Olympiads in Informatics, 2025, Vol. 19, 1–25

DOI: 10.15388/ioi.2025.01

Interactive Problem Solving in the Classroom: Experiences with Turing Arena Light in Competitive Programming Education

Giorgio AUDRITO[1], Luigi LAURA[2], Alessio ORLANDI[3], Dario OSTUNI[4], Romeo RIZZI[5], Luca VERSARI[3]

[1]*University of Turin, Italy*
[2]*International Telematic University Uninettuno, Italy*
[3]*Google*
[4]*Università degli Studi di Milano, Italy*
[5]*Università di Verona, Italy*
e-mail: giorgio.audrito@unito.it, luigi.laura@uninettunouniversity.net, oalessio@google.com, dario.ostuni@unimi.it, romeo.rizzi@univr.it, veluca@google.com

Abstract. Turing Arena Light (TAlight) is a contest management system designed having in mind the needs of classroom teaching, rather than competitive programming contests. In TALight all problems are interactive by default. This means the contestant's solution for a problem will always interact with the problem, in real time. In this paper, we discuss our experience of using Turing Arena light in the course of Competitive Programming, an optional course offered to students of various LTs and LMs at the Department of Computer Science at the University of Verona. The course is meant to teach the students how to solve algorithmic problems, by teaching them the most common algorithmic techniques and data structures, and how to actively use them to solve problems.

Keywords: competitive programming, programming contest, competitive programming education.

1. Introduction

Programming contest management systems are the backbone of competitive programming events, handling everything from problem distribution and solution submission to automated judging and live scoreboarding (Revilla *et al.*, 2008; Maggiolo and Mascellani, 2012; Maggiolo *et al.*, 2014). Over the years, these systems have evolved from ad-hoc scripts and manual procedures into sophisticated platforms that emphasize security, scalability, and fairness (Leal and Silva, 2003). Traditional contest systems like the **Programming Contest Control System (PC^2)**, used in ACM ICPC since the 1990s,

enabled basic contest operations (login, submissions, judging interface) and were reliable for on-site contests. However, many early systems required judges to manually run solutions or provided limited automation.

Turing Arena Light (TALight) is a new contest management system that distinguishes itself by focusing on simplicity*, interactivity, and flexibility*. It was conceived as a lightweight platform geared toward educational use and practice environments rather than large-scale contests (`https://github.com/turingarena/turingarena`). TALight's design philosophy is to keep the core system minimal and conceptually simple, delegating most functionality to problem-specific modules. Uniquely, all problems in TALight are treated as interactive by default, meaning a contestant's solution gets connected to and interacts in with a problem manager program that provides inputs and checks outputs. The interaction takes place in real-time, while nothing prevents the constant to monitor the interaction in full while also print-debugging his code or logging everything in local. With this, it is quite handy to design interactive problems. And, even with standard input/output problems, the real commitment of the problem maker is to provide as much feedback as possible so that all of the students get their chance to actively learn and no one is cut out.

2. Related Work

Competitive programming systems can be classified into three main categories, each serving distinct purposes while sharing some overlapping features.

Contest Management Systems (CMS) are sophisticated platforms specifically designed for formal competitions like the International Olympiad in Informatics (IOI), ACM-ICPC, or national olympiads. Systems like DOMjudge (Pham and Nguyen, 2019), CMS (Maggiolo and Mascellani, 2012; Maggiolo *et al.*, 2014), and Kattis (Enstrom *et al.*, 2011) provide robust infrastructure for high-stakes, in-person events. They focus on security, reliability, and scalability to handle numerous concurrent submissions while maintaining fair evaluation conditions. These systems typically include features like real-time scoreboards, detailed analytics for judges, and stringent sandboxing mechanisms to ensure solution integrity. CMS platforms prioritize standardized evaluation environments where all participants compete under identical conditions with controlled resource limitations.

Online Judges serve a broader educational purpose by providing continuous access to problem-solving opportunities outside formal competitions. Platforms like Codeforces, LeetCode, and SPOJ host extensive problem libraries that users can attempt at their own pace. Unlike Contest Management Systems, they emphasize learning progression through difficulty-ranked challenges, detailed performance statistics, and community engagement via discussion forums and editorials. While they can host virtual contests, their primary value lies in self-directed practice. Many online judges incorporate gamification elements like ratings, badges, and streaks to motivate continued participation. Furthermore, since these systems have, in some cases, order of thousands of differ-

ent tasks, there is a vast literature related to the development of recommender systems able to suggest a suitable task depending on the learner's abilities (Audrito *et al.*, 2020; Fantozzi and Laura, 2020, 2021a, 2021b); also the problem of plagiarism is addressed (Iffath *et al.*, 2021). We refer the interested reader to the surveys of Wasik *et al.* (2019) and Watanobe *et al.* (2022).

Classroom Ad-hoc Tools like Turing Arena Light are specifically tailored for educational settings where pedagogical considerations outweigh competitive rigor. These systems prioritize ease of use, interactive problem types, and flexibility to accommodate diverse learning objectives. Unlike the standardized environments of CMS platforms, classroom tools often allow students to work in familiar development environments on their own machines. They typically feature simplified interfaces, immediate feedback mechanisms, and support for interactive problems that engage students through real-time interactions. While less suited for large-scale competitions, these tools excel at reinforcing classroom concepts and providing instructors with meaningful insights into student progress.

3. Turing Arena Light

In this chapter we introduce *Turing Arena light*, the spiritual successor of *Turing Arena* (`https://github.com/turingarena/turingarena`). *Turing Arena light* is a contest management system that is designed to be more geared towards the needs of classroom teaching, rather than competitive programming contests. It strives to be as simple[1] as possible, while being very flexible and extensible.

While we will discuss each point in more detail later, as an overview the design of *Turing Arena light* focuses on the following aspects:

- **Simplicity:** the design of *Turing Arena light* tries to keep things as simple as possible, while achieving the desired functionalities. While a meaningful objective metric for simplicity is hard to define, the current implementation of *Turing Arena light* consists of only 2197 lines of code (`https://github.com/romeorizzi/TALight`).
- **Interactivity:** in *Turing Arena light* all problems are interactive by default. This means the contestant's solution for a problem always interacts in real-time with the problem. In particular, a problem in *Turing Arena light* is defined by the problem *manager*, which is a program that interacts with the contestant's solution and gives a verdict at the end of the interaction. By being interactive by default, *Turing Arena light* allows a wider range of problems to be implemented with less effort, while not causing much overhead for non-interactive problems.
- **Flexibility:** *Turing Arena light* is designed to be able to run on all major operating systems, and allow solutions and problem managers to be written in any programming language, while still being able to guarantee a certain level of security. To

[1] Simple might mean very different things, in this context it is conceptual simplicity.

achieve this, *Turing Arena light* only consists of a small core written in Rust (Matsakis and Klock, 2014), whose main purpose is to spawn the process of the problem manager on the server, to spawn the process of the contestant's solution on its own machine, and to connect the standard input and output of the two processes. Thus, the contestants' code is never run on the server, and the problem manager can run without a sandbox, being trusted code written by the problem setter.

- **Extensibility:** as stated in the previous point, *Turing Arena light* only consists of a small core that has the fundamental role of spawning to processes and connecting their standard input and output. All the other functionalities are implemented by the problem manager itself, possibly using a common library of utilities. This allows the problem setter to implement any kind of problem, while still being able to use the same contest management system.

4. Architecture and Design

The fundamental idea behind *Turing Arena light* is to have two programs that talk to each other through the standard input and output channels. One of the two programs is the problem *manager*, which is a program that interacts with a solution to give it the input and evaluate its output, and eventually give a verdict. The other program is the *solution*, which is the program written by the contestant that is meant to solve the problem.

While this is not too far off from what other contest management systems do, the two main differences are that in *Turing Arena light* these two programs run on different machines, and the interaction between them is done in real-time. This is unlike mainstream contest management systems, where the two programs run on the same machine (like in DOMjudge (Eldering *et al*., 2020), CMS (Maggiolo and Mascellani, 2012) and Codeforces (`https://codeforces.com/`)), or where the interaction is not done in real-time (like in the old Google Code Jam (`https://codingcompetitionsonair.withgoogle.com/#codejam`) and Meta Hacker Cup (`https://www.facebook.com/codingcompetitions/hacker-cup`)).

In the following subsections we will discuss the components of *Turing Arena light* and how they interact with each other. We will start from the problem manager, going through the server and the client, and finally discussing the user interface.

4.1. *Problem Manager*

A problem in *Turing Arena light* is defined as a set of *services* and a set of *attachments*. A service is a program that can be spawned with a set of well-defined parameters, and that will ultimately interact with the solution. An attachment is a generic file that can be attached to the problem and downloaded by the contestant, such as the statement of the problem, or a library that the contestant can use in their solution.

A service defines which parameters it accepts, and the accepted values for each parameter. Parameters can be either strings or files. Each string parameter has a regular

```
%YAML 1.2
---
public_folder: public
services:
  free_sum:
    evaluator: [python, free_sum_manager.py]
    args:
      numbers:
        regex: ^(onedigit|twodigits|big)$
        default: twodigits
      obj:
        regex: ^(any|max_product)$
        default: any
      num_questions:
        regex: ^([1-9]|[1-2][0-9]|30)$
        default: 10
      lang:
        regex: ^(hardcoded|hardcoded_ext|en|it)$
        default: it
  help:
    evaluator: [python, help.py]
    args:
      page:
        regex: ^(free_sum|help)$
        default: help
      lang:
        regex: ^(en|it)$
        default: it
```

Fig. 1. Description file for a problem in *Turing Arena light.*

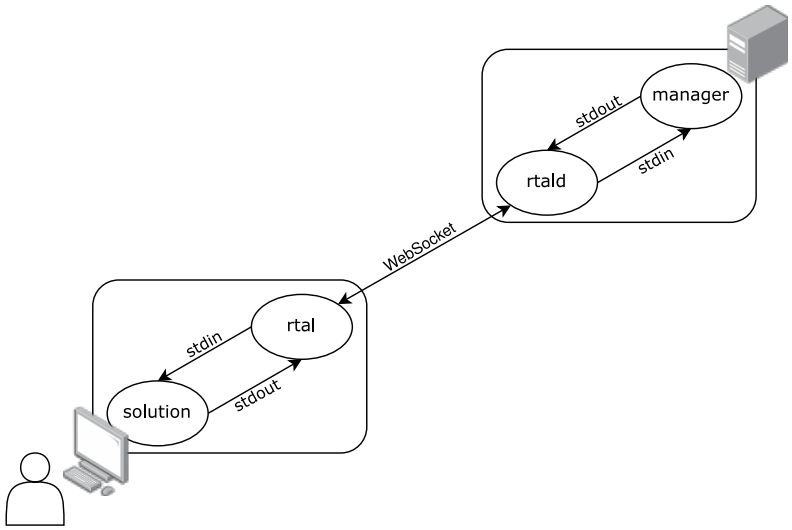

Fig. 2. Architecture of *Turing Arena light.*

expression that defines the set of accepted values and a default value. Furthermore, a service defines which program will be invoked with the given parameters: the problem *manager* (also called the *evaluator*). The attachments are just regular files in a folder on the file system.

The description of a problem is contained in a file called `meta.yaml`, which is a YAML (Ben-Kiki *et al*., 2009) file. The file contains the description of all the services, and their parameters, and the directory of the attachments. An example of a `meta.yaml` file is shown in Fig. 1. Thus, a problem in *Turing Arena light* is represented by a folder containing a `meta.yaml` file, and all the files and subdirectories needed for services and attachments.

4.2. *Server*

After the problem manager, there is the server. The server is the beating heart of *Turing Arena light*: its role is to accept incoming connections from the clients, spawn the problem manager corresponding to the client requested problem and service, passing to it the parameters specified by the client, and finally connect the standard input and output of the problem manager to the client.

Note that up to this point, *Turing Arena light* is merely a specification of how the problem is defined and how the interaction between the problem manager and the solution should happen. This opens up the possibility of having multiple implementations of the *Turing Arena light* framework, since the specification is very simple and does not require any particular technology, such as sandboxing.

Currently, there is only one implementation of the *Turing Arena light* framework, which is `rtal` (*Rust Turing Arena light*). It is written in Rust (Matsakis and Klock, 2014), and it is the reference implementation of *Turing Arena light*. The server component, `rtald`, is a small program that, given a folder containing problems, listens for incoming connections from the clients, and spawns the correct problem manager, and relays the standard input and output of the problem manager to the client via a protocol based on WebSockets (Fette and Melnikov, 2011).

4.3. *Client*

On the other side of the network[2] there is the client. The client is the program that the contestant runs on their machine to connect to the server and interact with the problem manager. Its role is to connect to the server, send the request for a problem and a service, send the string and file parameters for the service, and finally spawn and attach itself to the standard input and output of the solution running on the local machine of the contestant.

[2] Which might even be on the same machine, if both the server and the client are running on the same machine.

Once everything is up and running, the client will send the standard output of the solution to the server, which will relay it to the problem manager, and forward on the standard input of the solution all the incoming data from the server. Basically, the client is a proxy that connects the standard input and output of the solution to the server.

Like the server, there is also a `rtal` component for the client, also called `rtal`. This client component is a command line program that takes as parameters the address of the server, the problem and the service, and the parameters for the service. It also takes the command to run the solution. The client will then connect to the server, send the request for the problem and service, and spawn the solution with the given command, proxying the data between the solution and the server.

4.4. *User Interface*

As far as the contestant is concerned, what they must do is to write a solution to the problem in their favourite programming language. The only requirement is that it reads from the standard input and writes to the standard output. To read the problem statement, the contestant can download the attachments of the problem using the client. The client will download the attachments and save them on the local machine of the contestant.

Once the solution is ready, the contestant can run the client passing the right parameters, including the command to run their solution. The client will then connect to the server, send the request for the problem and service, and spawn the solution with the given command. Note that the solution is spawned and run on the local machine of the contestant, which means that the contestant has full freedom on which files it can read and write, which resources it can use, and so on. This is unlike other contest management systems that support real-time interaction, where the solution is run on a sandboxed environment on the server.

The ability to run the solution on the local machine opens to many possibilities. For example, the contestant can precompute some large set of data, save it on their machine, and then use it during the interaction with the problem manager to speed up the computation. Another example is the potential to use external libraries, multithreading, or even GPU computation. All of this is possible because the solution is run on the local machine of the contestant, where they have full control, and not on the server.

5. Implementation Details

As mentioned in the previous section, *Turing Arena light* currently has only one full implementation, which is *Rust Turing Arena light* (`rtal`). Like the name suggests, it is written in Rust (Matsakis and Klock, 2014). The choice of language was motivated by the fact that Rust is a systems programming language, and thus it is well suited for writing low-level programs that need to interact with the operating system and other pro-

grams. Furthermore, one key factor is portability: Rust is a compiled language whose compiled binaries require only minimal external dependencies to run, which makes it ideal to produce distributable binaries. This is important because *Turing Arena light* is meant to be used by students, which might not have the technical knowledge to install and configure a complex system. Having a single binary that can be downloaded and run without any configuration is a big advantage.

The implementation of *Turing Arena light* is split into three components: the server (`rtald`), the client (`rtal`), and the checker (`rtalc`). All three components share some common parts. The main one is the problem description definition, also known as the `meta.yaml` file. The definition can be found in Fig. 3. The definition is written using Rust structures which are then serialized to and deserialized from YAML using *serde* (`https://serde.rs/`), a serialization framework for Rust. `rtalc` is a small independent command-line program that takes as input a directory containing the problem description, and checks that the description is valid and matches the content of the directory. This is useful to check that the problem description is correct before uploading it to the server.

```rust
pub const META: &str = "meta.yaml";

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Problem {
    pub name: String,
    pub root: PathBuf,
    pub meta: Meta,
}

#[derive(Debug, Default, Serialize, Deserialize, Clone)]
pub struct Meta {
    pub public_folder: PathBuf,
    pub services: HashMap<String, Service>,
}

#[derive(Debug, Default, Serialize, Deserialize, Clone)]
pub struct Service {
    pub evaluator: Vec<String>,
    pub args: Option<HashMap<String, Arg>>,
    pub files: Option<Vec<String>>,
}

#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct Arg {
    #[serde(with = "serde_regex")]
    pub regex: Regex,
    pub default: Option<String>,
}
```

Fig. 3. Problem description definition in *Rust Turing Arena light*.

The two main jobs of the client and the server are process spawning and networking. For both of these tasks, `rtal` and `rtald` use the *tokio* (`https://tokio.rs/`) library, which is a framework for writing asynchronous programs in Rust. For the process spawning part, there is nothing particularly interesting: the server spawns the problem manager, and the client spawns the solution. They then, through *tokio*, manage the channels of the standard input and output of the spawned processes. All the internal communication within the server and the client is done using the actor threading model (Hewitt *et al.*, 1973; Hoare, 1978).

For the networking part, the communication protocol between the server and the client is based on WebSockets (Fette and Melnikov, 2011). The protocol definition is shown in Fig. 4. The protocol is based on JSON (Crockford, 2006) messages, which are serialized and deserialized using *serde*. These messages are then exchanged between the server and the client using WebSockets. The interaction between the server and the

```rust
pub const MAGIC: &str = "rtal";
pub const VERSION: u64 = 4;

#[derive(Serialize, Deserialize, Debug)]
pub enum Request {
    Handshake {
        magic: String,
        version: u64,
    },
    MetaList {},
    Attachment {
        problem: String,
    },
    ConnectBegin {
        problem: String,
        service: String,
        args: HashMap<String, String>,
        tty: bool,
        token: Option<String>,
        files: Vec<String>,
    },
    ConnectStop {},
}

#[derive(Serialize, Deserialize, Debug)]
pub enum Reply {
    Handshake { magic: String, version: u64 },
    MetaList { meta: HashMap<String, Meta> },
    Attachment { status: Result<(), String> },
    ConnectBegin { status: Result<Vec<String>, String> },
    ConnectStart { status: Result<(), String> },
    ConnectStop { status: Result<Vec<String>, String> },
}
```

Fig. 4. Network protocol definition in *Rust Turing Arena light*.

client is shown in Fig. 2. Using WebSockets enables a client of *Turing Arena light* to be implemented as a web application.

Both `rtal` and `rtald` run their spawned processes in an unsandboxed environment. This is done to avoid the complexity of sandboxing, but we argue that it does not pose a major security risk. The reason is that, for the client, the program being run is the contestant's own written solution, which is run on their local machine. Thus, the contestant has full control over the program, and can do whatever they want with it. For the server, the program being run is the problem manager, which is written by the problem setter. Thus, as long as the problem setter is trusted, there is no need to sandbox the problem manager. This is usually the case, as the problem setter is the one who also is responsible for the server where the `rtald` program is running. If this is not the case, then `rtald` can be run in a virtualized environment, such as a Docker container (Merkel *et al.*, 2014), to mitigate the risk of a bug in the problem manager that could cause unauthorized access to the server.

5.1. *Problem Manager Libraries*

So far we have discussed the architecture, the design and the implementation of Turing *Arena light*. However, we have not yet discussed how the problem manager is implemented. As mentioned in the previous sections, the problem manager is a program that interacts with the solution, and gives a verdict at the end of the interaction. The problem manager, just like the solution, has to communicate with its counterpart, which is the solution, using the standard input and output channels. Thus, the problem manager has full freedom on how to interact with the solution, as long as it does so using the aforementioned channels.

While this grants the problem maker a great deal of freedom, it also means that the problem maker has to potentially write a lot of boilerplate code each time they want to implement a new problem. To mitigate this problem, a problem maker can create a library of utilities that can be used to implement the problem manager. This library can be based on a particular style of problems, so that the problem maker can offer a consistent experience to the contestants.

In our case, we wrote a library called `tc.py`. A snippet of the library is shown in Fig. 5. This library allows to write a *old-Google-Code-Jam* like problem by only writing the code essential to the problem, and leaving all the boilerplate code to the library. What the manager has to implement is a function that generates a test case, and a function that evaluates the solution given by the contestant on a test case. The library will then take care of the rest, including enforcing the time limit, generating the right number of test cases, and assigning and storing the score for the solution. Note that with the *Turing Arena light* there is no way to enforce the memory limit, as the solution is run on the local machine of the contestant. However, the time limit can be enforced by measuring how much time passes between the sending of the input and the receiving of the output. While this is not a very precise measurement, it is good enough for distinguishing between solutions that have very different computational complexities.

```
class TC:
  def __init__(self, data, time_limit=1):
    self.data = data
    self.tl = time_limit

  def run(self, gen_tc, check_tc):
    output = open(join(environ["TAL_META_OUTPUT_FILES"], "result.txt"), "
        w")
    total_tc = sum(map(lambda x: x[0], self.data))
    print(total_tc, flush=True)
    tc_ok = 0
    tcn = 1
    for subtask in range(len(self.data)):
      for tc in range(self.data[subtask][0]):
        tc_data = gen_tc(*self.data[subtask][1])
        stdout.flush()
        start = time()
        try:
          ret = check_tc(*tc_data)
          msg = None
          if isinstance(ret, tuple):
            result = ret[0]
            msg = ret[1]
          else:
            result = ret
          if time() - start > self.tl:
            print(f"Case #{tcn:03}: TLE", file=output)
          elif result:
            print(f"Case #{tcn:03}: AC", file=output)
            tc_ok += 1
          else:
            print(f"Case #{tcn:03}: WA", file=output)
          if msg is not None:
            print(file=output)
            print(msg, file=output)
            print(file=output)
        except Exception as e:
          print(f"Case #{tcn:03}: RE", file=output)
          print(file=stderr)
          print("".join(traceback.format_tb(e.__traceback__)), e, file=
              stderr)
        tcn += 1
    print(file=output)
    print(f"Score: {tc_ok}/{total_tc}", file=output)
    output.close()
```

Fig. 5. Snippet of the python version of the competitive-programming like problem manager library for *Turing Arena light*.

As the name suggests, the `tc.py` library is written in Python (Van Rossum *et al.*, 2007), and it is meant to be used with problem managers written in Python. This works great for problems where the optimal solution plays well with Python, however in problems where the performance of the solution is critical, having the problem manager written in Python may make the evaluation of the contestant's output too slow. To mitigate this problem, we ported the `tc.py` library to Rust, thus creating the `tc.rs` library (`https://github.com/dariost/tal-utils-rs`). By using Rust as the program-

```
CREATE TABLE users (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    other TEXT
);

CREATE TABLE problems (
    name TEXT PRIMARY KEY
);

CREATE TABLE submissions (
    id INTEGER PRIMARY KEY,
    user_id TEXT NOT NULL,
    problem TEXT NOT NULL,
    score INTEGER NOT NULL,
    source BLOB NOT NULL,
    address TEXT,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (problem) REFERENCES problems(name)
);
```

Fig. 6. SQLite schema for database used by `tc.py` and `tc.rs`.

ming language for the problem manager, the whole execution of the problem manager is much faster. The functionality of the two libraries is the same, and they are interoperable with each other. This means that in a single contest, the problem maker can use both Python and Rust problem managers.

Turing Arena light has no built-in support for saving the results of the contest, as this job is left to the problem manager. This is done to allow the problem maker to have full control over how the results are saved. In `tc.py` and `tc.rs` we implemented a simple database that saves the results of the contest in a SQLite (Owens, 2006) database. The schema of the database is shown in Fig. 6. The database provides a way to save the results of the contest, and it enables contestants to see their position in the ranking during the contest, using a service defined in *Turing Arena light*.

6. Graphical User Interface

The Rust implementation of *Turing Arena light* only comes with a command line interface for the client. While this is enough to run the contest, it is not very user friendly. Contestants have to remember the right parameters to pass to the client, and the less experienced ones might have trouble working with a terminal. To mitigate this problem, a graphical user interface for the client was developed.

A web application was developed as a new client for *Turing Arena light* (`https://talco-team.github.io/TALightDesktop/`). A screenshot of the application is shown in Fig. 7. It was developed using the *Angular* framework (Green and Seshadri, 2013), and it is written in TypeScript (`https://www.typescriptlang.org/`). The

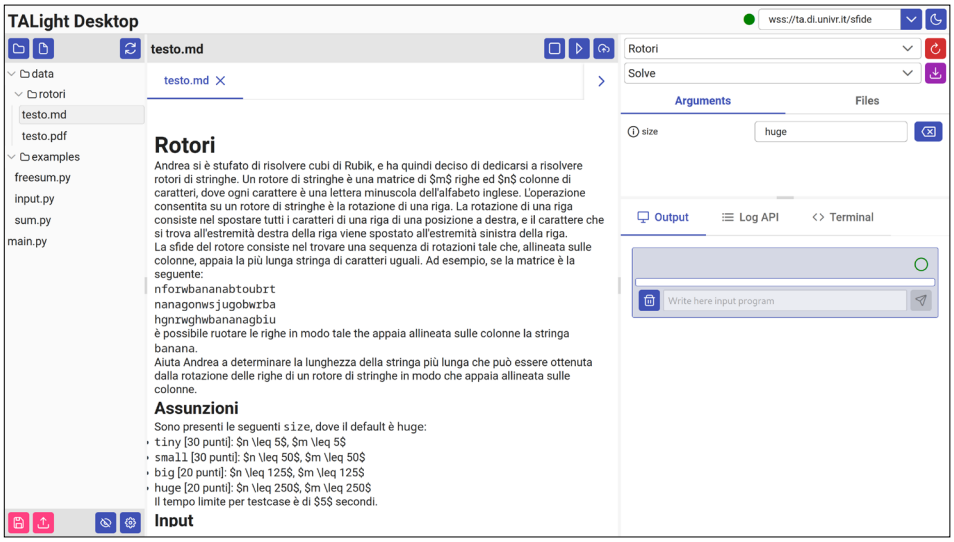

Fig. 7. Graphical user interface of T*uring Arena light.*

peculiar thing about this application is that aside from offering all the functionalities of the command line client, it also offers a way to write the solution directly in the browser. Not only that, but the solution is run directly in the browser, without the need to install any additional software. This functionality is currently only available for Python solutions, but it could be extended to other languages as well. To do this, the Python interpreter has been compiled to JavaScript, using *Pyodide* (`https://pyodide.org/`). This allows to run Python code directly in the browser. Thus, the contestant can do everything from an integrated environment in its browser.

Aside from running the solution in the browser, the web application also implements an emulated file system within the browser. This allows the contestant to send file parameters and receive file attachments and file outputs, all from the browser. Another useful feature that derives from having a file system is the ability to save and restore the working environment. This is useful for example when the contestant is working on a problem, and they want to save their progress and continue working on it later. Another scenario is when a template is provided to the contestant, and they can start working directly on it. The file system can be exported as a tar archive, or can be stored in the cloud using either GitHub (`https://github.com/`), Google Drive (`https://drive.google.com/`), or OneDrive (`https://onedrive.live.com/`). They can be later imported back from a tar archive or from the cloud, specifically from GitHub.

7. Experience in the Classroom

Turing Arena light has seen a good amount of use in some of the courses of the Computer Science department at the University of Verona. In particular, it has been used in

the courses of *Algorithms and Data Structures*, *Operations Research* and *Competitive Programming*. In this section we will discuss the experience of using *Turing Arena light* in the course of *Competitive Programming*.

The course of *Competitive Programming* is a course that is offered to the students of the department of Computer Science at the University of Verona. The course is meant to teach the students how to solve algorithmic problems, by teaching them the most common algorithmic techniques and data structures, and how to use them to solve problems. The course is structured in two parts: the first part is a series of lectures where the theory is explained, and the second part is a series of practical lessons where the students are given problems to solve, and they have to write a solution to the problem.

The practical lessons are done in a computer lab, where the students have access to a computer with the *Turing Arena light* client (`rtal`) installed. We prepared a body of problems that the students can solve, and we give them a problem to solve during each lesson. The problems are themed around the topic of the lecture, so that the students can practice the theory they learned during the lecture. The students have access to the problems both during the lesson, and at home, so that they can practice on their own. To achieve this, we have a server running `rtald` that is accessible from the Internet, and the students can connect to it from the client. The whole implementation of *Rust Turing Arena light* is released under the `MPL-2.0` license, which allows the students to download and use the client and the server for free.

Particular emphasis has been put on interactive problems. Since *Turing Arena light* allows to implement interactive problems with little effort, and they are kind of scarce in other contest management systems, we decided to focus on them. In particular, focusing on interactive problems allows us to give the students problems that do not focus on the time to compute the solution, but rather on the ability to interact with the problem manager or how many queries they can make. The kind of problems that are best suited for this are problems that involve some kind of game, where the contestant has to play against the problem manager. Another format is where the contestant has to guess some hidden information, and the problem manager gives hints to the contestant. In both cases the problem gives the contestant a sense of *playing a game*, rather than *solving a problem*, which is a good way to keep the students engaged.

Retaining the students' attention is more difficult in a classroom setting rather than in a competitive programming contest. In a contest, the contestants are motivated by the fact that they are competing against other contestants, and they want to win. Thus, they challenge themselves, they can be motivated to solve problems and learn on their own. In a classroom setting, the students are usually only motivated by the fact that they have to pass the exam, and they are not motivated to learn anything more than what is needed for that. Thus, it is important to keep the students engaged and make them interested about the topic, in order for them to then be motivated to learn more on their own. Interactive problems are a way to move towards this goal, as they can be more engaging than other kinds of problems.

As an example of such a problem, following this paragraph there is a problem that was given in the first laboratory lesson of the course.

Anna and Barbara's game Anna and Barbara discovered a new game: it is played on a V vector of n natural numbers. The first player picks a number from one end and takes it, then the second player does the same, and the game continues like this until the vector becomes empty. The player whose sum of the numbers taken is greater.

Anna always likes to play as the first player, while Barbara always wants to always go second. You want to play a game, but you have already seen the vector that will be used. Use this information to choose who to play against in so that you are sure not to lose and win the game!

Assumptions The following `size` are present, where the default is `big`:

- `small`: $n <= 8$, $max(V) <= 20$.
- `big`: $n <= 50$, $max(V) <= 10^6$

The sum of the values of V is always odd.

The time limit for testcase is 5 seconds.

Interaction The first line contains T, the number of games that will be played. In each game you are given on the first line n, and on the second line the vector V of natural separated by space. At this point it is your turn, write 0 if you want to play first, or 1 if you want to play second. The game starts with the first player and alternates players until all numbers have been taken. The player whose turn it is must write `L` or `R` followed by a carriage return depending on whether he or she wants to choose the number furthest left or the one furthest to the right.

To get `AC` you must win the game. It is always possible to win the game by some choice of which player to be and the moves to make, regardless of what the opponent will do.

Example Lines beginning with < are those sent by the server, those that begin with > are those sent by the client.

```
< 2
< 4
< 0 8 5 4
> 0
> R
< R
> R
< L
< 4
< 7 4 5 3
> 1
< R
> L
< R
> L
```

7.1. *Exams*

Aside from the laboratory lessons, Turing *Arena light ha*s also been used for the exams of the course. The exam is structured in a fashion similar to a competitive programming contest: the students are given three problems to solve, each worth 100 points, and they have four hours to solve them. The exam is taken in a computer lab, where the students have access to a computer with the Turing *Arena light clien*t (`rtal`) installed, other than the usual tools for programming, like an editor, a C++ compiler and a Python interpreter. The students are allowed to use any programming language they want, and they can use any piece of documentation they want, as long as they do not communicate with other students. However, they do not have internet access, so they cannot look up solutions online, or use other fancy tools, like GitHub Copilot (`https://copilot.github.com/`).

In the one academic year the course was offered, we administered five exam sessions. At the end of each session, the results were published, having a randomly generated identifier for each student[3]. The results of the exams are shown in Figures 8, 9, 10, 11 and 12. As shown in the figures, the total number of students that took the exam across all sessions is 31. The participation to the exam started low, with only 4 students taking the first exam, but it increased over time, with 12 students taking the last exam. Students were allowed to take the exam multiple times, and some of them did, since they could improve their score by taking the exam again, and keeping the best score. The results of the exams become better over time, as the students got more opportunities to practice and become accustomed with the kind of problems that were given.

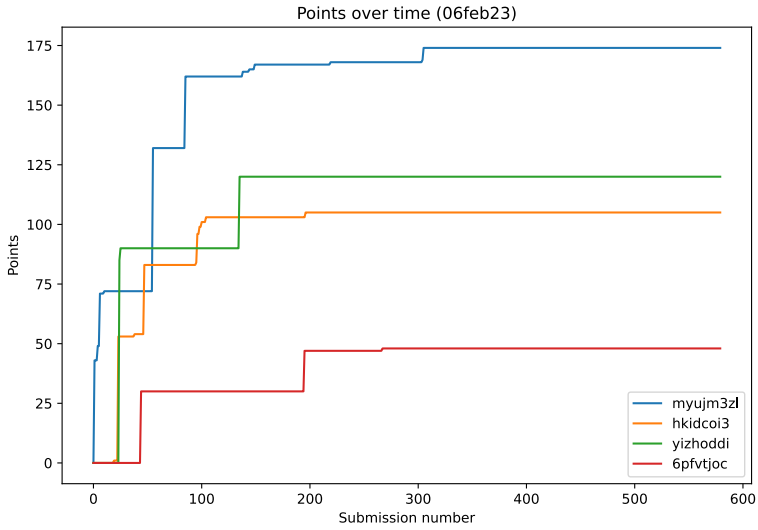

Fig. 8. Results of the 6th of February 2023 exam.

[3] If a student took the exam multiple times, they would have a different identifier each time.

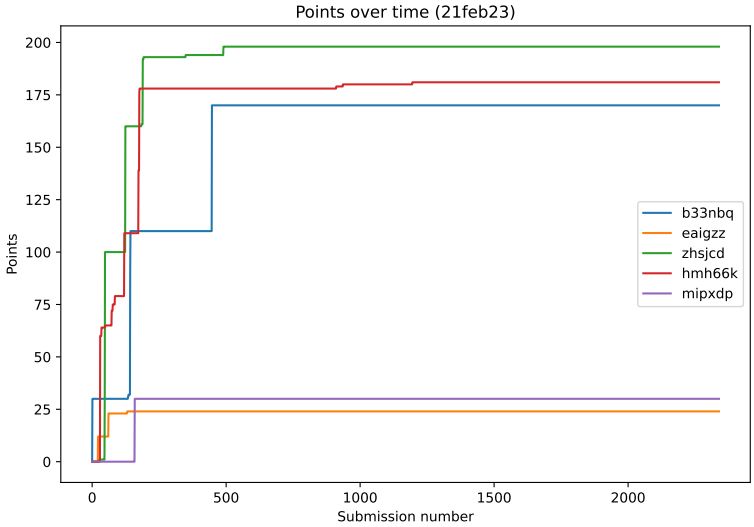

Fig. 9. Results of the 21st of February 2023 exam.

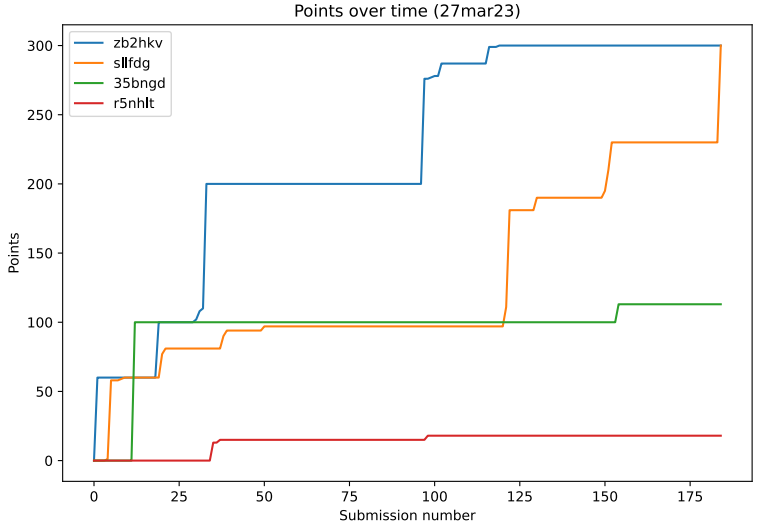

Fig. 10. Results of the 27th of March 2023 exam.

As an example of the problems given in the exam, following this paragraph there is a problem that was given in the exam session of the 22nd of June 2023.

Plumbing Luigi has begun his new adventure as a plumber, and now he is faced with his first job. In an old building there is a piping system that connects by joints the various apartments. Specifically, in this system there are n joints connected by pipes, and each pipe is connected to two joints. Between each pair of joints there is a piping path connecting them, and the number of pipes is $n - 1$. Each pipe has some length L_i.

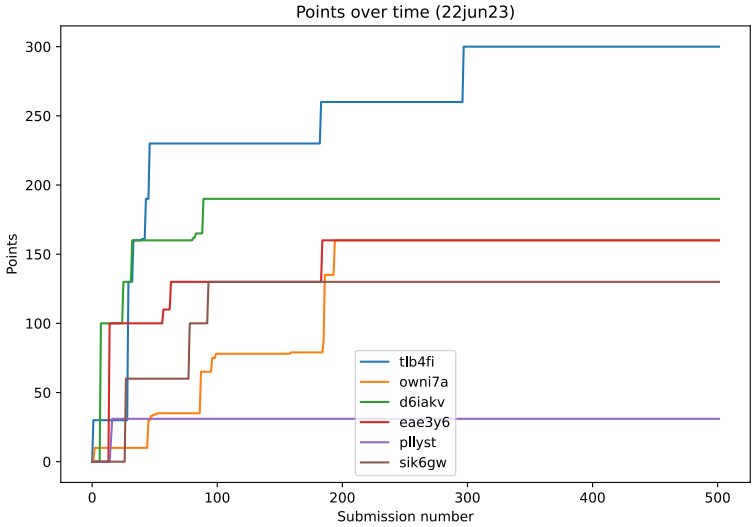

Fig. 11. Results of the 22nd of June 2023 exam.

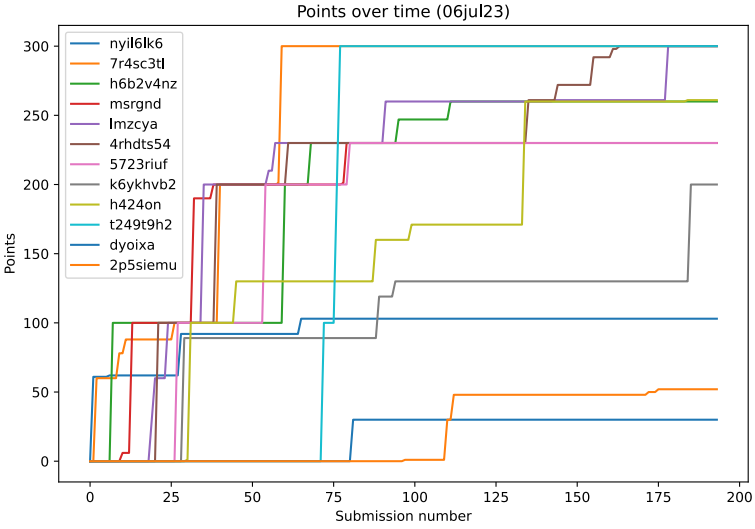

Fig. 12. Results of the 6th of July 2023 exam.

Luigi was called to calculate the total length of the piping system. However, Luigi does not have a diagram of the system, but he can measure the total length between two joints by running water between the two joints and measuring the time it takes to travel the path, thus measuring its length.

Luigi wants to finish the job as soon as possible so that he can move on to the next one, so he wants to calculate the total length of the piping system with as few measure-

ments as possible. Help him take the minimum number of measurements needed to calculate the total length of the piping system.

Assumptions The following `size` are present, where the default is `big`:

- `tiny` [30 points]: $n <= 45$, each joint is connected to at most 2 pipes
- `small` [30 points]: $n <= 45$
- `big` [40 points]: $n <= 50$

The maximum number of measurements that Luigi can make is 1000.

For each pipe i, L_i is an integer between 1 and 10000.

Interaction The first line contains T, the number of testcases to be solved. This is followed by T instances of the problem.

In each instance, initially the server sends n, the number of joints. Next, the client can make two kinds of requests:

- `? u v`: the client asks for the path length between joints u and v.
- `! l`: the client communicates the total length of the pipeline system, which is l.

Whenever the client makes a request of type `? u v`, the server responds with an integer, which is the length of the path between joints u and v.

The client can make at most 1000 requests of type `? u v`, after which it must make a request of type `! l` to terminate the interaction of the current instance.

Technical details While this problem has no time limit, sending 1000 queries and receiving as many responses can take a non-negligible amount of time.

However, it is possible to send queries in batches: if you do not need to know the result of the current query to send the next one, you can send all queries, and only after sending them do an explicit flush of the standard output.

In this way, all queries will be sent as a single packet, and all responses will be received as a single packet, greatly reducing communication time.

Example Lines beginning with < are those sent by the server, those that begin with > are those sent by the client.

```
< 1
< 3
> ? 0 1
< 4
> ? 0 2
< 2
> ? 1 2
< 6
> ! 6
```

Technical details While this problem has no time limit, sending 1000 queries and receiving as many responses can take a non-negligible amount of time.

However, it is possible to send queries in batches: if you do not need to know the result of the current query to send the next one, you can send all queries, and only after sending them do an explicit flush of the standard output.

In this way, all queries will be sent as a single packet, and all responses will be received as a single packet, greatly reducing communication time.

Example Lines beginning with < are those sent by the server, those that begin with > are those sent by the client.

```
< 1
< 3
> ? 0 1
< 4
> ? 0 2
< 2
> ? 1 2
< 6
> ! 6
```

7.2. *Survey*

After all the exams were administered, we asked the students to fill in a survey about their experience with *Turing Arena light*. The survey was anonymous, and it was done using Google Forms (`https://www.google.com/forms/about/`). The survey consisted of five questions:

- How much did you like the problems available in Turing Arena light (rtal)?
- How difficult did you find the problems proposed with Turing Arena light (rtal)?
- Did you find the interactive problems more interesting than the regular ones?
- How hard was to use Turing Arena light (rtal)?
- How strongly would you like for Turing Arena light to have a graphical user interface?

These questions were chosen to get a general idea of how the students felt about *Turing Arena light*, and to check if initial goals of *Turing Arena light* were being met. Note that this survey was conducted before the graphical user interface was available, so the last question was meant to check if the work being done on the graphical user interface was worth it.

- The responses for the first question are shown in Fig. 13. As can be seen from the results, the students liked the problems available in *Turing Arena light*. The average score is 4.07, which means that the problems were liked by the students.
- The responses for the second question are shown in Fig. 14. As can be seen from the results, the students found the problems proposed with *Turing Arena light* to be of slightlyabove-medium difficulty. The average score is 3.67, which means

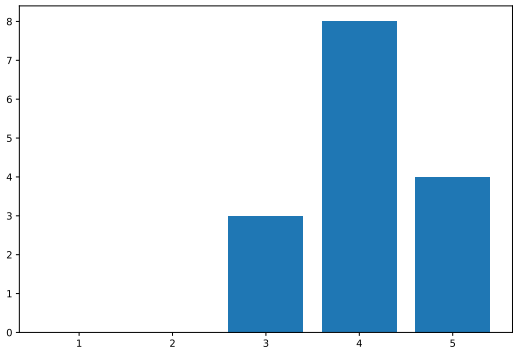

Fig. 13. Answers of question 1: *How much did you like the problems available in Turing Arena light (rtal)?*

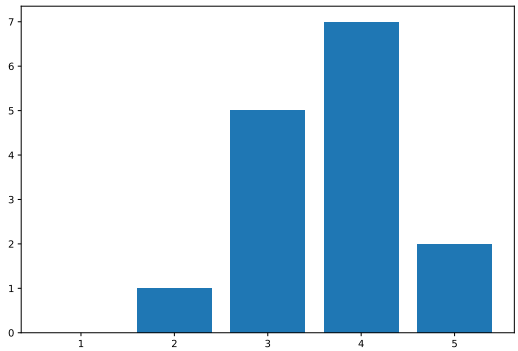

Fig. 14. Answers of question 2: *How difficult did you find the problems proposed with Turing Arena light (rtal)?*

that the problems were not too difficult, but they were not too easy either, although they were slightly on the hard side.

- The responses for the third question are shown in Fig. 15. As can be seen from the results, the students found the interactive problems to be more interesting than the regular ones. The average score is 3.80, which reinforces the idea that interactive problems are more engaging than regular ones.
- The responses for the fourth question are shown in Fig. 16. As can be seen from the results, the students found *Turing Arena light* to be easy to use. The average score is 2.20, which means that for the sample of students that took the survey, *Turing Arena light* did not pose any particular difficulty.
- The responses for the fifth and final question are shown in Fig. 17. As can be seen from the results, the students are kind of split whether they would like *Turing Arena light* to have a graphical user interface. The average score is 2.93, which means that the students are indifferent on average about wanting a graphical user interface, although there are some students that would strongly like it.

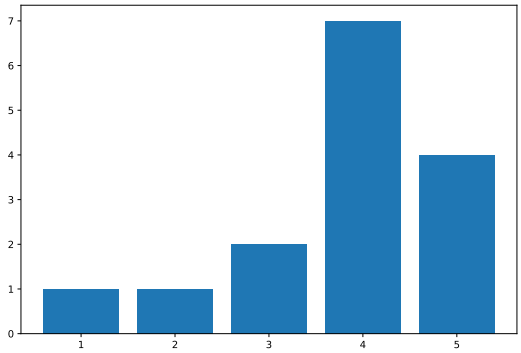

Fig. 15. Answers of question 3: *Did you find the interactive problems more interesting than the regular ones?*

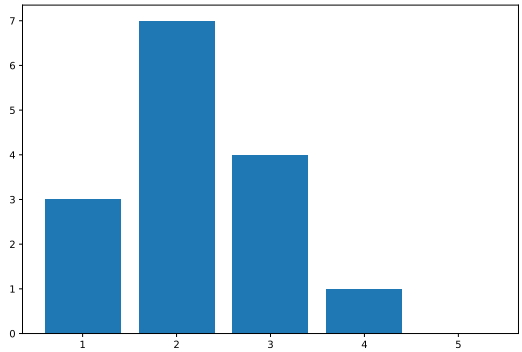

Fig. 16. Answers of question 4: *How hard was to use Turing Arena light (rtal)?*

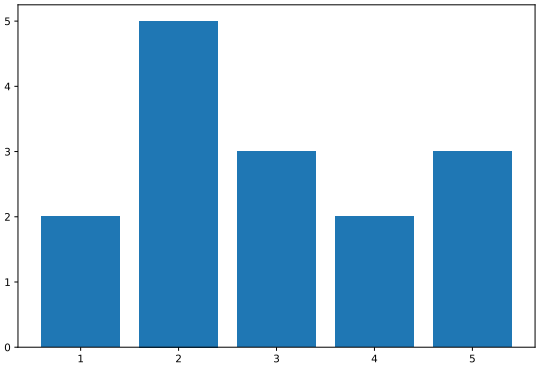

Fig. 17. Answers of question 5 of the post-exams survey: *How strongly would you like for Turing Arena light to have a graphical user interface?*

8. Future Directions

Turing Arena light has been developed enough to be used in a real-world classroom setting, and it has been used in the course of *Competitive Programming* at the University of Verona. It has been used for both the laboratory lessons and the exams, and it has been well received by the students. However, there is still a debate to be had in which direction *Turing Arena light* should move forward.

While the extreme flexibility of *Turing Arena light* made it possible to experiment a lot with different kinds of problems, it also made it difficult to find a common ground on which to standardize some common features, without having all of the problem manager libraries reimplement them. One such feature is the ability to save the results of the contest. While *Turing Arena light* does not have any built-in support for saving the results of the contest, it is possible to implement it in the problem manager. However, this means that each problem manager has to reimplement the same functionality, which is not ideal.

Moreover, some feature are implementable only by standardizing them at the core of *Turing Arena light*. One such feature is the ability of accurately measuring the time consumed by the solution. Right now, the time used by the solution is measured by measuring the time between the sending of the input and the receiving of the output. However, this is not a very accurate measurement, as it does not take into account the time spent sending and receiving the packets over the network. This is not a problem when the server and the client are on the same local network, as it happened in the course of *Competitive Programming*, but it becomes a problem when the server and the client are on different networks, such as when the server is on the Internet.

There is a solution to mitigate this problem, which is to encrypt the data, send it, then start the clock and send the decryption key. Doing it this way, one can eliminate the time spent sending the data, which can be a significant amount of time when the input is big. However, to implement such a solution, it would require to have some mechanism to make the problem manager and the core communicate on a meta-level to require this functionality from the core. However, such mechanism could cause a narrowing of the flexibility of *Turing Arena light*.

While the command-line interface has worked great for the course of *Competitive Programming*, it is not very probable that it would be fine for other courses with less *hardcore* students. Thus, the development of the graphical user interface continues, and it is planned to be tested in the next iteration of the course of *Competitive Programming*, and possibly in other courses with more *general* students.

References

Audrito, G., Di Mascio, T., Fantozzi, P., Laura, L., Martini, G., Nanni, U., Temperini, M. (2020). Recommending tasks in online judges. In: *Advances in Intelligent Systems and Computing*. Cham: Springer International Publishing, pp. 129–136.

Ben-Kiki, O., Evans, C., Ingerson, B. (2009). Yaml ain't markup language (yaml™) version 1.1. In: *Working Draft 2008-05* 11.

Crockford, D. (2006). *The Application/JSON Media Type for Javascript Object Notation (JSON)*. Tech. rep.

Eldering, J., Kinkhorst, T., van de Warken, P. (2020). *DOM Judge–Programming Contest Jury System. 2020*.

Enstrom, E., Kreitz, G., Niemela, F., Soderman, P., Kann, V. (2011). Five years with kattis – Using an automated assessment system in teaching. In: *2011 Frontiers in Education Conference (FIE)*. IEEE, Oct. 2011, T3J–1–T3J–6.

Fantozzi, P., Laura, L. (2020). Recommending tasks in Online Judges using Autoencoder neural networks. In: *Olymp. Inform.* (Dec. 2020).

Fantozzi, P., Laura, L. (2021a). A dynamic recommender system for online judges based on autoencoder neural networks. In: *Methodologies and Intelligent Systems for Technology Enhanced Learning, 10th International Conference. Workshops*. Advances in intelligent systems and computing. Cham: Springer International Publishing, pp. 197–205.

Fantozzi, P., Laura, L. (2021b). "Collaborative recommendations in online judges using autoencoder neural networks". In: *Advances in Intelligent Systems and Computing*. Advances in intelligent systems and computing. Cham: Springer International Publishing, pp. 113–123.

Fette, I., Melnikov, A. (2011). The websocket protocol. In:

Green, B., Seshadri, S. (2013). *AngularJS*. " O'Reilly Media, Inc."

Hewitt, C., Bishop, P., Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pp. 235–245.

Hoare, C.A.R. (1978). Communicating sequential processes. In: *Communications of the ACM* 21.8, pp. 666–677.

Iffath, F., Kayes, A., Tahsin Rahman, Md., Ferdows, J., Arefin, M., Sabir Hossain, Md. (2021). Online judging platform utilizing dynamic plagiarism detection facilities. In: *Comput. Rev. Esp. Hist. Contab.* 10, p. 47.

Leal, J., Silva, F. (2003). Mooshak: a Web-based Multi-site Programming Contest System. *Software: Practice and Experience.* 33, 567–581.

Maggiolo, S., Mascellani, G. (2012). Introducing CMS: A Contest Management System. In: *Olympiads in Informatics*, 6 (2012).

Maggiolo, S., Mascellani, G., Wehrstedt, L. (2014). CMS: a Growing Grading System. In: *Olympiads in Informatics*, 123.

Matsakis, N.D., Klock, F.S. (2014). The rust language. *ACM SIGAda Ada Letters*, 34(3), 103–104.

Merkel, D. *et al.* (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux j.* 239(2), 2.

Owens, M. (2006). *The Definitive Guide to SQLite*. Springer.

Pham, M.T., Nguyen, T.B. (2019). The DOMJudge based online judge system with plagiarism detection. In: *2019 IEEE-RIVF International Conference on Computing and Communication Technologies (RIVF)*. IEEE, pp. 1–6.

Revilla, M.A., Manzoor, S., Liu, R. (2008). Competitive learning in informatics: The UVa online judge experience. *Olympiads in Informatics*, 2, 131–148.

Van Rossum, G. *et al.* (2007). Python Programming Language. In: *USENIX annual technical conference*. Vol. 41.1. Santa Clara, CA, pp. 1–36.

Wasik, S., Antczak, M., Badura, J., Laskowski, A., Sternal, T. (2019). A survey on online judge systems and their applications. en. *ACM Comput. Surv.* 51(1), pp. 1–34.

Watanobe, Y., Rahman, Md.M., Matsumoto, T., Rage, U.K., Ravikumar, P. (2022). Online Judge System: Requirements, architecture, and experiences. en. *Int. J. Softw. Eng. Knowl. Eng.*, 32(06), 917–946.

G. Audrito is involved in the training of the Italian team for the IOI since 2006, and since 2013 is the team leader of the Italian team. Since 2014, he has been coordinating the scientific preparation of the OIS and of the first edition of the IIOT. He got a Ph.D. in Mathematics in the University of Turin, and currently works as a Junior Lecturer in the University of Turin.

L. Laura is currently the president of the organizing committee of the Italian Olympiads in Informatics that he joined in 2012; previously, since 2007, he was involved in the training of the Italian team for the IOI. He is Associate Professor of Theoretical Computer Science in Uninettuno university.

A. Orlandi helped train, tutor and coach potential candidates for, and members of, the Italian IOI delegation. He is currently a Principal Engineer at Google Switzerland.

D. Ostuni is currently the president of *Associazione Nazionale Programmazione Competitiva*, the largest Italian association of competitive programmers. He has been in the organization of the Italian Olympiad in Informatics since 2015. He got a Ph.D. in Computer Science from the University of Verona, and is currently a postdoctoral researcher at the University of Milan.

R. Rizzi begun training high school students for their participation toregional and national competitions of the Italian Olympiads in Informatics in 2001. Since 2004 he also begun training and coaching the Italian team to the International Olympiads in Informatics. He is full professor of Operations Research in the University of Verona.

L. Versari has been training Italian competitors in Informatics Olympiads since 2012 and Swiss competitors since 2023. He is a member of the IOI ITC, as well as part of the technical committees of multiple international competitions. He is currently a Software Engineer at Google.