eOS: The Exercise Operating System

Rui Mendes
Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
azuki@di.uminho.pt
https://orcid.org/0000-0002-5321-6863

José João Almeida
Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
jj@di.uminho.pt
https://orcid.org/0000-0002-0722-2031

Abstract

We present an architecture for a system for creating, adapting and evaluating programming exercises for students. The system is capable of generating exercise skeletons, automatically creating inputs and outputs, provide a way of creating a large number of exercises programmatically and allowing students to solve them while giving them feedback. Furthermore, it allows the creation of special comparators that can check whether the output of a given submission is equivalent to the expected one or simply check whether the above mentioned output corresponds to a correct solution.

2012 ACM Subject Classification Software and its engineering → Domain specific languages

Keywords and phrases domain specific language, code generation, automatic evaluation, testing

Digital Object Identifier 10.4230/OASIcs.SLATE.2018.5

1 Introduction

Evaluating students' performance in programming involves creating a large number of programming exercises and tools for estimating how well they solve them. The task of creating programming assignments forces teachers to devise these problems and to write them in a systematic fashion, not only concerning the task descriptions but also how they are evaluated. The usual methodology involves creating several scenarios that cover all the cases and check if the submissions solve them correctly. Thus, the team needs to create the inputs and corresponding outputs covering those cases. Furthermore, in many cases the formulation is rendered more difficult because there are several correct answers and this usually involves further complicating the problem by establishing a specific ordering (e.g., we want the first solution in the lexicographic order) or artificially simplifying the problem in order to get a deterministic answer (e.g., asking for the length of the minimum path length in a graph instead of one of the paths).

The goal of `eOS` is to help in this task. `eOS` will help create assignments by automatically generating program inputs and even getting the outputs by automatically generating them from the inputs by means of a solution. Furthermore, it is capable of handling comparators for increasing the ability of ascertaining whether a given solution is correct.

What sets `eOS` appart from other systems like CodeBoard [4], Stepik [15] or Mooshak [12] is the fact that it provides ways to programmatically create exercises by using scripting tools instead of using a web interface that, while user friendly, is time consuming when it is necessary to create many, often similar, problems. The second advantage is the fact that it is easy to use special comparators thus allowing the creation of programming exercises without having to artificially modify the system in order to get a single, deterministic answer to a given input. The third advantage is being able to create problems that involve creating one or more functions in a given, larger program.

2 On the automatic evaluation of programming assignments

It is not easy to write programs that are capable of automatically evaluating code. This is due to the fact that the automatic analysis of code is difficult. It is extremely hard to write a system that is capable of understanding what a piece of code does without running it and even to know if it will terminate. The usual approach for the automatic evaluation of programs is somewhat similar to unit testing [10]: it sets up a given environment and runs the program and analyzes its output. Most existing systems are either language dependent or often prefer to have the program read inputs from the *standard input* and write its result to the *standard output*. The authors are especially interested in systems that work in a wide spectrum of languages and thus favor the latter approach.

At the Department, the teaching staff uses automated evaluation in several courses and languages: we have been using such systems with `C`, `Python`, `Perl` and courses like introduction to programming, algorithms and complexity and bioinformatics. We also have courses that use other programming languages (e.g., `Java`) and subjects (e.g., machine learning, evolutionary computation) but don't use such systems mainly because the task of creating programming assignments is quite cumbersome and current systems are not capable of dealing with non determinism or the stochastic behavior inherent to such systems.

2.1 Overview of existing systems

We have extensively used Mooshak[12] and also Codeboard [4] or Stepik [15]. Mooshak is a system for managing programming contests on the Web that is used in the Maratona Inter-Universitária de Programação (MIUP) which is a 5 hour Portuguese programming contest that is part of the ACM International Collegiate Programming Contest for teams of three students. When a contest is created, Mooshak allows administrators to create problems. Creating a problem involves using a web interface with the mouse and keyboard, selecting things like the description, timeout, problem letter (from A to Z), if there are static or dynamic correctors and other details. Creating tests involves, again using the web interface, to create several tests where one selects what is the input, the output, context and number of points. One of the authors was involved in preparing the last MIUP contest and the task of setting up the system is somewhat daunting because the contest involved nine problems each with more than 10 test cases. It is possible to create special correctors (somewhat similar to what is presented in section 3.5) but the functionality is not well documented and it is almost never used. Thus, one is limited to check whether the expected and obtained outputs are exactly equal, to the point of there being a specific error message for the fact that both match if one removed all whitespace. Its limitations also involve that using floating point is avoided to the maximum and the authors have had several problems in the past when creating assignments that use floating point, to the point where these exercises are either simply not used or involve much more information about truncating errors and rounding.

Stepik was thought for creating courses and allows the creation of programming assignments. The programming assignments are created in Python and involve creating functions that generate program inputs, outputs and comparators between the expected and obtained outputs like our system. However, it is hard to manage the assignments, including moving them and each assignment must be created or edited using a web interface involving using the mouse and keyboard.

Codeboard can perform automatic evaluation by creating a project that outputs a special string as the last line of output. This involves creating the project using an IDE in the web interface and, for each assignment and language, creating a part of the code that grades the system. It is also possible to use of Java-JUnit, Haskell-HSpec or Python-UnitTest to help evaluate assignments. However, assignments are still created one at a time using a web interface and furthermore, they seem to be language specific.

2.2 Creating programming assignments

Creating programming assignments often involves:

1. Write the description of the assignment;
2. Create the inputs that will be passed to the programs submitted by the users in order to estimate their correctness;
3. Create the expected outputs;
4. Describe how close the outputs of the submission match the expected ones.

Most of these steps often involve repeatable effort. Creating the inputs often depends on the type of problem. For instance, problems over sequences or lists involve creating them according to a given rule (e.g., generate integer lists with random elements over a given interval) while most problems concerning graphs involve creating a graph with some characteristics (e.g., a geometric graph [13]). In order to create a candidate solution to a problem one needs to create some sort of algorithmic pipeline.

For instance, when creating a problem that involves path-finding, it is necessary to generate graphs, compute the distances between nodes, apply a shortest path algorithm (e.g., Dijkstra [3] or Floyd Warshall [5]) and present a path. Then, it is necessary to compare the path produced by the submission with the expected outputs in order to estimate the correctness of the solution.

This programming task's difficulty could even be tweaked by either providing the edges of the graph along with their weights or by providing the geometric coordinates of the vertexes or even some more indirect way of representing them.

The task of ascertaining the correctness of the submissions also involves repetition. If the expected output is for instance, a set of values, it is necessary to verify whether the user's submission provides the correct output but simply using a different ordering or, in the case of a graph path, if it is a valid one of the correct length.

Even writing the problem description is not without its fair share of repetition. In fact, the description of the program inputs and how the program should print its output could be reused for similar problems.

Often, it is necessary to create several problems using the same data structure. This means that it is usually possible to create an input generator for that data structure and use it in several problems. If one is able to combine this with a library that solves all the necessary tasks, it should be easy to create several programming tasks that will be able to correctly evaluate the students' ability to solve them.

3 Framework

The aim of `eOS` is to provide a framework, written in Python, to help the teaching staff with the task of creating problem assignments. It uses a command called `eos` with the following sub-commands (cf. Fig. 1 for the `eOS` process flow):

register This command allows the registration of a new problem;
evaluate This command takes a problem and a submission and evaluates the submission;

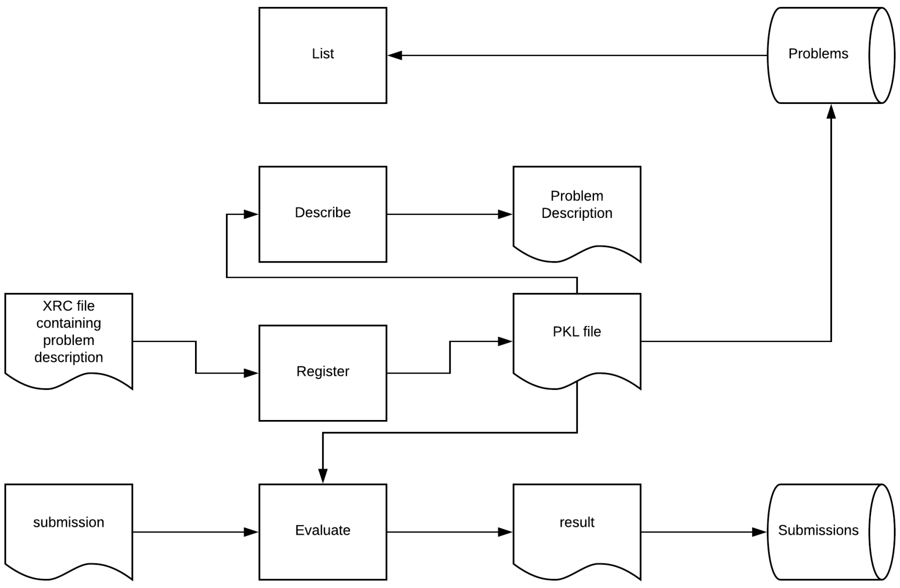


Figure 1 Process flow of eOS.

list This command lists all problems;

describe This command describes a problem;

language This command allows the definition of a new programming language.

3.1 Registering problems

In order to register problems, we conceived a very simple Domain Specific Language (DSL) [11]. It allows users to describe all the aspects of the problem including:

import In order to import Python modules;

input output The input or output;

description A problem description;

parse_input parse_output A function that parses the input or output, it takes a string and returns a list of strings;

solution A function that solves the problem, it takes a string corresponding to one of the inputs and returns a string that is the corresponding output;

template In the case where the problem only asks for a part of the code;

lang The language of the problem;

ntests The number of tests to show to the user (0 by default);

timeout The number of CPU seconds allowed for the program to run on each test (1 by default);

comparator A function that takes the input, expected output and obtained one and returns `True` or `False`;

source_invariant A function that takes the language used by the submission and the program source and returns `None` in case it can be accepted and a string containing an error message in case the program cannot be accepted.

Each commands starts with a `#` (e.g., the command is `#import`) and accepts one or more lines. These commands can be supplied with an exclamation point suffix in order to evaluate the argument of the command assuming it is Python code. In case the arguments supplied to a command consist of a a single word (i.e., they have no whitespace), the systems searches for a Python function with that name.

3.2 Describing inputs and outputs

In order to describe inputs, one uses the `#input` keyword. This involves creating a list of strings where each string corresponds to one of the inputs. This keyword can accept a function, given by its name, that produces the list. In this case, if the function has a docstring [2], it will be automatically added to the problem description.

By default, the input is several lines where each line corresponds to one input. In case something else is necessary, one can use the keyword `#parseinput` for supplying a function that performs the necessary parsing and produces the list of strings. It is also possible to supply the list of strings directly by appending an exclamation point to the input command. As an example, the following lines supply the same input:

```
#input
there is no place like home.
hello there!
#input! ['there is no place like home.\n', 'hello there!\n']
```

One can specify the output in exactly the same way or by supplying a solution by using the `#solution` keyword. This is a Python function that can be given by name and that takes a string corresponding to one input and produces a string corresponding to one output. If the solution is specified and the function has a docstring, it will be automatically used in order to document how the output is specified. No documentation of the input or output is performed if a template is used (since in this case the user does not have to worry about it).

3.3 A simple example

Let us assume that we aim to create a problem where the user has to read a sequence of integers and has to compute the longest increasing subsequence. We would need to write a text file, e.g, `lis01.xrc` that could have the following:

```
#description
Write a function called lis that takes a list of numbers and returns the
size of the longest increasing subsequence of non-consecutive integers
found.
#input
3 1 2 4
7 2 1 3 2 3 7 2 4
1 1 2 2 3 3 3 4 4 4 5 5 5 6
#output
3
4
14
#lang python
#template
[[function]]
```

```
lst = list(map(int, input().split()))
print(lis(lst))
#ntests 1
```

We define a template, that the submission must be in Python and that only the first of the three tests is shown to the user as feedback. It is also possible to supply the input by means of a function that returns a list of strings where each string is a test case and the output as a function that yields a list of strings. Templates can be used for other languages as long as they are defined (cf. section 3.8).

Instead of the output, we can supply a solution that is a function that computes the output given the input. In this case, we could have the following in file `lis02.xrc`:

```
#import example
#description
Write a program that reads a line containing a sequence of numbers
separated by spaces and prints the size of the longest increasing
subsequence of non-consecutive integers found.
#input get_input
#solution solve_lis
#ntests 2
#timeout 2
```

The module `example.py` contains the functions `get_input` and `solve_lis`. The function `get_input` doesn't take any arguments and returns a list of strings while the function `solve_lis` takes a string, which is an input and returns a string that is the output. If `get_input` has a docstring, it will by automatically appended to the problem description. The output description can also be taken from the docstring of the function `solution`. We also specify that the timeout is 2 seconds (instead of the 1 second default) and that the first two tests are supplied to the user as feedback. This problem asks the user to write the whole program, including reading the input and writing the output and accepts solutions in any language.

If we want to register the problem `lis02.xrc` given above and evaluate a solution in a file called `sub01.py` we would write:

```
$ eos register lis02
Write a program that reads a line containing a sequence of numbers
separated by spaces and prints the size of the longest increasing
subsequence of non-consecutive integers found.
The input consists on a single line containing several integers
separated by spaces.
The output consists of a single integer representing the length
of the longest increasing subsequence of non-consecutive integers
found.
Input 1:
3 1 2 4
Output 1:
3
Input 2:
7 2 1 3 2 3 7 2 4
Output 2:
```

```
4
$ eos evaluate lis02 sub01.py
Ok!
```

Notice that since the functions `get_input` and `solve_lis` have docstrings, their documentation is added at the end of the problem description. If we evaluate a submission that doesn't pass all tests, the output will indicate a feedback, if available, how many tests were run, how many were correct, how many had errors and how many timed out.

```
$ eos evaluate lis02 sub02.py
correct: 1
error: Timeout
errors: 0
timeouts: 10
total: 11
```

The file `sub02.py` corresponds to a naive implementation and, as such, it can only solve 1 of the 11 problems in the CPU time given. We realize that our program wasn't able to solve 10 problems due to timeout.

3.4 Parsing the input or output

In case we wanted to create a task for counting the number of lines in the input, we would want a different way of specifying input since in this case, the input would be several lines. For this, we could use a function called `get_paragraphs` defined in `parse_inputs.py` that splits the text on blank lines. Notice that, in this case, no tests are shown to the user in case of failure.

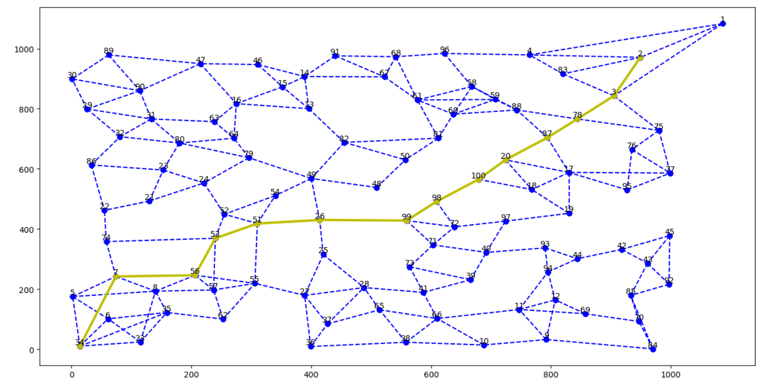
```
#import parse_inputs
#description
Write a program that counts the number of lines in the input.
#parse_input get_paragraphs
#input
d

e
f
#output
1
2
```

3.5 An example with special comparators

In many cases, it is more interesting to use a special comparator because there are many ways of supplying the same solution. For instance, if we ask for a path between two vertexes in a graph, there can be several paths between these nodes even if we are only interested in the shortest one. In this case, we have to supply a `comparator`.

```
#import graphs
#description Write a program that reads a graph and two nodes and writes
the shortest path between the two nodes.
```



```
196
1 2 249
1 3 421
1 4 426
2 3 172
2 4 193
2 83 184
...
99 71 125
99 72 101
100 18 123
100 20 110
34 2
```

Figure 2 Example of a generated graph with 100 vertexes and a path of minimum length from 34 to 2 and the corresponding input. The last line of the input corresponds to the origin and destination vertexes (34 and 2 respectively). There is another path of the same length that starts with [34, 33, 8, 56]. The figure was automatically created by the generator.

```
#input! generate_path_problems(10, 100)
#solution get_shortest_path
#comparator same_path_length
```

In this case, the function `generate_path_problems` was defined in `graphs.py` and generates 10 graphs with 100 nodes each along with the figures depicting the generated graphs (cf. Fig 2). The graph appears in the input with one line with an integer for the number of edges and one line for each edge with two node ids and the corresponding weight separated by spaces and a final line with the origin and destination nodes separated by a space. Notice that in this case we used `input!` because we have to evaluate the input as it is not simply the name of a function. The function `same_path_length` takes three arguments: the input, the expected output and the obtained one and should return `True` if both paths are from the same two vertexes and have the same length or `False` otherwise.

Another advantage of using comparators is that they can verify whether a given output is correct. As a rather contrived example, we could ask for a sorting algorithm and simply use the comparator to check whether the output produced by the submitted program contained all the elements in the input and they were ordered. Thus, we could simply supply the input and the comparator even though we didn't have a function that produced the output. The comparator can also be used to check if a given heuristic produces a solution of an acceptable quality (e.g., using the A* algorithm [9]).

3.6 Source invariants

In some cases, we may want to design a problem where the user cannot use a given function or library because we want to evaluate a given algorithm. For instance, let us suppose that we want to evaluate submissions that implement hash tables. In this case, we could create a function in Python similar to this one:

```
import re
def check_for_hash(lang, src):
RE = {'Java': r'java.util.(Hashtable|HashMap)',
        'C'   : r'#include\s*<search.h>'          }
    for line in src.splitlines():
        if re.search(RE[lang], line):
            return 'You may not use a library that implements hash tables'
```

and use it in the definition of a problem by adding `#source_invariant check_for_hash`.

3.7 Batch creation of exercises

The main advantage of `eOS` is to provide a programmatical way of creating several exercises. We will illustrate this concept by creating several problems that compute statistics over lists of integers.

```
problems = 'minimum maximum mean median variance'
text = """
#import my_list
#import fun_list
#description
Write a program that reads a line containing several integers separated
by whitespace and outputs their {name}
#input! gen_lists(range(1, 11))
#solution sol_{name}
"""
fun = """
def sol_{name}(inp):
    lst = list(map(int, inp.split()))
    return str({module}.{name}(lst))
"""
with open('fun_list.py', 'w') as FL:
    print('import my_list', file = FL)
    for num, prob in enumerate(problems.split()):
        with open("prob{:02}.xrc".format(num + 1), "w") as F:
            print(text.format(name = prob), file = F)
        print(fun.format(module = 'my_list', name = prob), file = FL)
```

`gen_lists` is a function that generates a random list of the given size. Thus, the specified input creates one list of each size ranging from 1 to 10. In this example, we first create 5 functions in a file called `my_list.py` (minimum, maximum, mean, median and variance) and this Python script creates a file called `fun_list.py` with a version of these functions that includes parsing the input and outputing the result as a string. This is done using the `fun` template that is used to create functions with the prefix `sol_`. This script also creates 5 files with names `prob01.xrc`, . . . , `prob05.xrc` using the template `text` corresponding to these 5 problems.

By executing this script and subsequently calling:

```
for prob in prob0?.xrc; do eos register $prob; done
```

on the command prompt, we register the 5 problems into the system and can then use them. This is a proof of concept that will soon be a part of the system to facilitate its reuse. Nevertheless, this strategy can easily be adapted to generate more exercises by adapting this script to other needs.

3.8 Defining new languages

`eOS` is able to evaluate problems defined in any language as long as one knows how to take a solution and *compile* it in order to create an executable and how to *run* it. The keyword `eos language` allows users to define new languages.

In order to define a new language, the user needs to define the following aspects:

name The language name;

compile The Unix shell command that compiles the program and creates the executable using `file` for the filename of the submission (this field may be omitted in case of an interpreted language);

run The Unix shell command that runs the program;

extension The extension or extensions for this language.

For instance, in order to define `C++`, one could create the following file called `lang_cpp`:

```
#name C++
#compile g++ -std=c++11 -Wall -Wextra -Werror [[file]]
#run ./a.out
#extension cc cpp cxx
```

And subsequently run the command `eos language lang_cpp`. This command only needs to be run once per system. As a subsequent example, let us imagine the scenario where we want to create exercises for a course where we want to evaluate the use of `flex` and `bison`. We can create a custom language `FlexBison` that expects a file, terminated by the extension `flbi`, archived with `tar` and compressed with `bzip2` containing two files, one named `lex.l` and another named `gram.y`:

```
#name FlexBison
#compile
tar xjf [[file]]
bison -d gram.y
flex lex.l
g++ gram.tab.c lex.yy.c -lfl -o parser
#run ./parser
#extension flbi
```

4 Discussion

Existing systems provide user friendly interfaces for creating problems (e.g. [15, 4, 12]). This is quite useful for creating exercises since it allows users to create them using a user friendly interface. However, there are several advantages to having a programmatic way of generating exercises. The first and foremost is the creation of a batch procedure for generating many exercises.

Thus, it is possible to use a library for generating data structures (e.g., trees or graphs) and create a large number of exercises that involve creating, updating or traversing the data structure or using frequently taught algorithms over that structure (e.g., path between two vertexes, checking whether it is a direct acyclic graph or a connected graph).

When devising problems, it is important to reserve inputs that test all functionalities of the task besides the ones shown to the users in order to prevent them from simply creating a program that prints the correct output given the input. As we intend to use this system both for helping students learn programming as to evaluate their success, it is possible to show all inputs to the students but we must caution the teaching staff about the obvious drawback involved.

It is very important to create inputs that correctly test the task being evaluated including all normal inputs and hedge cases (e.g., in the case of sorting a list, it is necessary to have a case using the empty list, a list with one element, lists with several elements, lists with repeated elements and lists with a very large number of elements). Inexperienced people can be tempted to only supply a few well behaved cases and later realize that very bad solutions were accepted. As a degenerate example, if all inputs for our sorting problem use lists of three integers, it is possible that a system using a few conditional statements and no loops is accepted. There are many more considerations when evaluating problems by supplying inputs and outputs, mainly it is necessary to ensure whether there are simpler algorithms than the ones we intend to evaluate that can solve the problem [6].

The main advantages of `eOS` are the following:

- It was thought with the Unix philosophy in mind;
- It is easy to create scripts for the batch generation of exercises;
- It allows importing Python modules to help create exercises;
- Most of the keywords accept a function that generates the needed value;
- It provides feedback to the user when it fails, and the problem setter can configure how many of the inputs are shown to the user;
- It automatically includes docstrings of functions used to generate the input and output to the program description thus sparing the problem setter to have to write them;
- It provides facilities for evaluating a program or several functions that are included in a larger program;
- It can evaluate code in almost any language and each problem can potentially receive submissions in several languages;
- It is quite easy to create special comparators for specific similarity measures or for evaluating heuristics (including non deterministic and stochastic ones);
- It logs all submissions to the system.

The main shortcomings of `eOS` are the following:

- In order to use the system fully, the teaching staff needs to know how to program in Python;
- There is currently no web interface neither for the teaching staff to create problems nor for students to use the system;
- As in most similar systems, one must test all hedge cases of a problem;
- There is currently no way to assessing the complexity of a program, its style or if plagiarism;
- It needs to be fully tested in real world conditions and it needs stress tests to ascertain its security to attacks.

5 Conclusions

The framework presented in this work helps the teaching staff to quickly generate several programming assignments by introducing ways to automate this task. This automation is due to the fact that inputs and outputs can be supplied in several ways including by using generators, specific parsers and solutions that automatically generate outputs. The fact that it is possible to import Python modules makes it possible to easily extend the system to attend specific needs. Another advantage of this system is being capable of using special correctors to accept solutions based on user supplied metrics of equivalence or proximity.

Currently, we have implemented a mail filter that receives submissions by email from university accounts. This allows us to use the system in our courses and provides an easy authentication of their submissions. We will evaluate this system in practice in the next semester in several courses in order to evaluate its performance. Some of the courses involved will use several programming languages and technologies (e.g., C, Python, Flex, Bison, BioPython, Machine Learning algorithms) and can involve up to 150 students.

Currently, `eOS` addresses some security concerns about compiling and running insecure code (e.g., forks, memory and disk limits) but more work is surely needed to mitigate all the possible security issues [7]. In the future, we intend to create a Web portal for students to submit their solutions by using this system and to specify a programmatic layer on top of the existing one to further automate the generation of exercises (cf. section 3.7). We aim to accomplish this by supplying several ways of generating inputs, outputs, solutions and special comparators by using several forms of functional composition. We intend to perform this by adding functionalities for documenting input generators and the ability of concatenating several generators in order to both create the input and automatically describe the input in the problem description.

We also intend to use this system for gathering statistics concerning the submission process that will allow us to use machine learning techniques to analyze the results and design intelligent tutors [1, 8, 16]. We also wish to incorporate the ability to check for code plagiarism by automating the task of using systems like MOSS [14].

References

1 Charu C. Aggarwal and Jiawei Han. *Frequent pattern mining.* Springer, 2014.

2 Guido van Rossum David Goodger. PEP 257 – Docstring conventions. Documentation, Python Software Foundation, 2001. URL: `https://www.python.org/dev/peps/pep-0257/`.

3 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

4 Christian Estler and Martin Nordio. Codeboard: A web-based ide to teach programming in the classroom, 2018. URL: `https://codeboard.io/`.

5 Robert W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

6 Michal Forišek. On the suitability of programming tasks for automated evaluation. *Informatics in education*, 5(1):63–76, 2006.

7 Michal Forišek. Security of programming contest systems. *Information Technologies at School*, pages 553–563, 2006.

8 Karam Gouda, Mosab Hassaan, and Mohammed J Zaki. Prism: An effective approach for frequent sequence mining via prime-block encoding. *Journal of Computer and System Sciences*, 76(1):88–102, 2010.

9 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

10 Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management.* John Wiley & Sons, 2007.

11 Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: a systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

12 José Paulo Leal and Fernando Silva. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.

13 Mathew Penrose. *Random geometric graphs*, volume 5. Oxford university press, 2003.

14 Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.

15 Stepic team. Stepic.org: Cloud-based digital learning environment for computer science. , `https://blog.stepik.org/`. URL: `https://stepik.org/`.

16 Mohammed J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1–2):31–60, 2001.