Received: 4 March 2017 | Accepted: 23 July 2017
DOI: 10.1002/cae.21868

RESEARCH ARTICLE

WILEY

Correctness assessment of a crowdcoding project in a computer programming introductory course

Sebastián Ferrán | **Alejandra Beghelli** | **Gonzalo Huerta-Cánepa** | **Federico Jensen**

Faculty of Engineering and Sciences, Universidad Adolfo Ibáñez, Viña del Mar, Chile

Correspondence
Alejandra Beghelli, Faculty of Engineering and Sciences, Universidad Adolfo Ibáñez, Av. Padre Hurtado 750, Viña del Mar, Chile.
Email: alejandra.beghelli@uai.cl

Abstract
Crowdcoding is a programming model that outsources a software project implementation to the crowd. As educators, we think that crowdcoding could be leveraged as part of the learning path of engineering students from a computer programming introductory course to solve local community problems. The benefits are twofold: on the one hand the students practice the concepts learned in class and, on the other hand, they participate in real-life problems. Nevertheless, several challenges arise when developing a crowdcoding platform, the first one being how to check the correctness of student's code without giving an extra burden to the professors in the course. To overcome this issue, we propose a novel system that does not resort to expert review; neither requires knowing the right answers beforehand. The proposed scheme automatically clusters the student's codes based solely on the output they produce. Our initial results show that the largest cluster contains the same codes selected as correct by the automated and human testing, as long as some conditions apply.

KEYWORDS
automated code correctness assessment, computer programming education, crowdcoding, software crowdsourcing

1 | INTRODUCTION

Crowdsourcing gathers a vast and diverse group of people distributed around the world working towards a common goal resorting to the "wisdom of crowds." The assumption under this new collaborative work model is that the crowd can perform a task with higher speed and quality than any expert. In the crowdsourcing language, a requester is an organization or person that submits a task to a crowdsourcing platform. The workers are the people willing to carry out the task and submit their contributions to the same platform. Wikipedia is a well-known example of crowdsourcing, but nowadays many organizations are using this collaborative work model to improve their business in diverse areas such as product design, drug development, mining, and software development [4,15].

Crowdcoding (also known as Software Crowdsourcing) builds on the idea of crowdsourcing for software development projects. In a crowdcoding-based development, the original project (either as a whole or divided into smaller/simpler tasks) is presented as a coding challenge to an online community of software engineers. A group of reviewers ranks the submissions, and the best ones are selected to build the original project. One of the advantages attributed to crowdcoding is the lowering of defect rate thanks to the various development capacity provided by different programmers [7].

We argue that crowdcoding can be successfully used in educative environments as a way to solve computer-related challenges from local communities. That is, resorting to students (workers) of computer programming courses to build software projects required by the local communities (requesters). Academically speaking, the benefit of carrying out a crowdcoding project in an educative environment is twofold: on the one hand, students get involved in the development of real-world projects, with real users and requirements, making them aware of the professional challenges they might encounter when graduating. On the other hand, many useful software projects can be built through the years thanks to the work force available in these courses, increasing the benefit of the university to the community.

From the literature [11], we know that there are two main factors that affect the success (in terms of on-time delivery and software quality) of a crowdcoding project: (1) the number of registered developers, and (2) the size of the software to be developed. The first factor refers to the chance of getting a better quality solution if the number of workers increases. The second is related to the complexity of the software project. The usual approach to undertaking crowdcoding-based developments is task decomposition. This decomposition is done to decrease complexity and the time required to solve the work given that several smaller tasks that can be tackled in parallel. Depending on the size of each task, they can be either a micro-task [8], which refers to minute-length tasks with low cognitive effort, as discussed in [3,6] or a complex and time-consuming sub-problem, which remain simpler than the original project. Nevertheless, it is worth noticing that the benefit of presenting a software project as a collection of sub-tasks might be lost by the effort of putting back together the small pieces of code. Such job can be daunting as it involves using well-defined interfaces and testing procedures as well as intensive coordination and communication.

In the context of a programming course, the two factors mentioned above can be controlled. For the first factor, the number of students enrolled in a typical programming course is usually significantly higher than the number of registered coders on a site such as TopCoder, in which only 2–15 coders finally submit their work in a competition [17]. Therefore, the chances of high-quality solutions increase. Regarding the complexity of the tasks, the team of instructors must ensure that the software project is divided into simple sub-tasks, suitable to the programming skills of the students.

Nevertheless, there are two other challenges to overcome: (1) due to the inexperience of students, it is very likely that many of the submitted codes are incorrect; and (2) correct codes might not be efficient. In this paper, we focus on the first challenge: how to maximize the functionality of the software instead of its performance.

Due to budget and time constraints, the system must require minimum human intervention to detect the correct codes for a given task. That is, lecturers and teaching assistants should not spend time detecting the correct codes, reducing the need for expert-based correctness detection to special cases only. As most current correctness detection systems used in crowdsourcing systems require human intervention or knowing the right answers beforehand [1,2,5,12,16,19], they cannot be applied to this system.

In this paper, we propose a novel way of detecting the correctness of submitted codes. The system does not resort to a human evaluation of codes and does not need to know the set of right answers beforehand. Instead, it is based on the analysis of the output data generated by each submitted code. Such data are processed and clustered based on their similarity. The cluster with the highest number of submissions is considered to contain the correct codes that will be used for the next stages of software development. We tested the system with four different sub-tasks given to 1st year engineering students taking the course of Computer Programming during the 1st academic term of 2016 at Universidad Adolfo Ibáñez, Chile. Preliminary results show that the system is highly accurate in detecting the set of correct codes when dishonesty levels and the complexity of the task are low.

The rest of this paper is as follows: Section 2 reviews related work, Section 3 discusses the principle upon which the code correctness detection system is built and its implementation details, Section 4 describes the methodology used to test the system while Section 5 presents and discuss the results obtained. Section 6 concludes the paper.

2 | RELATED WORK

In the area of crowdsourcing, several correctness detection systems have been used. In the following, we describe the most common techniques used, as reported in references [1,2,5,12,16,19], adapted to the perspective of a crowdcoding task in an academic environment.

2.1 | Expert review

An expert (or team of experts using consensus majority) analyzes all the submitted codes to discard the incorrect ones and determine the best implementation for the required task.

2.2 | Real-time support

Workers (students) receive anonymous real-time expert feedback while they are still working on the task (synchronous feedback) or after delivering their first version of their code (asynchronous feedback). Based on such feedback, the final code can be improved. Self-assessment from workers was shown to achieve similar quality than external assessment [3], but these results were not obtained for coding tasks.

In an educative context, previous approaches follow the classical scheme of lecturers or teaching assistants reviewing the submitted work. Nevertheless, given that the software is built with a client in mind, the review process will overwhelm lecturers. We aim to decrease the time devoted to this task to the minimum. Additionally, beginner programming students might find difficult to self-assess their code if we want to perform a real-time support among peers.

2.3 | Workflow management

Quality control tasks are added to the set of sub-tasks required to complete the project [6]. Examples of quality control tasks are: another set of workers improve the work submitted by the original developers (by merging several works or adding new material), vote for the best piece of work or review and mark the work submitted (also known as crowdsourcing testing). Evaluating the impact of such technique in an educational environment (where the workers might be programmers at the beginner level), has not been assessed yet.

2.4 | Contributor evaluation

It relies on the quality of the developer. That is, it works under the assumption that if the developer is competent, then the submission is of excellent quality with high probability. Translated to an educational environment, it would mean that the student with the best marks would submit the best codes with high probability. Such approach would discourage non-top students from submitting their codes, which is one educational objective of the educational crowdcoding.

2.5 | Output agreement

Mostly used for tasks as tagging or describing images, videos or clips [10,18], workers independently provide a description for a specific data they are given. If workers agree on their output (using the same keywords to describe the data presented to them), such description is considered correct. Taken into the educational system, this technique would require (at least) two persons to look at the output data of the same code and based on the data, decide whether the code is correct or not. If both determine the code to be correct, then the code is classified as such. Again, this system is intensive in human intervention.

2.6 | Input agreement

Originally used to tag music, workers simultaneously receive an input (a tune) and must describe it [9]. The descriptions entered by the workers are shared among them. Based on that information, workers must decide whether they were given the same input (tune). If they decide so and the tune was the same, the descriptors are accepted as a quality answer. As this system generates descriptors rather than classifications, there is no direct translation of this scheme to code correctness detection.

2.7 | Ground truth

Answers given by the workers (i.e., the output data generated by the submitted codes) are compared to the "right answer" that defines the required correctness. This scheme is the approach used by automated tests, where a test for each of the requested features is created specifying the output value that must be returned by the code for each given input. In this case, the right answers must be known beforehand and prepared by an expert. For tasks where the right answers are not known, this technique is not feasible. The latter may be frequent in our case since problems come from real-life, and the lecturers may not be experts in the field.

2.8 | Spammers removal

A pre-processing of the submitted codes is carried out to remove submissions from spammers who intentionally upload bad quality work. This is a reality in systems that offer economic incentives, but it should not be a significant concern in academic environments.

2.9 | Behavior analysis

The actions of the workers during the task are monitored (e.g., number of clicks, number of mouse movements, number of checkboxes accessed, time to complete the task) and the quality of their work is inferred based on their behavior. The system has been tested in web search tasks where a supervised classifier is trained using the behavior of professional judges [5]. No studies have been performed to extend this approach to the context of academic crowdcoding.

Given that none of the mentioned approaches satisfy our requirement of minimum human intervention (except the last one, but only once the training stage has been already carried out), we developed a new system for code correctness detection. The new system can is an automated version of an output agreement system, and it is described in the next section.

3 | PROPOSED SYSTEM

3.1 | The principle behind the system

The correctness detection system is built based on an observation made through many years of teaching computing programming to first-year students and supported by other researchers [14]: inexpert programmers make different types of mistakes. As a result, the set of codes generated by such programmers tends to exhibit a high variability in terms of outputs for the same input. In other words, all correct codes generate the same output for a given input. Instead, for a code to be incorrect, any combination of mistakes can be made.

If we group correct and incorrect codes in different clusters, we hypothesized that: (1) the incorrect codes will be spread in different smaller clusters given the high number of combinations of mistakes, and (2) the cluster with the highest number of codes will contain the set of correct codes.

Formally, if $M = \{m_1, m_2, \ldots, m_{|M|}\}$ is the set of all possible mistakes that can be present in a given code, for a code to be correct the condition ($!m_1$ AND $!m_2$ AND $!m_3 \ldots$ AND $!m_{|M|}$) must be met, while for a code to be incorrect, the condition (m_1 OR m_2 OR $m_3 \ldots$ OR $m_{|M|}$) must be met. Therefore, depending on the combination of mistakes made several different outputs can be generated by the set of incorrect codes. For simplicity, from now on the expression "combination of mistakes" will be replaced by the word "error." The condition for our hypothesis to be valid can be written as:

$$E[N_c] > E[N_i],\ \forall i \in \{1, 2, \ldots, \Omega\} \tag{1}$$

where N_c is the random variable denoting the number of correct codes, N_i the random variable denoting the number of incorrect codes generating the same output due to error of type i, Ω is the number of different incorrect outputs (that is, the number of errors) and E[x] denotes the expectation of random variable x.

Let p_i be the probability of the i-th error taking place (i.e., the probability of the i-th incorrect output). Thus, the condition stated in Equation (1) can be re-written as:

$$N \prod_{i=1}^{\Omega} (1 - p_i) > Np_i,\ \forall i \in \{1, 2, \ldots, \Omega\} \tag{2}$$

where N is the number of codes analyzed.

By assuming a worst-case condition (i.e., all errors have the same probability, equal to the highest value, $\max\{p_i\}$), Equation (2) can be numerically solved for different values of Ω and $\max\{p_i\}$. Figure 1 shows the relationship between these values.

It can be seen that, for our hypothesis to be valid, the probability of any error must be lower than 0.5. Otherwise, even with just one type of error, the number of codes in the cluster with the correct ones will not exceed the number of codes in the cluster with the incorrect ones. As the number of possible errors increase (typically, with the complexity of the code), the probability of error must decrease significantly for the biggest cluster to contain the correct codes. For example, for a code with 20 different types of errors, the probability of error must be lower than 0.1 to make sure that the biggest cluster contains the correct codes.

Under an ideal situation, the probability of error should be low. However, there are conditions for which such value could increase over the required threshold: misunderstanding of concepts taught in lessons, misunderstanding of the assignment, ambiguous assignment requirements, high complexity tasks or cheating might lead to many codes with the same incorrect output. For the first case, for example, Pea et al. [14] show that novice programmers (in our case, students) made errors commonly when creating code since they behave as if they were talking to a person, instead of programming a machine. For all cases, preventive measures can be taken: take small formative quizzes to make sure students have understood the core concepts taught in lessons, make sure the micro-task is well defined using examples of output data in normal and extreme cases, keep the complexity of tasks low and warn the students that submissions will be automatically compared to each other for copy detection. Without such preventive measures in place, the system might not work properly.

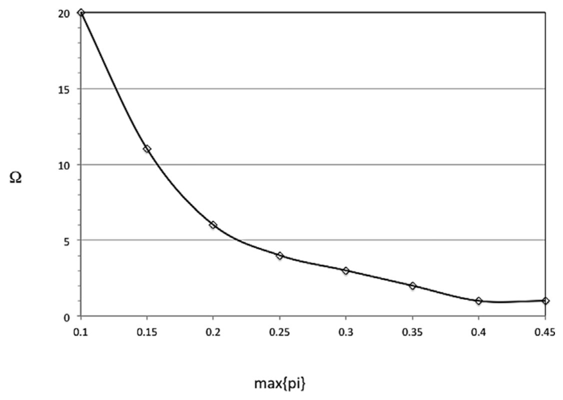

FIGURE 1 Maximum value of error probability versus the total number of errors

3.2 | Implemented system

The clustering of codes according to their output data was carried out as shown in the schematic of Figure 2. First, all submitted codes are collected for further processing (Stage 1, Code collection). Next, every code is compiled (this first version of the system was built to work with C source codes) and fed with the same set of different input data. The output data are stored in a text file. Thus, there is a different output data file per code submitted (Stage 2, Automatic generation of output files). Finally, the code clustering process is carried out by storing every set of identical text files in a specific folder (Stage 3, Code clustering). The implementation details of each stage are described in the following.

3.2.1 | Stage 1

The code collection was performed using a Moodle platform where the students must upload the source codes generated.

3.2.2 | Stage 2

The code is pre-processed to standardize all the output. This standardization consists on modifying all the calls to the printf

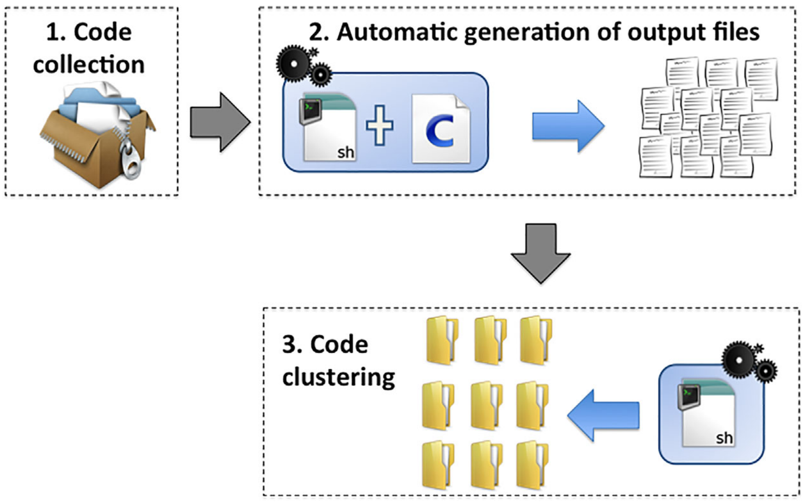

FIGURE 2 Schematic of code correctness detection system

method to remove all the customized text (letters, spaces, and numbers) coded by the students so only the output of the algorithms is left as the output, with no extra text.

Next, a C program (flow diagram of Figure 3), was built to compile each submitted code, inject different input data to it and store the output data to a single file (one file with all output data for each submitted code).

As a result of this stage, as many text files as codes uploaded by the students were generated (except for those codes that could not be compiled). These text files, containing the output data generated by each submitted code, were assigned the extension .tmp.

3.2.3 | Stage 3

Once the files with the output data of each code were generated, the process of code clustering (i.e., grouping the different files in folders according to their content) starts. The clustering is performed by running a script that checks for difference between two files. If there are no differences then the code are considered to be equivalent.

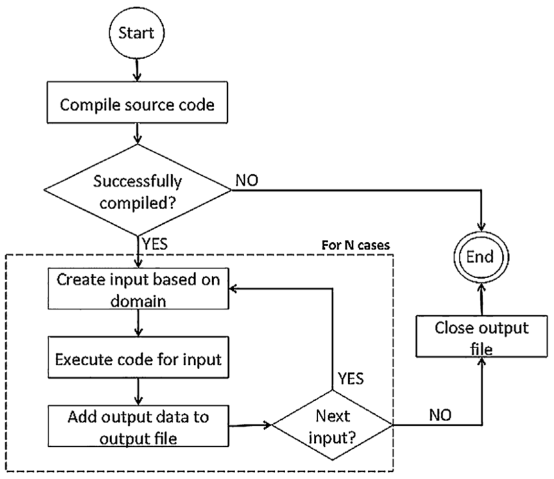

FIGURE 3 Flowchart describing the output data generation process

The previous stages are implemented as the following shell script:

```
#!/bin/bash

containsElement () {
  local e
  for e in ${@:2}; do [[$e == $1]] && return 1; done
  return 0
}

#pre-process#
mkdir p processed
find . name *.c | sed s|^\./||| xargs I myfile spatch no-show-
diff sp-file printf.cocci myfile o processed/pf_myfile

#run#
cd processed
cp f ../Makefile .
find . name pf_*.c exec sh c make ${0%.c} {} \;

#output#
mkdir p output
for i in pf_*.c
   do
   ./GenOutput $i
   done

#cluster#
cd output
clustered = ()
for f in *.output; do
   containsElement $f ${clustered[@]}
   if [$? == 0]; then
```

```
    clustered+=($f)
    touch ${f%.output}.cluster
    for g in *.output; do
      containsElement $g ${clustered[@]}
      if [$? ==0 a $f != $g]; then
        echo $f vs $g
        if diff w i B $f $g>/dev/null; then
          echo $g>> ${f%.output}.cluster
          clustered+=($g)
        fi
      fi
    done
  fi
done
```

4 | METHODOLOGY

To check whether the correct codes were stored in the folder with the highest number of files for different situations, we performed four tests with different code assignments given to the first-year engineering students of the Computer Programming course at Universidad Adolfo Ibáñez. As preventive measures, we announced that an automated copy detection system would be used, we defined simple assignments (except for the 4th assignment) as they represented the micro-tasks of a bigger project, we provided a set of examples on the output data and we held practical sessions where students were monitored for eventual misconceptions. The 4th assignment was of higher complexity, so we could evaluate the impact of complexity on the effectiveness of the system.

The first assignment (A1) consisted on displaying the left half of a triangle made of symbols #, where the height of the triangle was a parameter entered by the user (with a maximum value of 23). For example, for a height equal to four, the following figure had to be printed:

```
   #
  ##
 ###
####
```

The second assignment (A2) consisted on encrypting a sentence entered by the user using Caesar cipher (a shift cipher). The user also entered the encryption key. Symbols different from English alphabet should not be encrypted. The third assignment (A3) consisted of building a hybrid sorting algorithm (a mixture between bubble and selection sort). Finally, the fourth assignment (A4) consisted on the development of a set of sorting algorithms to order data entered by the user. The data should be ordered using three sorting algorithms: bubble, selection, and insertion sort. The output of the algorithm was the number of comparisons carried out by each algorithm. This experiment was of higher difficulty (thus, it cannot be considered as a sub-task) and it was optional for students that wanted to tackle it. At the Viña del Mar campus, 12 students participated.

We evaluated 327 A1-codes from students from Campus Peñalolén, 97 A2-codes from students from Campus Viña del Mar, 75 A3-codes from students from Campus Viña del Mar, 193 A3-codes from students from Campus Peñalolén and 12 A4-codes from students from Campus Viña del Mar.

Due to the performance limitations imposed by the machine used to run the correctness test, every assignment was divided into groups of 60 codes, randomly chosen. For A1 we generated ten groups of 60 codes each. For A2 we analyzed three groups of 60 codes each. For A3 we generated two and six groups for the set of 75 and 193 A3-codes, respectively.

TABLE 1 Results for A1-codes

Group	NC1	NC2	NC3	N_{Human}	N_{Auto}	Human (%)	Automated (%)
1	**14**	11	10	14	13	100	100
2	17	**11**	0	11	10	0	0
3	12	19	**13**	13	12	0	0
4	19	**9**	9	9	9	0	0
5	17	**15**	9	15	13	0	0
6	17	**11**	11	11	10	0	0
7	20	**17**	0	17	15	0	0
8	**12**	12	10	12	9	100	100
9	19	**14**	0	14	13	100	100
10	13	12	**9**	9	8	0	0

For each group, the value highlighted with bold font identifies the cluster with the actual correct codes (e.g., Cluster 1 in Group 1).

TABLE 2 Percentage of codes detected as copy

Group 1		Group 2		Group 3		Group 4		Group 5	
1 (14)	**7%**	1 (17)	35%	1 (19)	0%	1 (19)	26%	1 (17)	59%
		2 (11)	**0%**	**2 (13)**	**0%**	**2 (9)**	**0%**	**2 (15)**	**27%**
Group 6		**Group 7**		**Group 8**		**Group 9**		**Group 10**	
1 (17)	65%	1 (20)	50%	**1 (12)**	**17%**	1 (19)	37%	1 (13)	8%
2 (11)	**0%**	**2 (17)**	**12%**	2 (12)	8%	**2 (24)**	**7%**	**3 (9)**	**0%**

The folder with the set of correct codes is highlighted with bold font.

As a way of comparison, a human expert and an automatic test also checked the codes for correctness. The automated test compares the output of the code submitted by the student to the output generated by a code created by a senior programmer.

5 | RESULTS

In the following, the results obtained after applying the proposed system (code clustering), the automated test and the human review to the different sets of codes are presented.

5.1 | Results for A1-codes

Table 1 shows the results obtained for the ten groups made of 60 randomly selected student's codes. For each group, only 2 or 3 clusters with similar number of codes were generated. We denote by C1 the cluster with the highest number of codes, by C2 the cluster with the second highest number of codes and by C3 the cluster with the lowest number of codes. The columns of Table 1 represent the following characteristics of each group:

- Columns NC1, NC2, and NC3 show the number of codes in clusters C1, C2, and C3, respectively.
- Columns N_{Human} and N_{Auto} correspond to the number of codes detected as correct by a human reviewer and the automated test, respectively.
- Columns Human and Automated show the percentage of codes detected as correct by the human and the automated revision that are also present in C1.

For each group, the value highlighted with bold font identifies the cluster with the actual correct codes (e.g., Cluster 1 in Group 1).

From these results, it can be seen that in only two out of ten clusters the folder with the highest number of codes (C1) contained the correct codes. This finding highlights the fact that the same type of error was made by many students, increasing its probability beyond the value for which our hypothesis is valid.

To investigate whether a cheating condition was present (that would increase the probability of occurrence of the same type of error), we analyzed the codes using the MOSS (Measure of Software Similarity) system. MOSS is a platform developed at Stanford University to detect syntactic and semantic similarities between codes [13].

Results are shown in Table 2. For each group, the percentage of copy in the folder with the highest number of codes and the folder with the actual correct codes is listed. We present two columns: (1) one with the identification of the folders generated by the clustering method along with the number of codes in that folder; and (2) another with the corresponding percentage of codes in that folder that were classified as copies by the MOSS system. The folder with the set of correct codes is highlighted with bold font.

It can be seen that the average percentage of detected copied codes in the folders with the highest number of codes is equal to 30%, as opposed to just 6.1% in the folders with the correct codes. In this case, it seems that by copying the same mistakes, students artificially increased the number of incorrect codes with the same output, distorting the results. As the academic sanction for cheating was high, we expect the "copy effect" to decrease in subsequent assignments.

5.2 | Results for A2-codes

In this case, for the three groups of codes analyzed, there was a single folder with a number of codes significantly higher

TABLE 3 Results for A2-codes

Group	NC1	N_{Human}	N_{Auto}	Human (%)	Automated (%)
1	**34**	34	32	100	100
2	**32**	32	30	100	100
3	**41**	41	39	100	100

For each group, the value highlighted with bold font identifies the cluster with the actual correct codes (e.g., Cluster 1 in Group 1).

TABLE 4 Results for A3-codes, Campus Viña del Mar

Group	NC1	N_{Human}	N_{Auto}	Human (%)	Automated (%)
1	**12**	12	12	100	100
2	**15**	15	15	100	100

For each group, the value highlighted with bold font identifies the cluster with the actual correct codes (e.g., Cluster 1 in Group 1).

TABLE 5 Results for A3-codes, Campus Peñalolén

Group	NC1	N_{Human}	N_{Auto}	Human (%)	Automated (%)
1	**16**	16	14	100	100
2	**11**	11	9	100	100
3	**16**	16	15	100	100
4	**10**	10	9	100	100
5	**13**	13	12	100	100
6	**14**	14	13	100	100

For each group, the value highlighted with bold font identifies the cluster with the actual correct codes (e.g., Cluster 1 in Group 1).

than the rest. Table 3 shows the results obtained for the three groups made of 60 randomly selected codes.

With the A2-codes the hypothesis was completely validated, as the folder with the highest number of codes was also the folder with the correct codes. As expected, due to the strict academic sanction for cheating, the percentage of copied codes dropped significantly with just a 5% of codes detected as highly similar by the MOSS system (in average, 8%, 0%, and 7% for groups 1, 2, and 3, respectively). This highlights the fact that the level of cheating affects the accuracy of the system, dependent on low values of probability of error for the hypothesis to hold.

5.3 | Results for A3-codes

In this case, as with the A2-codes, there was a single folder with a significantly higher number of codes than the rest. Tables 4 and 5 show the results for the three correctness assessment methods for the Viña del Mar and Peñalolén campuses, respectively. Once again, the biggest cluster contained the correct codes.

5.4 | Results for A4-codes

Six clusters were formed after analyzing the outputs. From them, only one cluster was composed of more than one element, but that cluster did not contain the correct codes. From the clusters of size one, two of them contained the correct codes. The difference between these two submissions was that one student added the sorted data as output. Otherwise they were equivalents. This last experiment shows that the complexity of the task it is another aspect that must be carefully controlled for the biggest cluster to contain the correct codes.

6 | CONCLUSIONS

In this paper, we report on preliminary results on the use of an automatic system to evaluate the correctness of codes written by first-year engineering students. The novelty of the scheme lies in not requiring human intervention; neither knowing the correct output before hand. The core assumption for the effectiveness of the system is that the reasons behind an incorrect code are diverse (different mistake combinations generate different incorrect codes). Instead, correct codes must comply with every single requirement to function properly. Hence, when clustering codes according to their outputs (codes with the same outputs belong to a cluster), incorrect codes spread across several different clusters while all correct codes belong to a single cluster, allegedly the biggest cluster.

However, for this assumption to apply (correct codes are found in the biggest cluster), several conditions must be met: cheating must be minimized, the complexity of the tasks must be kept low (so they can really be considered as micro-tasks), requirements must be clear and misconceptions must be removed before performing the task.

We have evaluated the accuracy of the system by evaluating 5 groups of codes. We have found out that the system was able to detect the right set of codes in 3 out of 5 preliminary experiments. In the experiments where the system was not successful, some of the required conditions did not apply. In one case it seems that the high level of copy distorted the results (as the probability of unlikely errors was artificially increased) and in the other case, the complexity of the task increased the probability of errors over the threshold and decreased the number of submissions. Even in these cases, it is worth noticing that the correction detection is significantly simplified, as the reviewer needs to review just a few folders instead of each code separately.

Further research will focus on extending the experiments to a higher number of assignments to increase the statistical validity of results and including performance tests to detect the most efficient among the selected correct codes.

REFERENCES

1. M. Allahbakhsh, et al., *Quality control in crowdsourcing systems*, IEEE Internet Comput. **17** (2013), 76–81.
2. O. Alonso, *Implementing crowdsourcing-based relevance experimentation: An industrial perspective*, Inf. Retr. **16** (2013), 101–120.
3. S.P. Dow, et al., Sheperding the crowd yields better work. 2012 ACM Conf. Computer Supported Cooperative Work, Seattle, USA, 2012, pp. 1013–1022
4. M. Hossain, I. Kauranen, *Crowdsourcing: A comprehensive literature review*, Strat. Outsourc. **8** (2015), 2–22.
5. G. Kaza, I. Zitouni, "Quality management in crowdsourcing using gold judges behavior", Proceedings of the 9th ACM International Conference on Web Search and Data Mining, San Francisco, California, USA, 2016, pp 22–25.

6. A. Kittur, B. Smus, R. E. Kraut, CrowdForge: Crowd-sourcing Complex Work. 24th Ann. ACM Symp. User Interface Software and Technology, Santa Barbara, USA, 2011, pp. 43–52.
7. K.R. Lakhani, D. A. Garvin, E. Lonstein, *TopCoder(A): Developing software through crowdsourcing*, Harvard Business School Case, Boston, 2010.
8. T. D. LaToza, et al., Microtask programming: Building software with a crowd. 27th Annual ACM Symposium on User Interface Software and Technology. New York, USA, 2014. pp. 43–54.
9. E. Law, L. von Ahn, Input-Agreement: A New Mechanism for Collecting Data Using Human Computation Games. SIGCHI Conference on Human Factors in Computing Systems, Boston, USA, 2009, pp. 1197–1206.
10. E. Law, et al., TagATune: A Game for Music and Sound Annotation. 8th International Conference on Music Information Retrieval, Vienna, Austria, 2007, pp. 361–364.
11. K. Li, et al., "Analysis of the Key Factors for Software Quality in Crowdsourcing Development: An Empirical Study on TopCoder.com," IEEE 37th Annual Computer Software and Applications Conference, Kyoto, Japan, 2013, pp. 812–817.
12. K. Mao, et al., *A survey of the use of crowdsourcing in software engineering*, J. Syst. Software **126** (2017), 57–84.
13. MOSS [Internet]. Available online at: http://moss.stanford.edu. Last access: 7th August 2017.
14. R. D. Pea, *Language-Independent conceptual 'bugs' in novice programming*, J. Educ. Comput. Res. **2** (1986), 25–36.
15. P. A. Rosen, *Crowdsourcing lessons for organizations*, J. Decis. Syst. **20** (2011), 309–324.
16. M. Saengkhatiyya, M. Sevandersson, U. Vallejo, *Quality in crowdsourcing − How software quality is ensured in software crowdsourcing*. Master Thesis. Lund University.
17. TopCoder [Internet]. Available online at: https://www.topcoder.com/tc?module = CompList&ph = 125. Last access: 7th August 2017.
18. L. Von Ahn, L. Dabbish, Labeling images with a computer game. SIGCHI Conference on Human Factors in Computing Systems, Vienna, Austria, 2004, pp. 319–326.
19. J.B.P. Vuurens, A.P. de Vries, *Obtaining high-quality relevance judgements using crowdsourcing*, IEEE Internet. Comput. **16**, 20–27.

S. Ferrán obtained the professional title of informatics engineer from Universidad Adolfo Ibañez in 2016. He now works as a financial risk analyst in banking.

A. Beghelli received the BEng and MSc degrees in Electronic Engineering from Universidad Técnica Federico Santa Maria (UTFSM, Valparaiso, Chile) in 1993 and 2001, respectively, and the PhD degree from University College London, UK, in 2006. Currently, she works as senior lecturer in the Faculty of Engineering and Sciences at Universidad Adolfo Ibañez. She has received several awards: Best Student, promotion 1996 (UTFSM Alumni Association), Best Female Graduate of UTFSM (Zonta International, 1996), the M.H. Joseph Prize for academic excellence in the field of Engineering (British Federation of Women Graduates, 2003), LEOS Graduate Student Fellowship (IEEE Lasers and Electro-Optics Society, 2004), Excellence in Teaching (UTFSM 2006, 2007, 2008, 2009), and the Universidad Adolfo Ibañez Award for Academic Contribution (2015). Her research interests are in resource allocation in optical networks, engineering education, and design engineering. She has authored or co-authored more than 80 refereed journal and conference papers.

G. Huerta-Cánepa is a professor in the Faculty of Engineering and Science at the Universidad Adolfo Ibáñez. Dr. Huerta-Cánepa holds a PhD degree in Information and Communications from KAIST, South Korea. His primary research interest is the Internet of Things and its application to daily life, from education to e-health. He has received best paper awards in conferences and journals and was the recipient of the first Qualcomm fellowship award in Korea, in 2011. Besides his research interests, he loves programming and has participated in several programming contests, and he is trying to merge this passion with his research topics.

F. Jensen is pursuing his undergraduate studies in Computer Science at Universidad Adolfo Ibañez. He works as a software developer and as a research assistant in the Applied Bioengineering Laboratory developing biometric sensors.

How to cite this article: Ferrán S, Beghelli A, Huerta-Cánepa G, Jensen F. Correctness assessment of a crowdcoding project in a computer programming introductory course. *Comput Appl Eng Educ*. 2018;26:162–170. https://doi.org/10.1002/cae.21868