Better feedback for educational online judges

Anaga Mani[1], Divya Venkataramani[1], Jordi Petit[2] and Salvador Roura[2]
[1]*SASTRA University, Tanjore (India)*
[2]*Universitat Politècnica de Catalunya, Catalonia (Spain)*

Keywords: Online Programming Judges, Automatic Assessment, Data Mining.

Abstract: The verdicts of most online programming judges are, essentially, binary: the submitted codes are either "good enough" or not. Whilst this policy is appropriate for competitive or recruitment platforms, it can hinder the adoption of online judges on educative settings, where it could be adequate to provide better feedback to a student (or instructor) that has submitted a wrong code. An obvious option would be to just show him or her an instance where the code fails. However, that particular instance could be not very significant, and so could induce unreflectively patching the code. The approach considered in this paper is to data mine all the past incorrect submissions by all the users of the judge, so to extract a small subset of private test cases that may be relevant to most future users. Our solution is based on parsing the test files, building a bipartite graph, and solving a Set Cover problem by means of Integer Linear Programming. We have tested our solution with a hundred problems in Jutge.org. Those experiments suggest that our approach is general, efficient, and provides high quality results.

1 INTRODUCTION

It is an established fact that practice is fundamental to learn computer programming. Whether at secondary, high school or university level, instructors usually assign programming problems to beginner students in order to help them acquire this skill. However, it is unquestionable that, for instructors, correcting programming assignments can be a slow, boring and error-prone task. Fortunately, programming problems are ideal candidates for automated assessment; see (Douce et al., 2005) for a review on this topic, (Ihantola et al., 2010) for an updated list of such systems and (Kurnia et al., 2001; Cheang et al., 2003; Joy et al., 2005) for some direct experiences in the classroom.

Currently, programming online judges are widely used to assess solutions to programing problems. Online judges are web based systems that offer a repository of programming problems and a way to submit solutions to these problems in order to obtain an automatic verdict on their behavior when run on different public and private test cases designed to be as exhaustive as possible. The solutions of such problems are complete programs or functions with a well defined specification.

Historically, online judges were targeted to train participants for important programming contests as the IOI[1] or the ACM-ICPC[2]. UVa[3], Timus[4], Sphere[5] and Codeforces[6] are just a few examples of such judges. More recently, some judges as TopCoder[7] and InterviewStreet[8] were built as a recruitment platform where companies can find and try to hire highly skilled programmers.

Additionally, some online judges were designed with an educative purpose in mind. Some notable examples are EduJudge[9], URI Online Judge[10] and Jutge.org[11]. These systems offer an organized repository of graded problems, and integrate the features of an online judge with those of a learning management system to assist both students and instructors. We refer to (Verdú et al., 2012; Tonin et al., 2012; Petit et al., 2012) for an overview of their respective capabilities, and to (Giménez et al., 2009) for an experience on the usage of Jutge.org in a first-year introductory programming course.

[1]http://www.ioinformatics.org
[2]http://icpc.baylor.edu
[3]http://uva.onlinejudge.org
[4]http://acm.timus.ru
[5]http://www.spoj.pl
[6]http://codeforces.com
[7]http://www.topcoder.com
[8]http://www.interviewstreet.com
[9]http://eduvalab.uva.es/en/projects /edujudge-project
[10]http://www.urionlinejudge.com.br
[11]http://www.jutge.org

A common feature of online judges is that their verdict is, essentially, binary: Either the submitted solution is correct or it is not. Clearly, these binary verdicts hinder the adoption of online judges in educative settings. Indeed, some students may feel frustrated when their solution is evaluated as wrong but no example where it fails is offered by the system. Likewise, some instructors may be reluctant to use a tool for which they cannot access the private test cases. Since beginners tend to write overly complicated solutions, online judges often detect errors that can be very hard to find by just inspecting the source code, even for experienced instructors.

As long as online judges require to keep secret their private test cases, there does not seem to be much to do to increase the feedback. However, in some particular settings it is feasible to automatically correct some wrong solutions (Singh et al., 2013). Another possible approach, the one used in this paper, is to relax a bit the secrecy and consider that, for educative purposes, it could be positive to reveal a subset of the private test cases.

An obvious solution would be to just show users an instance for which their code fails. However, we feel that just providing an arbitrary example where a program fails would encourage sloppy-thinking programmers, who first recklessly write a wrong program, and afterwards try to patch it (several times) using the given counterexamples, which, worse still, could be too insignificant again. Clearly, this is not the right way to proceed. Instead, we believe that the counterexamples provided (at least to the instructors, and perhaps also to the students; see the conclusions of this paper) should be as relevant as possible. By "relevant" we mean that these test cases should probably be helpful to most of the users. The goal of this paper is to show how to create these relevant test cases.

Our solution, which we call *the distiller*, consists in data mining past incorrect submissions sent by all the users of an online judge. We will present some examples that indicate that this "distillation" process, which we have implemented in Jutge.org, is quite general and efficient. Although we lack a quantitative measure for the quality of the produced test cases, we will show that, in general, a set of a few small test cases is enough to make (almost) *all* bad submissions fail. Therefore, it could be argued that those *automatically generated* test cases capture somehow the essence of the problem, and thus should be of great help when revealed to some users.

To the best of our knowledge, this is the first time that such a study has been conducted.

Organization. The paper is organized as follows: In Section 2, we give a more detailed overview of the main elements of online judges and present some basic definitions. The basic description of the problem we tackle is given in Section 3, together with the main ingredients of our solution. An overview of the results obtained by our tool for a few selected problems from Jutge.org is shown in Section 4. Section 5 discusses how to use the distilled test cases.

2 BACKGROUND

In this section, we first summarize the main elements and resources of online judges, then properly define the different types of test cases and submissions we are interested in.

Main elements of online judges. Although online judges have differences among them, they all have a common set of elements and resources and a common way to interact with them. On one hand, judges keep information on their users and provide functionality to register, log in, etc.

On the other hand, judges offer a repository of problems. Problems consist of a statement (typically in HTML or PDF format), some test cases, and some internal specification on how to run the test cases under certain constraints as time or memory limits.

The basic interaction between a user and the judge is the user submitting a solution to some problem in the repository. At this point, the judge will queue the submission to check for its correctness using the test cases and, subsequently, the judge will provide a verdict to the user.

Processing a submission is a somehow difficult task, because it must be performed in a secure way that defends the judge against possible attacks. We shall ignore these important implementation topics as they are not relevant to the current work, but refer the interested reader to (Leal and Silva, 2003; Forišek, 2006; Petit et al., 2012).

Classification of test cases. Besides the statement, problem setters define two kinds of test cases:

- *Public test cases*: Test cases that are visible by the users. They are usually included in the statement of the problem, in order to clarify it and to provide a few examples of the expected input and output.
- *Private test cases*: Internal test cases that are never disclosed to the users. They are used to check, in the most possible exhaustive way, the

correctness of the submissions. These tend to include hand made tricky inputs and automatically generated inputs.

Most problems expect solutions that read from the standard input and write to the standard output. Thus, the different test cases are stored in pairs of files, one for the input and another for the correct output.

Because of practical reasons (for instance, the overhead of creating a new process to do very small work), many individual test cases are usually joined in a few files. To clarify this, consider the problem of computing the greatest common divisor of two numbers. Assume that this problem has n individual test cases (say, $n = 10^5$). Then, instead of having n input files and their corresponding n output files, we rather have a single input file with the n pairs of numbers and one output file with the n correct answers. Of course, the problem statement asks to read a sequence of pairs of numbers and, for each one, write their gcd. Moreover, in addition to this big file, the problem setter will often add a few small files, to check, e.g., the correct processing of a file with just one pair of numbers or with no numbers at all.

In the following, we will stress this distinction by denoting by *test cases* the individual tests that are joined in a *test file*. In general, problems have a few test files (say, at most a dozen) with many test cases (say, some thousands). When needed, we will also distinguish between *input test files* and *correct output test files* and between *input test cases* and *correct output test cases*. Likewise, we will use the terms *user output test files* and *user output test cases* for the outputs generated by a (possibly incorrect) submission from a user.

Classification of submissions to problems. Online judges traditionally classify submissions with a marking system that is mainly binary. A solution is either *"Accepted"* (AC) when it efficiently produces satisfactory results on all the test cases run by the judge, or it is *"Rejected"*. Despite of that, *"Rejected"* submissions usually carry some additional information to provide a better verdict:[12] *"Compilation Error"* (CE), meaning that the submitted code cannot even be successfully compiled, *"Execution Error"* (EE) meaning that some error appeared during the execution of some test case (possibly reporting a refined diagnostic as division by zero, invalid memory access, memory limit exceeded, or the most common time limit exceeded...) or *"Wrong Answer"* (WA), meaning that, despite a successful execution, the user output does not match the correct output. This last verdict indicates a solution that is fast enough and does not crash, but which is wrong. Note that the reason for a WA verdict can range from a subtle bug in the code to a program that has little or no sense at all. In any case, the contents of the private test cases are never disclosed to the user.

[12] See, e.g., https://www.jutge.org/documentation/verdicts or http://uva.onlinejudge.org/index.php?option=com_content&task=view&id=16&Itemid=31

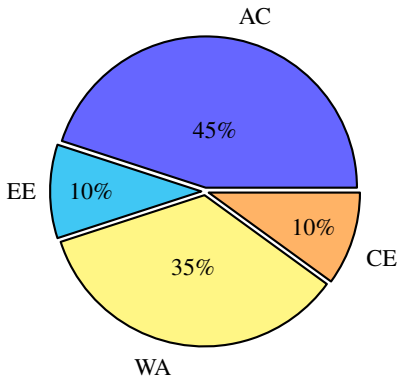

Figure 1: Current distribution of verdicts for Jutge.org.

Figure 1 shows the current distribution of verdicts for Jutge.org.

For the purpose of this work, we are interested in an alternative classification of submissions:

- *"Good submissions"*: Submissions that pass all test cases, public and private. These correspond to *"Accepted"* submissions.
- *"Bad submissions"*: Submissions that pass all public test cases but fail in some private test cases.
- *"Horrible submissions"*: Submissions that do not even pass all public test cases.

In the rest of the paper we shall not be interested in good submissions (because, being good, they do not help us to find relevant test cases) nor in horrible submissions (because these are so bad that they do not reflect a minimum of quality). Thus, we focus in bad submissions.

Please observe that the *good, bad* and *horrible* terms are labels that we only associate to certain types of submissions and should not be applied to their authors. Moreover, these terms exclusively apply to the black-box correction process of the judge, which is based on the functional correctness and efficiency of the submitted codes, but not on other important qualities, such as readability, succinctness, elegance, etc.

3 PROBLEM DESCRIPTION AND SOLUTION

Consider any particular problem in the repository of an online judge. Our goal is to extract a relevant test file with a small set of private test cases for which all bad solutions fail for at least one test case in the test file. For simplicity, we will generally assume that the number of test cases in the relevant test file must be minimized. When necessary, we will point to the minor changes to apply so to reduce the total size of the file itself.

In order to discover these relevant data sets, we assume that the following information is available:

- A set of private input test files together with their corresponding correct output test files.
- A "large enough" collection of bad submissions.
- For each input test file and each bad submission, the corresponding user output file.

In the case of Jutge.org, all this information is permanently stored in the system databases, but in other judges the user output files are not kept and should be regenerated by resubmitting the bad submissions.

Let n denote the total number of test cases for the problem, and let m be the total number of available bad submissions.

Our solution is conceptually described in Figure 2 in the next column. In the following, we provide further details for some of those steps.

Splitting files (steps 1, 2 and 3). To split an input test file consists in parsing it in order to extract its individual input test cases. Output test files are split in a similar way.

The splitting phase is highly dependent on the specific syntax prescribed for the input and output files. In the worst situation, an ad-hoc program to parse the input and the output is necessary for each problem. Fortunately, a large fraction of elementary problems share the same types of input/output general parsing schemes, possibly involving a few parameters. Of course, each individual problem must still be tagged by hand to specify how to split its inputs and outputs.

These are the most frequent parsing schemes that we have identified among those implemented in our tool:

- *No split:* No split needs to be done, as all the input in each test file represents a unique test case.
- *Split by lines:* Each line of the file corresponds to an individual test case.

1. Split all input test files into a list of input test cases $[I_1, \ldots, I_n]$.
2. Split all correct test files into a list $[C_1, \ldots, C_n]$, so that C_i is the correct output for I_i.
3. For each bad submission $1 \leq j \leq m$, split all its user output test files into a list $[O_1^j, \ldots, O_n^j]$, so that O_i^j is the output of the j-th bad submission for the i-th test case.
4. Build a bipartite graph:
   - Its n left vertices correspond to each test case.
   - Its m right vertices correspond to each bad submission.
   - For each submission j that fails on a test case i, (i.e., $O_i^j \neq C_i$), add an edge between i and j.
5. Compute a minimum subset $S \subseteq \{1, \ldots, n\}$ of left vertices whose induced edges hit all right vertices.
6. Output all the I_k, for all $k \in S$, as the relevant test cases for the problem.

Figure 2: *Distiller* algorithm.

- *Split by words:* Each word of the file corresponds to an individual test case. Here, a word is understood as a sequence of non-blank characters.
- *Split by K words:* Each K consecutive words of the file corresponds to an individual test case.
- *Split with counter:* Each test case is made up of several elements (lines, words, K words,...) and the number of them is specified with an initial counter.
- *Split with separator:* Each test case is made up of several elements (lines, words, K words,...) and they are separated with a special separator to be specified as a parameter.

In the case that a problem needs a non predefined splitting scheme, our implementation allows us to define a customized scheme (see Section 4).

Solving the Set Cover problem (steps 4 and 5). Step 4 builds a bipartite graph where left vertices corresponds to test cases, right vertices to submissions, and edges correspond to failure to report the correct output for a particular submission on a particular test case. Step 5 asks for solving the following problem: Given a bipartite graph $G = (U, V, E)$, find a minimum subset S of U such that each vertex in V has some neighbor in S. Figure 3 illustrates it.

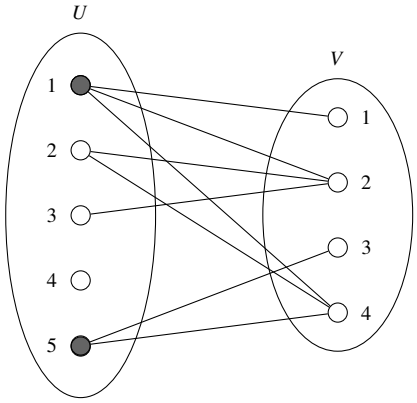

Figure 3: The Set Cover problem: Given a bipartite graph $G = (U, V, E)$, find a minimum subset S of U such that each vertex in V has some neighbor in S. In this example, the minimum solution is obtained with $S = \{1, 5\}$.

It turns out that this problem is one of the formulations of the *Set Cover problem*, which is a classical problem in computer science. Its decisional version is **NP**-complete, as proved in (Karp, 1972). Some heuristics have been considered to efficiently obtain solutions of good quality for this problem. It is known (Chvatal, 1979) that the obvious greedy algorithm (at each stage, choose the test case connected the largest number of still uncovered submissions) has a logarithmic approximation ratio. Unfortunately, this is essentially the best possible ratio under the usual $\mathbf{P} \neq \mathbf{NP}$ assumption (Alon et al., 2006).

Despite the theoretical intractability of Set Cover, we are able to exactly solve our instances using Integer Linear Programming (ILP). Let x_i be the binary variable that encodes whether $I_i \in S$ or not. Also, let $E_{i,j} \in \{0, 1\}$ indicate whether there is an edge connecting the i-th test case with the j-th submission. Then, the set cover problem is modeled as

$$\text{minimize} \sum_{i=1}^{n} x_i$$

subject to

$$\sum_{i=1}^{n} x_i \cdot E_{i,j} \geq 1 \qquad \forall j = 1, \ldots, m$$

$$x_i \in \{0, 1\} \qquad \forall i = 1, \ldots, n.$$

Output distilled set (step 6). In order to output the input test cases that make up the distilled set, these must be joined in the opposite way that they were split. To do so, each parsing scheme must implement this reverse operation.

Further details. Let us finally point to some additional details relevant to our implementation:

- To speed-up the running time of our solution, our complete implementation does not work with the output test cases themselves, but with numeric hashes instead. This enhancement keeps true negatives, but can introduce false positives (a wrong test case might be deemed correct) with a tiny probability, which we decide to ignore for the sake of efficiency.
- In some rare cases, we have found that some bad submissions are "*really bad*". These are detected by observing that the number of user output test cases is different than the number of correct output test cases or, when being equal, these differ in too many positions. To avoid junk into the graph, our algorithm ignores these submissions in a parameterizable way.
- In order to optimize the total size of the distilled set, we can simply change the objective function of the ILP so to weight each input test case by its size. Other variations such as maximizing the number of bad submissions caught by a distilled set of a given maximal size could be defined in the same style.

4 RESULTS

We have implemented a program in Python using a commercial product (Gurobi, 2013) to solve the ILP.

For testing, we have chosen about a hundred different problems from the introductory programming course of Jutge.org and their corresponding submissions of about 7200 users since September 2006. In the sake of providing results on the most popular problems, the selection was based on their number of submissions.

Each of these problems with identifier *id* is freely available at `http://www.jutge.org/problems/`*id*. All statistics are at the time of writing this paper.

Validating dates (P29448). This problems asks for a program that reads a list of triples of integers and, for each one, prints whether it is or not a valid date in the Gregorian calendar. Its public test cases are shown in Table 1(a).

Any experienced programmer would agree that this problem is easy but presents a few complications for a beginner, because it requires a thorough case analysis for the validation on the number of days for each month, with the additional difficulty of copying with leap years.

This problem has five private test files, which feature a total of 129458 elemental test cases (128163

Table 1: Results for problem "Validating dates".

(a) Public samples

Sample input	Sample output
30 11 1971	Correct Date
6 4 1971	Correct Date
4 8 2001	Correct Date
29 2 2001	Incorrect Date
32 11 2005	Incorrect Date
30 11 2004	Correct Date
-20 15 2000	Incorrect Date

(b) Statistics

Submissions	6903
Accepted	1600
Rejected	5303
Compilation Error	378
Wrong Answer	4856
Execution Error	69
Users with accepted problem	1115
Users with rejected problem	276
Good submissions	1600
Bad submissions	3718
Horrible submissions	1585

(c) Distilled input test cases

29 2 4900	13 9 1800	29 2 1870
7 2 1900	28 0 1852	31 2 1964
31 8 8298	31 1 7709	1 12 2002
29 1 1942	31 12 5741	30 1 3157
30 2 1864	29 2 6592	0 2 1910
31 7 3535	31 9 1948	29 2 2000
31 2 2000	32 12 2008	0 3 2000
5 13 2000	31 11 2000	

without duplicates). The statistics for this problem are shown in Table 1(b).

The parsing scheme for this problem was set to *split by 3 words* and the parsing scheme for the output was set to *split by lines*. The application of the distillation process took about 35 seconds (including 1.10 seconds to exactly solve the Minimum Set Cover problem with ILP) and produced a distilled set with 23 test cases, shown in Table 1(c).

As could be expected, those test cases feature limiting values for the number of the day (such as 0 and 32) and stress years (such as 2000) for which the leap year rule may be wrongly coded.

Roman numbers (P18298). This problems asks for a program that reads several (decimal) numbers and prints their equivalent Roman numbers. All numbers are in the range $[1,3999]$, and the rules to transcribe to Roman numbers are given. Its public test cases are shown in Table 2(a).

Table 2: Results for problem "Roman numbers".

(a) Public samples

Sample input	Sample output
1	1 = I
4	4 = IV
10	10 = X
16	16 = XVI
2708	2708 = MMDCCVIII
999	999 = CMXCIX
3005	3005 = MMMV

(b) Statistics

Submissions	2338
Accepted	1352
Rejected	986
Compilation Error	178
Wrong Answer	758
Execution Error	50
Users with accepted problem	1087
Users with rejected problem	39
Good submissions	1352
Bad submissions	867
Horrible submissions	119

(c) Distilled input test cases

2490	3470	1000	3049

This problem has three private test files, which feature 127977 elemental test cases (but just 3999 without duplicates; the remaining ones are there just to ensure that the programs are not too slow). The statistics for this problem are shown in Table 2(b).

The application of the distillation process to this problem took five seconds, and produced a distilled set with just four test cases, shown in Table 2(c).

Anagrams (P71916). This problems asks for a program that reads n pairs of lines and tells, for every pair, if the lines are anagrams of each other (i.e., if one line can be obtained from the other simply reordering the letters and forgetting the spaces and the punctuation symbols). It has the peculiarity that the number n is given at the beginning of the input. Its public test cases are shown in Table **??**, and its statistics are shown in Table 3(a).

This problem uses the predefined *split by lines* output scheme, but requires an ad-hoc input scheme. We need to specify how to parse the input tests (for this case, the easiest is to forget the first line and group the remaining lines in pairs) and how to rewrite them (print the number of pairs followed by the pairs themselves). The following simple code is all we need to do this customization:

Table 3: Results for problem “Anagrams”.

(a) Public samples

Sample input

```
3
ROME.
MORE.
AVE MARIA, GRATIA PLENA, DOMINUS TECUM.
VIRGO SERENA, PIA, MUNDA ET IMMACULATA.
ARMY.
MAY.
```

Sample output

```
YES
YES
NO
```

(b) Statistics

Submissions	2454
Accepted	922
Rejected	1532
Compilation Error	274
Wrong Answer	949
Execution Error	309
Users with accepted problem	636
Users with rejected problem	136
Good submissions	922
Bad submissions	1093
Horrible submissions	439

(c) Distilled input test cases

```
6
R.
RRR.
.
RR.
,,.
,.
A.
.
WINSTON CHURCHILL.
I'LL CRUNCH THIS NOW.
.
.
```

```
class CustomInputSplitter ( Splitter ):
    def read ( self , f ):
        return group(f. readlines ()[1:], 2)
    def write ( self , inputs , f ):
        print >>f, len( inputs )
        for input in inputs :
            print >>f, input[0] + "\n" + input[1]
```

The generated distilled file for this problem is shown in Table 3(c) and evidences some corner cases.

Brackets and parentheses (P96529). We finally consider a problem that is more difficult from an algorithmic point of view: To read words made up of brackets and parentheses and, for each one, tell whether the brackets and the parentheses close correctly. Its public test cases are shown in Table **??**(a).

Table 4: Results for problem “Brackets and parentheses”.

(a) Public samples

Sample input	Sample output
`[]([]())`	yes
`(([])()`	no
`[](`	no
`(([]))`	yes

(b) Distilled input test cases

```
(
[]
][
]]
)]
[[[
([)[]
[((())]]
[[(())](
[(()[])][([]])(())]
```

In this case, the distillation process produced 11 individual test cases out of the 1766 private test cases. However, the distilled set has a big issue: its total size is quite big (18360 characters). Clearly, although these test cases catch all bad submissions, they are too long to be helpful to humans. Our solution to this trouble consisted in modifying the ILP so as to minimize the total size of the distilled set by weighting each variable in the objective function by the length of the corresponding input. Doing so, we obtained 13 test cases, whose total size is 1254 characters, and where three test cases still have more than 350 characters each. By removing these three long test cases, we got the distilled set shown in Table 3(b), which still catches 91% of the bad submissions.

5 CONCLUSIONS AND FUTURE WORK

In this paper we have considered how to identify small and relevant test cases for problems in online judges. The main idea of our solution is to data mine past wrong (but not too wrong) submissions to automatically extract the relevant test cases.

To do so, we have provided an elegant solution that we have applied and tested on several problems of Jutge.org, a virtual learning environment for computer programming. The results suggest that our solution is general, efficient, and generates high quality

test cases. Taking into account the amount of effort spent by problem setters in creating huge and exhaustive private test cases, it is almost deceiving to see *a posteriori* that only a few of these test cases are, in fact, relevant.

These relevant test cases can improve the feedback of educative online judges which, until now, just report a negative verdict in the event of a wrong submission. However, an important question that remains to be addressed is how to use these distilled sets in the workflow of educative online judges.

A first option would be to disclose the distilled test cases to students that request them after they have submitted a bad submission. A similar approach would be to disclose only one distilled test case for which the student's code fails (assuming there is at least one, which should happen most of the times). Although this could be a way to help beginners to spot their errors and fix them, it also could induce a "trial and error" behavior similar to the one already criticized in the introuction of this paper. At the very least, the counterexample where the code fails, being part of the distilled set, should provide a deeper insight to the users than just a random counterexample.

An alternative to mitigate the above option would be to keep distilled test cases private from students, but available to instructors. Looking both to the code of their mentored students and to the distilled test cases (where the code fails), instructors could be able to provide better individual answers, including, if deemed suitable, some of the distilled test cases.

The community of instructors using Jutge.org is currently debating these options and some variations of them.

Acknowledgements. We thank all the users of Jutge.org: without their numerous mistakes, this work would have never seen the light.

REFERENCES

Alon, N., Moshkovitz, D., and Safra, S. (2006). Algorithmic construction of sets for *k*-restrictions. *ACM Trans. Algorithms*, 2(2):153–177.

Cheang, B., Kurnia, A., Lim, A., and Oon, W.-C. (2003). On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131.

Chvatal, V. (1979). A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235.

Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *ACM Journal on Educational Resources in Computing*, 5(3).

Forišek, M. (2006). Security of Programming Contest Systems. In Dagiene, V. and Mittermeir, R., editors, *Information Technologies at School*, pages 553–563.

Giménez, O., Petit, J., and Roura, S. (2009). A pure problem-oriented approach for a CS1 course. In Hermann, C., Lauer, T., Ottmann, T., and Welte, M., editors, *Proc. of the Informatics Education Europe IV*, pages 185–192.

Gurobi (2013). Gurobi Optimizer Reference Manual. `http://www.gurobi.com`.

Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Procs. of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93. ACM.

Joy, M., Griffiths, N., and Boyatt, R. (2005). The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3).

Karp, R. M. (1972). Reducibility Among Combinatorial Problems. In Miller, R. E. and Thatcher, J. W., editors, *Complexity of Computer Computations*, pages 85–103.

Kurnia, A., Lim, A., and Cheang, B. (2001). Online judge. *Computers & Education*, pages 299–315.

Leal, J. P. and Silva, F. (2003). Mooshak: a web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581.

Petit, J., Giménez, O., and Roura, S. (2012). Jutge.org: an educational programming judge. In *Procs. of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 445–450. ACM.

Singh, R., Gulwani, S., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In Boehm, H.-J. and Flanagan, C., editors, *PLDI*, pages 15–26. ACM.

Tonin, N., Zanin, F., and Bez, J. (2012). Enhancing traditional algorithms classes using URI online judge. In *2012 International Conference on e-Learning and e-Technologies in Education*, pages 110–113.

Verdú, E., Regueras, L. M., Verdú, M. J., Leal, J. P., de Castro, J. P., and Queirós, R. (2012). A distributed system for learning programming on-line. *Computers & Education*, 58(1):1 – 10.