2016 IEEE 40th Annual Computer Software and Applications Conference

Toward More Robust Automatic Analysis of Student Program Outputs for Assessment and Learning

Chung Keung Poon
School of Computing and Information Sciences
Caritas Institute of Higher Education
Hong Kong
ckpoon@cihe.edu.hk

Tak-Lam Wong
Department of Mathematics and Information Technology
The Hong Kong Institute of Education
Hong Kong
tlwong@ied.edu.hk

Y. T. Yu, Victor C. S. Lee, Chung Man Tang
Department of Computer Science
City University of Hong Kong
Hong Kong
{csytyu, csvlee}@cityu.edu.hk, c.m.tang@my.cityu.edu.hk

***Abstract*—Automated analysis and assessment of students' programs, typically implemented in automated program assessment systems (APASs), are very helpful to both students and instructors in modern day computer programming classes. The mainstream of APASs employs a black-box testing approach which compares students' program outputs with instructor-prepared outputs. A common weakness of existing APASs is their inflexibility and limited capability to deal with admissible output variants, that is, outputs produced by acceptable correct programs that differ from the instructor's. This paper proposes a more robust framework for automatically modelling and analysing student program output variations based on a novel hierarchical program output structure called HiPOS. Our framework assesses student programs by means of a set of matching rules tagged to the HiPOS, which produces a better verdict of correctness. We also demonstrate the capability of our framework by means of a pilot case study using real student programs.**

Keywords- automated assessment technology; computer science education; learning computer programming; program output variant; student program analysis

I. INTRODUCTION

Computer science educators have unequivocally reported the many facets of difficulties in the teaching and learning of computer programming [2][19]. One of the key success factors to overcome these difficulties is to provide extensive exercises for students to practise their coding and debugging skills [11][16][19]. While setting appropriate exercises and practice tasks for students is by no means easy, assessment of students' work and provision of timely and quality feedback are even more challenging due to the sheer volume of effort involved [8][18][26]. In response, universities worldwide have developed *automated program assessment systems* (APASs) [6][12][30], which are now routinely used in conventional classes as well as personalized and blended learning courses [24].

Different APASs vary in their designs and features, but the majority of them possess the core function of assessing the functional correctness of students' programs. This is typically done by automatically executing students' programs against a suite of pre-defined test cases and comparing the programs' actual outputs with the instructor's expected outputs. The latter task requires a mechanism, technically known as a *test oracle*, for determining the correctness of program outputs [27]. In the field of software testing, the general problem of *test oracle automation* is well known to be challenging. Implementation of test oracles in existing APASs is often too simplistic, rigid and incapable of being tailored to support the intended educational outcomes of the exercises [7][25]. This is because multiple correct (or *admissible*) programming solutions to an exercise may produce different outputs (called *output variants*). A program which the human instructor accepts to be correct (or admissible) could be inappropriately rejected by a rigid test oracle in an APAS. This phenomenon is common and has been a root cause of many educationally undesirable effects on teaching and learning that can substantially compromise the benefits of an APAS in practice [2][11][21][25][30]. Such a technical limitation of APASs is precisely the research problem that we are going to address in this paper.

The rest of this paper is organized as follows. Section II reviews current practice and related work. In Section III, we propose a robust framework for automatically modelling and analysing student program output variations based on a novel structure called HiPOS that has been inspired by research work in the areas of natural language processing and information retrieval. Our framework assesses student programs by using a set of matching rules which produces a better verdict of program output correctness. Section IV presents a pilot case study that demonstrates the capability of our framework applied to the assessment of real student programs with encouraging results. Finally, Section V concludes this paper.

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.208

II. CURRENT PRACTICE AND RELATED WORK

A. The Test Oracle Problem in APASs

Previous research has studied the technical issues in core APAS functions (such as course administration, assignment and feedback management) [6] as well as explored means of exploiting an APAS or extending its functions to improve teaching and learning. Some of the researched issues are plagiarism detection [14], grading styles [11], authoring and design of exercises [21], assessment of students' testing skill [29], and generation of feedback to students [8][16], etc.

In this paper, our focus is on the assessment of functional correctness of students' programs. Existing APASs usually assess the correctness of the student's program by executing it with a suite of instructor's tests. The program is deemed correct if its (actual) output in every test run is accepted to be correct. A mechanism to determine the correctness of outputs is known as a *test oracle* [27]. Generally, test oracle automation is very challenging. But in assessing student programs, the instructor normally has either a correct program or the exact correct outputs in mind. As such, the oracle problem is typically circumvented by using the *output comparison method* [2][12][19], which matches the actual output texts with the correct ones. Most of the well-known APASs, such as BOSS [14], Ceilidh/CourseMarker [11], Curator [4], HoGG [18], PASS [30] and PETCHA [21], use this method in either its rudimentary or an adapted form. The output comparison method not only works for text-based programs which return texts as outputs, it may also be adapted to non-text-based programs by using wrapper code to convert their outputs into text strings [18]. Other forms of outputs, such as GUI, can be handled in different ways [2].

While the output comparison method generally works well for exercises with unambiguously correct outputs, it can also be unsatisfactory for exercises with multiple correct (or *admissible*) solutions which produce *output variants*, that is, outputs that deviate from the expected [25]. For example, students are often insensitive to small deviations in spaces and punctuations. When each actual output deviates from the expected "slightly", an APAS with a naïve test oracle easily rejects the whole program as incorrect. The instructor could then have a hard time explaining why the program is treated as wrong when the outputs are indistinguishable to the student's eye. These cases were common [1][30], causing students frustrations (spending hours on "invisible" bugs), confusions and feeling of being unfairly or harshly treated. They gave comments like "Sometimes it is right to you but wrong to the automark" or "It's over-sensitive" [22]. Some complained their APAS to be "too fussy" or "too picky with spaces" [14], "dangerously precise", "problematic" or "[taking] up much of my time debugging missing spaces in the output". The test oracle problem in APAS is widely recognized, yet seldom formally researched.

B. Strategies Used in Practice

To avoid conflicts with students, many instructors resort to writing "air-tight" program specifications with unusually restrictive and ultra-fine formatting details for ensuring unambiguity of correct outputs [13]. Students are also pre-warned that strict conformance is demanded [4]. Such a strategy may reduce student complaints but not dissatisfaction. It is also counter-educational: it renders the exercises spurious, limits the types of exercises, distracts students from the essentials of the exercise, and inhibits creativity [6][13].

In reviewing many influential APASs, Douce et al. [6] summarized the *main disadvantage* of using an APAS as "the restrictions that apply to what can be assessed automatically, that is, only clearly defined questions with completely specified interface for the overall solution", and that "an assessment engine cannot award additional marks for creative design or innovative solutions". Frontline educators also echoed that the rigidity in output assessment as the *main drawback* of an APAS [1].

Apart from ultra-fine specification, there is a *filtering strategy* which enhances APASs by filtering out certain characters (usually whitespaces or punctuation marks) and performing simple character conversions (such as converting all letters into lowercase) before comparing the outputs [2][14][30]. This strategy is popular due to its technical simplicity. Usually based on ad hoc rules, this strategy is only effective in some cases but not generally. Some researchers adopt an *avoidance strategy*, such as completely avoiding output variants by using a unit testing framework [29] or requesting students to write partial programs (drivers or stubs) so that the APAS only needs to test return values which are either unformatted or uniformly formatted by some custom wrapper code [21]. But strict unambiguity can be impractical (which needs to list all cases), unreasonable (for instance, why is `2.3` admissible but not the more precise value `2.30`?), undesirable (when formatting itself is a goal of the exercise, e.g., "generate a calendar of the month"), expensive (time-consuming to design) and unwelcomed (stripping away creativity and interests) [13].

C. Existing Formal Approaches

Some formal approaches have been proposed to increase the flexibility of output matching in APASs. Some APASs require the instructor to write regular expressions [18], shell scripts [12] or parser scripts using *lex* and *yacc* [13] to recognize admissible variants by pattern matching. This strategy does relax the test oracle's rigidity, but not all instructors would like to write scripts for every exercise. Even if the instructor is competent, the scripts can be error-prone and tedious to write. Also, the limited expressiveness of simple scripting will restrict their matching capability. These drawbacks fundamentally hinder the widespread adoption of such *scripting strategies*.

Recently, finer and more elaborate frameworks have been proposed that promise a better solution to the problem. For example, Tang et al. [25] adopt a *token pattern approach* (TPA) framework to capture the program output structure so that fine-grained matching rules can be designed for the components of the expected output and fed into an APAS for automatic processing without the need to write scripts.

Encouraging success with the use of the TPA framework has been reported in the literature [26]. For example, consider the programming exercise **Ex.1** as follows.

Ex.1: *Write a program that asks the user to enter 3 numbers, obtains the 3 numbers from the user and prints a message showing the average of the 3 numbers.*

When the input is "`56 81 86`", an expected output is:

S^0: "`The average of 3 numbers is 74.33.`"

Apart from S^0, the instructor may also consider the following actual outputs admissible because, to a human interpreter, they have essentially the same meaning as S^0.

S^1: "`Mean of 3 numbers: 74.3333`"

S^2: "`74.33 is the average of 3 numbers.`"

The expected output S^0 has a sentence-like structure with two components containing values that vary among different test cases, namely, "`3`" and "`74.33`". In the TPA framework, the output text is split into component items called *tokens* to capture its meaningful parts of information. A *token pattern* is a string of tokens extracted from the expected output, each token being tagged with its *type*, *value* and associated *matching rules*. Matching rules are criteria for determining correctness when the expected output token is compared with the corresponding actual output token. With suitable choices of matching rules [26] that ignore the non-essential items (such as the word "`is`" and the fullstop "`.`" in S^0,) and allow the use of synonyms (such as "`Mean`" in place of "`average`") and slightly different degrees of precision of the key numeric values, the TPA framework can automatically accept S^1 as an admissible output variant. However, it was observed that some exercises are notably harder to assess automatically than others [23]. Also, modelled as a sequence structure, a token pattern in the TPA framework may be unable to capture the characteristics of certain types of outputs, such as S^2 in which the subject and object of the sentence are reversed. Inspired by the TPA framework, this paper proposes a more robust framework that represents the program output as a tree-like structure based on natural language processing (NLP) models.

A different framework is proposed by Fonte et al. [7] using a *Flexible Dynamic Analyzer* (FDA) which compares the meaning of program outputs and expected outputs by performing a *semantic-similarity analysis*. They designed a domain-specific language called *Output Semantic-Similarity Language* (OSSL) to specify the output structure and semantics as the basis for the desired comparison and awarding partial marks for partially correct outputs. However, there are some unresolved issues in the FDA framework. First, the scalability of the framework is unclear. In general, there can be a large number of admissible output variants for a test case. When a program exercise requires a large number of test cases to verify its correctness, the effort of manually listing all output variants of every test case can be very tedious and easily become intractable. Second, when the output is not just a simple construct (such as sequence or set) but a complex composition of a number of elementary items, it is also unclear how the framework can specify the output structure as well as its different components, both of which in general may vary among different test cases.

TABLE I. TOKENIZATION OF EXPECTED OUTPUT AND OUTPUT VARIANTS

Tokens	tok_1	tok_2	tok_3	tok_4	tok_5	tok_6	tok_7	tok_8
Expected Output S^0	`The`	`average`	`of`	`3`	`numbers`	`is`	`74.33`	`.`
Output Variant S^1	`Mean`	`of`	`3`	`numbers`	`:`	`74.3333`		
Output Variant S^2	`74.33`	`is`	`the`	`average`	`of`	`3`	`numbers`	`.`

III. THE HIPOS FRAMEWORK

A. Problem Definition

Like TPA, our proposed framework also begins with tokenization of the expected output. Let us first present the general problem in a formal notation. Denote the expected output by the sequence representation as

$$S^0 \equiv (tok_1^0, tok_2^0, \ldots, tok_{|S^0|}^0),$$

where tok_j^0, refers to the j-th token of S^0 and $|S^0|$ refers to the length of S^0 (that is, the number of tokens in S^0). We also denote the actual output produced by the program written by student i as S^i, which is an output variant also represented as a sequence of tokens, that is, $S^i \equiv (tok_1^i, tok_2^i, \ldots, tok_{|S^i|}^i)$. TABLE I shows a sample of the tokenized expected output S^0 and output variants S^1 and S^2 for the exercise **Ex.1**.

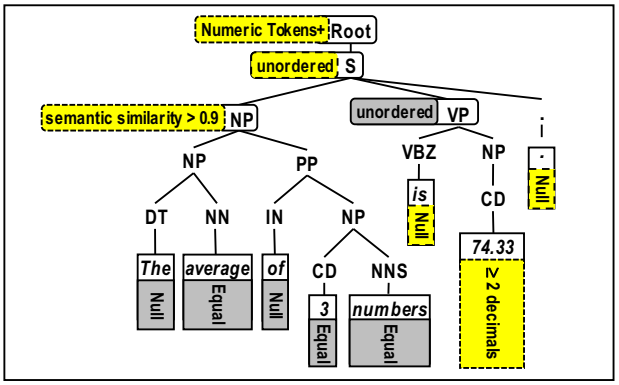
Formally, the test oracle problem for an APAS can be defined as follows: Given an expected program output S^0 and a set of matching rules M, the *Admissible Output Variant Problem* is how to automatically determine the admissibility of an output variant S^i with respect to S^0 and M.

Like TPA, the matching rules in our framework can be customized by individual instructors. Ideally, an APAS should automatically generate default matching rules that specify the instructor's assessment requirements, but a one-size-fits-all default is possible only when all instructors use the same judgment criteria. A recent exploratory study [24] has demonstrated that different instructors may have reasonably good overall agreement in the admissibility of output variants, but the agreement is, unsurprisingly, not 100%. With this regard, a user-friendly and flexible mechanism is desirable for instructors to design and customize the matching rules to suit their teaching needs.

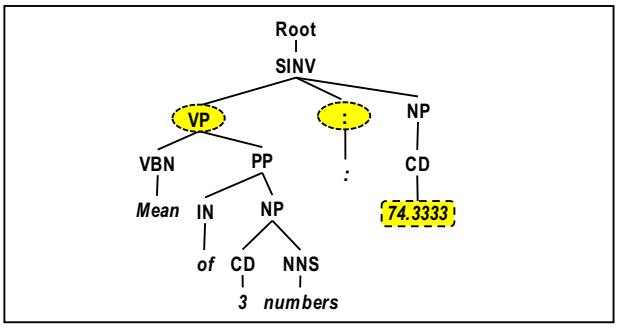
B. HiPOS and Program Output Assessment

To obtain a better verdict of correctness of different output variants, we have designed an ordered tree-like structure called *Hierarchical Program Output Structure* (HiPOS) to model the program outputs. Formally, let the ordered tree H^i be the HiPOS representing a particular program output S^i. Each H^i consists of two different types of nodes, namely, *leaves* and *internal nodes*. Each leaf of H^i, denoted by $Leaf_j^i$, corresponds to the token tok_j^i of the program output. Let $Node_k^i$ be the k-th internal node of H^i. The subtree rooted at an internal node corresponds to the block of tokens that together represent a meaningful part of the program output.

For example, Figure 1 shows the HiPOSs of the expected output S^0 and the output variant S^1. Each internal node of a HiPOS is labelled by a tag in the tagset used for natural language parsing [20]. The parsed tree models the syntactic structure of a sentence in natural language processing (NLP), which is widely used for document summarization [9], information extraction [15] and information retrieval [17]. In particular, our problem bears similarity with Web information extraction because the pages of most commercial Web sites are also automatically generated by computer programs and NLP techniques are commonly utilized to analyze the similarity and difference between Web pages in order to extract useful and relevant information [5]. **Ex.1** requires the learners to write a program to generate a message about the average of three numbers, which is expected to be akin to computer program generated texts. Inspired by the similarity of the tasks, we adopt an automatic natural language parser [3] to construct the two HiPOSs.



(a) HiPOS H^0 of S^0: `The average of 3 numbers is 74.33.`



(b) HiPOS H^1 of S^1: `Mean of 3 numbers: 74.3333.`

Figure 1. The HiPOSs of the expected output S^0 and the output variant S^1.

Note: (1) Each internal node of the HiPOS is labelled by a tag used for natural language parsing. (**S**=Simple declarative clause; **NP**=Noun phrase; **VP**=Verb phrase; **PP**=Personal pronoun; **DT**=Determiner; **NN**=Noun, singular or mass; **IN**=Preposition or subordinating conjunction; **CD**=Cardinal number; **NNS**=Noun, plural; **VBZ**=Verb, 3rd person singular present)

(2) Determination of admissibility of a program output is based on the HiPOS H^0 and tagged matching rules of the expected output. The shaded boxes (yellow or gray) in H^0 refer to the instructor-specified matching rules. The yellow shapes (boxes/ovals) with dotted edges in both HiPOSs refer to the matching rules or nodes triggered for comparison in our examples.

Based on the HiPOS of the expected output, an APAS can automatically generate a set of frequently used matching rules as default, which the instructor may customize if desired, and tag them to the appropriate nodes of the HiPOS.

Instead of comparing the two HiPOSs (in Figure 1) as a whole, which is a tree comparison problem, they will be matched in parts successively. Figure 1(a) shows the HiPOS H^0 of S^0 tagged with matching rules chosen by the instructor. Consider S^1: `"Mean of 3 numbers: 74.3333"`, whose HiPOS H^1 is depicted in Figure 1(b). The matching rule "The result needs to contain at least one numeric token.", denoted as **Numeric Tokens+** and tagged to the root node of H^0, will firstly be triggered to check if S^1 contains at least one numeric token. Our framework will then traverse H^1 using depth-first-search and check if the part of the sentence represented by any subtree of H^1 is similar in semantics to the part `"The average of 3 numbers"` of S^0. Our framework finds that the phrase `"Mean of 3 numbers"`, represented by the subtree of H^1 rooted at the internal node labelled **VP**, satisfies the corresponding rule "The semantic similarity value needs to exceed 0.9.", denoted as **semantic similarity > 0.9** in Figure 1(a), according to [10] (see also the online phrase similarity service provided by UMBC [28]). Next, H^1 is traversed to search for the part `"is 74.33"` (or one with the same semantics) of S^0. Note that the leaf labelled **VBZ** (containing the word `"is"`) serves as a "wildcard" according to the tagged matching rule "This token matches any token, including Null.", denoted as **Null**. Hence, it matches the internal node of H^1 labelled **:**, containing `":"`. Moreover, the leaf of H^0 containing `"74.33"` matches the leaf shown as **74.3333** of H^1 because the latter satisfies the rule "The token needs to match with 2 or more decimal places.", tagged to the former and denoted as **≥ 2 decimals**. Finally, matching of the last leaf of H^0 containing `"."` may be omitted as it is tagged with the rule denoted as **Null** in Figure 1(a). Thus, S^1 will be considered admissible though it is not exactly the same as S^0.

In a similar manner, our framework can automatically determine that S^2: `"74.33 is the average of 3 numbers."` is admissible despite the reversal of subject and object of the sentence, which is allowed according to the matching rule "Allow different orders of the subtrees.", denoted as **unordered** and tagged to the internal node labelled **S** just below the root node of H^0. Limited by space, further details of the comparison are omitted here.

IV. A Case Study

We conducted a case study to evaluate the effectiveness of our framework by comparing its verdicts of admissibility of program outputs with those produced by an existing APAS. We collected 18 programs written by computer science undergraduate students of a local university. They were studying an introductory course on C++ programming. They were required to attend lectures and tutorials every week. In one of the tutorials, students were asked to work on an exercise **Ex.2** about the use of "array" as shown below.

Ex.2: *Write a program such that it first accepts an integer n* (`1 < n ≤ 20`) *which represents the number of students in a class. For each student, read the student's mark m* (`0 ≤ m ≤ 50`). *Store all marks into an array.*

After all marks are read, the program prints the average mark and the bar chart of the marks. You may assume that the input is valid (*that is, data type is correct and the integers are within the valid range*).

Example input:

```
3 30 20 50
```

Example output for the above input:

```
Average = 33.33
******************************
******************
**************************************************
```

Upon completion of this tutorial, students were expected to be able to use one-dimensional array to solve relevant problems. The instructor designed a number of test cases for testing students' programs, including the example input and output shown in **Ex.2**. The existing APAS adopted in this study assessed the correctness of a program by directly comparing the actual output and the expected output, allowing limited variation in the actual output by trimming blanks at the beginning and end of the text string and performing character case conversion before comparison. In parallel, we applied our framework to determine the admissibility of students' program outputs. For each output, the corresponding HiPOS was automatically generated by the natural language parser of our framework. Figure 2 shows the HiPOS of the example output and the instructor-specified matching rules. Although the output was not in the form of a grammatical sentence or paragraph, the natural language parser could still successfully parse it to form an ordered tree-like structure as shown in Figure 2.

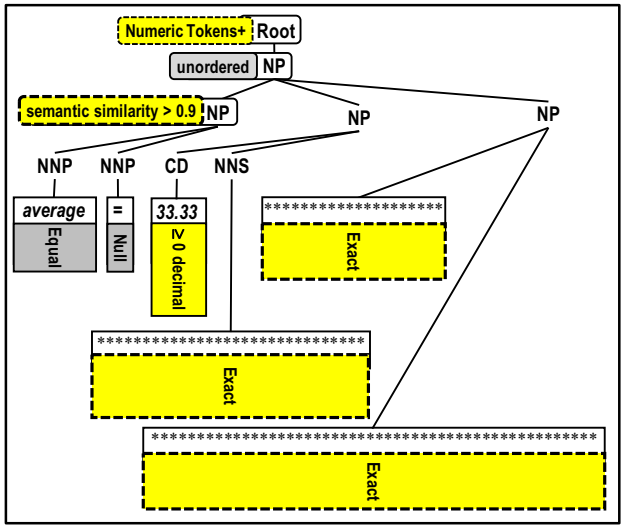


Figure 2. The HiPOS of the example expected output for **Ex. 2**.

Note: See explanatory notes in Figure 1. (**NNP**=Proper noun, singular)

TABLE II. OUTPUT VARIANTS PRODUCED BY STUDENTS' PROGRAMS

Output ID	**Output variants** ('↵' denotes the newline character)

1	`Average = 33.33↵`
	`******************************↵`
	`******************↵`
	`**************************************************↵`

2	`Average = 33.3333↵`
	`******************************↵`
	`******************↵`
	`**************************************************↵`

3	`Average=33.33↵`
	`******************************↵`
	`******************↵`
	`**************************************************↵`

4	`Average = 33↵`
	`******************************↵`
	`******************↵`
	`**************************************************↵`

5	`33↵`
	`******************************↵`
	`******************↵`
	`**************************************************↵`

6	`33.33↵`
	`******************************↵`
	`******************↵`
	`**************************************************↵`

7	`******************************↵`
	`******************↵`
	`**************************************************↵`

8	`Average = 33.33↵`
9	`Average = 33.3333↵`

Among the 18 programs submitted by students, there were 9 different output variants only, as shown in TABLE II. Output 1 was exactly the same as the expected output. Outputs 2, 3 and 4 were all rejected as inadmissible by the existing APAS. However, these output variants were actually admissible as judged by the instructor. Indeed, Output 2 differs from the expected output only by the precision used for printing the average value but still satisfies the specified matching rule denoted as **≥ 0 decimal**. Output 3 deviates from the expected output only by the insignificant formatting spaces around the equality sign. Output 4 was produced by the unexpected use of the integer data type, but this is still acceptable as it is not relevant to the primary objective of the exercise, and the rule only specifies the average value to match with 0 or more decimal places.

Outputs 5 and 6 were "partially admissible" because they satisfy all the matching rules except the one denoted as **semantic similarity > 0.9** and tagged to a node labelled **NP** in Figure 2. However, the instructor could always adjust this matching rule by deleting it or reducing the similarity threshold 0.9 if deemed appropriate according to needs. Output 7 violated the rule denoted as **Numeric Tokens+** and tagged to the root node in Figure 2, and was rightly rejected as inadmissible. Finally, both Outputs 8 and 9 were considered inadmissible because they did not satisfy the matching rules, denoted as **Exact**, that required printing the exact bar charts of the marks.

To conclude, in this case study, our framework could automatically assess the admissibility of output variants in line with what the instructor would have in mind.

V. Conclusion

We have presented a robust and adaptable framework for automatically determining the admissibility of students' submitted programs. We employ a natural language parser to construct a novel Hierarchical Program Output Structure (HiPOS) to effectively capture the relations of the output tokens. The APAS can automatically tag a set of frequently used matching rules to the HiPOS of the expected output as the automated criteria to determine the correctness of students' program outputs. The matching rules can be customized by the instructor to meet their teaching needs. To evaluate our framework, we have conducted a case study to analyse the output variants produced by the programs that were written by a group of computer science students. Compared with the performance of an existing APAS, our framework could more accurately assess the admissibility of output variants in accordance with the instructor's criteria.

Acknowledgment

The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. UGC/FDS11/E02/15).

References

[1] Academia Stack Exchange, Use of automated assessment of programming assignments. Last accessed: 20 December 2015. http://academia.stackexchange.com/questions/20578/use-of-automated-assessment-of-programming-assignments

[2] K. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," Computer Science Education, vol. 15, no. 2, pp. 83–102, 2005.

[3] D. Chen and C.D. Manning, "A fast and accurate dependency parser using neural networks," Proc. Conference on Empirical Methods in Natural Language Processing, pp. 740–750, 2014.

[4] Curator: an Electronic Submission Management Environment, Computer Science @ Virginia Tech. Last accessed: 20 December 2015. http://courses.cs.vt.edu/curator/

[5] B.B. Dalvi, W.W. Cohen, and J. Callan, "Websets: Extracting sets of entities from the web using unsupervised information extraction," Proc. ACM International Conference on Web Search and Data Mining, pp. 243–252, 2012.

[6] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," ACM Journal of Educational Resources in Computing, vol. 5, no. 3, Article 4, 2005.

[7] D. Fonte, D. da Cruz, A.L. Gançarski, and P.R. Henriques, "A flexible dynamic system for automatic grading of programming exercises," Proc. Symposium on Languages, Applications and Technologies (SLATE'13), pp. 129–144, 2013.

[8] S. Gulwani, I. Radiček, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments," Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), pp. 41–51.

[9] U. Hahn and I. Mani, "The challenges of automatic summarization," Computer, vol. 33, no. 11, pp. 29–36, 2000.

[10] L. Han, A. Kashyap, T. Finin, J. Mayfield, and J. Weese, "UMBC EBIQUITY-CORE: Semantic textual similarity systems," Proc. Joint Conference on Lexical and Computational Semantics, pp. 44–52, 2013.

[11] C. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas, "Automated assessment and experiences of teaching programming," ACM Journal on Educational Resources in Computing, vol. 5, no. 3, Article 5, 2005.

[12] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," Proc. Koli Calling International Conference on Computing Education Research (Koli Calling'10), pp. 86–93, 2010.

[13] D. Jackson, "Using software tools to automate the assessment of student programs," Computers and Education, vol. 17, no. 2, pp. 133–143, 1991.

[14] M. Joy, N. Griffiths, and R. Royatt, "The BOSS online submission and assessment system," ACM Journal on Educational Resources in Computing, vol. 5, no. 3, Article 2, 2005.

[15] Y. Jung, K. Stratos, and L.P. Carloni, "LN-Annote: An alternative approach to information extraction from emails using locally-customized named-entity recognition," Proc. International Conference on World Wide Web, pp. 538–548, 2015.

[16] K.M.Y. Law, V.C.S. Lee, and Y.T. Yu, "Learning motivation in e-learning facilitated computer programming courses," Computers and Education, vol. 55, no. 1, pp. 218–228, 2010.

[17] D.D. Lewis and K.S. Jones, "Natural language processing for information retrieval," Communications of the ACM, vol. 39, no. 1, pp. 92–101, 1996.

[18] D.S. Morris, "Automatic grading of student's programming assignments: An interactive process and suite of programs," Proc. ASEE/IEEE Frontiers in Education Conference (FIE 2003), pp. S3F-1–6.

[19] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devin, and J. Paterson, "A survey of literature on the teaching of introductory programming," ACM SIGCSE Bulletin, vol. 39, no. 4, pp. 204–223, 2007.

[20] The Penn Treebank Project. Last accessed: 20 December 2015. https://www.cis.upenn.edu/~treebank/

[21] R. Queirós and J.P. Leal, "PETCHA – A programming exercises teaching assistant," Proc. Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2012), pp. 192–197.

[22] H. Suleman, "Automatic marking with Sakai," Proc. Research Conference of the South African Institute of Computer Scientists and Information Technologists, pp. 229–236, 2008.

[23] C.M. Tang, Automated Testing of Student Programs using Token Patterns, MPhil thesis, Department of Computer Science, City University of Hong Kong, June 2011.

[24] C.M. Tang and Y.T. Yu, "An exploratory study on instructors' agreement on the correctness of computer program outputs," Proc. International Conference on Hybrid Learning (ICHL 2013), Lecture Notes in Computer Science (LNCS), vol. 8038, pp. 69–80.

[25] C.M. Tang, Y.T. Yu and C.K. Poon, "An approach towards automatic testing of student programs using token patterns," Proc. International Conference on Computers in Education (ICCE 2009), pp. 188–190.

[26] C.M. Tang, Y.T. Yu, and C.K. Poon, "An experimental prototype for automatically testing student programs using token patterns," Proc. International Conference on Computer Supported Education (CSEDU 2010), pp. 144–149.

[27] T.H. Tse, F.C.M. Lau, W.K. Chan, P.C.K. Liu, and C.K.F. Luk, "Testing object-oriented industrial software without precise oracles or results," Communications of the ACM, vol. 50, no. 8, pp. 78–85, 2007.

[28] UMBC Phrase Similarity Service. Last accessed: 20 December 2015. http://swoogle.umbc.edu/SimService/phrase_similarity.html

[29] Web-CAT, The Web-CAT Community. Last accessed: 20 December 2015. http://web-cat.org/

[30] Y.T. Yu, C.K. Poon, and M. Choy, "Experiences with PASS: Developing and using a Programming Assignment aSsessment System," Proc. International Conference on Quality Software (QSIC 2006), pp. 360–365.