On the suitability of programming tasks for automated evaluation

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

Abstract

For many programming tasks we would be glad to have some kind of automatic evaluation process. As an example, most of the programming contests use an automatic evaluation of the contestants' submissions. While this approach is clearly highly efficient, it also has some drawbacks. Often it is the case that the test inputs are not able to "break" all flawed submissions. In this article we show that the situation is not pleasant at all – for some programming tasks it is impossible to design good test inputs. Moreover, we discuss some ways how to recognize such tasks, and discuss other possibilities for doing the evaluation. The discussion is focused on programming contests, but the results can be applied for any programming tasks, e.g., assignments in school.

1 Introduction

Competitions similar to the International Olympiad in Informatics (IOI, [12]) and the ACM International Collegiate Programming Contest (ACM ICPC, [1]) 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 article 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.

2 Investigating recent 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 1, some more details are in Appendix A.

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 1 and Appendix A 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.

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 1: Task survey results.

As we show in Appendix A, 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 tasks was not suitable for black-box testing at all! In other words, it was impossible to design good test inputs for these tasks.

After writing most of this article we became aware of the fact that [11] 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 [14].

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.

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 [2] 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 [10] 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 [9].

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 [7].) 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).

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 [5].

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 combinatorial 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 A.

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 [5].

5 Theoretical results on testing

The general theoretical formulation of the testing problem (given are two Turing machines, do they accept the same language?) is not decidable – see [6]. 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.

It can be shown that a more exact formulation of our testing problem is an NP-hard problem. For the lack of space we present just the idea behind this claim: We will reduce the boolean formula satisfiability problem (SAT) to the problem of testing programs. Suppose we have an instance of SAT, i.e., a boolean formula with n variables. We will encode it as follows: Take the referrence solution and in the beginning of the program add a piece of code that splits the input into a sequence of bits, consider the first n bits to be the values of variables, and evaluate the formula for these values. If the formula turns out to be true, the modified program gives an incorrect answer and terminates.

Thus, deciding whether the contestant's program is correct is at least as hard as deciding SAT. This implies that there is no (known) way to do the automated testing efficiently, and at the same time to guarantee 100% accuracy.

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 [15]) 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 [13]. 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, [8]) 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 [8]). Discussing the evaluation of such tasks at the IOI is beyond the scope of this article.

7 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.

This article attempted to show that not all tasks are suitable for automated evaluation. We described some of the reasons behind this fact and some possible ways of recognizing such tasks. In our opinion, at the IOI it is important that both the host SC and the ISC are aware that these issues do exist and that they'll take them into consideration when selecting future competition tasks.

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.

A 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 [5].

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 [5]. 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 [3].)

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.

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 [4]), 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.

B Acknowledgements

The author would like to express his thanks to the Slovak Informatics Society (SISp) and the IOI for their partial financial support. The author is also indebted to all the anonymous reviewers. Their remarks significantly helped to improve the clarity of this paper.

C Biography

M. Forišek is currently a graduate student at the Comenius University in Slovakia. He received a Master's degree in Computer Science from this university in 2004. As a student he was a very successful participant in programming contests, he was awarded gold medals both at the IOI and at the ACM ICPC World Finals. In the past years he became an active member of the community around various programming contests. He was the head of the problem committee for several international programming contests, including several years of the Internet Problem Solving Contest and one year of the Central European Olympiad in Informatics. His research interests range from theoretical computer science to education of mathematics, informatics, and algorithms.

References

[1] ACM International Collegiate Programming Contest (2006). `http://icpc.baylor.edu/`

[2] Appel, K., and W. Haken (1997). Solution of the Four Color Map Problem. Scientific American 237(4):108-121.

[3] Berlekamp, E.R., and J.H. Conway, and R.K. Guy (1982). Winning Ways for your mathematical plays. Academic Press, New York.

[4] de Bruijn, N.G. (1946). A Combinatorial Problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49:758-764.

[5] Forišek, M. (2004). On suitability of tasks for the IOI competition. http://ksp.sk/~misof/ioi/tasks.html

[6] Hopcroft, J.E., and J.D. Ullman (1979). Introduction to Automata Theory, Languages, and Computation. Addison-Wesley.

[7] Hromkovič, J. (2005). Design And Analysis of Randomized Algorithms. Springer-Verlag.

[8] Internet Problem Solving Contest (2006). http://ipsc.ksp.sk/

[9] Karp, R.M., and M.O. Rabin (1987). Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2):249-260.

[10] Knuth, D.E., and J.H. Morris (Jr), and V.R. Pratt (1977). Fast pattern matching in strings. SIAM Journal on Computing 6(1):323-350.

[11] van Leeuwen, W.T. (2005). A Critical Analysis of the IOI Grading Process with an Application of Algorithm Taxonomies. Master Thesis at TU Eindhoven. http://www.win.tue.nl/~wstomv/misc/ioi-analysis/thesis-final.pdf

[12] International Olympiad in Informatics (2006). http://ioinformatics.org/

[13] TopCoder Algorithm Competitions (2006). http://www.topcoder.com/tc?module=Static&d1=help&d2=index

[14] Verhoeff, T. (2006). The IOI is (Not) a Science Olympiad. Informatics in Education, in print.

[15] Voda, P.J., and J. Komara, and J. Kluka (1994-2006). Clausal Language. http://www.ii.fmph.uniba.sk/cl/