A New Method to Increase Feedback for Programming Tasks During Automatic Evaluation

Test Case Annotations in ProgCont System

Piroska Biró[1,2], Tamás Kádek[3], Márk Kósa[1], János Pánovics[1]

[1]University of Debrecen, Faculty of Informatics, Dept. of Information Technology
Kassai út 26, 4028 Debrecen, Hungary
{biro.piroska, kosa.mark, panovics.janos}@inf.unideb.hu

[2]Sapientia Hungarian University of Transylvania
Faculty of Economics, Socio-Human Sciences and Engineering
Piaţa Libertăţii nr. 1, 530104 Miercurea Ciuc, Romania

[3]University of Debrecen, Faculty of Informatics, Dept. of Computer Science
Kassai út 26, 4028 Debrecen, Hungary; kadek.tamas@inf.unideb.hu

Abstract: The unexpected challenges posed by the pandemic also have transformed university education. Information technology is still the most advantageous field, as IT tools in education are more widespread. We have been using the ProgCont system for automatic evaluation of programming tasks since 2011 at the Faculty of Informatics of the University of Debrecen. The system's responsibilities have expanded over the years, and due to the pandemic, it will have to play a more significant role in self-preparation. Initially, we used the system to evaluate competitive tasks and later examinations. In this period, the feedback was limited to accepting or rejecting the submitted solutions. A submitted solution is accepted if the application produces the appropriate output for the problem's input. Usually, we test the submissions with several inputs (test cases) for each problem. To provide additional information about the reason for rejection, we would like to supplement test cases with comments (annotations) that identify the test cases' unique properties. Our goal is to help identify the subproblems that need improvement in case of a partially correct solution. In our article, we would like to present the potential of this development. We chose a problem that received an impressive number of solutions. We created new test cases for the problem with annotations, and by re-evaluating the submissions, we compared how much extra information students and instructors obtained using the annotations. The presented example proves that this new development direction is necessary for students' self-preparation and increases differentiated education possibilities.

Keywords: ProgCont system; programming education; automatic solution evaluation; test case annotations

1 Introduction

The emergence of the pandemic will radically reshape university education. In this form of training, it is possible to rely more strongly on students' independent work compared to secondary school and primary school education. At the university level, distance education is easier to introduce, and higher education institutions have also switched to this form of education. At the University of Debrecen, education could be restored to its traditional form only for two months in the year after the pandemic had started in March 15, 2020.

The Faculty of Informatics had several IT solutions to support education, the role of which suddenly and significantly increased during the pandemic. The ProgCont system that implements automatic evaluation of programming exercises is a good example.

We have been developing the system for almost a decade, during which time its usage has expanded significantly [3], [8], [9], [15]. In the context of distance education, we want to strengthen its role in self-preparation.

We considered using other existing systems: Mooshak [10], [14], PC^2 – Programming Contest Control [2], UVa Online Judge [16], [17], Bíró and Mester ELTE [5]. They were all outstanding imaginative applications [1], [6], [7], [11], [12], [18], yet they did not fit perfectly with local needs.

The ProgCont system was intended initially for automatic and objective evaluation of examinations and programming competition problems. By uploading the source code created as a solution, contestants received immediate feedback on whether or not their program was producing the appropriate output, making the solution of the problem acceptable or not. In case of a negative response, the competitor must alone identify the error in their program. We can also take advantage of the automatic evaluation system during our educational activities [13]; accordingly, the first examination problem sets and then practice problem sets have appeared in ProgCont.

Instructors using ProgCont formulated more and more different problems. Up to now,

- 45 competition problem sets,
- 241 examination problem sets,
- 11 practice problem sets

are available in the system with a total of 1 657 tasks. ProgCont supports C, C++, C#, Java, and Pascal programming languages by default (from 2011), and later it has become possible to use Python (from 2016) and Racket (from 2020).

Students often criticise that, although the evaluation is objective and automatic, it does not help correct a faulty program because it does not show the tests where the program does not perform well. The principle is that the test cases' content, apart

from an example usually given in the problem's description, is unknown. This practice makes it impossible for the submitted programs to focus on specific test cases instead of an algorithmic solution to the problem. It is possible to identify the test cases the application produces incorrect output for, but not the test cases' contents themselves. However, there would be no obstacle to exploring some test cases' characteristics without uncovering exact test content. To improve the feedback provided by ProgCont, we will introduce the possibility of using test case annotations from 2021 onwards.

The annotation of a test case is a short textual description that defines the subproblem examined with that particular test case. If we want to use annotations that identify the subproblems well, it could be necessary to modify the test cases.

In the following, we show the possibilities of annotations for a selected problem.

2 The Sample

We selected the problem that received the most submissions in the system so far, which means 1 387 submissions exactly. The problem has initially been a member of a problem set for the *High-Level Programming Languages 1* examination, and later it was published as a practice problem after the test.

> TASK[1]
>
> Write a program that reads times in 24-hour format from the standard input until end-of-file (**`EOF`**), one per line. The program should write to the standard output the 12-hour times corresponding to the given times. If the hours are less than 10, display the hours with one digit. The minutes should always appear with two digits. For example:

No.	Input	Output
1	0.02	12.02 am
2	11.58	11.58 am
3	12.32	12.32 pm
4	13.29	1.29 pm
5	22.17	10.17 pm

The selected assignment first appeared on March 11, 2014, on the day of the examination, and then it has been continuously available for the last seven years. In our article, we examine these seven years until March 11, 2021. During the examined period, we received 65 submissions resulting in compile error. Those are omitted from subsequent analyses because our system cannot run tests on those, so the actual number of submissions in the sample examined is 1 322.

[1] https://progcont.hu/progcont/100029/?pid=200502

The possible responses of the ProgCont system after the automatic evaluation are the following:

Compile error (`E-Cmp`): The submission is syntactically wrong. We are unable to execute the submitted program, so we cannot evaluate test cases on it.

Runtime error (`E-Run`): The execution of the program has failed, e.g., it is terminated with an error message.

Time limit exceeded (`E-Tme`): The execution of the program has been terminated forcibly after exceeding the given time limit.

Wrong answer (`E-Res`): The submission returns with incorrect output for the test case.

Presentation error (`E-Pre`): The submission returns with incorrect output for the test case, but the expected result differs in whitespace characters only.

Accepted (`Pass`): The submission returns with the correct output for the test case.

When a submission contains no compile (or syntactical) errors, then the system continues the examination with the help of at least one but usually more test cases. The evaluation result can be different for each test case; the final response depends on the errors' priority. The priority order of the response codes from highest to lowest are: **`E-Run, E-Tme, E-Res, E-Pre,`** and **`Pass.`**

3 Results

3.1 Findings from Original Test Cases

Initially, there were two test cases for the task. One of them was a short sample that also appears in the description of the task. The second test case contained all possible inputs, consisting of a total of 1 440 lines, each representing one task. Times appeared unordered in the test file. We have analysed similar problems in many ways before. Some important aspects are the comparison by source language and comparing different user groups' performance, which is impossible for this problem [3], [4], [8], [9]. Figure 1 shows what we can determine from the evaluation results of the submissions and the test cases. 30% of the submitted solutions completed the problem. The proportion of successfully passed tests is higher (37%). The reason for this difference is the fact that 13% of the submissions worked correctly only in one of the two test cases. Since the second test case contained all possible inputs, it is not difficult to guess that these programs failed on this second test case.

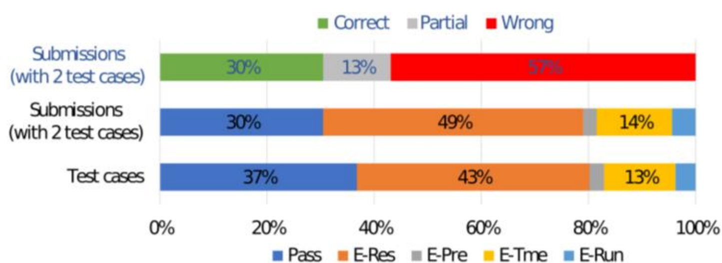

Figure 1
Distribution of evaluation results and partial solutions

From the point of view of our study, the partially correct solutions (or partial solutions for short) are the most interesting because in these cases, we could say more about the circumstances in which they are successful than those in which they are not. In this case, we can only talk about 167 partial solutions, which is only 13% of the solutions, according to Figure 1.

3.2 First Experiment

In the first experiment, we create new test cases for this problem and annotate them according to their different characteristics. In addition to the original two test cases, we prepared further 9 test cases. Evaluating the submissions with the new test cases, the number of correct solutions decreased, and the number of partial solutions increased (56%), which we summarised in Figure 2.

In the case of partial solutions (56%), we see a chance for the student to improve their existing program successfully. Without annotations, they have to perform the debugging process alone and create example inputs that bring out the program's error. With annotations, this process can be significantly simplified by comparing passed and failed test cases.

Table 1
Error distribution of 9 test cases

Test	Lines	Hours	Minutes	Pass	E-Res	E-Pre	E-Tme	E-Run
3	none	—	—	1 065	66	3	150	38
4	more	1–11	10–59	785	251	61	182	43
5	more	1–11	0–9	499	559	39	182	43
6	more	12	10–59	670	383	44	182	43
7	more	12	0–9	427	649	21	182	43
8	more	13–23	10–59	808	230	59	182	43
9	more	13–23	0–9	494	561	40	182	45
10	more	0	10–59	751	307	44	178	42
11	more	0	0–9	513	579	26	161	43

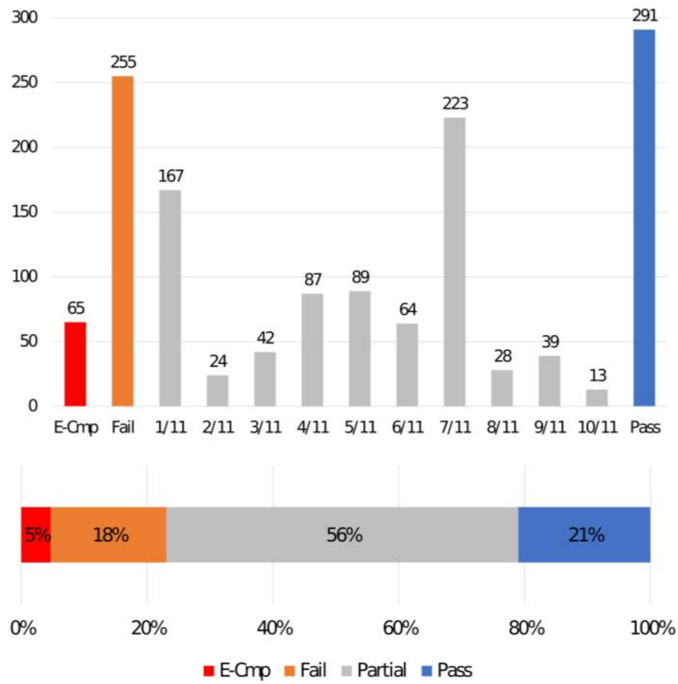

Figure 2
Distribution of partial solutions

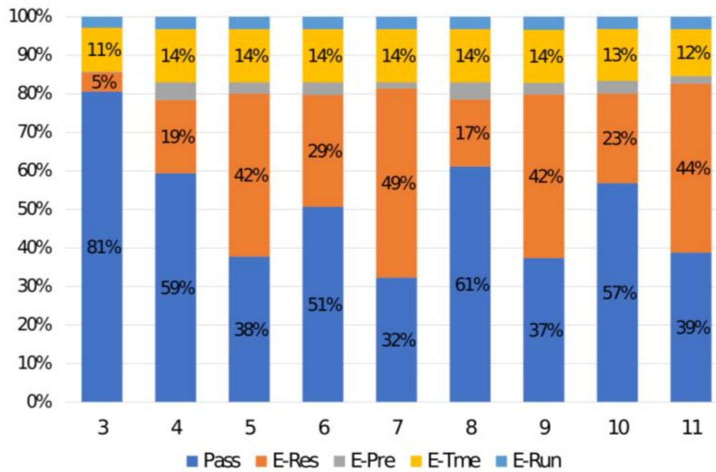

Figure 3
Error distribution of 9 test cases

The different features can be described in two categories; first, usually task-independent annotations:

`empty input = true:` the test case consists of an empty input file

`input type = text:` in the test case, text input must be processed

`line length = max 10:` the length of the input lines is up to 10 characters

`lines = none:` the test case does not contain any tasks

`lines = more:` the test case contains more than one task

The second and third annotations characterise all test cases of this problem due to the input specification. For some problems, **`empty input = true`** and **`lines = none`** may often differ. Test cases that include zero tasks may be nonempty files because they may record the number of tasks or contain some kind of end-of-input symbol.

Task-specific annotations:

`hour = 0:` the test case contains only tasks in which each time has an hour value of 0; i.e., the correct output is **`12.MM am`**

`hour = 1-11:` the test case contains only tasks in which each time has an hour value of 1–11; i.e., the correct output is **`1-11.MM am`**

`hour = 12:` the test case contains only tasks in which each time has an hour value of 12; i.e., the correct output is **`12.MM pm`**

`hour = 13-23:` the test case contains only tasks in which each time has an hour value of 13–23, i.e., the correct output is **`1-11.MM pm`**

`minute = 10-59:` the test case contains only tasks in which the minute has two digits

`minute = 0-9:` the test case contains only tasks in which the minute has one digit; therefore, the correct output starts with an initial 0

We can examine the results in two ways: per test case and per annotation.

Figure 3 and Table 1 show the number of errors per test case.

Initially, the ProgCont system only told us whether the output was correct or not, but later the submitter was enabled to check the correctness of their solution on a case-by-case basis. Table 1 and Figure 3 show that the number of successful solutions is small for test cases 5, 7, 9, and 11; a common feature of these test cases is that we have annotated them with **`minute = 0-9`**.

Using the annotations, we can group the test cases; this grouping indicates the above relationship in a much more direct way, as we can see in Table 2, Figure 4, and Figure 5.

The first five lines of Table 2 show a grouping by global annotations, and the rest show problem-specific annotations. We used one test case that contained no tasks; it appears as an empty input. For the selected problem, **`empty input = true`** and **`lines = none`** belong to this test case. For each syntactically correct solution, we evaluated this test case once. Most of the submitted programs (81%) worked correctly on the empty input (Figure 4). All other test files contain more than one task, so they all are annotated with **`lines = more`**. The annotation **`lines = one`** can highlight test cases that contain only a single task.

Table 2
Error distribution of 9 test cases by annotations

Annotation	Value	Count	Pass	E-Res	E-Pre	E-Tme	E-Run
empty input	true	1 322	1 065	66	3	150	38
input type	text	11 898	6 012	3 585	337	1 581	383
line length	max 10	10 576	4 947	3 519	334	1 431	345
lines	more	10 576	4 947	3 519	334	1 431	345
lines	none	1 322	1 065	66	3	150	38
hour	0	2 644	1 264	886	70	339	85
hour	1–11	2 644	1 284	810	100	364	86
hour	12	2 644	1 097	1 032	65	364	86
hour	13–23	2 644	1 302	791	99	364	88
minute	10–59	5 288	3 014	1 171	208	724	171
minute	0–9	5 288	1 933	2 348	126	707	174

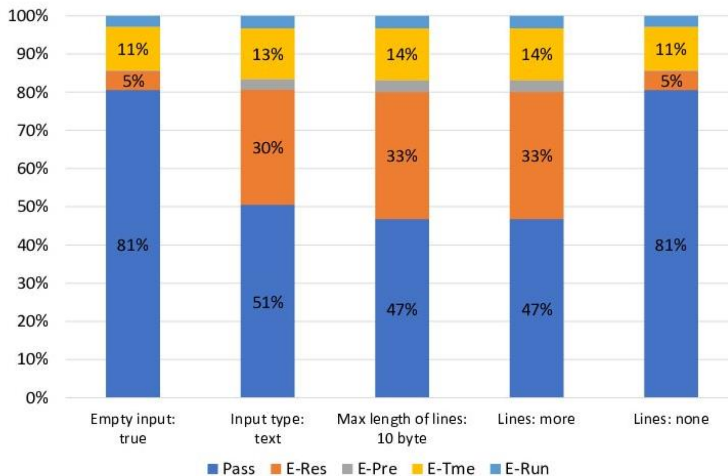

Figure 4
Error distribution of 9 test cases by global annotations

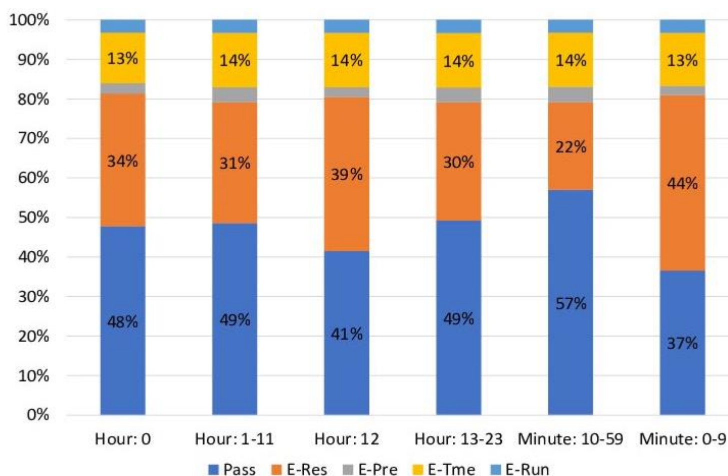

Figure 5

Error distribution of 9 test cases by problem-specific annotations

All nine new (annotated) test cases had to contain text input for this problem, so for all 1 322 syntactically correct submissions, 9 test cases had the annotation **`input type = text`** (we ran the submitted programs a total of 11 898 times). The last six lines of Table 2 show problem-specific annotations. It is clear from the table and the corresponding diagram (Figure 5) that the test cases with the annotation **`minute = 0-9`** are the least successful among the submitted solutions (37%); as for the hour, **`hour = 12`** is what causes difficulty (41%).

Presentation error (**`E-Pre`**) and runtime error (**`E-Run`**) are not typical at all among the submissions (4-5%). 14% of the submissions cause time limit exceeded error (**`E-Tme`**) in at least one of the test cases. 13% of those fail on all test cases for the same reason. The 1% **`E-Tme`** is due to the fact that several people read the input up to 0.0. If there is one such line in the input, the program stops with an erroneous result; otherwise, we get an infinite loop, which leads to an **`E-Tme`** error message. In another 13%, incorrect reads result in an infinite loop for each input (Figure 5).

Table 3

Submission verdict distribution by annotations

Annotation	Value	Pass	E-Res	E-Pre	E-Tme	E-Run	None
empty input	true	1 065	66	3	150	38	0
input type	text	305	60	0	147	35	775
line length	max 10	305	149	14	161	41	652
lines	more	305	149	14	161	41	652
lines	none	1 065	66	3	150	38	0
hour	0	483	274	21	161	41	342

hour	1–11	473	225	35	182	43	364
hour	12	407	363	21	182	43	306
hour	13–23	476	208	33	182	43	380
minute	10–59	542	172	30	178	42	358
minute	0–9	317	484	14	161	41	305

Figure 6 and Figure 7 show the ratio of accepted and rejected submissions based on the given annotation. Of course, the classification of some submissions can be uncertain because they can have different evaluation results on test cases with the same annotation. E.g., some tests classified with a particular annotation can pass, while others cannot. For example:

- There exists only one test case with the **`empty input = true`** and **`lines = none`** annotations, so each solution can be classified certainly. We used all the other global annotations on all of the remaining eight new test cases, and based on these, hardly half of the submitted solutions can be classified (Figure 6).
- In the case of **`minute = 0-9`**, we can see that 53% of the submissions clearly cannot handle the one-digit minute on input, so they are incorrect; this is the most common problem found in the submissions (Figure 7).

Global annotations are less informative than problem-specific annotations.

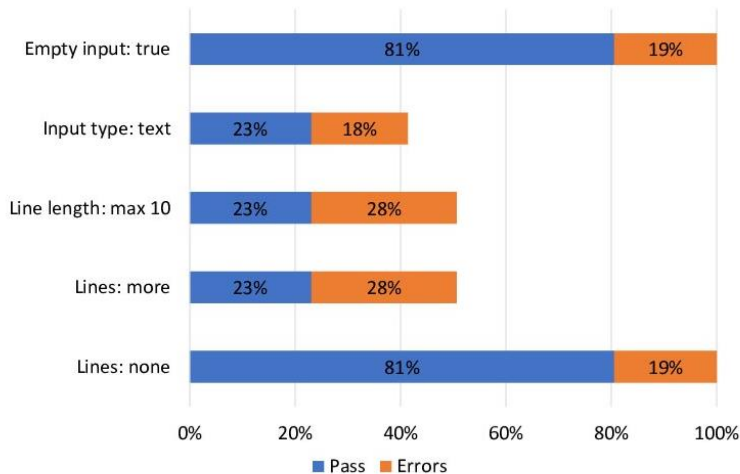

Figure 6
Error distribution of solutions by global annotations

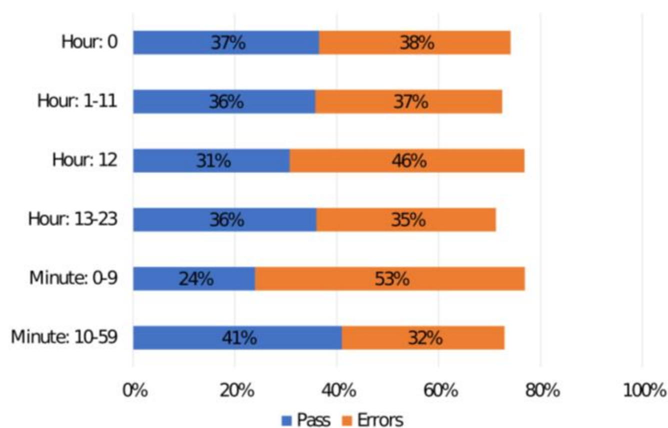

Figure 7
Error distribution of tests by problem-specific annotations

3.3 Second Experiment

In the first experiment, we tried to collect the cases we considered problematic and made test cases annotated accordingly. We grouped the times in the input into four categories by hours. In the second experiment, we discarded this preliminary suggestion and tentatively generated separate test cases for each of the 24 hours, yielding a total of 49 test cases (empty input and 24 hours with one- and two-digit minute values). We can see from the results, summarised in Table 4, that the hours previously classified with the same annotation behave very similarly. E.g., the difference between `hour = 1` and `hour = 2` is minimal.

In addition to the new test cases, the `minute = 0-9` and `minute = 10-15` annotations have 24–24 test cases. Figure 8 shows the weak correlation (0.59) of the test cases belonging to these two categories.

Table 4
Error distribution of 49 test cases by annotations

Annotation	Value	Count	Pass	E-Res	E-Pre	E-Tme	E-Run
empty input	true	1 322	1 066	65	3	150	38
input type	text	64 778	32 831	18 623	2 346	8 861	2 117
line length	max 10	63 456	31 765	18 558	2 343	8 711	2 079
lines	more	63 456	31 765	1 8558	2 343	8 711	2 079
lines	none	1 322	1 066	65	3	150	38
hour	0	2 644	1 276	876	70	339	83
hour	1	2 644	1 329	761	104	364	86
hour	2	2 644	1 336	752	104	364	88
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

hour	11	2 644	1 366	739	89	364	86
hour	12	2 644	1 105	1 024	65	364	86
hour	13	2 644	1 334	762	98	364	86
hour	14	2 644	1 344	744	104	364	88
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
hour	23	2 644	1 356	743	95	364	86
minute	10–59	31 728	19 567	5 328	1 439	4 364	1 030
minute	0–9	31 728	12 198	13 230	904	4 347	1 049

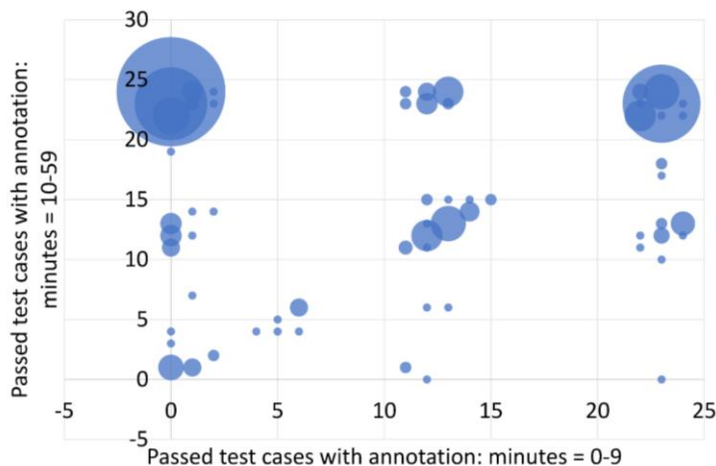

Figure 8
Passed test cases with annotation on minutes

As the number of successfully passed tests in test cases where the minute consists of a single digit increases, the number of successfully passed test cases where the minute contains two digits does not increase. The second experiment did not yield a new result; it merely confirmed that the original annotations were defined well. In this case, it is not advisable to stick to the version with more test cases, because it results in a longer evaluation time without additional information.

Conclusions

Our article presented a new method that helps students and educators alike in solving programming problems. The selected example demonstrates the functionality of the method. Merely by creating several test cases, the ratio of partial solutions increased from 13% to 56%. The created nine new test cases check well-separable subproblems. The annotations associated with them help identify the subproblems that partial solutions cannot solve correctly. In self-preparation, it gives students feedback that makes it easier for them to delineate and correct the error without asking the instructor for help. Educators can also keep track of which subproblems are the biggest challenge for students, helping them implement differentiated education.

Acknowledgement

This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.

References

[1] S. D. Benford, K. E. Burke, and E. Foxley, "A system to teach programming in a quality controlled environment," *The Software Quality Journal*, Vol. 2, pp. 177-197, 1993

[2] W. Bin, "Programming Contest Control System(PC^2) Application in Teaching," *Journal of Changzhou Vocational College of Information Technology*, 2005

[3] P. Biró and T. Kádek, "Automatic evaluation of programming tasks at the University of Debrecen," in *INTED2020 Proceedings*, 2-4 March, 2020 2020, pp. 3522-3527, DOI:10.21125/inted.2020.0994

[4] I. Falus, *Introduction to the methodology of pedagogical research (Bevezetés a pedagógiai kutatás módszereibe)* Budapest: Műszaki Kiadó, 2004

[5] G. Horváth, G. Horváth, L. Zsakó, "The Bíró and the Mester – the role of online assessment in programming education. (A Bíró és a Mester – az online értékelés szerepe a programozás oktatásában)," in: Károly K, Homonnay Z (eds) *Mérési és értékelési módszerek az oktatásban és a pedagógusképzésben*, ELTE Eötvös Kiadó, 2017

[6] D. Jackson and M. Usher, "Grading student programs using ASSYST," in *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, 1997, pp. 335-339

[7] D.W. Juedes, "Experiences in Web-based grading", *33rd Annual Frontiers in Education*, IEEE, 2003, Vol. 3, pp. S3F: 27-32

[8] T. Kádek and P. Biró, "The ProgCont API: innovative evaluation of solutions to programming assignments. (A ProgCont API: programozási feladatok megoldásainak újszerű kiértékelése)," in *ENELKO SzámOkt 2019*, 2019, pp. 191-195

[9] T. Kádek and P. Biró, "Effects of distance education on the ProgCont system. (A távolléti oktatás hatásai a ProgCont rendszerre.)," in *ENELKO SzámOkt 2020*, 2020, pp. 104-109

[10] J. P. Leal and F. Silva, "Mooshak: a Web-based multi-site programming contest system," *Software: Practice and Experience*, Vol. 33, No. 6, pp. 567-581, 2003

[11] A. K. Mandal, C. Mandal, and C. Reade, "A System for Automatic Evaluation of Programs for Correctness and Performance," in *Web Information Systems and Technologies*, 2007, pp. 367-380

[12] S. Manzoor, "Analyzing programming contest statistics," *Perspectives on Computer Science Competitions for (High School) Students*, Vol. 48, 2006

[13] P. Biró and T. Kádek, "The Mathability of Computer Problem Solving with ProgCont," *Acta Polytechnica Hungarica*, Vol. 19, No. 1, pp. 77-91, 2022, DOI:10.12700/APH.19.1.2022.19.6

[14] M. Rubio-Sánchez, P. Kinnunen, C. Pareja-Flores, and Á. Velázquez-Iturbide, "Student perception and usage of an automated programming assessment tool," *Computers in Human Behavior*, Vol. 31, pp. 453-460, 2014

[15] R. Tóth, M. Kósa, T. Kádek, and J. Pánovics, "The development of evaluation systems at the Faculty of Informatics, University of Debrecen," in INTED2019 Proceedings, 2019, pp. 5552-5559

[16] E. Verdú, L. M. Regueras, M. J. Verdú, J. P. Leal, J. P. de Castro, and R. Queirós, "A distributed system for learning programming on-line," *Computers & Education*, Vol. 58, No. 1, pp. 1-10, 2012

[17] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A Survey on Online Judge Systems and Their Applications," *ACM Comput. Surv.,* Vol. 51, No. 1, Jan. 2018

[18] I. M. Wirawan, A. R. Taufani, I. D. Wahyono, and I. Fadlika, "Online judging system for programming contest using UM framework," in 2017 *4th International Conference on Information Technology, Computer, and Electrical Engineering* (ICITACEE) 2017, pp. 230-234