving.com/Edutainment/g1/index.php.

[Par04] Ivailo Partchev, *A visual guide to item response theory.*

[Pet] Peter Košinár and Vladimír Koutný and Richard Kralovič, *A linux kernel patch to have exact processor time measurement*, http://people.ksp.sk/~misof/ioi/timepatch-2.4.25.

[PF] Xu Pengcheng and Ying Fuchen, *Peking University Judge Online*, http://162.105.81.201/soft/JudgeOnlineSetup.rar.

[PO07] Pavel S. Pankov and Timur R. Oruskulov, *Tasks at Kyrgyzstani Olympiads in Informatics: Experience and Proposals*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Poh06] Wolfgang Pohl, *Computer Science Contests for Secondary School Students: Approaches to Classification*, Informatics in Education **5** (2006), 125–132.

[Poh07] ———, *Computer Science Contests in Germany*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[RG07] Pedro Ribeiro and Pedro Guerreiro, *Increasing the Appeal of Programming Contests with Tasks Involving Graphical User Interfaces and Computer Graphics*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[RML08] Miguel Revilla, Shahriar Manzoor, and Rujia Liu, *Competitive Learning in Informatics: The UVa Online Judge Experience*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 2, 2008.

[Sny05] Jan A. Snyman, *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*, Springer Publishing, 2005.

[Son02] Jeff Sonas, *The Sonas Rating Formula – Better than Elo?*

[Sph] Sphere Research Labs, *Sphere Online Judge*, `http://www.spoj.pl/`.

[TC] *TopCoder Algorithm Competitions*, `http://www.topcoder.com/tc?module=Static&d1=help&d2=index`.

[Thi06] Gert Thiel, *When is it appropriate to ban a player from a chess club?*, 2006, `http://www.chesscircle.net/forums/general-chess-forum/3108-when-is-it-appropriate-to-ban-a-player-from-a-chess-club.html`.

[Top08] TopCoder Inc., *Algorithm Competition Rating System*, Tech. report, 2008.

[Top09] ———, *Algorithm Competition Data Feeds*, 2009.

[TY02] Andrew L. Tepper and Josh M. Yelon, *The eGenesis Ranking System: A highly cheat-resistant, zero-sum ranking system for multi player games*, Tech. report, 2002.

[TY04] ———, *Tournament Ranking System*, Tech. report, 2004.

[USA] *USA Computing Olympiad (USACO)*, `http://www.uwp.edu/sws/usaco/`.

[UVa] *University of Valladolid Online Contest System*, `http://acm.uva.es/contest/`.

[vdLH96] Wim J. van der Linden and Ronald K. Hambleton (eds.), *Handbook of Modern Item Response Theory*, Springer, 1996.

[vdV06] Willem van der Vegt, *The CodeCup, an annual game programming competition*, Presented at Perspectives on Computer Science Competitions for (High School) Students, 2006. http://bwinf.de/competition-workshop/RevisedPapers/3_vanderVegt_rev.pdf.

[Ver] Tom Verhoeff, *Settling Multiple Debts Efficiently: An invitation to computing science*, Informatics in Education **3**, 105–126.

[Ver90a] ———, *Guidelines for Producing a Programming-Contest Problem Set*, Personal Note. http://www.win.tue.nl/~wstomv/publications/guidelines.pdf.

[Ver90b] ———, *Guidelines for Producing a Programming-Contest Problem Set*, An unpublished personal note.

[Ver97] ———, *The Role of Competitions in Education*, Presented at Future World: Educating for the 21st Century. http://olympiads.win.tue.nl/ioi/ioi97/ffutwrld/competit.pdf.

[Ver04] ———, *Concepts, Terminology, and Notations for IOI Competition Tasks*, Presented at IOI 2004. http://scienceolympiads.org/ioi/sc/documents/terminology.pdf.

[Ver06] ———, *The IOI is (not) a Science Olympiad*, Informatics in Education **5** (2006), 147–159.

[VG88] C. David Vale and Kathleen A. Gialluca, *Evaluation of the Efficiency of Item Calibration*, Applied Psychological Measurement **12** (1988), no. 1, 53–67.

[VKK09] P. J. Voda, J. Komara, and J. Kluka, *Clausal Language*, 1994-2009, http://www.ii.fmph.uniba.sk/cl/.

[vL05] Wolter T. van Leeuwen, *A Critical Analysis of the IOI Grading Process with an Application of Algorithm Taxonomies*, Master's thesis, 2005.

[VVJ96] N. D. Verhelst, H. H. F. M. Verstralen, and M. G .H. Jansen, *A logistic model for time-limit tests*, ch. 10, pp. 169–186, Springer, 1996.

[Wei] Eric Weisstein, *Eric Weisstein's World of Mathematics*, http://mathworld.wolfram.com/.

[WY06] Hong Wang and Baolin Yin, *Visualization, Antagonism and Opening – Towards the Future of the IOI Contest*, Presented at Perspectives on Computer Science Competitions for (High School) Students, 2006. http://bwinf.de/competition-workshop/Submissions/9_WangYin.pdf.

[WYL07] Hong Wang, Baolin Yin, and Wenxin Li, *Development and Exploration of Chinese National Olympiad in Informatics (CNOI)*, Olympiads in Informatics (Valentina Dagiene, ed.), vol. 1, 2007.

[Yak06] Bogdan Yakovenko, *50% Rule Should Be Changed*, Presented at Perspectives on Computer Science Competitions for (High School) Students.