



UNIVERSITY OF VERONA

DEPARTMENT OF COMPUTER SCIENCE

DOCTORAL PROGRAM IN COMPUTER SCIENCE

DESIGN AND EVALUATION OF
COMPETITIVE PROGRAMMING PLATFORMS
FOR COMPUTER SCIENCE EDUCATION

DARIO OSTUNI

ADVISOR

PROF. ROMEO RIZZI

COORDINATOR

PROF. FERDINANDO CICALESE

CYCLE XXXVI – S.S.D. INF/01

Contents

1 Introduction	1
1.1 Thesis structure	3
2 Competitive programming	5
2.1 Problems	5
2.1.1 Kinds of problems	5
2.1.2 Structure of a problem	7
2.1.3 A couple of examples	8
2.1.4 Anatomy of a solution	12
2.2 Problem solving techniques and topics	14
2.2.1 Computational complexity	14
2.2.2 Dynamic programming	16
2.2.3 Greedy algorithms	21
2.2.4 Divide and conquer	23
2.2.5 Strings	25
2.2.6 Graphs	27
2.2.7 Computational geometry	29
2.2.8 Number theory	31
2.3 Programming languages	33
2.3.1 C	33
2.3.2 C++	34
2.3.3 Pascal	35
2.3.4 Python	35
2.3.5 Java	36
3 Competitions	39
3.1 International Olympiad in Informatics	40
3.1.1 Italian Olympiad in Informatics	41
3.1.2 CMSocial	50
3.2 International Collegiate Programming Contest	59
3.2.1 SouthWestern Europe Regional Contest	63
4 Turing Arena light	67
4.1 Architecture and design	68
4.1.1 Problem manager	68
4.1.2 Server	68
4.1.3 Client	70

4.1.4	User interface	70
4.2	Implementation details	71
4.2.1	Problem manager libraries	74
4.3	Graphical user interface	76
4.4	Experience in the classroom	78
4.4.2	Exams	83
4.4.3	Survey	87
4.5	Future directions	88
5	Code Colosseum	91
5.1	Design, Motivations and Goals	92
5.1.1	Visualization	94
5.1.2	Tournaments	94
5.1.3	Simplicity	94
5.2	Implementation details	95
5.3	Pilot Experience	97
5.3.1	Royal Game of Ur	97
5.3.2	Double-elimination Tournament	97
5.3.3	Experience and Feedback	98
5.4	Findings and further directions	99
5.5	Replay functionality	99
5.6	Graphical user interface	100
5.7	Additional games	101
5.7.1	Checkers	101
5.7.2	Chess	101
5.8	Future directions	102
6	Conclusions	105
6.1	Future works	107
7	Other works	109
7.1	AI playing <i>Touhou</i> only from pixels	111
7.1.1	Introduction	111
7.1.2	Dataset Generation	112
7.1.3	Semantic Segmentation Networks	114
7.1.4	Experiments and Results	117
7.1.5	Conclusions and Future Works	120
7.2	SMT, MILP and SAT models for DTP	123
7.2.1	Introduction	123
7.2.2	The Disjunctive Temporal Problem	124
7.2.3	Existence of integer schedules	127
7.2.4	Encoding DTP into SMT	128

7.2.5	Encoding DTP into MILP	130
7.2.6	Encoding DTP into SAT	133
7.2.7	Experimental Evaluation	138
7.2.8	Conclusions and future work	142
Bibliography		145

1

Introduction

Competitive programming is a mind sport where participants solve well-known algorithmic problems in a limited amount of time. It has found popularity among high-school and university students whose field of study is related to computer science. Some of the most popular competitions draw more than 50000 participants from all over the world [1]. Solving problems in competitive programming requires a lot of skills: knowledge of algorithms and data structures, ability to write code quickly and without bugs, and the ability to come up with a solution to a problem in a limited amount of time. The participants that want to be successful in competitive programming need to acquire all of these skills through learning and practice.

These elements make competitive programming a great tool to aid the teaching of computer science. The skills that are specific to competitive programming are also useful in the realm of computer science. Moreover, competitive programming is a fun activity that can motivate students to learn more about computer science by solving algorithmic problems and learning new solving techniques. Competitive programmers create communities where they share their knowledge and help each other improve. This creates a great environment for students to learn and practice their skills. Furthermore, competitive programming also has a competitive aspect to it, that enables students to test their skills against other like-minded people.

Competitions are an important part of competitive programming. They are the primary engine that drives the creation of communities. There exist many competitions worldwide, with the two most popular ones being the *International Olympiad in Informatics* (IOI) for high-school students and the *International Collegiate Programming Contest* (ICPC) for university students. Participating in these competitions is a great way to test one's skills and to meet other competitive programmers. However, it is also a challenge to organize them. There are many aspects of organizing a contest, like creating the problems, curate the technical aspects of the contest, and deal with the logistics of the contest.

One of the main technical aspects of a contest is the *contest management system* used. A contest management system is a software that is used to manage the contest. Its purpose is to automate the process of evaluating the solutions submitted by the participants. To do so, it provides an environment for the problemsetter to create and test the problems, and it enables the participants to submit their solutions and receive feedback. Moreover, it usually provides a scoreboard that shows the current standings of the contest, keeping track of the score of each participant. While many contest management systems exist, they are usually made to run contests in compet-

itive programming competitions, and they are not necessarily suited to be used as a teaching tool in a classroom environment.

In this thesis we present two novel contest management systems that are designed with an educational purpose in mind. The first one is *Turing Arena light*, a lightweight contest management system that is geared towards interactivity and ease of use. The second one is *Code Colosseum*, a contest management system that is designed to host challenges between programs written by the participants. Both of these systems have been created to provide a tool that can create engagement and interactivity in a classroom environment.

To evaluate the effectiveness of these systems, we have tested them both in a real-world application. For *Turing Arena light*, we have used it as a companion for a course on competitive programming at the University of Verona, both for exercises and exams. For *Code Colosseum*, we have used it in a contest for high-school and university students, where the participants had to write programs that played a game against each other. We evaluated the effectiveness of these systems by collecting feedback from the participants.

Furthermore, we present in this thesis some of the work done on the organization side of two competitions. The first one is the *Italian Olympiad in Informatics* (OII), the national competition that selects the Italian team for the *International Olympiad in Informatics* (IOI). We present a report of the technical challenges we faced in organizing the 2020 edition of the OII during the COVID-19 pandemic. We also present some analytics on the learning progress of the participants of the OII and their teachers on a platform called *CMSocial*. This platform has been developed by the OII team to provide an online platform for participants to practice and engage with the community. The second one is the *SouthWestern European Regional Contest* (SWERC), the European regional contest for the *International Collegiate Programming Contest* (ICPC). We present some statistics on the editions of the contest we organized in 2022 and 2023.

Finally, we draw some conclusions on the work done in this thesis based on the feedback collected from the participants of the experiences organized with the contest management systems presented in this thesis. After the conclusions, we present some research works that we have done that are unrelated to the main topic of this thesis. These are a work on the creation of AIs that are able to play the game of *Touhou* only by looking at the screen, and a work on the encoding of instances of the *Disjunctive Temporal Problem* (DTP) into *Satisfiability Modulo Theories* (SMT), *Mixed Integer Linear Programming* (MILP), and *Satisfiability* (SAT) models.

1.1 Thesis structure

This thesis is organized into the following chapters:

1. **Introduction:** we introduce the topics that will be discussed in this thesis, and we present the structure of the thesis.
2. **Competitive programming:** we introduce the field of competitive programming, how the problems are structured, what solving techniques are used to solve them, and the strengths and weaknesses of the programming languages used in competitive programming.
3. **Competitions:** we present the competitions that are most relevant to competitive programming, and we present the contributions that we have done to the organization of two of these competitions.
4. **Turing Arena light:** we present *Turing Arena light*, a contest management system that we have developed that is designed to be used as a teaching tool, specifically geared towards interactivity and ease of use. We also present the results of its usage in a course on competitive programming at the University of Verona.
5. **Code Colosseum:** we present *Code Colosseum*, a contest management system that we have developed that is designed to host challenges between programs written by the participants, providing a way to visually see what the programs are doing. We also present the results of its usage in a contest for high-school and university students.
6. **Conclusions:** we draw some conclusions on the contributions given in this thesis, based on the analytics and feedback collected from the participants of the works presented in this thesis.
7. **Other works:** we present two research works that we have done that are unrelated to the main topic of this thesis. These are a work on an *AI playing Touhou from pixels* and a work on *SMT, MILP and SAT models for DTP*.

2

Competitive programming

Competitive programming is the art of solving well-known computer science problems faster than your peers [2]. It is a sport, a hobby, a way to learn algorithms and data structures, and problem solving [3, 4, 5].

2.1 Problems

In competitive programming the main focus is solving the problem. The problem is a well-defined computational task that can be solved either completely or with a good approximation in a reasonable amount of time by a computer. The problem is usually given in the form of a short text that describes the problem with a metaphor, a set of constraints, and a set of examples. The problem is usually given in a natural language, such as English, and the solution is usually written in a programming language.

There are many kinds of problems, and each problem requires its own set of problem solving skills, solving techniques, algorithms and data structures. The job of the person trying to solve the problem, usually called a *contestant*, is to read the statement, understand the problem, elaborate an algorithmic solution and implement it in a programming language. This can require a vast array of skills and knowledge in the field of computer science, that go beyond the mere ability to code [6].

The problem maker instead is tasked with creating a problem that has a well-known solution that can be found and implemented in a reasonable amount of time, but that is not trivial to find and implement. Aside from coming up with the problem, the problem maker also has to create a set of test cases that will be used to validate the solutions of the contestants. The test cases are usually generated by a program that is called the *generator*, and the correctness of the solutions is usually validated by another program that is called the *validator*. Generally, a solution is considered correct if it passes all the test cases, and incorrect if it fails at least one test case.

2.1.1 Kinds of problems

One way to categorize problems is according to the way the solution interacts with the problem.

Batch problems: these are the most common, and simple, kind of problems. A *batch* problem gives an input and expects from the solution an output that will be validated by a validator against that input. The output of the validator can either be a *correct* or *incorrect* verdict, or a *score* that gives a measure of how good the solution is. *Batch* problems can be seen as *one-shot* problems, since there is only one interaction between the solution and the problem.

Interactive problems: these are problems that require the solution to interact with another program, usually called the *manager*. The interaction is usually done through the standard input and output of the solution program. The interaction can be done in various ways, such as a question-answer game, or a request-response protocol. At the end of the interaction, like for the *batch* problems, the manager will give a verdict, that can either be *correct*, *incorrect*, or a *score*. These problems are called *interactive* because the solution and the manager interact multiple times, and both parties can adapt their behaviour based on the previous interactions.

Output-only problems: these are problems that give a fixed input for the whole problem instead of generating them on-the-fly or keeping them secret. With this kind of problem the solution can usually use as much time and resources as it needs. The solution will produce an output that will be validated by a validator. As with the previous kinds, the validator will give a verdict, that can either be *correct*, *incorrect*, or a *score*.

Black-box problems: these are the interactive equivalent of the *output-only* problems. In *black-box* problems the solution will interact with a *black-box* and has to find some information about it. In *black-box* problems the interactions are usually limitless, and the black box, or black boxes, are fixed for the problem, and not secretly held or generated on-the-fly. After having interacted with the black box, the solution will produce an output that will be validated by a validator. As with the previous kinds, the validator will give a verdict, that can either be *correct*, *incorrect*, or a *score*.

Two-steps problems: these are problems that require **two** solutions that interact with each other. Usually the input of the problem is given to the first solution, that will produce an output that will be given as the input of the second solution. The output of the second solution will be validated by a validator. The output generated by the first solution will also be given to the validator to make it generate its correctness verdict (usually it uses the length of the output of the first solution, but it can check for other parameters as well). As with the previous kinds, the validator will give a verdict, that can either be *correct*, *incorrect*, or a *score*.

Optimization problems: while all the other kinds of problems usually have a well-defined optimal solution to reach, *optimization* problems are specifically designed to have an optimal solution that is out of reach for the solution to find in a reasonable amount of time. The solution will usually have to find a good approximation of the optimal solution, and the validator will give a score that will be higher the better the approximation is. In this case the validator can still give an *incorrect* verdict if the solution is not valid, but it will never give a *correct* verdict. Usually *optimization* problems are in the form of *output-only* problems, but they can also be in the form of *batch* problems.

2.1.2 Structure of a problem

When presented with a competitive programming problem, the contestant usually sees a fairly common structure for the problem. The various parts are listed and described in the following paragraphs.

A short¹ text that describes the problem with a metaphor². This text is called the *statement* of the problem. In international competitions the statement is usually given in English, but in local competitions it can be given in the local language.

If the interaction with the problem is not trivial, such as in the case of *interactive* problems or problems that interact with custom functions, there is an *implementation details* section that describes how the solution should interact with the problem. The contents of this section can vary a lot, but usually it is a description of the protocol that the solution should use to interact with the problem. For instance, in IOI-like problems where the solution does not read from the standard input and does not have to implement a `main` function, the *implementation details* section will describe which function the solution should implement, and which other functions are available to interact with the problem.

A set of *constraints* that describe the limits of the problem. These constraints can be on the input, on the output, on the resources, or on the time. For instance, the constraints can be that the input will be at most 100 numbers long and that the solution will have at most 1 second to produce the output.

Some problems may have subtasks, that are groupings of test cases that satisfy stricter constraints than the general ones. Solving these subtasks will give a partial score, and solving all the subtasks will give the full score. For instance, a problem may have a subtask that requires the solution to run in quadratic time, and another subtask that requires the solution to run in linear time. Solving the first subtask will give a partial score, and solving both subtasks will give the full score. Usually the subtasks are ordered by difficulty, so that the first subtask is the easiest, and the last subtask is the hardest.

A description of how the input is formatted and how the solution should read it. For instance, the input can be a number n , a newline, and then n space-separated numbers. The input is usually given in a form such that it can be easily read by the solution, even if it is written in a programming language that does not have support for fancy input parsing. In some cases where the input might be very big, the input might be given in a more compact form, such as a binary file.

A description of how the output should be formatted and how the solution should write it. For instance, the output can be a single number. The output is usually much smaller than the input, and can employ techniques such as printing numbers modulo

¹Sometimes short can mean several pages long.

²Although sometimes it can be cut and dry.

a big prime number to limit the size of the output. Usually, the output requested by the problem is some kind of number that gives the answer to the problem without necessarily asking the solution. For instance, in a problem where the minimum path in a graph is requested, the output asked will probably be the length of the path, and not the path itself.

If the problem is an *optimization* problem or it wants to give some kind of partial scoring, there will be a description of how the validator will score the solution. In this case, the output will be a full proper solution to the problem, and not just the numerical answer, so that the validator can check the correctness of the solution and give it a score based on how good the solution is. For instance, in a problem where the solution has to find the maximum path in a graph, the output will be the path itself, and the validator will check that the path is valid and will give a score based on the length of the path.

2.1.3 A couple of examples

As an example, here is the statement of the problem *Ordinamento a paletta* from the 2016 edition of the Italian Olympiad in Informatics [7].

Ordinamento a paletta (paletta)

Romeo attended a special barbecue party where the cook handled a large number of hamburgers in an amazing way. When the hamburgers needed to be flipped, the cook was able to do that on three consecutive burgers with a single spatula (paletta), quickly and in a single shot! This inspired Romeo for a new sorting problem called *paletta-sort*.

Given an array V storing all the integers from 0 to $N - 1$ (where array positions are from 0 to $N - 1$), the only feasible operation in the paletta-sort is called *ribalta*: it replaces the integers A, B, C in three consecutive positions of V with their flipped values C, B, A in this order. You are required to help Romeo to understand if paletta-sort can sort V : if it is so, say how many *ribalta* operations are needed.

Implementation

You shall submit one file having extension `.c`, `.cpp` or `.pas`.

You need to implement the following function:

- C/C++: `long long paletta_sort(int N, int V[]);`
- Pascal: `function paletta_sort(N: longint; V: array of longint) : int64;`
- N is an integer representing the number of elements to sort.
- V is an array, indexed from 0 to $N - 1$, containing the elements to sort.

- The function has to return the number of *ribalta* operations to sort V , or -1 if the latter cannot be sorted in this way.

The grader will call the function `paletta_sort` and will print the returned value to the output file.

Grader

In the directory for this problem there is a simplified version of the grader used during evaluation, which you can use to test your solutions locally. The sample grader reads data from `stdin`, calls the function that you should implement and writes to `stdout` in the following format.

The input file is made of 2 lines, containing:

- Line 1: integer N .
- Lines 2: values $V[i]$ for $i = 0, \dots, N - 1$.

The output file is made of a single line, containing:

- Line 1: the value returned by the function `paletta_sort`.

Constraints

- $1 \leq N \leq 1500000$.
- $0 \leq V[i] \leq N - 1$ for $i = 0, \dots, N - 1$.

Scoring

Your program will be tested against several test cases grouped in subtasks. For each test case you will get the following factor.

- 1: If you compute the minimal number of *ribalta* operations.
- 0.2: If array V can be sorted and you compute any non-negative number (that is, you can distinguish if V can be sorted or not).
- 0: All the remaining cases.

For each subtask, its score is given by the product of the points below times the above factor for the worst test case in the subtask.

- Subtask 1 [0 points]: Examples.
- Subtask 2 [20 points]: $N \leq 100$.
- Subtask 3 [30 points]: $N \leq 5000$.
- Subtask 4 [20 points]: $R \leq 100$ (or V cannot be sorted).
- Subtask 5 [25 points]: $N \leq 100000$.
- Subtask 6 [5 points]: No limitations.

Example 1

Input

5
2 0 4 3 1

Output

-1

Example 2

Input

6
2 3 0 5 4 1

Output

3

Explanation

In the first example it is not possible to sort V .

This problem, *Ordinamento a paletta*, is a typical *batch* problem in the style of the International Olympiad in Informatics (IOI) [8]. The problem is given in English, and the solution can be written in C, C++, or Pascal. The problem uses both subtasks and partial scoring.

As another example, here is the statement of the problem *Incognita* from an exam of the *Sfide di Programmazione* course in the University of Verona.

Incognita

Giorgio Giovanni is an explorer of ancient ruins. During one of his expeditions, he discovered an ancient temple, which he is now in the process of exploring. However, at the entrance to the temple there is a contraption that requires the input of n integers to be activated.

A mysterious old man approaches Giorgio, and tells him that he knows the numbers to be entered, but cannot tell him directly. However, he is willing to give him some information, for the right price. The old man introduces himself as Diego B.

Diego is willing to answer the following kind of questions: Giorgio will have to indicate a symbol between +, - and 0 for each number, and Diego will tell him the sum of the numbers assigned the symbol +, minus the sum of the numbers assigned the symbol -. We denote by k the number of + and - in Giorgio's question. To answer this question, Diego will ask for a payment of $\lceil \frac{n}{k} \rceil$ coins.

Giorgio has brought only b coins with him, and he cannot go back for more. Help Giorgio formulate the questions to ask Diego, so that he can find out all the numbers without using more than b coins.

Constraints

In this problem n is always equal to 100. The numbers to be guessed are between -10^6 and 10^6 .

The parameter `size` indicates the number of testcases required, the default is 100. It can be any integer between 1 and 100.

For testcase t (between 1 and 100), the value of b is:

$$\min(\max(\lceil 19000 * 1.06^{-t} \rceil, 100), 10000)$$

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 numbers to be guessed, and b , the number of coins available to Giorgio.

The client can send requests to the server, which can be of two types:

- `? <s1> <s2> . . . <sn>`: the client asks the server to calculate the sum of the numbers assigned the symbol `+`, minus the sum of the numbers assigned the symbol `-`, where s_i is the symbol assigned to the i -th number. The server responds with an integer, which is the answer to the query. The client must pay $\lceil \frac{n}{k} \rceil$ coins for this query, where k is the number of `+` and `-` in the query. If the client does not have enough coins, the testcase verdict will be RE. Only `+`, `-` and `0` can be used as symbols;
- `! <v1> <v2> . . . <vn>`: the client tells the server that the numbers to be guessed are v_1, v_2, \dots, v_n . If the numbers are correct, the testcase verdict will be AC, otherwise it will be WA. After this query, the server will move on to the next testcase.

Technical details

While this problem has no time limit, sending thousands of queries and receiving as many responses can take a not inconsiderable amount of time.

However, it is possible to send the 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 the 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
< 2 4
> ? + 0
< -6
> ? 0 -
< -9
> ! -6 9
```

This problem, *Incognita*, is an *interactive* problem in the style of *Turing Arena light* [9]. The problem was originally given in Italian, but it has been translated to English for this thesis. In this problem there are no subtasks, but the single test cases are worth one point each and get gradually harder.

2.1.4 Anatomy of a solution

We have seen that a solution is a computer program that solves a problem. But what does it mean to solve a problem? There are usually three requirements that a solution must satisfy to be considered correct, thus solving the problem.

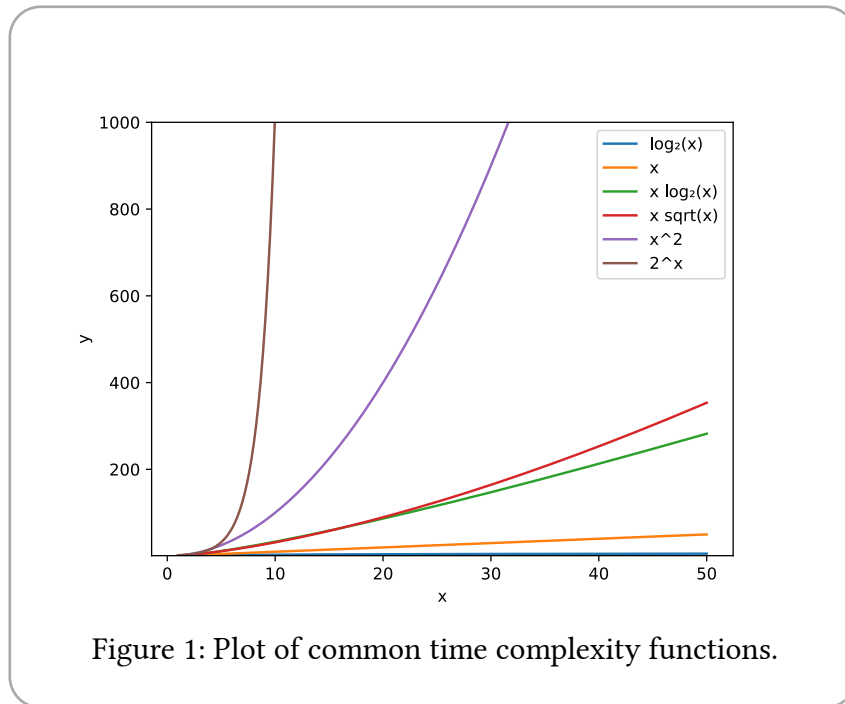
Correctness

For every possible input of the problem, the solution *should* produce the **correct** output. Ideally, this would be what we are going to test, however determining if a solution is correct for every possible input is a known undecidable problem [10]. Instead, we can test the solution against a set of inputs that are considered representative of the whole set of possible inputs. These inputs are called the *testcases*. They are generated either as a fixed set for the problem, or on-the-fly for each attempted solution. The program that generates the *testcases* is called the *generator*.

Given this set of inputs, the solution is considered *correct* if it produces the correct output for every input. If it fails to produce the correct output for at least one input, it is considered *incorrect*. Thus, the first requirement for a solution is to be correct, in the sense that it produces the correct output for every input in the set of *testcases*.

Time limit

The solution *should* be **fast**. Fast can mean different things: it can either refer to how machine-optimized the solution is, or how fast it is in terms of algorithmic time complexity. In competitive programming we look at the latter meaning of fast. This is because we want to test the solution of the problem on increasingly bigger inputs. If we have a solution *A* that is very machine-optimized, but has a less-than-optimal time complexity, and a solution *B* that is not as machine-optimized, but has a better time complexity, then *B* will always become faster than *A* if the input is big enough.



We can clearly see from Figure 1 that different time complexities have drastically different growth rates. Even the most machine-optimized solution that has a time complexity of $O(n^2)$ will never be able to solve a problem with $n = 1000$ even if it runs on a supercomputer for a million years³.

In an ideal world⁴ we would be able to have a way to compute the time complexity of a solution. Alas, this is not possible, since the only general way to do it would be to test it on every possible input. Thus, the only metric we can accurately measure is the time it takes for the solution to run on a single input. The strategy used is to fix a certain amount of time, called the *time limit*, for the solution to run on any single test case. This time limit can vary from hundreds of milliseconds to several seconds, depending on the problem. By generating test cases that become bigger and bigger, we can see how the solution behaves as the input grows. If the solution has a good time complexity, it will be able to solve the bigger test cases within the time limit, otherwise, even if it is very machine-optimized, it will inevitably fail to solve the bigger ones at some point.

Thus, the second requirement for a solution is to be fast enough, in the sense that it can solve all the test cases of the problem within the time limit. However, what we are really trying to measure is its time complexity, and the time limit is just a proxy for it.

³ $2^{1000} \approx 10^{300}$, the number of seconds in a million years is around $3 \cdot 10^{13}$, so a supercomputer would need to do over 10^{286} operations per second, which is 268 orders of magnitude more than the current fastest supercomputer, which clocks in at around 10^{18} operations per second.

⁴More like an *ideal math world*.

Memory limit

The solution *should* be **memory efficient**. Like for the time limit, the efficiency of the memory usage of a solution can have two meanings: a practical one, and a complexity one. And, applying the same arguments we did for the time limit, we would like to measure the memory complexity of a solution, but in real world we are limited to measuring the actual memory usage of the solution for a specific input.

In the same setting of solutions A and B as before (the former machine-optimized but with a bad memory complexity, the latter not as machine-optimized but with a good memory complexity), we can see that the memory usage of A will grow much faster than the memory usage of B as the input grows, even if the constant factor of memory usage of A is smaller. As in the previous example, if we take a solution with memory complexity $O(n^2)$, it will never be able to solve a problem with $n = 1000$, because the available information storage in the observable universe is about $6 \cdot 10^{80}$ bits [11], which is about 220 orders of magnitude less than the required amount of memory⁵.

Thus, the third requirement for a solution is to be memory efficient, in the sense that it can solve all the test cases of the problem within the memory limit. However, what we are really trying to measure is its memory complexity, and the memory limit is just a proxy for it, like the time limit.

2.2 Problem solving techniques and topics

Solving a problem requires a lot of knowledge and skills in the field of computer science. In this section we will see some of the most common techniques and topics that occur in competitive programming problems. A competitor that wishes to do well in competitive programming should be familiar with all of these techniques and topics.

2.2.1 Computational complexity

In competitive programming the foundation for judging the efficiency of a solution resides in the analysis of its computational complexity. Computational complexity is the study of the resources needed to solve a problem. In this context the resources are the time and the memory needed to solve the problem, as function of the size of the input.

Before running a solution for a certain problem, it is important to have an estimate of how much time and memory it will need to run to completion. For this reason, we care about how much time and memory a solution will need in the worst possible case for a given input size, so that we can know in advance if it is reasonable to run such a solution on a given input. This is called the *worst-case* analysis of the solution.

⁵ $2^{1000} \approx 10^{300}$

Notation	Definition
$f(n) \in O(g(n))$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in o(g(n))$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \Omega(g(n))$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$
$f(n) \in \omega(g(n))$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$
$f(n) \in \Theta(g(n))$	$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \sim g(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$

Table 1: Landau's symbols definition for $f : \mathbb{N} \rightarrow \mathbb{N}^+$ and $g : \mathbb{N} \rightarrow \mathbb{N}^+$.

While it is important to have an estimate, it is not needed to have an exact measure of the time and memory needed by the solution. This is because, especially for the time, the exact measure can vary a lot depending on the machine that runs the solution, and other variables such as the compiler used, how loaded the machine is in that moment, and so on. Thus, we usually only care about the *asymptotic* behaviour, up to a constant, of the resources needed. This means that we only care about how the resources needed grow as the input grows.

To express the asymptotic behaviour of a function we use the *Bachmann-Landau's symbols* [12, 13], which are defined in Table 1. The two most common symbols are O (big O) and Θ (Theta). Big O is an estimate that says that a function can grow at most as fast as another function, up to a constant. Theta is an estimate that says that a function grows as fast as another function, up to a constant. For instance, if we have the function $f(n) = 2n^2 + 3n + 1$, then $f(n) \in O(n^2)$, $f(n) \in O(n^3)$, $f(n) \in \Theta(n^2)$, but $f(n) \notin O(n)$ and $f(n) \notin \Theta(n^3)$. Note that if we have a function $f : \mathbb{N} \rightarrow \mathbb{N}^+$ such that $f(n) \in \Theta(n)$, then $f(n) \in O(n)$.

We use this symbols to express the asymptotic behaviour of the time and memory functions of a solution. For instance, if we have a solution that does a linear scan of an array of length n from the input, then its time complexity is $\Theta(n)$, although in the competitive programming world it is more common to use a less powerful estimate even if we know the lower bound to be tight. Namely, we would say that the solution

runs in $O(n)$ time. We can do this estimate because regardless of what programming language we are using, or what machine we are running the solution on, the time needed to do a basic operation such as a constant read, a sum of two numbers, or a comparison of two numbers, is some constant time $\Theta(1)$.

With this building blocks we can now express how good (or bad) is a solution for a certain problem. This gives the competitor the ability to do some crude guesses on the ability of a solution to solve a problem within the resource limits imposed by the problem. For instance, it is common practice to estimate that the number of operations that a computer can do in one second is around $10^7 \sim 10^8$. Thus, if we have a solution that has a time complexity of $O(n^2)$, the maximum input size that we can solve in one second is around 10^4 . If we see that the problem requires us to solve an input of size 10^5 , and we have a quadratic solution, then we can already guess that it will not be fast enough to solve the problem.

2.2.2 Dynamic programming

Dynamic programming [14] is a problem solving technique that tries to solve a problem by breaking it down into smaller pieces, solving those pieces, and then combining the solutions to solve the original problem. Dynamic programming bases itself on the ideas of recursion.

Recursion

Recursion⁶ is a way to model solutions to problems that have the following properties:

- If the input is small enough⁷, then the solution can be computed trivially. This is call the *base case*.
- For all other inputs, we can break down the input into smaller pieces (usually called *subproblems*), and we can compute the solution for the original input by combining the solutions of the smaller pieces. This is called the *general case*.

If a problem has this properties, then we can solve it by designing a recursive function that on the *base cases* just returns the solution directly, since it is trivial to compute, and on the *general case* breaks down the input into smaller pieces, calls itself recursively on those pieces, and then uses the solutions of the smaller pieces to compute the solution of the original input.

Example

You have a bathroom that is composed of $n \times 1$ empty cells. You want to cover it with tiles, such that each cell is covered by some tile and no two tiles overlap. You have an

⁶To understand *recursion* you first need to understand *recursion*.

⁷It also might be the case that the input is big, but has some special properties that makes it trivial to solve

unlimited amount of tiles of size 1×1 and 2×1 . How many ways are there to cover the bathroom with tiles?



Let's first try to enumerate all the possible ways to cover the bathroom with tiles for small values of n .

With $n = 0$ there is only 1 way to cover the bathroom, which is doing nothing.


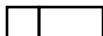
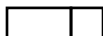
With $n = 1$ there is also only 1 way to cover the bathroom, which is using a single 1×1 tile:

- 

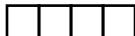
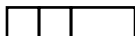
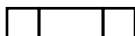
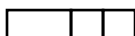
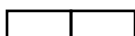
With $n = 2$ there are 2 ways to cover the bathroom, which are using two 1×1 tiles, or using a single 2×1 tile:

- 
- 

With $n = 3$ there are 3 ways to cover the bathroom:

- 
- 
- 

With $n = 4$ there are 5 ways to cover the bathroom:

- 
- 
- 
- 
- 

With some patience we can enumerate all the possible ways to cover the bathroom with tiles for small values of n . However, as n gets bigger it becomes harder and harder to enumerate them because the number of possibilities grows, and it turns out that it grows *exponentially*. So by working by hand and *guessing* all the possibilities there is a high probability, as n grows, that we will miss some of them.

To solve this problem *methodically* we can try to model it as a recursive function. Let's call this function $f(n)$. We first need to find the *base cases* of the function. We already computed by hand the values of $f(n)$ for n up to 4, so, if needed, we can use them as *base cases*.

For figuring out how to define the general case, we need a key observation: if we need to cover an $n \times 1$ bathroom with tiles, the last tile we put will either be a 1×1

tile, or a 2×1 tile. If it is a 1×1 tile, then we need to cover the remaining $n - 1$ cells, and if it is a 2×1 tile, then we need to cover the remaining $n - 2$ cells. However, the problem of covering $n - 1$ or $n - 2$ cells is the same problem we are trying to solve, but smaller, so we can just delegate its computation to the function f itself.

For this observation to be true, the bathroom must at least have 2 cells, otherwise we can't put a 2×1 tile in it. Thus, the only *base cases* we need are $f(0) = 1$ and $f(1) = 1$. For the general case we know that if we put a 1×1 tile as the last one the number of ways to cover the bathroom is $f(n - 1)$, and if we put a 2×1 tile as the last one the number of ways to cover the bathroom is $f(n - 2)$. Thus, the general case is $f(n) = f(n - 1) + f(n - 2)$.

We can now write a recursive function that computes the number of ways to cover a bathroom of size n :

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ f(n - 1) + f(n - 2) & \text{otherwise} \end{cases}$$

By using this function we now have a *methodical* way to compute the answer. However, if we try to use this definition alone, we will run into a *performance* problem: note that the only number that we can start adding up from is 1. This means that if the answer is k , we will need to at least evaluate the function f for k times.

Now that we have the function f , let us see the values of $f(n)$ for n up to 20:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946

Some may have noticed that the answers to this problem are actually the numbers in the Fibonacci sequence [15]. This is because the function f we defined is actually the recursive definition for the Fibonacci sequence. This sequence grows exponentially, which means that the number of operations that we need to compute the answer will grow exponentially as well. However we can speed up the computation by using the aforementioned *dynamic programming* technique.

Memoization

Let's look at the pseudocode for the computation of the f function in Figure 2. The basic issue with this recursive approach is that we are computing the same values over and over again. For instance, if we want to compute $f(5)$, we will need to compute $f(4)$ and $f(3)$, and to compute $f(4)$ we will need to compute $f(3)$ and $f(2)$, and so on. This means that we will compute $f(3)$ twice, and $f(2)$ three times. One key observation is that we can store the values of f that we already computed, and reuse them when we need them again. This technique is called *memoization*.

By memorizing the values of f , if we now compute $f(5)$ we will first call $f(4)$ and $f(3)$, $f(4)$ will call $f(3)$ and $f(2)$, and so on. When the computation of $f(3)$ will have


```

f(n):
  if n ≤ 1:
    return 1
  else:
    return f(n - 1) + f(n - 2)

```

Figure 2: The pseudocode for the recursive function f .

finished, we will store the value of $f(3)$ for future use, since it does not matter how or how many times we compute $f(3)$, the result will not change. Once $f(4)$ will have returned, we will still need to use the value of $f(3)$, but this time we will not need to compute it again, since we already have it stored. This means that we will only compute $f(3)$ once, and we will reuse it every time we need it.

One has now to wonder: how much time are we saving by using *memoization*? And how much memory are we using to store the values of f ? To figure out the first answer we need to know two things:

- how many different states do we have to visit in our computation?
- how much time does it take to compute each state?

For this example, both of these questions are fairly easy to answer. When computing $f(n)$, for the definition of f , it will compute $f(n)$, $f(n - 1)$, $f(n - 2)$, $f(n - 3)$, ..., all the way down to $f(1)$ and $f(0)$. It cannot go up or go further down than that, since the definition of f does not allow it. Thus, the number of states that we need to visit is $n + 1$, which is $\Theta(n)$. For the second question, both cases in the definition of our function f only require a constant amount of time to compute: in the *base case* we only need to return a constant value, and in the *general case* we only need to add two numbers that we can look up in our *memoization* table. Thus, the time it takes to compute each state is $\Theta(1)$.

Since the number of states that we can visit while computing our answer is $\Theta(n)$, and the time it takes to compute each state is $\Theta(1)$, the total time it takes to compute the answer is $\Theta(n) \cdot \Theta(1) = \Theta(n)$. This is a huge improvement over the exponential time it took before. The formula to compute the time complexity of a dynamic programming solution in general is *number of states* \times *time to compute each state*.

What about the memory usage? How much memory do we need to store the values of f ? The answer is similar to the previous one: we need to store $n + 1$ values, this $\Theta(n)$, and each value takes a constant amount of memory, thus $\Theta(1)$. It follows

```

f(n):
  let a = 1
  let b = 1
  for i = 1 to n:
    let c = a + b
    a := b
    b := c
  return a

```

Figure 3: The pseudocode for the *bottom-up* version of f .

that to store the values of f we need $\Theta(n) \cdot \Theta(1) = \Theta(n)$ memory. Like for the time complexity, this formula holds in general: the amount of memory needed to store the values of a dynamic programming solution in the *memoization* table is *number of states* \times *memory needed to store each state*.

Top-down vs bottom-up

This approach that we have taken to solve the problem using dynamic programming is called *top-down*. To refresh our memories let's look at the pseudocode for the recursive function f in Figure 2⁸. This approach is called *top-down* because we start from the top of the recursion tree, go through the *general cases*, and we go down until we reach the *base cases*. However, this is not the only way to do it.

Another way to do it is to start from the *base cases*, go up through the *general cases*, until we reach the top of the recursion tree. This approach is called *bottom-up*. The key to *bottom-up* dynamic programming is to start from the *base cases*, and then compute the *general cases* in order, until we reach the result we need. This is done for the bathroom tiling problem in as shown in Figure 3.

We start with a and b set to 1, since those are the values of $f(0)$ and $f(1)$. Then, we compute $f(2)$ by adding a and b together, and storing the result in c . Then, we update a and b to be b and c respectively, and we repeat the process until we reach $f(n)$. At this point, a will contain the value of $f(n)$, and we can return it.

The *bottom-up* approach has some benefits over the *top-down* one. The first one is that it avoids the use of recursion, which, when executed on a real-world machine, has an overhead in its execution. The second one is that it can avoid the storage of

⁸Note that it lacks the memorization of the results.

all the values of f in the *memoization* table, since it only needs to store the ones that are needed for the computation of the next value. This may improve drastically the memory utilization for some problem. For instance, as can be seen with the *bottom-up* approach for the bathroom tiling problem in Figure 3, we only need to store the last two values of f to compute the next one, so we only need $\Theta(1)$ memory to compute $f(n)$. However, note that this is not true in general, and the *bottom-up* approach can still need all the previously computed values to compute the next one.

However, the *bottom-up* approach has some drawbacks as well. The main one is that it is generally harder to find the correct formulation, since we need to think backwards about the problem and come up with which pieces we need to compute first, and how to combine them to compute the next ones. Another one is that the function f may not have a trivial ordering of the states, and it may be hard to figure out which states we need to compute before we can move to the next ones. This is not true with the *top-down* approach, since the ordering of the states is implicit in the recursion definition.

2.2.3 Greedy algorithms

When modeling a solution to a problem with a recursive function, and then using dynamic programming to speed it up, the *general case* does the following: when presented with a choice of what to do next, it tries all the possible choices, and then it either combines them in some way, or it picks the best one. We have to try all the possible choices because we don't know *a priori* which one is the best one.

Let us take as an example the following problem. You have a list of n numbers, and you want to pick a subset of them such that the resulting sequence is alternating. That is, if you start from the first number, then the second number must be smaller than the first one, the third number must be bigger than the second one, the fourth

```

f(i, d, v, n):
  let ans = 1
  for j = i + 1 to n - 1
    if d is up and v[j] > v[i]
      ans := max(ans, f(j, down, v, n) + 1)
    if d is down and v[j] < v[i]
      ans := max(ans, f(j, up, v, n) + 1)
  return ans

```

Figure 4: The pseudocode for the dynamic programming version of f .

number must be smaller than the third one, and so on. Or vice versa, if you start from the first number, then the second number must be bigger than the first one, the third number must be smaller than the second one, the fourth number must be bigger than the third one, and so on. How many numbers can you pick at most?

We can model this problem with a recursive function f that takes as input the index of the number we are currently considering, and the direction we are going in (either *up* or *down*). What we want to do is try to *attach* the next number to the current sequence in the direction we are going in, and to that we have potentially up to n choices, because each time we can try attach all the higher numbers, if going up, or all the lower numbers, if going down, after the current one. After having tried them all, we can pick the choice that gives us the best result, and return it. The code for this strategy is shown in Figure 4. To solve the problem with this function f we just need to try all the possible starting points, and for each of them we need to try both directions, and then pick the best result among all of them. Note that we can apply *memoization* to f .

How fast is this solution? Since this is a dynamic programming solution, the time complexity is *number of states* \times *time to compute each state*. The number of states is $2 \cdot n$, since we have n possible starting points, and for each of them we have two possible directions. The time to compute each state is $O(n)$, since we need to try all the possible choices, and they can be up to n . Thus, the total time complexity is $O(n^2)$. Can we do better?

Local optimality

Indeed, we can. In the previous approach when we were confronted with a choice of what to do next, i.e. which next number to attach to the current sequence, we tried all the possible choices and picked the best one, because in general when we have to choose between multiple ones we don't know which one will lead to the best global result. That is, in general choosing the best local result does not lead to the best global result. However, in this problem, this isn't the case. If we are going up, we know that at some point we will have to go down, so if we have a sequence of increasing numbers, the best choice is always to attach only the last one of the increasing subsequence. The same is true if we are going down. Knowing this, we don't have to scan all the possible choices, but we can just pick the best local one, in this case the last number of the current upwards or downwards streak. This approach is called *greedy*, because when presented with immediate opportunities, it always picks the best one without thinking about the future.

It is important to say that the greedy approach is not valid in general. If the problem does not have the property that the best local choice leads to the best global choice, then the greedy approach will find a suboptimal solution. However, in the cases where it works, like with this one, it is usually much faster than the dynamic programming approach. The code for the greedy approach is shown in Figure 5. The

```

f(v, n):
  if n ≤ 1:
    return n
  let ans = 1
  if v[0] < v[1]
    let d = up
  else:
    let d = down
  for i = 1 to n − 1
    if d is up
      if v[i] < v[i − 1]
        d := down
        ans := ans + 1
    else:
      if v[i] > v[i − 1]
        d := up
        ans := ans + 1
  return ans

```

Figure 5: The pseudocode for the greedy version of f .

time complexity of this approach is $O(n)$, since we only need to scan the input once, which is a big improvement from the $O(n^2)$ of the dynamic programming approach.

2.2.4 Divide and conquer

Divide and conquer, or *divide et impera* in Latin, is yet another problem solving technique that bases itself on the ideas of recursion. The idea of *divide et impera* is to break down a problem into smaller pieces, solve those pieces, and then combine the solutions to solve the original problem. This is similar to dynamic programming, but the difference is that in dynamic programming the pieces are usually overlapping, while in *divide et impera* they are a partition, with possibly some missing pieces that are discarded, of the original problem.

To illustrate this technique, let us tackle the problem of sorting a list of numbers. You have a list of n numbers, and you want to sort them in increasing order. How can we do this? There are many algorithms that can solve this problem, from the more naive ones, like *bubblesort* [16], to more advanced ones, like *Shellsort* [17], *Timsort*

```

quicksort(v):
  if  $\text{len}(v) \leq 1$ 
    return v
  let p = v[0]
  let l = []
  let r = []
  for i = 1 to  $\text{len}(v) - 1$ 
    if  $v[i] < p$ 
      l := l + [v[i]]
    else:
      r := r + [v[i]]
  return quicksort(l) + [p] + quicksort(r)

```

Figure 6: The pseudocode for the *quicksort* algorithm.

[18], *introsort* [19] and *quicksort* [20]. The last one mentioned, *quicksort*, will be our focus since it uses a *divide et impera* approach to sort the numbers.

The idea behind *quicksort* is a recursive one. If the array we are trying to sort is empty or has only one element, then it is already sorted, and we don't need to do anything on it. Otherwise, we can pick an element from the array, called the *pivot*, and then we can partition the array (minus the *pivot*) into two parts: the first part contains all the elements that are smaller than the pivot, and the second part contains all the elements that are not smaller than the pivot. Having done this partition, we know that all the elements in the first part will have to go before the pivot when sorted, and all the elements in the second part will have to go after the pivot when sorted. Thus, the only thing left to do is to sort the first part and the second part. To do that we can just call the same procedure recursively on the first part and the second part, and let the recursion do its job. Once the recursion has finished, we will have two sorted arrays, one containing all the elements that are smaller than the pivot, and one containing all the elements that are not smaller than the pivot. To get the final sorted array we just need to concatenate the two sorted arrays with the pivot in the middle. This is illustrated in Figure 6.

But how fast is this algorithm? The time complexity of *quicksort* depends on the choice of the pivot. If we were to always pick the element that turns out to be the median of the array, then the time complexity would be $O(n \log n)$. This is because for each depth of the recursion we need to scan the whole array once, and this costs

$O(n)$, and the depth of the recursion is $O(\log n)$, since each time we partition the array we divide it in two parts, and we stop when we reach an array of size 1. However, if we were to always pick the first element of the array as the pivot, then the time complexity would be $O(n^2)$. This is because if we pick the first element as the pivot, and the array is already sorted, then we will always partition the array into an empty array and an array of size $n - 1$, and we will do this n times, so the resulting time complexity will be $O(n^2)$.

To mitigate the issue of picking a bad pivot, we have two options. The first one is to pick a random element as the pivot. This way, the probability of picking a bad pivot is low, and the expected time complexity is $O(n \log n)$. The second one is to pick an element which is close to the median of the array. This way, we can guarantee that the two partitions are always balanced up to some factor. To select such a *pivot* we can use the *median of medians* algorithm [21], which can find an element in a region near the median of an array in $O(n)$ time. This way, the time complexity of *quicksort* can be guaranteed to be $O(n \log n)$.

Thus, as illustrated by *quicksort*, *divide et impera* bases itself on the idea of breaking down a problem into smaller independent pieces, solving those pieces, and then combining the solutions to solve the original problem. Other examples of *divide et impera* algorithms are *binary search*, *branch and bound* [22] and *fast Fourier transform* [23].

2.2.5 Strings

Strings are a fairly common appearance in competitive programming. Strings are used to represent text, and they are usually represented as an array of characters. In competitive programming the most common alphabet from which the characters are taken are the lowercase letters of the English alphabet, i.e. a, b, c, \dots, z . In any case it is very rare to see strings that have characters outside of the ASCII range in a competitive programming problem.

Since strings are nothing more than an array of characters, we can use all the techniques that we use on arrays on strings as well. For instance, it is possible that the solution to a problem involves sorting a string, or doing dynamic programming on a string. However, there are some problems and techniques that are specific to strings⁹.

String matching

String matching is the most basic problem about strings in competitive programming. You have a string t , usually called the *needle*, and a string s , usually called the *haystack*. You want to find out if the *needle* appears in the *haystack*, in other words, if t appears as a substring of s . For instance, if $s = \text{banananasso}$ and $t = \text{ananas}$, then

⁹Technically, any integer array can be represented as a string with a sufficiently large alphabet, so we could use string algorithms on plain integer arrays.

```

matches(s, t):
  let n = len(s)
  let m = len(t)
  for i = 0 to n - m - 1
    let j = 0
    while j < m and s[i + j] = t[j]
      j := j + 1
    if j = m
      return true
  return false

```

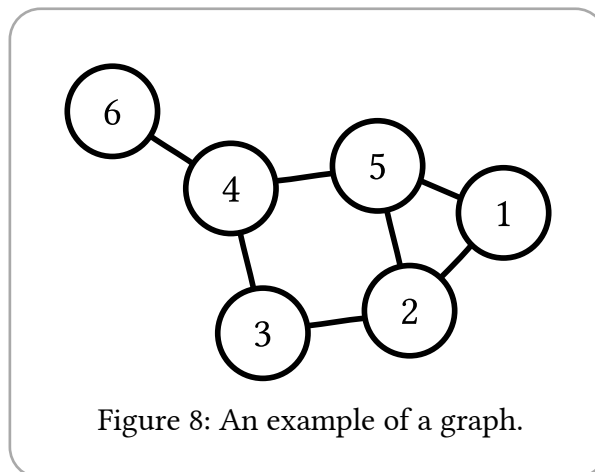
Figure 7: The pseudocode for naive string matching.

t appears in s starting from the fourth character. If $t = \text{pear}$, then t does not appear in s .

Variations of this problem include counting the number of times the *needle* appears in the *haystack*, or finding all the positions where the *needle* appears in the *haystack*. To solve the basic problem of whether the *needle* appears or not in the *haystack*, we can use the naive algorithm shown in Figure 7. The idea behind this algorithm is to scan the *haystack* from left to right, and for each position check if the *needle* appears starting from that position. While this algorithm works, how fast is it? If we call, as it is usually done when talking about this problem, n the length of the *haystack*, and m the length of the *needle*, then the outer loop will run up to $n - m$ times, which is $O(n)$. The inner loop will run up to m times, which is $O(m)$. Thus, the total time complexity of this algorithm is $O(nm)$.

In everyday practice, it is rare that this algorithm reaches its worst case scenario of $O(nm)$, since it is usually only necessary to check a few characters of the *needle* before the inner loop rejects the current match. However, in competitive programming it is very common to see test cases that are designed to make the algorithms run in their worst case scenario. For instance, we could have a test case where $s = a^{1000000}$ and $t = a^{1000}b$. In this case, the inner loop will not reject the current match until it has scanned all the a 's of the *needle*, and thus it will do $\Theta(nm)$ operations.

Can we do better? Yes, there exist algorithms that can solve this problem in $O(n + m)$ time, such as the *Knuth-Morris-Pratt* algorithm [24], the *Aho-Corasick* algorithm [25], the *Rabin-Karp* algorithm [26] and the *Z-algorithm* [27, 28]. Variations of these algorithms can also lead to solving different problems specific to strings, like finding



the longest palindrome in a string, which can be done with a variation of the *Z-algorithm* called *Manacher's algorithm* [29].

2.2.6 Graphs

Graphs are a very common topic in competitive programming. A graph is a collection of nodes (or vertices) that are connected by edges. Usually, the formal way to describe a graph in a mathematical way is to say that a graph is a $G = (V, E)$, where V is the set of nodes, and $E \subseteq V \times V$ is the set of edges. An example of a graph is shown in Figure 8.

Graphs come in many varieties, some of which are:

- *directed* and *undirected*: in a directed graph the edges have a direction, while in an undirected graph the edges don't have a direction.
- *weighted* and *unweighted*: in a weighted graph the edges have a weight, while in an unweighted graph the edges don't have a weight.
- *simple* and *non-simple*: in a simple graph there are no self-loops and no multiple edges, while in a non-simple graph there can be self-loops and multiple edges.

```
dfs(v, G, p):
  p[v] := true
  for u in G[v]
    if not p[u]
      dfs(u, G, p)
```

Figure 9: The pseudocode for a depth-first search.

```

dfs( $v, G$ ):
  let  $n = \text{len}(G)$ 
  let  $p = [\text{false}]^n$ 
  let  $q = [v]$ 
  while not empty( $q$ )
    let  $v = \text{pop}(q)$ 
    if not  $p[v]$ 
       $p[v] := \text{true}$ 
      for  $u$  in  $G[v]$ 
        if not  $p[u]$ 
          push( $q, u$ )

```

Figure 10: The pseudocode for a breadth-first search.

Some combinations of these properties have special names. For instance, a graph that is directed and does not contain cycles is called a *directed acyclic graph* (DAG), while a graph that is undirected and does not contain cycles is called a *tree*.

Graphs can be represented in many ways. The most basic one is called an *adjacency matrix*. An adjacency matrix is a matrix A of size $n \times n$, where n is the number of nodes in the graph, such that $A_{i,j} = 1$ if there is an edge from node i to node j , and $A_{i,j} = 0$ otherwise. This representation has the advantage that is very easy to check if there is an edge between two nodes, since we just need to check the value of the corresponding cell in the matrix. However, it has the disadvantage that it takes $\Theta(n^2)$ memory, which can be a big value if the graph has many nodes, and a waste of memory if the graph is sparse, i.e. it has few edges. Also, knowing which nodes are connected to a given node requires a linear scan of a whole row of the matrix.

A more compact, and almost universally used in competitive programming, representation of graphs is called the *adjacency list*. An adjacency list is an array of dynamic arrays of integers. The array has size n , where n is the number of nodes in the graph, and each element of the array is a dynamic array of integers. The i -th element of the array contains the list of nodes that are connected to node i . This representation has the advantage that it takes $\Theta(n + m)$ memory, where n is the number of nodes and m is the number of edges, which is a big improvement over the adjacency matrix representation if the graph is sparse. Also, to know which nodes are connected to a given node we just need to look at the corresponding dynamic array in the array. This comes at the cost of having to scan a whole dynamic array if we want to know

if there is an edge between two nodes, but this operation usually is not needed, and if we need it we can just use a hash table of the edges.

There are many problems that can be modeled as graphs. For instance, the problem of finding the shortest path between two intersections in a city can be modeled as a graph, where the nodes are the intersections, and the edges are the roads. There also exists many algorithms that work on graphs. For instance, the problem of finding the shortest path between two nodes in a graph can be solved with *Dijkstra's algorithm* [30].

Although there are many algorithms on graphs, there are two that are the building blocks for many others. The first one is the *depth-first search* (DFS). The idea behind DFS is to visit all the nodes in a graph, and to do that we start from a node, and then we visit all the nodes that are reachable from that node, and then we visit all the nodes that are reachable from those nodes, and so on. This is done by using recursion, as shown in Figure 9.

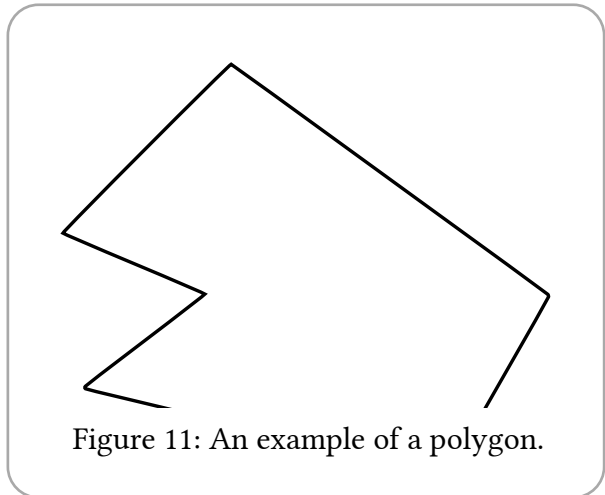
The second one is the *breadth-first search* (BFS). The idea behind BFS is to visit all the nodes in a graph, and to do that we start from a node, and then we visit all the nodes that are reachable from that node, and then we visit all the nodes that are reachable from those nodes, and so on. This might sound familiar, and that's because it is the same idea behind DFS. However, while DFS uses recursion, BFS uses a queue, as shown in Figure 10.

While both DFS and BFS achieve the task of visiting programmatically all the nodes in a graph, they do it in different ways. DFS visits the nodes in a depth-first order, going deep in the graph until it reaches a dead end, and then backtracking and going deep again. BFS visits the nodes in a breadth-first order, visiting all the nodes that are at the same distance from the starting node, and then visiting all the nodes that are at the next distance from the starting node, and so on. These two algorithms form the base for many other algorithms on graphs, and they are used in many problems.

2.2.7 Computational geometry

Computational geometry is a topic that sometimes appears in competitive programming problems. Computational geometry problems involve geometric objects such as points, lines, polygons, circles, and so on. In competitive programming computational geometry is almost exclusively confined to two dimensions.

Computational geometry problems usually require a lot of basic knowledge of algebraic operations on geometric objects, like addition, subtraction of vectors, doing the dot and cross products on vectors, computing the norm of a vector, and so on. These operations are usually not hard to implement, but they are very error prone, and it is very easy to make a mistake or remember a formula with a slight mistake in it, like forgetting a minus sign.



```
area(P):  
  let n = len(P)  
  let a = 0  
  for i = 0 to n - 1  
    let j = (i + 1) mod n  
    a := a + P[i]x · P[j]y - P[i]y · P[j]x  
  return a / 2
```

Figure 12: The pseudocode for calculating the area of a polygon.

Other than the basic operations, there are some properties that are important to know in order to exploit them to solve problems. The main one is convexity: a polygon is convex if for any two points inside the polygon, the line segment that connects them is also inside the polygon. A non-convex polygon is shown in Figure 11. Another important aspect is the fact that when dealing with computation geometry problems, we are also dealing with floating point numbers, and thus we have to be careful when doing comparisons between them, since two numbers that should be equal might end up being slightly different due to the imprecision of floating point numbers.

Aside from these basic operations and theory, and the problems that arise from them, the two most common problems, and thus algorithms for solving them, in computational geometry are the *convex hull* problem and the *polygon area* problem. Some of the algorithms to compute the convex hull of a set of points are *Gift wrapping* [31], *Graham scan* [32], *Quickhull* [33], *Monotone chain* [34] and *Kirkpatrick-Seidel* [35].

The *polygon area* problem is the problem of computing the area of a polygon. The polygon might not be convex, but it shouldn't be self-intersecting. The common algorithm to solve this problem is *shoelace formula* [36, 37]. The idea behind this algorithm is to compute the area of the polygon as the sum of the areas of the triangles that are formed by the edges of the polygon and a fixed point. The fixed point can be any point, but it is usually chosen to be the origin, since it simplifies the computation of the area of the triangles. The pseudocode for this algorithm is shown in Figure 12.

Aside from having to know a lot of theory to approach computational geometry problems, these problems tend to require creative solutions, not unlike the ones for the other categories, but at the additional cost of the knowledge burden. This makes computational geometry problems more suitable for more experienced competitive programmers.

2.2.8 Number theory

Number theory is a topic that appears in competitive programming problems either as in the form of the problem itself, or as a tool needed to solve some aspect of prob-

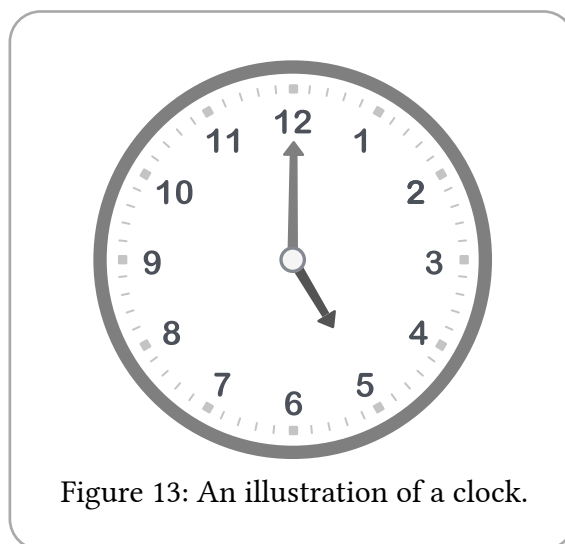


Figure 13: An illustration of a clock.

```
gcd(a, b):  
  if b = 0  
    return a  
  else  
    return gcd(b, a mod b)
```

Figure 14: The pseudocode for the greatest common divisor.

```

modexp( $b, e, m$ ):
  if  $e = 0$ 
    return 1
  let  $r = \text{modexp}(b, e/2, m)$ 
   $r := (r \cdot r) \bmod m$ 
  if  $e \bmod 2 = 0$ 
    return  $r$ 
  else:
    return  $(r \cdot b) \bmod m$ 

```

Figure 15: The pseudocode for the fast modulo exponentiation algorithm.

lems on other topics. Number theory is the study of the integers and their properties. Most of the number theory used in competitive programming is modulo arithmetic, which is the study of the integers modulo a given number m .

In modulo arithmetic, also called *clock arithmetic*, the integers are represented as a circle of size m , and the operations are defined as if the integers were on a clock. For instance, if $m = 12$, then $1 + 1 = 2$, $3 + 7 = 10$, but $11 + 3 = 2$, since $11 + 3 = 14$, and 14 is equivalent to 2 modulo 12. This is the same as saying that $11 + 3 = 2$ if we were to use a clock to represent the integers, since 3 hours after 11 o'clock is 2 o'clock. An illustration of this is shown in Figure 13.

We can also define the other basic operations on integers in modulo arithmetic. For instance, if we want to compute $a - b \bmod m$, we can do so as $a + (m - b) \bmod m$. This is because adding m to a number does not change its value modulo m , but it ensures that we are dealing with a positive number. Multiplication is fairly straightforward, since we can just multiply the two numbers and then take the result modulo m . Division is a bit more complicated. In competitive programming, when modulo division is involved, the modulo m is always a prime number, because it makes division much easier. In fact, division is performed by finding the multiplicative inverse of the divisor modulo m , and then multiplying the dividend by the multiplicative inverse. The multiplicative inverse of a number $a \bmod m$, if m is prime, is $a^{m-2} \bmod m$, due to *Fermat's little theorem*.

On the algorithms side, one of the common number theory algorithms, that is also useful for computing the multiplicative inverse quickly, is the *fast modulo exponentiation* algorithm. The pseudocode for it is shown in Figure 15. The key idea behind it is the observation that if we want to compute $b^e \bmod m$ and e is even, we can compute

$b^{\frac{e}{2}} \bmod m$ and then square it. This avoids having to compute $b^e \bmod m$ directly, which otherwise would take $O(e)$ time. With this observation, we can compute $b^e \bmod m$ in $O(\log e)$ time by using that observation in the even cases, and just multiplying by b in the odd cases.

Another common number theory algorithm is the *greatest common divisor* (GCD) algorithm. The greatest common divisor of two numbers a and b is the largest number that divides both a and b . There are actually many algorithms to compute the GCD, such as *Lehmer's algorithm* [38] and the *binary GCD algorithm* [39]. However, the most common one is the *Euclidean algorithm*, which is shown in Figure 14. The GCD can be used in a standalone way in competitive programming, where the problem is to compute the GCD of some numbers in some fashion, or it can be used as a building block for other algorithms.

2.3 Programming languages

When doing competitive programming, it is important to choose and know well a programming language. There are programming languages that are more suitable for competitive programming than others, and there are programming languages that are more suitable for some problems than others.

In this section we will discuss some of the most common programming languages used in competitive programming, and we will discuss their points of strength and weakness. Note, however, that the overwhelming majority of competitive programmers use C++ as their main programming language [40].

2.3.1 C

The C programming language was created in 1972 [41], and it is one of the most used programming languages in the world, particularly in the field of systems programming. It is considered a low-level programming language, since it is very close to the hardware, and it is very fast. It is also a quite simple¹⁰ programming language. An example of a program written in C is shown in Listing 1.

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return EXIT_SUCCESS;
}
```

Listing 1: Hello world in C.

¹⁰C is simple in the sense that does not provide too many abstractions from the hardware.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return EXIT_SUCCESS;
}
```

Listing 2: Hello world in C++.

In competitive programming C has been a staple language for a long time, however it has been superseded by C++ over the years. C has the advantage of being very fast, and thus it is a good choice for competitive programming, where you want your programs to be as fast as possible. However, it has the disadvantage of not providing many abstractions that are useful for competitive programming, even the most basic ones like dynamically sized arrays.

Having C++ that is a superset¹¹ of C, with C++ having the same speed as C and providing a very rich set of abstractions in its standard library, most competitors have migrated to, or directly started with, C++.

2.3.2 C++

The C++ programming language was created in 1985 [42], and is by far the most used programming language in competitive programming. Sometimes considered a high-level programming language, sometimes considered a low-level one, C++ is a programming language with a very rich set of features, and a very large standard library. An example of a program written in C++ is shown in Listing 2.

As speed goes, C++ is as fast as C, so it is a good choice for competitive programmers in this regard. C++ provides a huge standard library that contains many useful abstractions when solving competitive programming problems. For instance, it provides dynamic arrays, hash tables, sets, priority queues, stacks, and so on. It also provides many useful algorithms, like binary search, sorting, string matching, and so on. This gives competitive programmers using C++ a big advantage over competitive programmers using other languages with less rich standard libraries.

One slight disadvantage of C++ is that it is a very vast programming language, and thus it takes a long time to learn it well. Although this is usually not a problem for competitive programmers, since they usually only use a very small subset of the language. Another problem is that compilation error messages can be very cryptic,

¹¹Technically C++ is not a strict superset of C for the lack of a couple of features exclusive to C, like the restricted attribute.


```
program helloworld;  
begin  
  writeln('Hello world!');  
end.
```

Listing 3: Hello world in Pascal.

especially when related to the standard library and templates. C++ also suffers from the dreaded *segmentation fault*, which is a runtime error that happens when a program tries to access memory that it shouldn't access. This is the number one reason of crashes in competitive programming C++ solutions, and it can be very hard to debug. However, there are now powerful tools that can help with this, like *valgrind* [43] and the *address sanitizer* [44].

2.3.3 Pascal

Pascal is a programming language that was created in 1970 [45], and it was one of the first programming languages to be designed with the goal of being easy to learn and use. It gained a lot of traction in the beginning of competitive programming competitions, however it has lost traction over the years, mainly due to the rise of C and C-like languages in competitive programming. An example of a program written in Pascal is shown in Listing 3.

Pascal got a good traction in the beginning of competitive programming competitions because it was the language of choice to be taught in many schools and universities. This meant that many competitors were already familiar with it, and thus they used it in competitions. However, as time went on, C and C-like languages became more and more popular, and thus Pascal lost traction. Nowadays, Pascal is rarely used in competitive programming competitions.

On the technical side, Pascal has similar advantages and disadvantages as C. It is a very fast programming language, and it is quite simple. However, it does not provide many abstractions, and thus it is not very suitable for competitive programming in the current landscape.

2.3.4 Python

```
if __name__ == "__main__":  
    print("Hello world!")
```

Listing 4: Hello world in Python.

The Python programming language was created in 1991 [46], and it is one of the most on-the-rise programming languages in the world. It is considered a high-level programming language, and it is easy to learn and to use. Its use has become pervasive in machine learning and data analysis applications [47, 48, 49]. An example of a program written in Python is shown in Listing 4.

Python is becoming a popular programming language among competitive programmers, and on some platforms, like *Codeforces*, it is the second most used programming language after C++ [40]. This is due to its strengths, which are its simplicity of writing code using it, and its *batteries included* approach, which means that it provides a very rich standard library. This makes it a very good choice for competitive programming, since it allows to write code very quickly, as it provides many useful abstractions, allowing to write very concise code.

However, Python has a very big disadvantage, which is its speed. Python is a very slow programming language, and thus it is not a good choice for competitive programming in this regard, since it is very important for competitive programming solutions to be fast. Some mitigations to this problem include having different time limits if submitting a solution in Python, which is sometimes seen as an unfair advantage, and using the *PyPy* [50] implementation of Python, which is a just-in-time compiler for Python that can make Python programs run much faster.

2.3.5 Java

The Java programming language first appeared in 1995 [51, 52], and it is a very used programming language in the corporate and business world. It is considered a high-level programming language, and it is relatively simple. An example of a program written in Java is shown in Listing 5.

The popularity of Java in competitive programming is similar to the one of Python, although while Python's popularity is on the rise, Java's popularity is on the decline. This is because Java provides strengths and weaknesses that are similar to the ones of Python, but in a more muted way. Java is not as slow as Python, but it is not as fast as C either. Also, Java is very verbose, and thus it is not as quick to write code in it as it is in Python. Still, Java, on some platforms, like *Codeforces*, is the third most used programming language after C++ and Python [40].

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Listing 5: Hello world in Java.

Historically Java has seen little use at the top of the competitive programming scene, with most of the top competitors using C++. It was seen as a valid choice for solving a problem only if the problem required some abstraction that was not available in C++, but was available in Java, like integers with arbitrary precision.

3

Competitions

In the previous chapter we talked about how solving problems is the core of competitive programming. In this chapter we will expand on that and talk about problems and people are brought together to create competitive programming competitions.

While solving problems is already by itself an interesting activity, making a competition out of it adds a lot of value. Competitions are a great way to motivate people to solve problems, and to bring people together to solve problems. Competitions are also a great way to learn from others, and to share your knowledge with others.

There exist many different competitions that are held regularly all over the world, both offline and online. In this chapter we will focus on the two most important and prestigious ones: the *International Olympiad in Informatics* (IOI) and the *International Collegiate Programming Contest* (ICPC). But before we dive into these competitions, let us first briefly talk about the other competitions:

- The *Google Code Jam* [53] was a yearly online competition organized by Google. It was held from 2008 to 2022. It was the largest competition held by a private company. Its format was a multi-round contest where the top contestants from each round would advance to the next round. The final round would be held on-site at the Google offices. The contest format was a testcase-based format, where contestants would ask the system to generate a set of testcases, then download the testcases, compute the answers locally and upload the answers to the system. This was the case for all editions except for the last two, where the solutions would be evaluated on their servers. The scope of the competition was to assist Google in finding talented programmers to hire.
- The *Meta Hacker Cup* [54] is a yearly online competition organized by Meta. It has been held since 2011. Its concept and format are identical to the Google Code Jam of the first years. The scope is also the same, to assist Meta in finding talented programmers to hire. Unlike the Google Code Jam, the Meta Hacker Cup is still being held, and it is now the largest competition held by a private company.
- *Codeforces* [55] is a website that hosts online competitions. It was launched in 2009 by Mike Mirzayanov and other competitive programmers. It is the largest online competitive programming platform. It hosts competitions regularly and has a huge library of problems. It also has a large community of competitive programmers that share their knowledge on the platform using blog posts and comments.
- *AtCoder* [56] is another website that hosts online competitions. It is the second biggest online competitive programming platform after Codeforces and it is

based in Japan. While not as big as Codeforces, AtCoder has a reputation for having high quality problems.

Aside from the major competitions, there are also many smaller competitions, from the minor international ones, to the local ones organized by universities and schools. These competitions are usually organized by competitive programmers themselves, and they are a great way to get started with competitive programming and to start learning more about it, while also having fun.

3.1 International Olympiad in Informatics

The International Olympiad in Informatics (IOI) [57] is the most prestigious competition for competitive programmers. It is an individual annual competition for high school students that was founded in 1989. Every year, each nation can send a team of up to four contestants to the competition. The competition is held in a different country every year. The competition lasts two days, and each day the contestants are given three problems to solve. The contestants are given five hours to solve the problems for each day.

The IOI is considered to be the most important competition for competitive programmers. To be able to participate in the IOI, your country must select you to be part of the team that it will send to the competition. Each country has its own selection process.

As far as the format of the competition goes, the IOI is a two day competition where each day the contestants are given three problems to solve. Each contest lasts five hours. Each problem can award up to 100 points, and each problem is divided into subtasks, each of which awards a certain amount of points. A subtask is a grouping of testcases that respect certain constraints that are easier to solve than the full problem. If all the test cases in a subtask are solved, then the points for that subtask are awarded.

The achievements of the contestants are measured by the number of points that they have scored. The contestants are ranked by the number of points that they have scored, and the contestants with the highest scores are awarded medals. The number of medals awarded is determined by the number of contestants that participate in the competition. The top 50% of the contestants are awarded medals, with the top $\frac{1}{12}$ being awarded gold medals, the next $\frac{1}{6}$ being awarded silver medals, and the next $\frac{1}{4}$ being awarded bronze medals.

The IOI competition usually uses CMS [58] as its contest management system. CMS is a contest management system that was developed by the Italian competitive programming community for the IOI that was held in Italy in 2012. CMS enables the contestants to submit their solutions to the problems, and it automatically evaluates the solutions on designated server machines called the *workers*.

The IOI organization is divided into three main bodies: the *International Committee* (IC), which has administrative responsibilities, the *International Scientific Committee* (ISC), which has to select and prepare the problems for the competition, and the *International Technical Committee* (ITC), which has to prepare and oversee the contest management system and the infrastructure for the competition. All three bodies are elected by the General Assembly, which is composed of the representatives of the participating countries.

Aside from the competition, IOI also hosts an annual scientific conference called the *IOI Conference*, where researchers and practitioners in the field of competitive programming meet to discuss the latest developments in the field.

3.1.1 Italian Olympiad in Informatics

In Italy, the national expression of the IOI is the *Italian Olympiad in Informatics* (*Olimpiadi Italiane di Informatica*, OII). The OII is a yearly competition for high school students that started in 2000 in order to select the Italian team for the IOI. It is organized by the Italian Ministry of Education.

The OII is organized as a series of competitions:

- First, there is the *school* competition, which is held in each school. The school competition is a written test that contains mathematical, programming and algorithmic problems. The test is the same for all schools. The best students from each school are selected to participate in the next competition.
- The second level is the *territorial* competition, which is held in key schools of each region. The territorial competition is a programming contest that is held on Terry [59]. The best students nationwide are selected to participate in the next competition.
- The third level is the *national* competition, which is held in a different city every year. The national competition is a programming contest that is held on CMS [58]. The top 20 ~ 30 students are selected to participate in the next stages.
- The final level of selection is the *training camp*, colloquially known as *Volterra*, due to the city where it is usually held. In these training camps, the students are trained by the coaches of the Italian team for the IOI. In each training camp, the students are given a series of programming contests that are held on CMS. The best students from the training camp are selected to go to the next training camp, until the final team is selected. The number of training camps varies from year to year, but it is usually around three or four.

We¹² worked on the OII for many years, and in 2021 we¹³ published a paper [60] that described the technical adjustments that had to be made to the OII during the

¹²The author of this thesis, Dario Ostuni (University of Verona)

¹³Giorgio Audrito (University of Turin), William Di Luigi (AICA), Luigi Laura (Uninettuno University), Edoardo Morassutto (AICA) and Dario Ostuni (University of Verona)

COVID-19 pandemic. In this paper we also describe how the OII is organized and how it works. What follows is a reworked version of the paper.

The COVID-19 pandemic is having a pervasive effect worldwide, including local, national, and international Olympiads in Informatics. Most national Olympiads had to be moved online, posing many serious challenges. Help across countries is of utmost importance in this context, enabling a successful continuation of the IOI during global hard times. We share the experience gained and tools produced during a year of online Olympiads in Italy, hoping other countries can take advantage of these (freely available) tools and suggestions for their own Olympiads.

Introduction

The *Olimpiadi Italiane di Informatica (OII)*, or Italian Olympiads in Informatics [61], are laid out in phases spanning a two-year length: the **Phase-1** of the N -th edition takes place in November-December of the year $N - 1$, the **Phase-2** takes place in April of the year N , and finally the **Phase-3** takes place in September of the year N . The highest-ranked students in this phase are then selected for participation in training camps during the whole scholastic year (**Phase-4**), ultimately leading to the selection of the Italian team at the IOI for the year $N + 1$. The selection process starts with a scholastic pen-and-paper test involving between 10k and 15k participants in Phase 1, which are then reduced to about 1k-2k participants for a regional programming competition in Phase 2. About one hundred students are selected for the national programming contest in Phase 3, and 20-30 are allowed to enter the training camps in Phase 4.

As a result of the COVID-19 pandemic, all four phases of the OII were affected and thus had to be moved to a fully online setting. Phase 1 of the OII 2020 (which happened at the end of November 2019) was the last onsite competition. After that, Phase-4 of the 2019 edition, Phase-2, Phase-3, and Phase-4 of the 2020 edition, and Phase-1 and Phase-2 of the 2021 edition had all to be moved online after being postponed by a few months in the hope that schools would reopen later in the year. This forced shifting required developing new tools and solutions, which ensured an overall successful year of Olympiads in Italy. We share these (freely available) tools and solutions to benefit other countries that may need them for their upcoming Olympiads. The following three sections present our experience with the first three phases of the OII in order. Phase 4 has not been included since it did not require the development of innovative solutions for the limited number of participants.

Phase-1 (scholastic)

In this phase, students have to answer a “quiz-based” exam with multiple-choice questions and (numeric) open questions of a logical or algorithmic nature. This contest is usually held in about 500 high schools, each of them with a designated “contact

person” in charge of distributing the paper-based exam to the students, proctoring them during the exam, collecting the completed exams when the time is up, and finally computing the score of each student.

The large number of participants and the “standardized test” style of the contest made this phase a significant challenge in moving it to an online setting.

Going online

We evaluated several possible solutions to hold this phase online. We initially wanted to restrict as much as possible any possibility of cheating, so we considered using some online platform that would present the questions in random order and with a fixed time to answer each question (e.g., five minutes) while preventing “go back” to discourage students from sharing answers since they would have limited time to answer them.

In the end, implementing this idea would be a significant challenge considering the available time and workforce and the high requirements for the overall load on the system. We also decided that we did not want to disrupt the experience so much for the students by forcing them to answer questions quickly and making them unable to change their answers.

The strategy we finally went with was to stick as much as possible to the “paper-like” feel of the exam by providing them with a simple PDF file so the students could immediately see every question and choose in which order to solve it. However, we randomized the order of the questions in the file and put randomized data in the actual exercises. The idea behind this was to trick cheating students into suggesting each other wrong answers (since the questions would look almost the same but would have slightly different data) or at least spend a significant amount of time decoding which “version” of a question they had.

Finally, to ensure that we did not have downtime caused by a load of tens of thousands of users making requests to our servers simultaneously, we offloaded everything we could to external services, like **Twitter** and **Google Forms**.

Randomization of the questions

In order to prepare questions with fully or partially randomized data, we developed a small program called **randomTeX** [62]. This program uses Jinja2 as the template engine (with a few configuration tweaks necessary to resolve some language ambiguities because LaTeX and Jinja2 have some common syntax, like the opening and closing curly brackets). To feed the data to the template, randomTeX uses an approach similar to the YAML front-matter in Markdown documents.¹⁴

¹⁴For further details, see <https://jekyllrb.com/docs/front-matter>.

We prepared a separate LaTeX file for each of the 20 exercises in the exam, and the data in the YAML front matter of the file itself defined the variations of each exercise. This approach was very flexible since we could easily add new versions of the same exercise by changing only the exercise file slightly and without touching the LaTeX code.

After some tests, we settled on generating four variations of each exercise. Although we could easily have more than four, we deemed this number enough for our needs, and it ultimately made it possible for us to verify that every variation made sense manually. We also avoided multiple-choice questions in favor of numeric open questions to simplify the correction process and minimize the information available to cheating students.

This, combined with a simple randomization in the presentation order of the questions, made it possible to generate *a different PDF for every student*. On the first page of the PDF file, we provided information such as the “exam ID” and the First and Last name of the student to make it clear that the exams were different.

Distribution of the exams

In order to facilitate distribution of the exams (while avoiding a high load on our servers), we decided to start distributing the PDF files well ahead of time: we provided a download link 6 hours in advance. Each PDF file, although with different content, was protected by the same password. To disclose the password to every student at the exact starting time of the contest, we used the “scheduled tweet” functionality on Twitter.

Specifically, each student downloaded their custom PDF file ahead of time. As the start of the exam approached, he or she visited the OII Twitter account and started monitoring the page for updates. As soon as the contest started, the account automatically tweeted the password that unlocked all the different PDFs.

Collection of answers

We offloaded the problem of collecting answers to an external service: Google Forms. We created a purposefully straightforward form: a field for the “exam ID” and 20 questions numbered from 1 to 20 without any question detail (as the order was different across students) and with an open numeric field to specify the answer. The idea was to collect the answers most reliably and then perform the actual validation once the contest ended. One detail worth mentioning is that we generated the “exam ID” to make it easy to reconstruct in case the student mistypes it: we concatenated five random words.

To reduce the possibility of last-minute connection issues that students might have had, we specified in the instructions on the first page of the PDFs that it was rec-

ommended to submit frequently: this was possible because we enabled “Edit after submit” in the form’s settings.

The plan worked well for most of the contest. Unfortunately, several students did not read or did not follow our advice and submitted right at the end of the contest: this probably triggered some malicious activity alert (the form started to ask students to solve a CAPTCHA before submitting), and this caused some students not to be able to submit their answers in time. To address this, we had to reopen the submission window (we duplicated the Google Form and linked it from the previous form) specifically for “late submissions”, and we informed students to use the new link if they were not able to submit their answers in time, while also reminding them that we reserved the right to accept or discard those late submissions.

Evaluating the submissions was straightforward: When we generated the PDF files, we stored the list of correct answers for each file. After the contest, we downloaded the spreadsheet with the answers and, using the “exam ID,” we compared each student’s answers with the correct ones. For each exercise, we graded only the last submission that included that exercise (since there were no penalties for wrong answers).

Possible improvements

In case we will have to go online again next year, there are a few things that we could improve:

- Instead of using a single Google Form for all the participants, it might be better to use multiple Google Form instances (e.g., ten different forms). We can easily modify the PDF template of the exam so that each student will see in his/her exam PDF a *different link* to the submission form. This might reduce the likelihood of triggering the CAPTCHA requests.
- Upon performing a statistical analysis of correct vs wrong answers for each exercise, we noticed that in one of the four variations of one specific exercise, there was a significantly higher rate of correct answers. This was caused by the fact that when we slightly changed the numbers, we introduced an “easier optimal strategy” for the exercise resolution, which was suboptimal in the other variations. Although this did not significantly affect the results, we were very concerned. We are considering introducing some automated validation step in the randomTeX program (e.g., by writing a validation script for each exercise that takes the YAML front-matter as input and outputs a boolean value), which could help us verify that some conditions are valid for all variants of an exercise (e.g., the optimal solution being unique, and so on).

Phase-2 (regional)

In this phase, students have to complete a programming contest, grouped in *territorial* zones based on the geographical position of their school. Italy is divided into 20 regions with very different populations and many schools. Since this contest was onsite,

this heterogeneity used to be an issue: participants from the same region ranged from hundreds to below a dozen (e.g., from 7 to 210 participants in the last onsite contest). In order to mitigate this problem, regions had to be split into *venues* of roughly the same size (from 7 to 51 in the last onsite contest), according to the capacity of the school hosting the contest.

Unlike Phase-1, which mainly evaluates logical and code-reading skills to reach as many students as possible, Phase-2 focuses on selecting students who can write code to solve actual programming tasks theoretically and practically by writing a program that, given an input file, produces the correct output. Time and space complexity are not explicit focuses of this phase, as opposed to IOI-like competitions, so every problem is output-only, and no explicit time limits are given.

In order to evaluate these skills effectively, Phase 2 does *not* use the same judge system used in the IOI, but rather one that we specifically designed for this back in 2017, called **Terry** [59]. After accessing the Terry interface, the participants can choose a task and download an input file (which is unique for each student and changes every time a submission is attempted), run their program locally with the downloaded file as input, and finally upload the produced output file along with the source code of the program producing it (for plagiarism avoidance) and immediately receive a score. This approach was heavily inspired by the early format of the Google Code Jam competition and allows us to grade submissions of a considerable number of students with relatively few resources (as we do not need to run the students' code, except in selected cases) and allows the students to use a broad set of supported languages.

After the contest ends, we perform plagiarism checks. We used to do them using JPlag [63], which only supports a subset of the allowed languages. For this reason, we recently developed Starplag [64], which computes the similarity between two source files regardless of their programming language by implementing a variation of the Levenshtein distance where text substitutions (i.e., variable renaming) are considered.

Going online

This was the first phase to be moved online since it was scheduled to take place in April 2020, right after the March COVID-19 restrictions took place in Italy. Terry was not meant to work as an online judge but was designed to be deployed “offline” in each onsite venue. Since every venue had a designated reference person, we could easily communicate with the students through this person. If a student had a question on the tasks, the reference person (e.g., a teacher in the venue's school) would forward it to us, and we would then provide an answer, which would be finally relayed to the student.

Since this process does not work for an online contest, we had to write a new component for Terry to handle questions, answers, and announcements. This new component worked well enough, although leaving margins for future improvements.

In particular, it showed all the questions in a “stream” as they arrived at us, making it difficult to reconstruct the context of a question. We underestimated the importance of a “conversation” with a student: a question from a user would often implicitly reference a previous question that the user sent us. Sometimes, a question could not simply be replied to with an answer, and we had to “answer” the question with another. For the next year, we plan to develop a conversation-like interface addressing this issue.

Proctoring more than a thousand students from home with available resources in real-time was unfeasible, so we adopted the standard approach of online contests, where only after-contest checks are performed. After the contest was over and before extracting the ranking, we examined the submitted source files, looking for evidence of plagiarism or irregularities.

Technical aspects

The contest server was designed to run inside a virtual machine sent to the venue hosts to require minimal technical knowledge from them. The virtual machine only had to serve that venue (around 50 students) and be as light as possible since we had no control over the quality of those servers. For this reason, Terry uses SQLite as DBMS: it is light and fast enough for our needs.

Scaling a system designed to handle a few dozen users to support a few thousand users can be risky. To limit this risk, we decided to keep each Italian region in a separate shard of the system, with its independent instance of Terry. This way, each instance only had to process a fixed fraction of the users, also allowing us to migrate each region independently in case of failure of some instance.

We used Docker containers for shipping the replicas and docker-compose for orchestrating them. Ideally, each instance should have used its host for complete separation of the load; however, we ended up renting 8 VMs on Google Cloud Platform,¹⁵ assigning each of the 20 regions to a specific VM in a way designed to distribute the load uniformly.

Besides the 8 VMs, we also set up a *coordinator* VM to host the communication component and some utility scripts. Among these extra tools, we also had an instance of Grafana¹⁶ displaying many metrics collected by Prometheus:¹⁷ at [65] and [66] you can find a snapshot of our dashboards.

You can see the machines we had (named with Greek letters) from those dashboards, where *phi* is the coordinator. The communication platform handled an average of 60 requests per second, while the other VMs had a utilization close to zero for most of the contest duration. Unfortunately, due to a misconfiguration of nginx (a

¹⁵<https://cloud.google.com>

¹⁶<https://grafana.com>

¹⁷<https://prometheus.io>

wrong request rate limit), the expected spike at the beginning of the contest caused nginx to terminate connections, not allowing them to reach Terry's backend (see the "503" tab in the dashboard [65]). This was fixed in about half an hour; in the future, it will be necessary to stress test the system with reasonable loads, not only with *denial of service* rate of requests.

Since the users were partitioned into different servers, we had to ensure each user connected and logged in to the correct container. This was accomplished through a DNS pointing each user to the correct VM and then to the correct instance of Terry within that VM. More precisely, we had separate URLs for each region (e.g., `lom.territoriali.olinfo.it` for Lombardy), adding a CNAME record to this domain pointing to one of the VMs, for example, `alpha.territoriali.olinfo.it`, that in turn is an A record pointing to the VM's public IP.

This extra step allowed us to quickly move one container into a different VM by simply spawning it and changing the content of the CNAME record (a low TTL was set to support this). Fortunately, this measure was unnecessary as the system ran smoothly after the rough start.

Phase-3 (national)

Since 2011, the Italian national phase has used the Contest Management System (CMS) [58] as its platform to host the contest; CMS is also the core of the training platform available for students [67] and has also been used in team olympiads [68]. The contest is IOI-like: 5 hours and (usually) 3 problems to solve. The only significant differences lie in the difficulty level of the problems, which are usually easier than IOI ones [69], and the set of allowed programming languages: in the Italian national phase, only C and C++ are allowed.

Before the pandemic, this phase was a 3-days onsite event hosted by a different high school or educational institution each year. About 100 students admitted to this phase will travel to the contest site with their regional contact person.

After the contest, the top 20-*ish* students would continue their journey to the training camps held in the Italian city of *Volterra*, in *Tuscany*, during which the Italian team for the IOI would be selected.

Going online

As the contest moved entirely online, we had to deal with several problems, the major one being proctoring. While during the previous phases, the problem of potential cheating was relatively irrelevant and could be dealt with offline tools, with only about 100 contestants, the potential problem of cheating had to be dealt with appropriately. Thus, much of the organization of the online contest revolved around this point.

The competition itself was still hosted with CMS, as its use onsite or online is similar. We let the students use their machines to compete in the contest and set up a 3-layered proctoring system:

1. All contestants were assigned to one of 14 different Zoom¹⁸ meetings where they would have to be in for the whole duration of the contest, with the camera on and the microphone unmuted. A human proctor from the staff was assigned to each of them, having to look over approximately eight contestants while also helping as a communication facilitator.
2. All contestants were required to compete from inside a virtual machine provided by the organization, having it full-screen for the whole duration of the contest. Aside from blocking internet access, the virtual machine also recorded the contestant's screen and sent the resulting stream to the server.
3. All contestants were required to run a custom-made proctoring program on the host machine, called *oii-proctor* [70], which would check for misbehaviors (such as opening a browser or leaving the virtual machine) and report them.

Furthermore, to mitigate possible unexpected cheating behaviors, the difficulty and novelty of the problems were increased relative to previous years, and the number of people selected for the training camps was also increased to 29. This was also made possible by the simpler logistics and lower budget required by an online training camp, which used to be limiting factors for this number.

The handling of the contest was overall successful, with only some secondary issues:

- contestants with bad internet connections or very slow PCs had, inevitably, a disadvantage;
- the increased difficulty of the problems almost halved the average score: in 2018 it was 79, in 2019 it was 82.34 and in 2020 only 49.7; this also caused the cut for bronze medals to still be 0 points as late as at 2 hours and 30 minutes into the contest [71];
- there was a greater incidence of contestants *forfeiting* the contest: eight contestants forfeited the contest this year, while in onsite contests, this number is usually zero or one.

To our knowledge, there were no cheating attempts during the contest.

Technical aspects

The server fleet was composed of a central server and a variable number of worker machines, which were used to host CMS's *cmsWorker* service. All machines were VMs hosted on Google Cloud. The central server hosted CMS, the service for collecting the screen streams from the contestants' VMs, and the service for collecting reports

¹⁸<https://zoom.us>

from *oii-proctor*. In total, we spent around 100 euros for the whole cloud infrastructure. We also set up Grafana and Prometheus to monitor the system: you can find the dashboards at [72] and [73].

The operating system chosen for the contestants' VMs was Ubuntu MATE due to its commonality and relatively low resource requirements. When started, it would prompt a login screen, which sent the inserted credential to the central server, which would send back a Wireguard [74] configuration file: Wireguard was used to set a secure connection between the VM and the server, and the login served to keep track of which user was on which machine. Then, during the contest, a service on the VM would take a screenshot every 30 seconds and send it to the central server. This allowed us to monitor the contestants further. Also, on all VMs, an SSH daemon was installed to allow us to enter the machine via the Wireguard tunnel.

oii-proctor is a program written in Rust [75] and distributed as a static binary for all three major operating systems that served as an added measure to ensure some proctoring for the host machine. After asking for a login to identify the user, *oii-proctor* collects information about the running processes to check if any browser is open and if the VirtualBox process is still alive and sends this information to the central server every 15 seconds. While doing these background jobs, it also puts up a small amount of a *security theater* [76]: to keep contestants under the impression that *oii-proctor* is constantly active doing some work, thus making a would-be-cheater contestant more aware of being monitored, *oii-proctor* constantly prints meaningless information at irregular intervals of time, masking when the actual checks are done.

Conclusions

We described how we moved all the phases of the Italian Olympiads in Informatics online. We hope that our experience, as well as the tools developed and the other ones mentioned, can be helpful to other national or local programming contest organizers.

3.1.2 CMSocial

As a companion tool for the Italian Olympiad in Informatics, an online platform called *CMSocial* was developed. *CMSocial* is the training portal for the Italian students that participate in the OII. It has a large library of problems, both from national and international competitions. *CMSocial* is a fork of CMS [58], the contest management system that is used for the IOI.

Aside from providing training for the students, *CMSocial* is also used to hold training courses to Italian high school teachers. In 2018 we¹⁹ published a paper [69] that investigated the effectiveness of the platform through its analytics. It also provides a

¹⁹William Di Luigi (AICA), Paolo Fantozzi (CTL), Luigi Laura (University of Rome “La Sapienza”), Gemma Martini (AICA), Edoardo Morassutto (AICA), Dario Ostuni (University of Verona), Giorgio Piccardo (CTL), and Luca Versari (University of Pisa)

comparison between the students crowd and the teachers crowd. What follows is a reworked version of the paper.

We discuss using Analytics in *CMSocial*, an online programming contest training system. We first provide an overview of the challenges in training for programming contests. Then we discuss the data collected in these years using *CMSocial*, a platform devoted to the training of students for the Italian Olympiads in Informatics (Olimpiadi Italiane di Informatica - OII), and analyze them by comparing two distinct groups of users in two distinct platforms built on *CMSocial*, one devoted to students and one to their teachers. Most notably, the two groups are more similar than expected when dealing with programming contest training.

Introduction

Programming contests, competitions in which participants face a set of tasks that require writing computer programs, usually using efficient algorithms, are significant and influential in the process of learning computer programming and, more generally, computer science [77, 78, 79, 80, 81, 82].

We use the data provided by two platforms based on the same training system, *CM-Social* [67], to compare the two distinct groups that use the platform: on one side secondary school students training for the Italian Olympiads in Informatics (Olimpiadi Italiane di Informatica - OII), and on the other side their teachers, i.e. secondary school teachers, that enroll in a self-paced online course of computer programming aimed at helping them to prepare students for OII; the teachers have a small programming contest at the end of each course, and if they solve at least three out of seven tasks they are granted a certification for the course. Most notably, the two platforms are almost identical, but there is no link between them, so they are the perfect experimental testbed for comparing the two groups.

We collected data for more than four years, involving approximately one thousand teachers and more than three thousand students, faced with a set of 153 and 401 tasks available on the platforms.

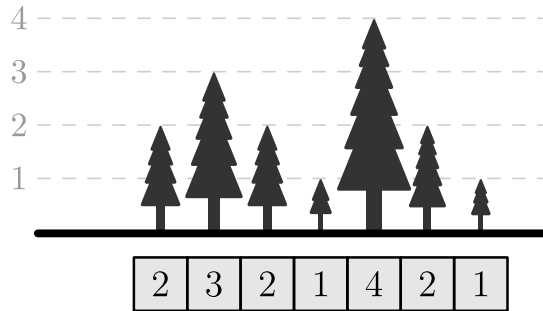
Programming Contests, skills, and platforms

Programming Contests

A *programming contest* is a competition in which the contestants face a set of programming tasks, also called problems, to be solved in a limited amount of time and under some time and space constraints. For example, we report the task *Taglialegna* statement, which was used in the final 2014 edition of OII.

Abbatti S.p.A. (which is the Italian brand of *tearDown INC*) is a big enterprise that works in the field of tree felling. In particular, it is been a few years since it started

improving in tearing down *barky trees*, a peculiar kind of trees which is very tall and thick. This particular species grow in a very tidy way: the woods made of these trees are actually a long horizontal line of trunks, placed at one decameter (32, 8 feet) one another. Each one of the trees has a particular height, which is expressed by a positive number (decameters).

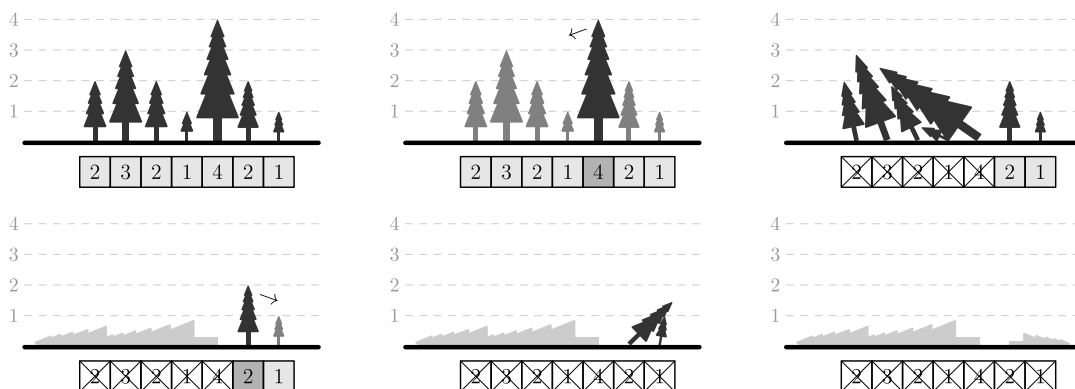


Tearing down one of these trees is a very hard thing to do and, although tearDown INC employees the most advanced technologies on the market, it is a very time consuming activity, since barky trees' bark is incredibly thick. The workers have the opportunity to choose in which direction (left or right) the tree should fall after the cut.

Each time a barky tree falls down it hits the trees that haven't been torn down which are placed on its falling trajectory; in other words, it tears down each tree which is closer than its height in the direction of the fall. Since the number of barky trees in this woods is huge, this dynamic of the fall creates a domino effect.

In order to be the best enterprise in barky tree felling tearDown INC developed a system which is able to scan the whole wood, choosing which trees should be cut by workers and in which directions they should fall with the aim of cutting all the trees. It is important to recall that it is in the best interest of the enterprise to minimize the number of trees that need to be torn down by workers directly. Your role in this situation is to implement the system for tearDown INC.

Below, we can see an example of a solution to the instance shown in the picture above: It is enough to cut two trees:



A single task can be broken into subtasks of increasing complexity: basic techniques are enough to solve some subtasks within the given time and space limits, while the most difficult ones require particular algorithmic techniques and data structures.

Among the programming contests, we mention:

- The International Olympiads in Informatics (IOI) are an annual programming competition for secondary school students patronized by UNESCO. <http://www.ioinformatics.org/>
- The ACM International Collegiate Programming Contest (ICPC) is a multitier, team-based programming competition operating under the auspices of ACM. <https://icpc.baylor.edu/>
- The recent International Informatics Olympiads in Team (IIOT), which started in 2017, is a team competition like ACM ICPC, different from IOI (individual competition). Currently, only four nations are involved: Italy, Romania, Russia, and Sweden. <https://iio.team/>
- Google Code Jam is based on multiple online rounds concluding in the World Finals. <https://code.google.com/codejam/>.
- The Facebook Hacker Cup is *an annual worldwide programming competition where hackers compete against each other for fame, fortune, glory, and a shot at the coveted Hacker Cup*. <https://www.facebook.com/hackercup/>

Students participating in programming contests generally have several programming contest platforms at their disposal, including Codeforces, USACO, COCI, TopCoder, Codechef, and HackerEarth, that run contests with different periodicities.

The programming languages allowed in the competitions vary considerably: for example, IOI and IIOT accept only C, C++, and Pascal; ICPC also adds Python (both 2 and 3), Java, and Kotlin to the list.

Algorithmic skills and techniques

The programming contests mentioned in the previous section focus on a large set of algorithmic skills that includes

- Basic Data Structures
- Complete Search, Divide and conquer, Greedy techniques
- Dynamic Programming (basic ideas)
- Dynamic Programming (advanced topics)
- Basic Graph algorithms (e.g., connected components, distance between two nodes)
- Advanced Graph algorithms (e.g., network flows, matching)
- Mathematical problems
- Computational Geometry
- String Processing

- Advanced topics

Some programming contests, such as IOI, have a syllabus²⁰, while others refer to general algorithmic skills. There are some books, such as the ones of Skiena [83], Halim [84], and the recent one of Laaksonen [85], that provide essential material about algorithms, data structures, and heuristics needed in programming contests.

Online training platforms

Besides the already mentioned platforms that provide online programming contests and usually have the archive of the previous years' tasks, there are several online training platforms; we mention the well-known UVa Online Judge²¹, Sphere Online Judge²² (SPOJ) and Open Kattis²³.

All these platforms are, roughly speaking, vast repositories of tasks, and it is possible to submit solutions and be part of a global ranking based on the overall number of tasks solved.

The online training system: *CMSocial*

In this section, we provide basic information about our online training platform, *CM-Social*, which is based on the *Contest Management System* (CMS) [58, 86], the grading system used in several programming competitions, including IOI. CMS was designed and coded almost exclusively by three developers involved in the Italian Olympiads in Informatics: Italy hosted IOI 2012, and therefore, since 2010, it started the development of CMS, which was used/tested in the OII finals 2011 and a few months later, was the grading system of IOI 2012. CMS version 1.0 was released in March 2013, and since then, it has been used in both IOI 2013 and 2014, together with several other programming competitions in the world [86].

Details about *CMSocial* can be found in [67], and the source code is freely available²⁴ we implemented three distinct platforms using *CMSocial*:

1. **OII-training** is the platform devoted to students interested in OII. We can see a screenshot of the home page in Figure 16. This platform has approximately 180 problems spanning several techniques and difficulties, ranging from regional contests to the IOI level. Furthermore, the tests from the first selection of OII (schools selection) are available as interactive online forms. So far, we have yet to advertise this platform in the schools since we are considering it in the beta testing phase.

²⁰The IOI syllabus is available at the URL <http://people.ksp.sk/~misof/ioi-syllabus/>

²¹<https://uva.onlinejudge.org/>

²²<http://www.spoj.com/>

²³<https://open.kattis.com/>

²⁴<https://github.com/algorithm-ninja/cmsocial>

2. **DIGIT** is the platform dedicated to teachers: We realized this platform in a project sponsored by the MIUR, aiming to build a self-paced online computer programming course focused on the olympiads in informatics. The idea was to train the teachers so they could be able to train their students. Thus, this platform is currently the richest of the three regarding contents and functionalities. We can see a screenshot of the home page in Figure 17. There are video courses in C/C++ and Pascal programming, Algorithms and Data structures, and some basic video tutorials, including how to use the platform to submit a solution. There are also some lecture notes, and all the material can be distributed to students; the video lectures are also available on the OII channel on YouTube.
3. **IOI-candidates** is the last platform, and the only one not publicly available, since it is devoted to the winners of the Italian national contest preparing for IOI. This platform has been the original motivation to develop the whole *CMSocial* system. This platform has all the problems available to the other two platforms and a *reserved* set of problems that we use in the contests to rank the students. The students are asked to refrain from discussing this problem in public forums or social networks since we usually reuse them after a few years.

In the following section, we report our findings from the comparison of the data from the first two platforms, **OII-training** and **DIGIT**; the third one, **IOI-candidates**, has a minimal number of participants with a high focus on the IOI so it is outside the scope of this work.

Experimental findings

Our datasets

As mentioned before, our datasets collect all the information available in two platforms based on the same training system, *CMSocial*: the data collected from the first platform - **OII-training**, devoted to students preparing for the Italian Olympiads in Informatics, and the data collected from the second platform - **DIGIT**, devoted to teachers participating in a self-paced online course.

The data of both platforms span slightly more than four years, from December 2013 to April 2018. It is summarized in Table 2: as we can see, as expected, **OII-training** has more participants and more problems since it belongs to an active community, while **DIGIT** has more or less a standard set of problems that increases slightly in each edition with the addition of the problems of the previous edition final contest. In Table 3, we can see the details of the editions of the teachers' course built on the **DIGIT** platform. In particular, besides the start and end dates, we can see the number of enrolled participants, i.e., the ones who applied for the course, and the (much smaller) number of certificates granted to participants who completed the small programming contest at the end of the course.

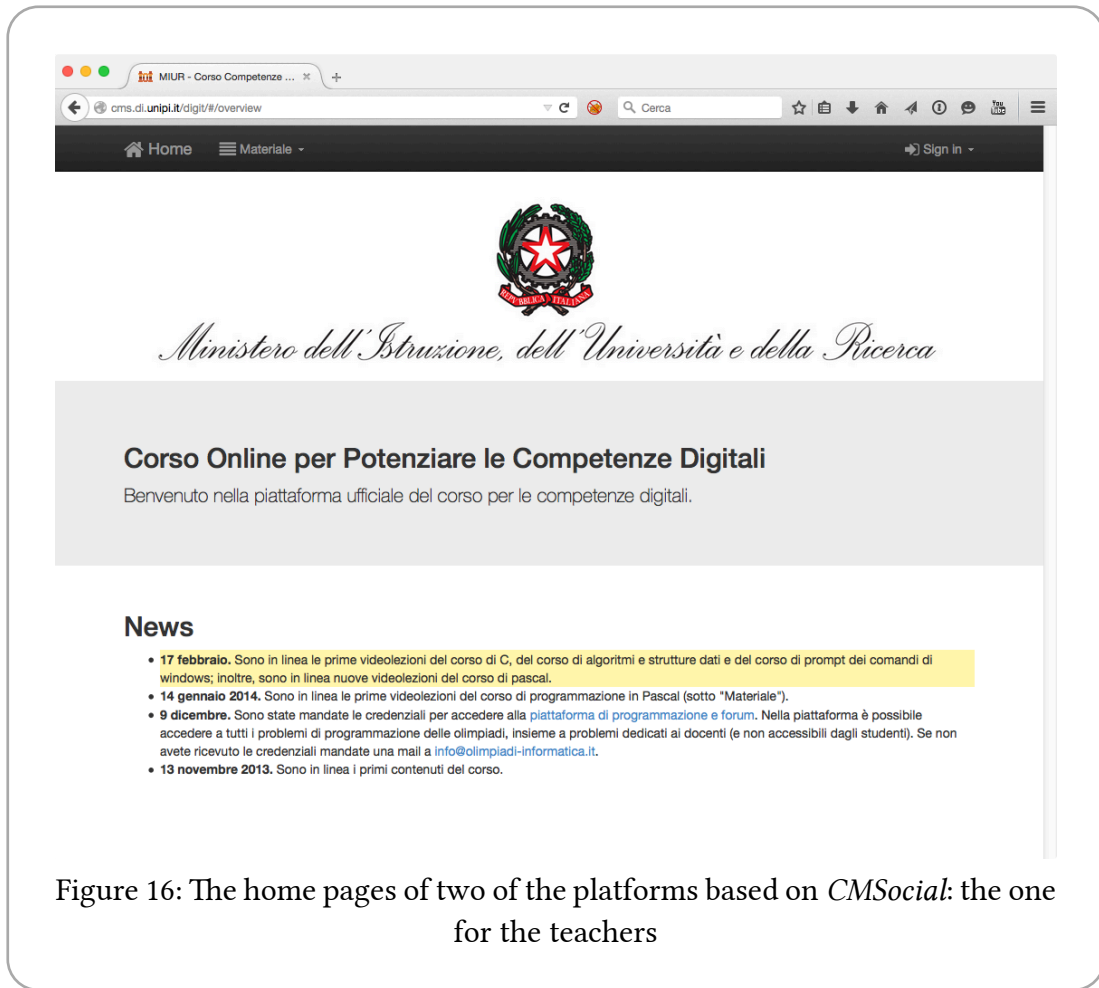


Figure 16: The home pages of two of the platforms based on *CMSocial*: the one for the teachers

Data Analysis

Let us begin by plotting the distribution of the sum of the scores in Figure 18: the two plots have a similar shape, as we can see in Figure 19 where we plot the density distribution to reduce the size difference between students and teachers; as we can see, the students' tail is much longer, and this is due to the availability of a more extensive set of problems, as seen in Table 2.

In Figure 20 (students) and Figure 21 (teachers) we can see, for each problem available in the platform, the number of participants that solved it, in decreasing order. As we can expect, both the plots have a similar shape, with a bias in the teachers' plot due to the problems in each final contest: for example, the fact Table 3) that in the sixth edition, 232 teachers completed the contest means that they solved at least three problems from the proposed set of seven.

So far, the only difference between students and teachers is due to the different contests of the two platforms. Let us dig more and try to understand the approach of single users when faced with the difficulty of the problems. Since there is no difficulty grade for each problem, we will estimate it with the percentage of people that solved

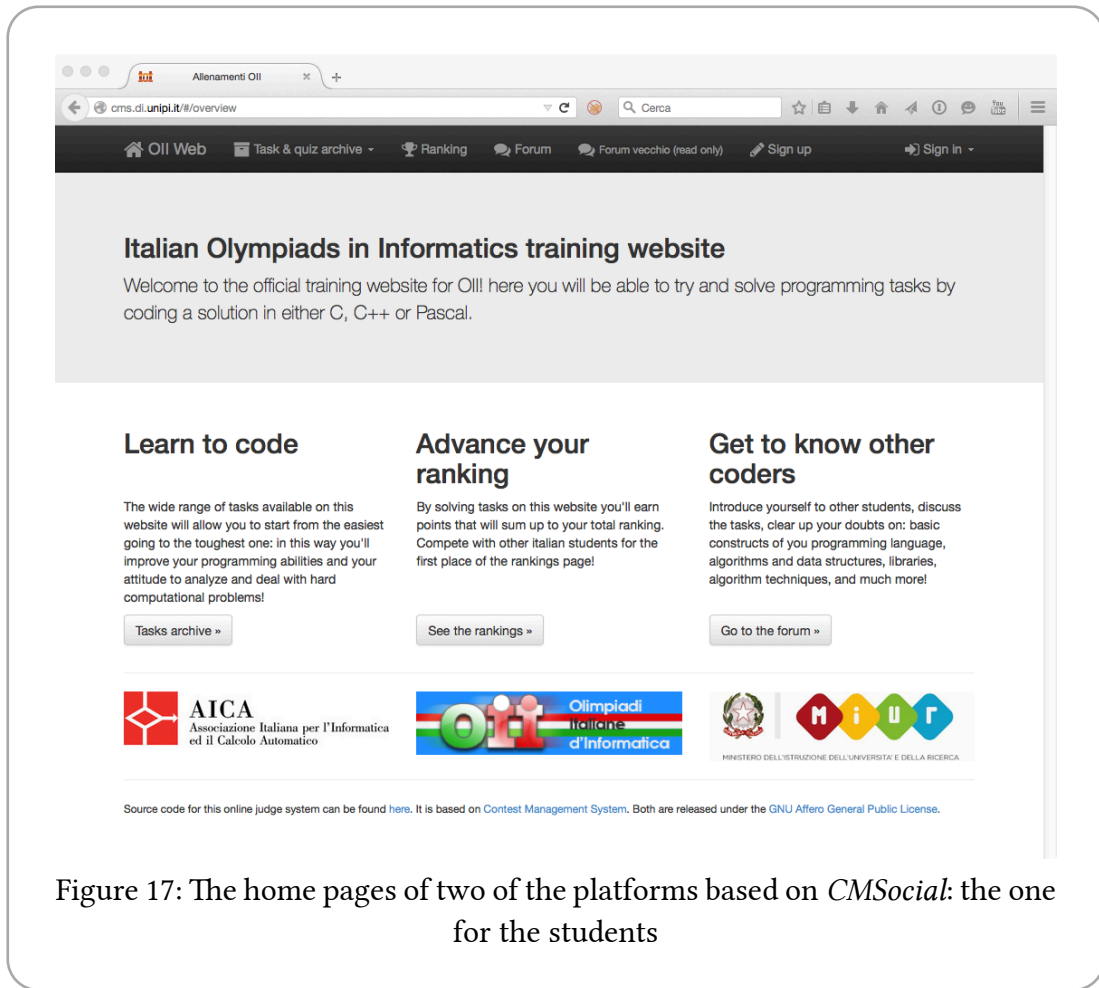


Figure 17: The home pages of two of the platforms based on *CMSocial*: the one for the students

it for our study; thus, a problem solved by many will be considered easier than one solved by a few. In particular, the difficulty D of a problem will be

$$D = 1 - \text{percentage of people that solved it}$$

In this way, a problem solved by 100% will have difficulty $D = 0$, and a problem solved by no one will have difficulty $D = 1$. In Figure 22, we can see the correlation between the sum of the weighted rank of the problems and the number of problems solved. The correlation is very high in both cases (students and teachers), and the two

Platform	# Participants	# Problems
OII-training (Students)	3543	401
DIGIT (Teachers)	1006	153

Table 2: Number of participants and available problems in the platforms.

Edition	Start	End	# Enrolled	# Certificates
1	12/2013	03/2014	315	54
2	03/2014	10/2014	506	93
3	11/2014	03/2015	691	87
4	10/2015	02/2016	211	139
5	04/2016	09/2016	1032	74
6	06/2017	10/2017	1387	232
7	12/2017	05/2018	1345	NA

Table 3: Number of enrolled participants and available problems in the platforms.

cases are very similar, as we can see in Figure 23, where we zoom in on the left part of Figure 22.

Summing up, the behavior of both groups on the two platforms is very similar. We observe in all the plots a tendency for teachers towards easier problems, which is very likely due to the easy problems in the final contest of each course edition.

Findings

We addressed using Analytics in *CMSocial*, an online programming contest training system. In particular, we analyze the behavior of two distinct groups of users in two distinct platforms built on *CMSocial*, one devoted to students and one to their teachers. As observed, the two groups are more similar than one would expect when dealing with programming contest training.

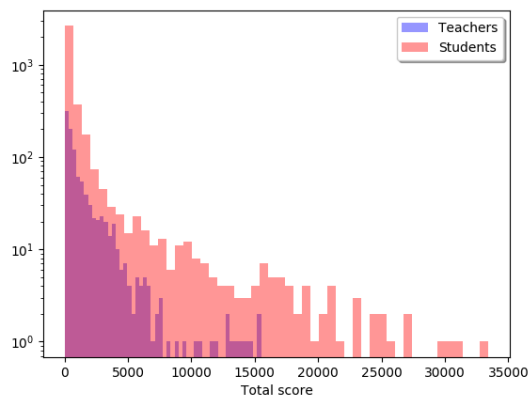
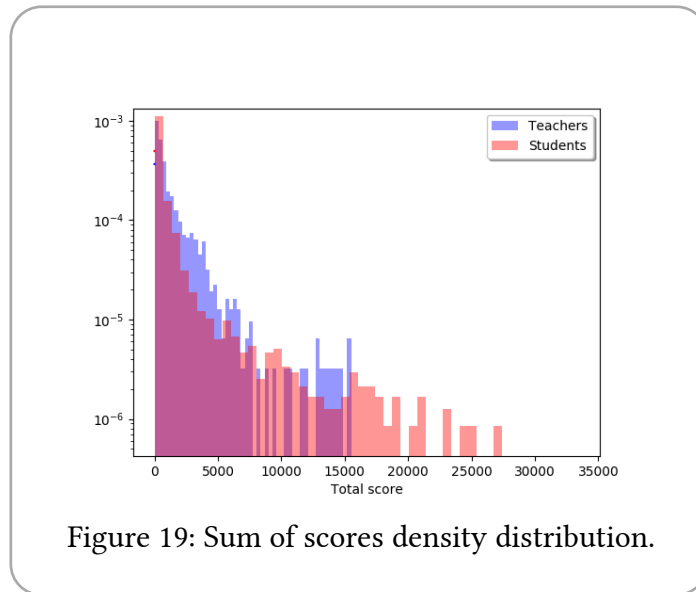


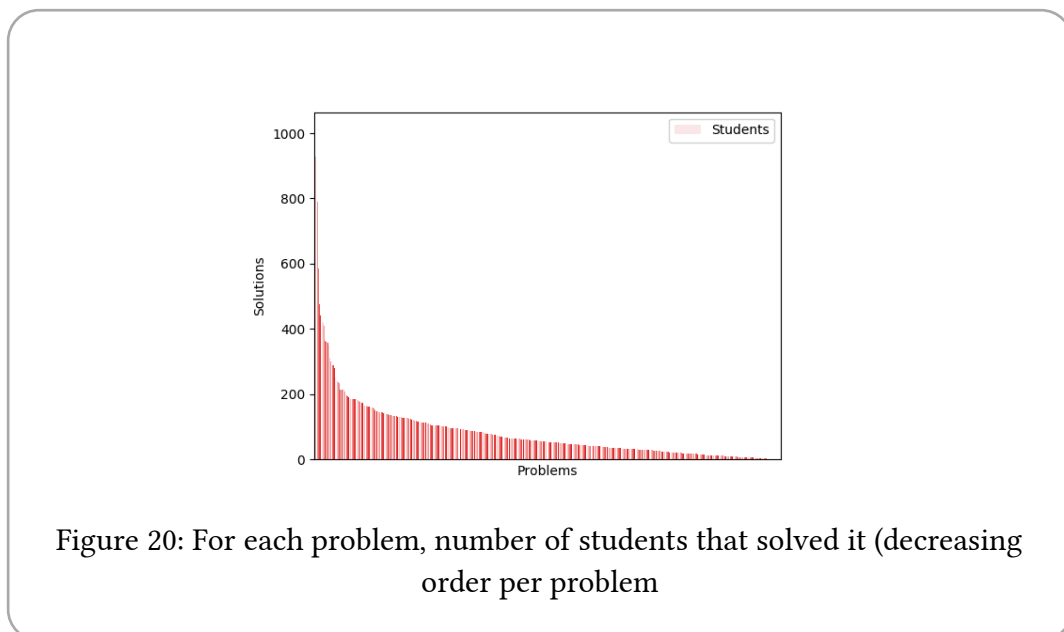
Figure 18: Distribution (histogram) of the sum of the scores.



We are currently working on a more refined estimate of the difficulty and on improving the platform with a framework that can allow recommendations, under the form of suggestions, to the learner about the following programming problem to undertake, and that can foster motivation in students through a lightweight, badge-based, gamified approach [87].

3.2 International Collegiate Programming Contest

The International Collegiate Programming Contest (ICPC) [1] is the most prestigious competition for competitive programmers at the university level. It is an annual team



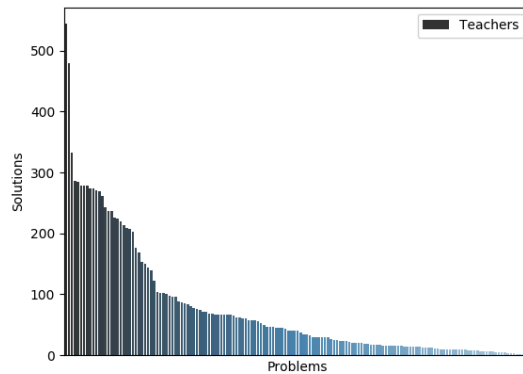


Figure 21: For each problem, number of teachers that solved it (decreasing order per problem)

competition for university students that was founded in 1970. As such, it is the oldest competition for competitive programmers that is still running today.

The standard format of an ICPC contest is a five hours contest where the contestants compete in teams of three. Each team must be composed by students from the same university. The contestants are given a set of problems to solve, and they must submit solutions to the problems. The scoring on a problem is binary: either the solution is correct, or it is not. The contestants are ranked by the number of problems that they have solved, and the time that it took them to solve them.

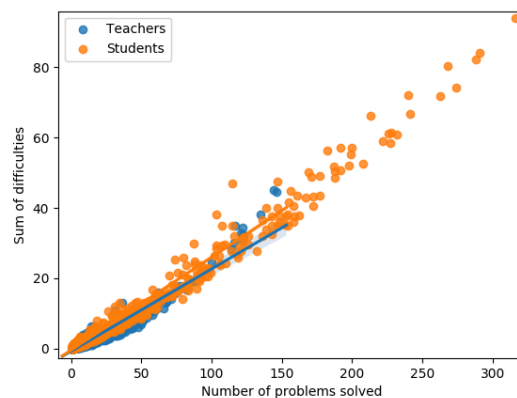


Figure 22: Correlation between the number of problems solved and the sum of the difficulties of the problems.

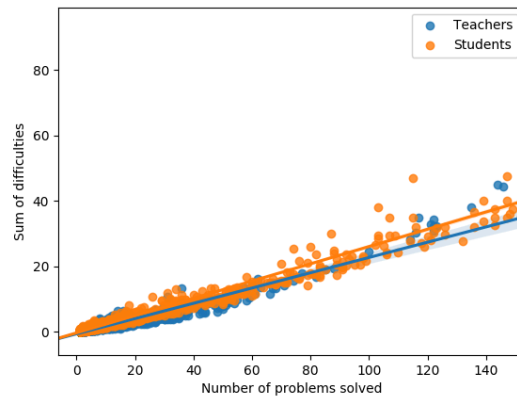


Figure 23: Correlation between the number of problems solved and the sum of the difficulties of the problems. (detail view from Figure 22).

The team that has solved the most problems is ranked first, and the team that has solved the least problems is ranked last. If two teams have solved the same number of problems, then the team that has solved them in the least amount of time is ranked first, and the team that has solved them in the most amount of time is ranked last. The time that it took a team to solve a problem is the time that has passed since the start of the contest until the submission of the first correct solution to the problem, plus a penalty of 20 minutes for each incorrect submission to the problem. The team that has the least amount of time is ranked first, and the team that has the most amount of time is ranked last.

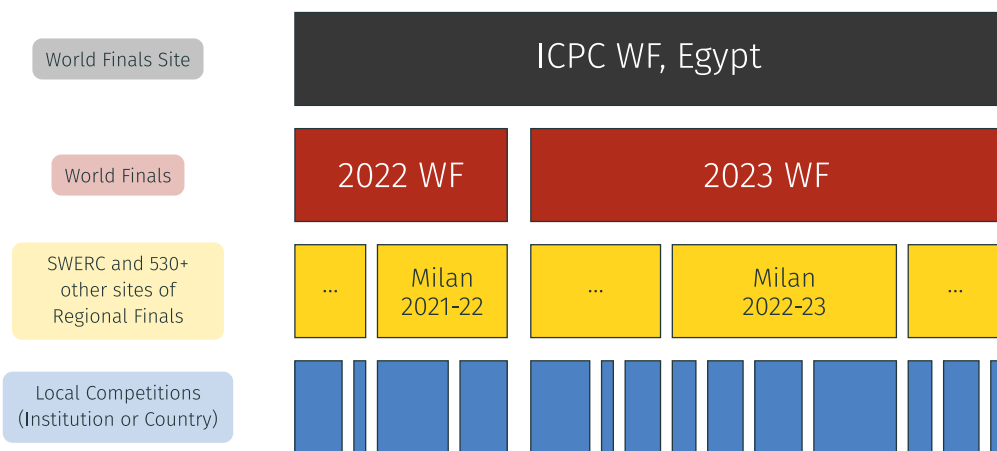
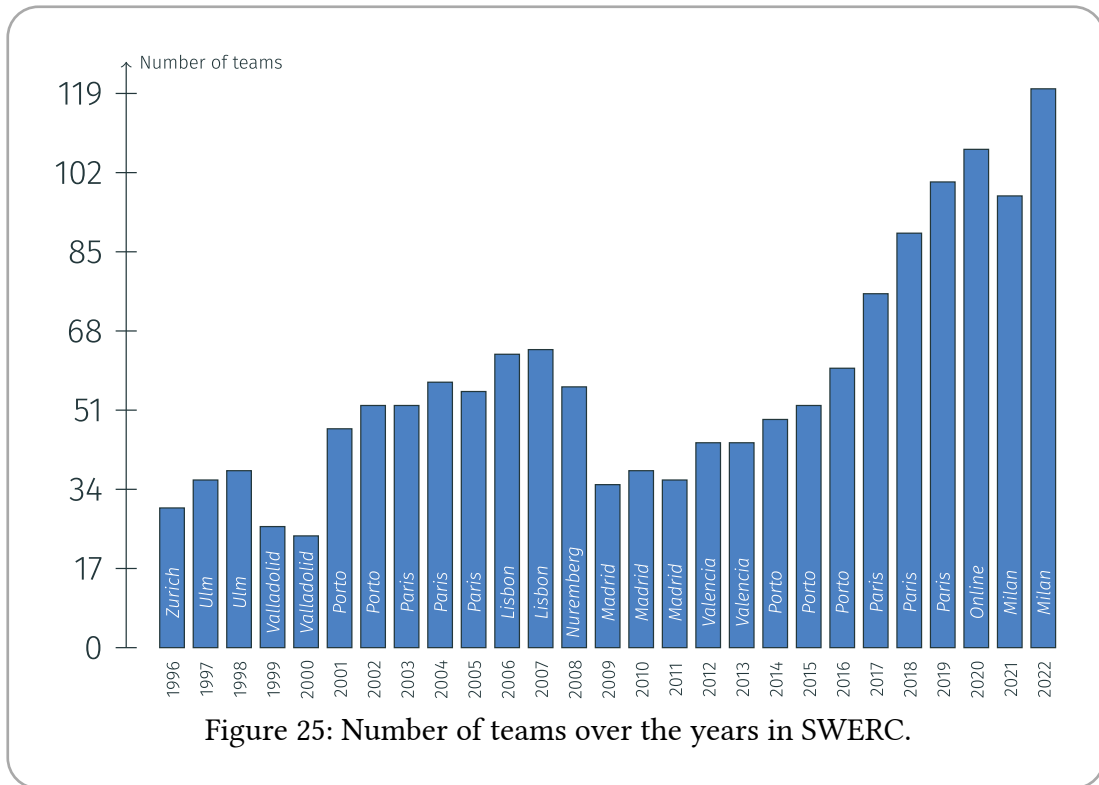


Figure 24: An example schedule for the ICPC.

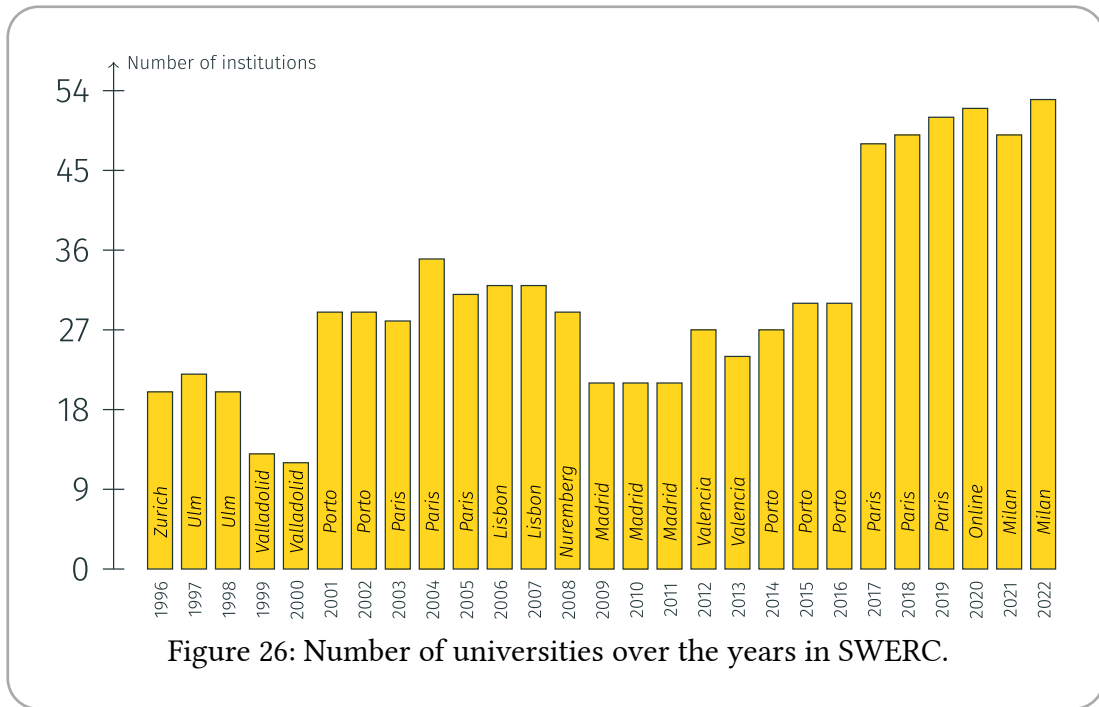


There are two contest management systems that are used for the ICPC: PC² [88] and DOMjudge [89]. The competition is organized in stages:

- First, there are the *university* selections, which are held in each university. The university selects independently their best teams to participate in the next stage. The format of the selection contest is up to the university.
- The second stage is the *regional* contest, which groups together universities from a certain set of countries. There is only a single contest during the event. The best teams from each regional contest are selected to participate in the next stage.
- The third stage is the *superregional*, or *continental*, contest. There is such a contest for each continent of the world. It is also a single day contest. The best teams from each superregional contest are selected to participate in the next stage.
- The final stage are the *World Finals*, which is the final stage of the competition. It is a single day contest that is held in a different university every year. The contest awards four gold medals, four silver medals and four bronze medals. The top team is awarded the title of World Champion.

Some regions still have to implement the full stage pipeline and skip the superregional stage. Such an example for the European region is illustrated in Figure 24.

The ICPC is organized by the ICPC Foundation, which is a non-profit organization that is based in the Baylor University in Waco, Texas. The ICPC Foundation is responsible for the organization of the World Finals, and it supervises the organization of the other stages.



3.2.1 SouthWestern Europe Regional Contest

The SouthWestern Europe Regional Contest (SWERC) [90] is the regional contest for the SouthWestern Europe region. It is a regional contest that groups together universities from Spain, Portugal, France, Italy, Switzerland and Israel. Like the other regional contests, it is a single day contest that is held in a different university every year²⁵. The contest awards two gold medals, four silver medals and eight bronze medals.

The organization of the contest is handled by a local committee that is lead by the *Regional Contest Director* (RCD). The RCD is responsible for the organization of the contest, for the raising of funds, and for the selection of the problemset. Usually the RCD is a professor from the university that is hosting the contest. The RCD usually delegates the scientific duties to a Chief Judge, who is responsible for the selection of the problemset, and to a Technical Director, who is responsible for the technical aspects of the contest.

We²⁶ organized the 2021-2022 and 2022-2023 editions of SWERC. The contest was hosted at the *Politecnico di Milano* in Milan, Italy. The second of these two editions was the biggest SWERC ever done, with 120 teams from 53 universities. The number of teams and universities over the years is shown in Figure 25 and Figure 26. The number of teams per university over the years is shown in Figure 27.

²⁵A university can host the contest multiple times.

²⁶The author of this thesis, Dario Ostuni (University of Verona), was RCD for SWERC from 2021 to 2023.

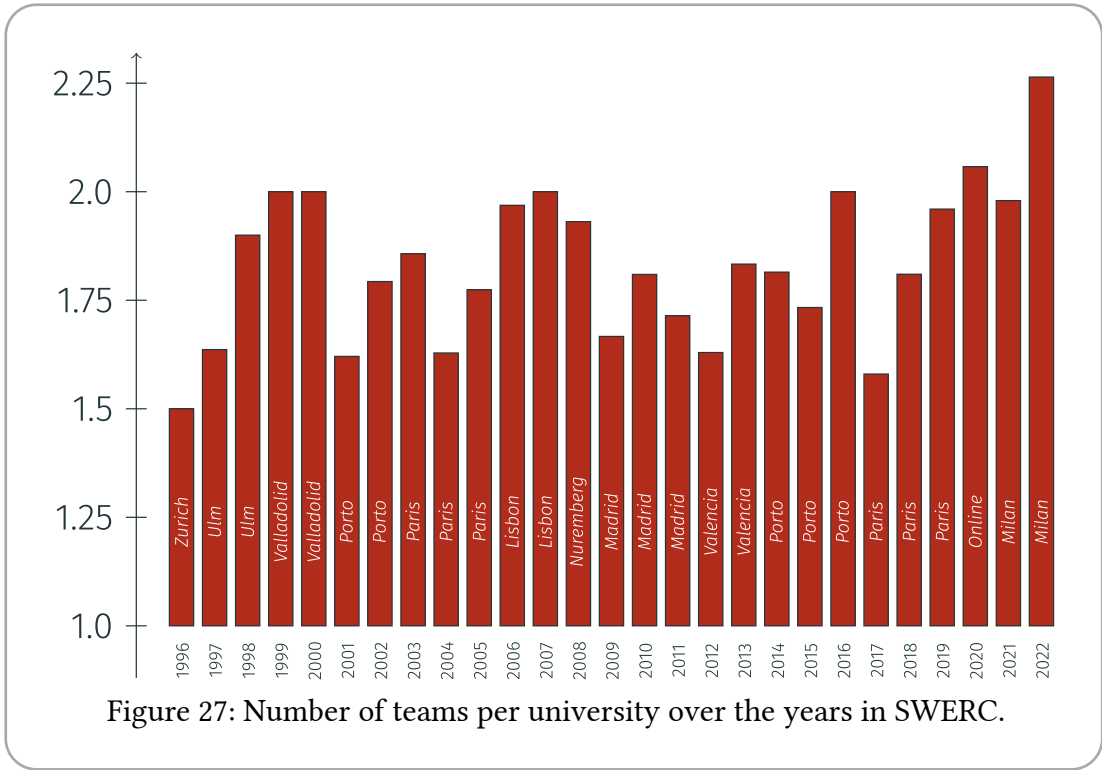


Figure 27: Number of teams per university over the years in SWERC.

To organize the contests we created a team of volunteers that helped us with the organization. The organization of the contest was split into five teams:

- The *organization* team: this team was responsible for all the logistics, financing, accounting and communications for the contest.
- The *technical* team: this team was responsible for the technical aspects of the contest, such as the contest management system, the network, the servers and the contestants' machines.
- The *judges*: this team was responsible for the creation and preparation of the problemset, and for overseeing the contest.
- The *media and artistic* team: this team was responsible for the creation of the media content for the contest, such as the graphics, the photos and the live singing performances during the opening ceremony.
- The *volunteers*: this team was responsible was composed of local volunteers that helped us with various tasks during the contest, such as the registration of the teams, the distribution of the material, and proctoring during the contest.

In these two editions we tried to innovate the contest by introducing new ideas and new organization concepts. For both years the judge team²⁷ was composed by people coming from all the countries of the SWERC region. This was done in order to have a problemset that is more representative of all the countries of the region. We

²⁷The judge team is the team of people that is responsible for the creation and preparation of the problemset.

worked to improve the opening ceremony, by removing as much as possible the long speeches and by introducing songs performed live during the ceremony. This was done in order to retain as much as possible the attention of the contestants during the ceremony.

4

Turing Arena light

In this chapter we introduce *Turing Arena light*, the spiritual successor of *Turing Arena* [91]. *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 simple²⁸ 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 [9].
- *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 achieve this, *Turing Arena light* only consists of a small core written in Rust [75], 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.

²⁸Simple might mean very different things, in this context it is conceptual simplicity.

4.1 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 [92], CMS [58] and Codeforces [55]), or where the interaction is not done in real-time (like in the old Google Code Jam [53] and Meta Hacker Cup [54]).

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.

4.1.1 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 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 [93] 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 Listing 6. 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.

4.1.2 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

```

%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

```

Listing 6: Description file for a problem in *Turing Arena light*

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 [75], 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 [94].

4.1.3 Client

On the other side of the network²⁹ 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.

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.

4.1.4 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.

²⁹Which might even be on the same machine, if both the server and the client are running on the same machine.

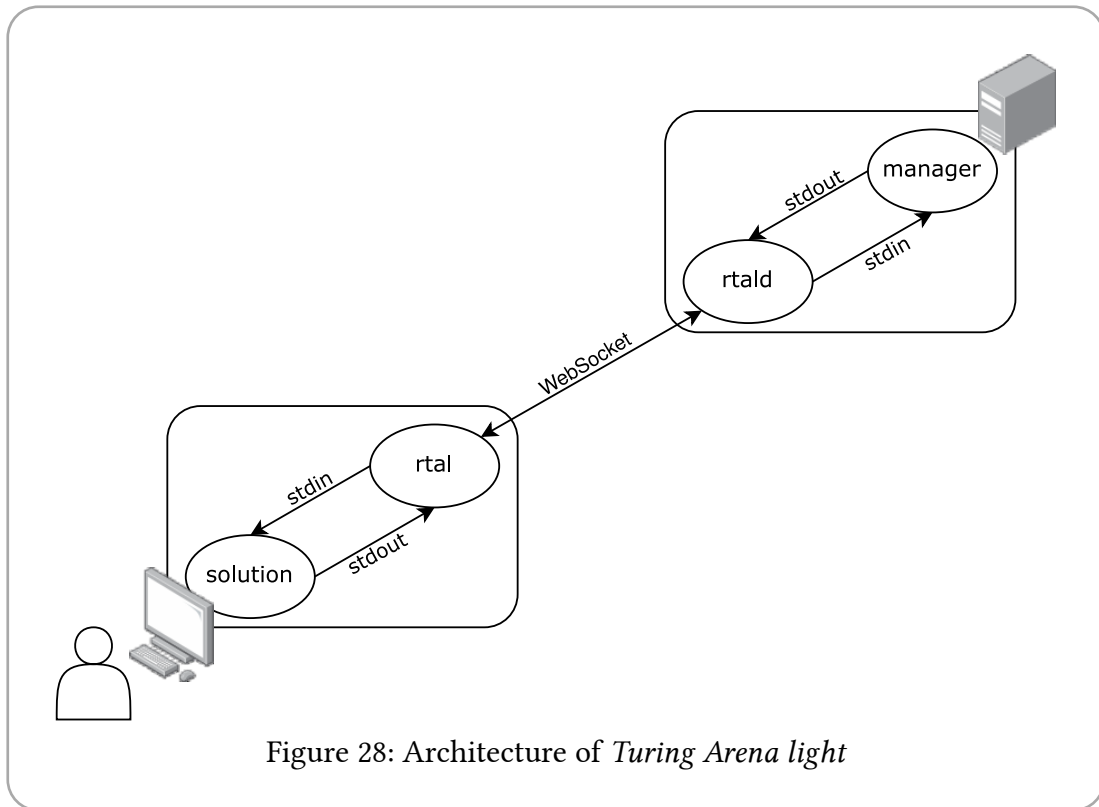


Figure 28: Architecture of *Turing Arena light*

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.

4.2 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 [75]. 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 programs. 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.

```

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>,
}

```

Listing 7: Problem description definition in *Rust Turing Arena light*

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 Listing 7. The definition is written using Rust structures which are then serialized to and deserialized from YAML using *serde* [95], 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.

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* [96] 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 com-

```

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> },
}

```

Listing 8: Network protocol definition in *Rust Turing Arena light*

munication within the server and the client is done using the actor threading model [97, 98].

For the networking part, the communication protocol between the server and the client is based on WebSockets [94]. The protocol definition is shown in Listing 8. The protocol is based on JSON [99] 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 client is shown in Figure 28. 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 [100], to mitigate the risk of a bug in the problem manager that could cause unauthorized access to the server.

4.2.1 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 Listing 9. 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.

As the name suggests, the `tc.py` library is written in Python [46], and it is meant to be used with problem managers written in Python. This works great for problems


```

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()

```

Listing 9: Snippet of the python version of the competitive-programming like problem manager library for *Turing Arena light*

where the optimal solution plays well with Python, however in problems where the

```

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)
);

```

Listing 10: SQLite schema for database used by `tc.py` and `tc.rs`

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 [101]. By using Rust as the programming 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 [102] database. The schema of the database is shown in Listing 10. 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*.

4.3 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.

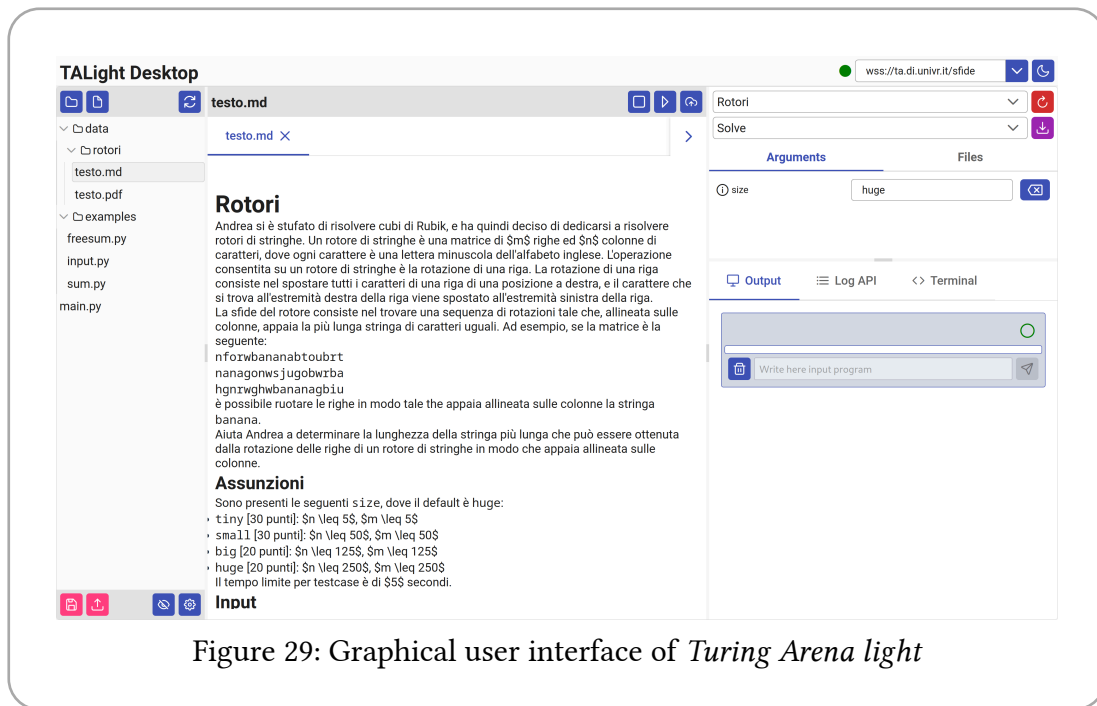


Figure 29: Graphical user interface of *Turing Arena light*

A web application was developed as a new client for *Turing Arena light* [103]. A screenshot of the application is shown in Figure 29. It was developed using the *Angular* framework [104], and it is written in TypeScript [105]. The 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* [106]. 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 [107], Google Drive [108], or OneDrive [109]. They can be later imported back from a tar archive or from the cloud, specifically from GitHub.

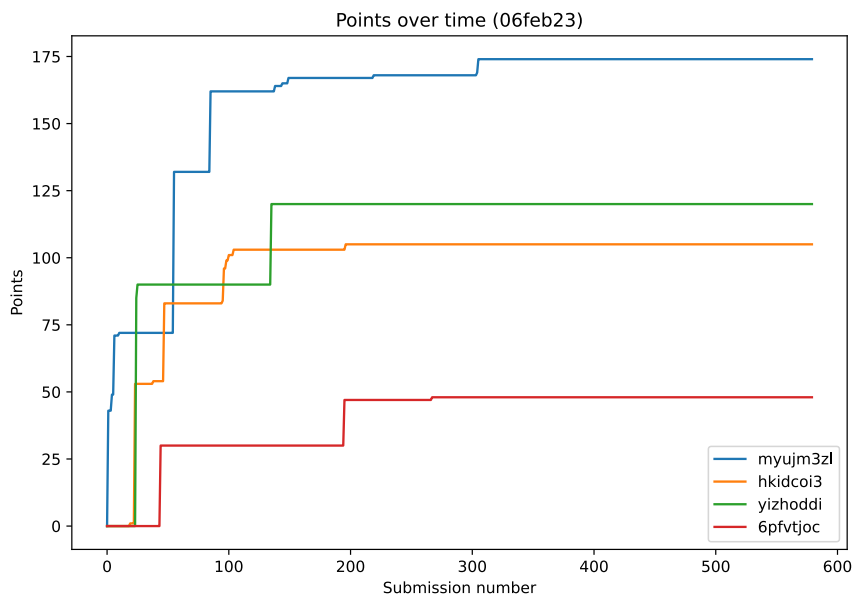


Figure 30: Results of the 6th of February 2023 exam

4.4 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

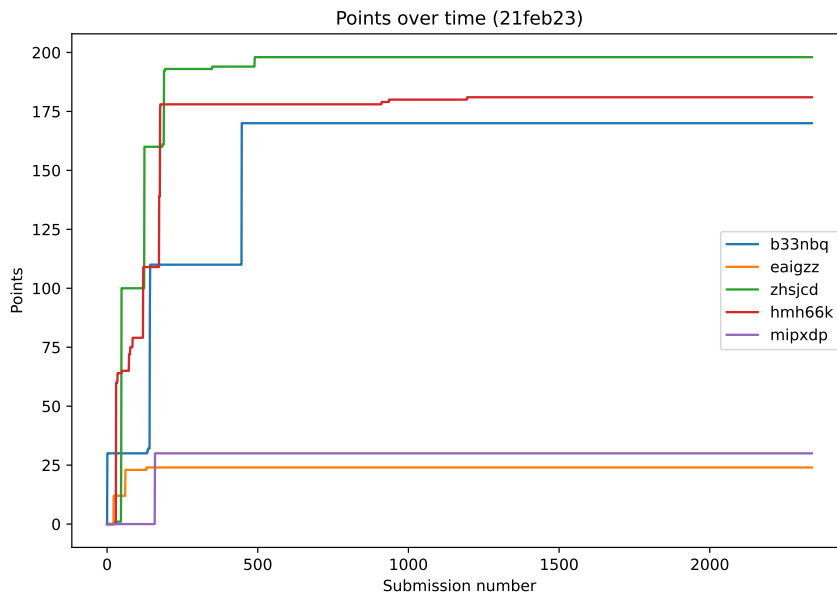


Figure 31: Results of the 21st of February 2023 exam

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

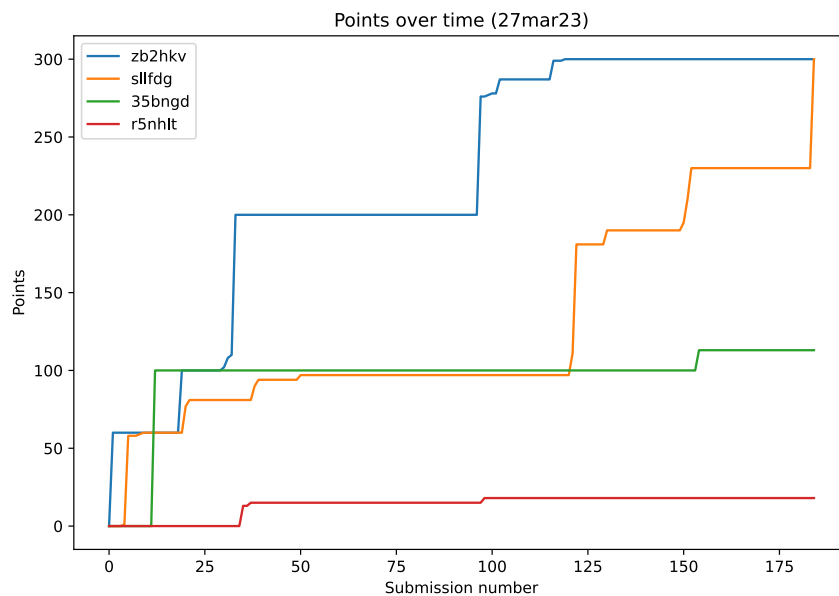


Figure 32: Results of the 27th of March 2023 exam

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!

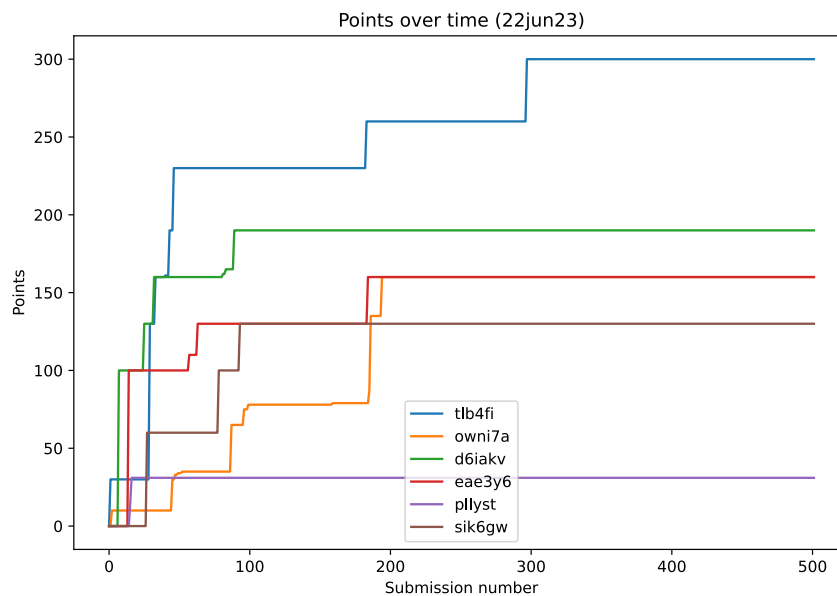


Figure 33: Results of the 22nd of June 2023 exam

Assumptions

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

- small: $n \leq 8, \max(V) \leq 20$.
- big: $n \leq 50, \max(V) \leq 10^6$

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.

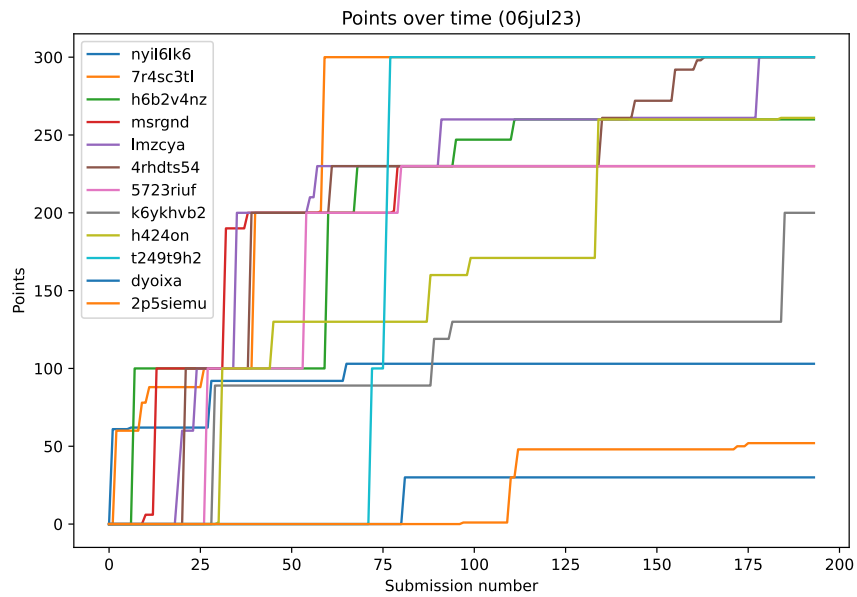


Figure 34: Results of the 6th of July 2023 exam

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

```

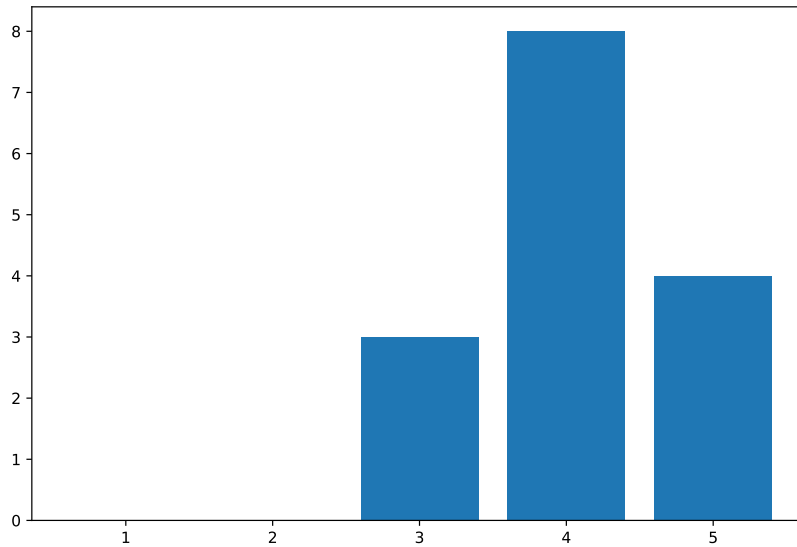



Figure 35: Answers of question 1 of the post-exams survey: *How much did you like the problems available in Turing Arena light (rtal)?*

4.4.2 Exams

Aside from the laboratory lessons, *Turing Arena light* has 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* client (`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 [110].

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 student³⁰. The results of the exams are shown in Figure 30, Figure 31, Figure 32, Figure 33 and Figure 34. 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

³⁰If a student took the exam multiple times, they would have a different identifier each time.

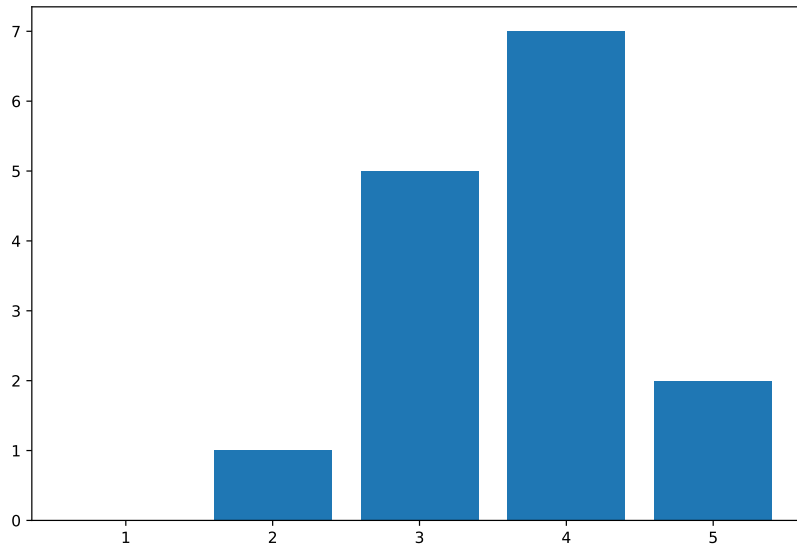


Figure 36: Answers of question 2 of the post-exams survey: *How difficult did you find the problems proposed with Turing Arena light (rtal)?*

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.

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 .

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.

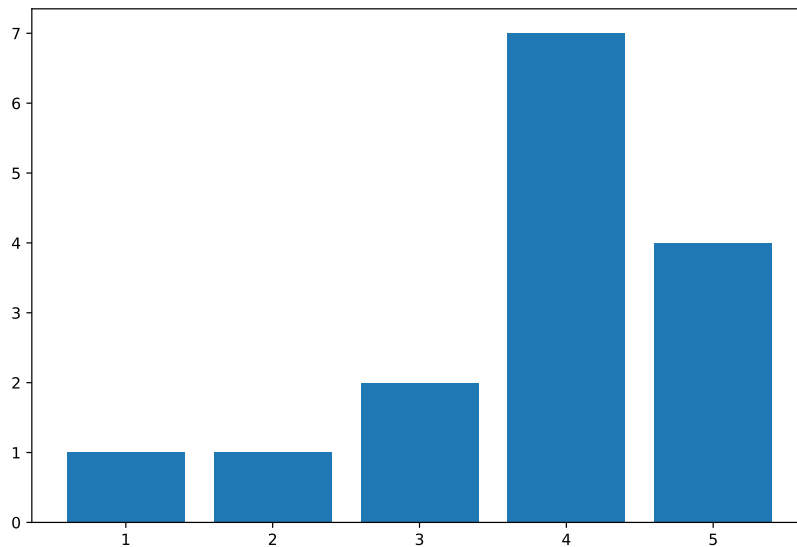


Figure 37: Answers of question 3 of the post-exams survey: *Did you find the interactive problems more interesting than the regular ones?*

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 measurements 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]: $n \leq 45$, each joint is connected to at most 2 pipes
- small [30 points]: $n \leq 45$
- big [40 points]: $n \leq 50$

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:

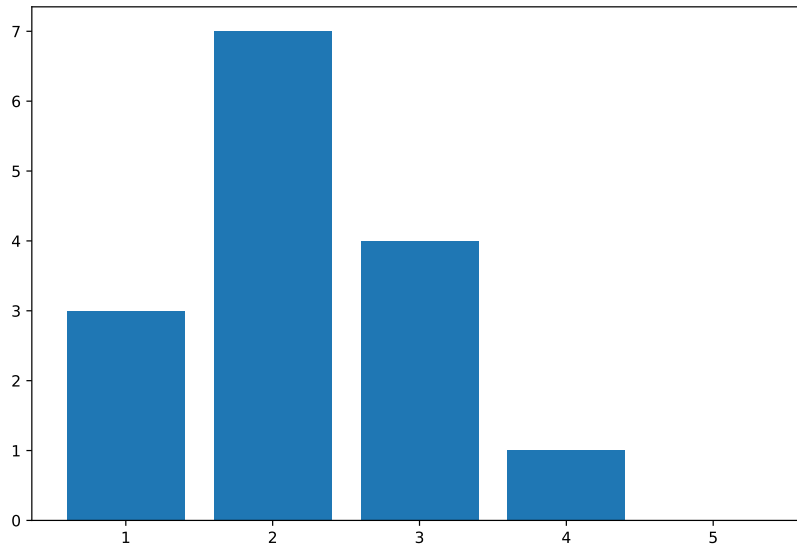


Figure 38: Answers of question 4 of the post-exams survey: *How hard was to use Turing Arena light (rtal)?*

- $? u v$: 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.

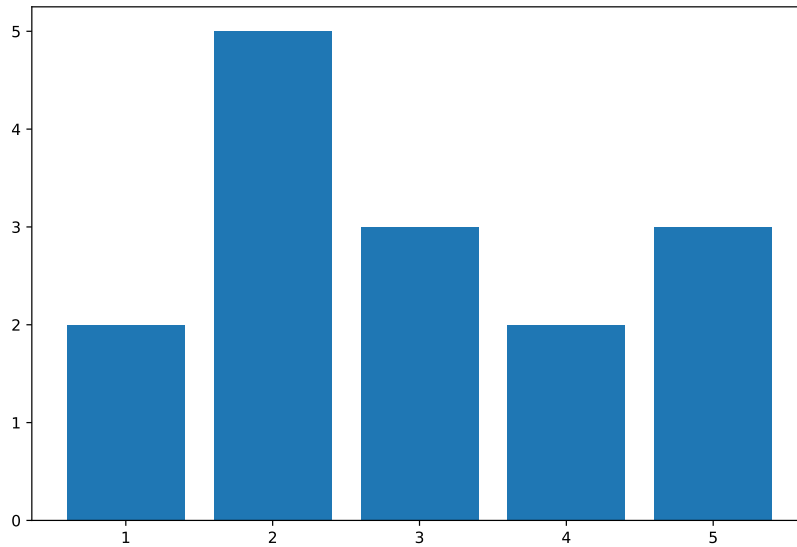


Figure 39: Answers of question 5 of the post-exams survey: *How strongly would you like for Turing Arena light to have a graphical user interface?*

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

4.4.3 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 [111]. The survey consisted of five questions:

1. How much did you like the problems available in Turing Arena light (rtal)?
2. How difficult did you find the problems proposed with Turing Arena light (rtal)?
3. Did you find the interactive problems more interesting than the regular ones?
4. How hard was to use Turing Arena light (rtal)?
5. 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.

1. The responses for the first question are shown in Figure 35. 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.
2. The responses for the second question are shown in Figure 36. As can be seen from the results, the students found the problems proposed with *Turing Arena light* to be of slightly-above-medium difficulty. The average score is 3.67, which means that the problems were not too difficult, but they were not too easy either, although they were slightly on the hard side.
3. The responses for the third question are shown in Figure 37. 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.
4. The responses for the fourth question are shown in Figure 38. 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.
5. The responses for the fifth and final question are shown in Figure 39. 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.

4.5 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, as shown by the results of the survey. 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 man-

ager. 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.

5

Code Colosseum

In this chapter we will introduce *Code Colosseum*, a platform to create and play real-time multiplayer games meant to be played by programs. We will present *Code Colosseum* as a complementary educational tool, akin to competitive programming but based on games instead of problems, to foster Computer Science education to a broader audience.

In 2021, we³¹ wrote a paper called *Make your programs compete and watch them play in the Code Colosseum* [112] which lays down the main ideas behind *Code Colosseum*, how it works and how it has been tested. What follows is a reworked version of that paper. After that, we will discuss the current state of the project, with all additions and improvements that have been made since then.

Games have a role in many aspects of science, technology, and society. Also, games attract human interest and offer unique learning opportunities. Indeed, the role of games in education has a long tradition.

We introduce *Code Colosseum*, a platform that takes competitive programming toward games instead of problems. By taking this direction, we aim to create a more engaging environment for students to compete in. The platform allows programs written by the contestants to compete in a real-time multiplayer game. The platform also allows us to spectate the matches between the programs. The design and implementation of *Code Colosseum* have been kept as simple as possible to facilitate participation, maintenance, and setup.

To assess the approach's effectiveness, we organized a tournament with 16 high school and university students as a pilot experience for *Code Colosseum*. In this tournament, they created programs to play the *Royal Game of Ur*, a board racing game. The feedback from the students about the experience was positive, and the suggestions received will be implemented for future experiences.

With the rise of *STEM*³² education, Computer Science has become one of the central subjects to be incorporated inside the curricula of secondary and primary education systems worldwide.

While the secondary education systems transit towards a more scientific-focused teaching of Computer Science, one way to give students an insight into it is through

³¹Dario Ostuni (University of Verona), Edoardo Morassutto (Politecnico di Milano) and Romeo Rizzi (University of Verona)

³²Science Technology Engineering Mathematics

extracurricular activities [113]. One such example is competitive programming, which has a long history of being used as an educational tool [114]. Many competitions in this field have an interest in fostering Computer Science education, such as the *International Olympiad in Informatics* [57], which is more competition-oriented and for high-school students only, the *Kangourou of Informatics* [115], which is more didactic-oriented and also open to middle school students, and, recently, *Codeforces* [116], the leading web platform for programming contests.

The role of competitive programming in computer science education has multiple facets: students become interested in various computer science topics and group up at the school level, national level, or even international level. The social aspect value is confirmed by the attention it receives in web platforms such as the American *TopCoder* [117], the Indian *CodeChef* [118], the Japanese *AtCoder* [56] and the Italian *CMSocial* [119]. Such communities strengthen students' interest in competitive programming and Computer Science. They also offer a place where knowledge and competence are emphasized and where students who might not find it at their schools can seek it.

While competitive programming fills an ever-expanding niche, many students may feel discouraged, seeing it merely as a competition for the best of the best. Furthermore, competitive programming might be less engaging for outsiders because the knowledge requirement to appreciate it is high.

We present *Code Colosseum*, a platform to expand competitive programming using games as its primary feature. Using games to increase engagement and motivation has proved effective in Computer Science education [120].

Other platforms that offer similar features already exist. For instance, *CodingGame* [121] and *CodeCombat* [122] are web platforms that let students learn programming with games; here, their programs can be developed in the platform web environment and are run on the platform server. One of the leading technical differences with *Code Colosseum* is that the students' programs are developed and run solely on their machines.

5.1 Design, Motivations and Goals

Creating a program (bot) that plays a game and let it compete against other bots within a shared and observable arena creates an engaging environment. For games where optimal play is out of reach or where there is a well-balanced component of luck, the matches can be exciting and instructive to observe, and the competition offers opportunities for social interactions. When the outcomes are somewhat predictable in advance, people can discuss the strengths and weaknesses of the bots and their strategies. This is even more so if the design and making of the bots is a team activity. Also, a match is engaging not only for insiders, who have direct interaction with their peers and share with them the same language and mental space, but also

for outsiders, who can be interested in just watching the matches, making enough sense of what is happening even without any prior Computer Science knowledge. This is very close to the idea behind *RoboCup* [123], which is a very successful competition where teams of high-school students assemble and program some robots to make them compete in a physical soccer-like game against those of other teams.

We propose *Code Colosseum*, a framework for creating and deploying real-time multiplayer games meant to be played by bots. The game could be a classic one, like checkers or any known card game; it could be a variant of a classic or an entirely new game. Both collaborative and competitive games are possible.

Code Colosseum is implemented as a client-server architecture, where the server manages the ongoing matches, and the clients connect the players' bots to the server, communicating over the net. Some of the main traits of *Code Colosseum* are:

- the server acts as the central and trusted hub for each match and always has complete information on the state of the game. To play a match, the clients need to connect only to the server;
- the server offers a lobby where you can create or join a match. When creating a match, one sets the number of players and how many of them should be covered by server-provided bots;
- the server can only manage a particular set of games. This set can be extended by implementing the rules and communication protocol for a new game;
- the player's bot can be any program. As such, it can offer an environment from which a human might directly play or assist an AI playing it;
- all matches can be publicly spectated.

The fact that the player's bot runs on their machine has several profound implications, among these:

- each participant can use their preferred programming languages, libraries, and tools without bearing on what is available on the server;
- during the matches, bots can make use of precomputed information and can leverage available hardware like GPUs;
- since each bot runs on the player's machine, standard debugging tools and techniques can be used to debug its logic and protocol implementation;
- bots do not add to the load of the server in terms of computing resources;
- it lifts the sandboxing requirement that would otherwise be needed for security reasons. This also helps in keeping a low complexity of design and implementation;
- since the client-server communication happens over the net, each participant needs a stable internet connection;
- the computational resources available to different players might wildly differ. The server cannot limit nor know such resources;

- in team games, the server has no control over the intra-team communication. For instance, in 2v2 card games without intra-team communication, the bots can privately communicate through other channels. Note, however, that *Code Colosseum* is still suitable for managing team games where the same team players can fully share their knowledge.

The following features of *Code Colosseum* are essential and must be further explored and developed.

5.1.1 Visualization

Spectating the game might be a way to get involved and prove interest or participation in the competition. Even a team coach without technical preparation might offer suggestions, opinions, or encouragement based on the matches they have seen. This sharing adds recognition and meaning while helping to promote a positive environment of interest and participation. Also, for peers and classmates, spectating the game might be the first step to getting involved. For these reasons, supporting the visualization of ongoing matches in an accessible way is an important feature that deserves further development. Unlike with usual competitive programming competitions, with games and tournaments, there is an unparalleled opportunity to obtain meaningful and immediately accessible visualizations that should certainly contribute to engaging and motivating all participants and spectators.

5.1.2 Tournaments

If the goal is to foster STEM education further, we must first comprehend the proper forms that make a proposal truly inclusive for a broader range of students. In particular, the social dimensions must be addressed since they relate to profound motivations. Tournaments fill that space and offer a powerful opportunity to catalyze the interest of both contestants and the broader community that might gather around them in a high school or university setting.

Also, tournaments bring a focus and a shared interest in matches and players. This is particularly true for elimination tournaments, where more spectators naturally follow the last matches. Indeed, players are naturally curious to see how the best players perform and learn from their strategies. Also, players can group up to watch and comment on the live matches. This offers further opportunities to socialize and exchange ideas and enthusiasm.

5.1.3 Simplicity

In the design and first implementation of the *Code Colosseum* platform, we adopted a minimal approach striving for simplicity. The importance of this point must be considered. We aim to get a system that can be used with ease to play matches and develop a new game. In the long term, communities around this platform might form in

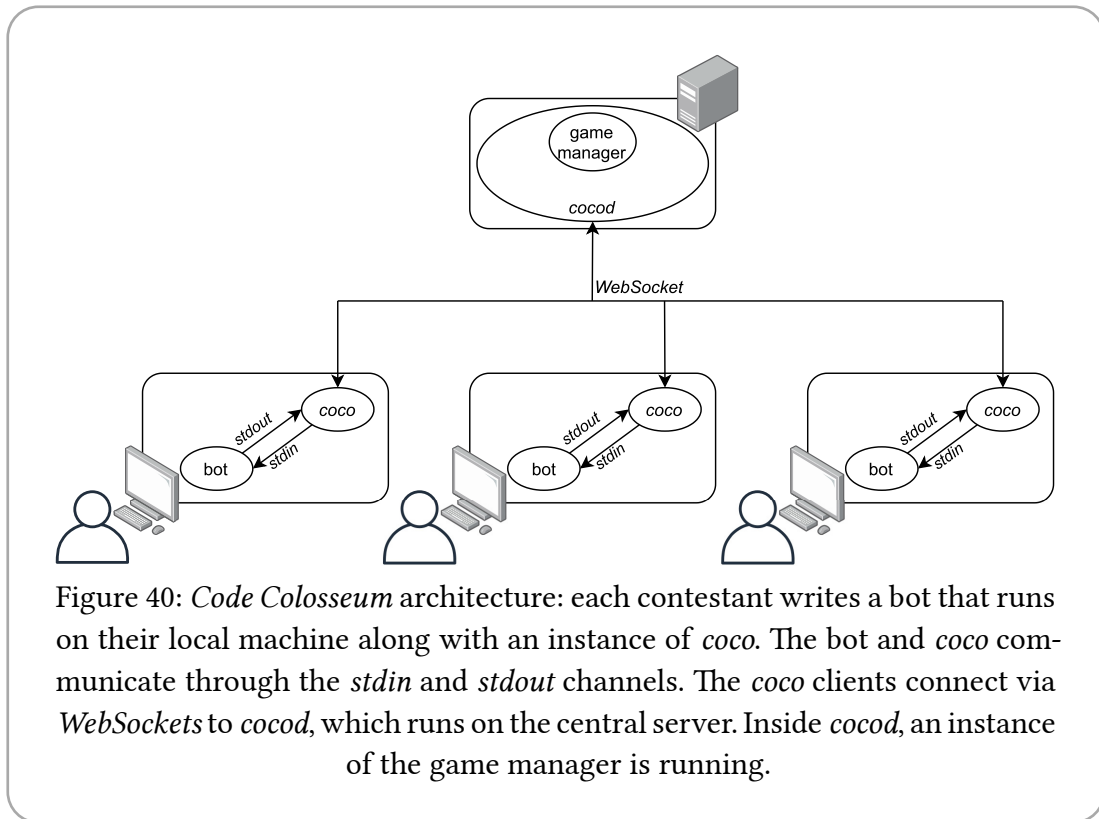


Figure 40: *Code Colosseum* architecture: each contestant writes a bot that runs on their local machine along with an instance of *coco*. The bot and *coco* communicate through the *stdin* and *stdout* channels. The *coco* clients connect via *WebSockets* to *cocod*, which runs on the central server. Inside *cocod*, an instance of the game manager is running.

high school or university settings. These communities will first be attracted by playing available games but might later try to produce their games.

5.2 Implementation details

Code Colosseum is a framework to build and play multiplayer games over a network. It has a client-server structure. It is designed for simplicity of setup and use. The server (*cocod*) and the client (*coco*) are written in Rust [75]. *Code Colosseum* is free software released under the Mozilla Public License 2.0 and can be found on GitHub [124]. Figure 40 shows a graphical overview of the architecture.

Both *cocod* and *coco* are written using the *Tokio* [96] asynchronous runtime. This allows the *cocod* server to handle many concurrent connections efficiently, making it suitable for hosting hundreds of matches simultaneously.

Code Colosseum provides no user authentication by design to keep its codebase and setup as simple as possible. The *cocod* server keeps a lobby of waiting-to-start and running matches. Anyone can create a new match for one of the supported games and join a waiting-to-start match. However, some limitations can be imposed, for instance, by providing a password at the time of creation to restrict participation access. Once the number of players needed to start the match is reached, the match automatically begins, and the players start playing. To create, list, and join matches,

the contestants use the *coco* client. All matches can also be spectated, even when they are already running or password-protected.

It is possible to add a new game to *cocod* by expanding it. In order to do so, a *game manager* has to be written. This is a program that, given n bidirectional pipes (for the players) and an output-only pipe (for the spectators), must implement the game logic and receive/send information from/to the players and the spectators accordingly. This minimal interface tries to lower the requirements to write a new *game manager*. Note, however, that this is not a trivial task: particular attention is needed when handling multiple data streams simultaneously, as some multi-threading or polling is needed, which, if managed incorrectly, can lead to problems such as deadlocks or synchronization errors. Moreover, even though real-time and turn-based games are possible, the former are more challenging to implement since they need to account for network latency.

The communication between the *coco* client and the *cocod* server is done using *WebSockets* [94] and a custom JSON-serialized protocol. Using *WebSockets* as the communication channel has several advantages over plain TCP channels:

- they can pass through HTTP proxies, which are common in schools and various institutions when the traffic is monitored and filtered;
- they are message-based rather than stream-based, which simplifies the handling of our message-based protocol;
- they can be put behind a standard HTTP reverse proxy to enable connection security (using HTTPS) and traffic shaping;
- a *WebSocket* client can be instantiated inside a web browser, allowing a web application to communicate directly with a *cocod* server.

When a contestant joins a match using the *coco* client with its bot, *coco* connects to the specified *cocod* server and waits until the match starts. When that happens, the *coco* client captures the *stdin* and *stdout*³³ channels of the bot, and virtually connects them to a bidirectional pipe of the *game manager* for that match. This is done using the previously established *WebSocket* communication channel with the *cocod* server.

Spectators can join a match using the *coco* client anytime. The *cocod* server will send real-time game updates as generated by the *game manager*. If the spectator joins an already running match, then the *cocod* server will first send all game data from the beginning of the match. The *coco* client does not provide a native visualizer for games. Hence, a custom program that reads the game data and prints a human-readable representation is needed. It has to be specified to *coco*, which will send the game data to its *stdin* stream.

³³ *standard input* and *standard output*.

Both *coco* and *cocod* are CLI³⁴ programs. While for *cocod*, this is not much of a problem, having a graphical client could be beneficial for tasks such as spectating a match. Note that *coco* allows the attachment of an arbitrary program for both playing and spectating games, opening the possibility of a GUI³⁵. Since the communication channel is a *WebSocket* and most of the implementation burden lies on the server, a web client could be written to provide most of the functionalities of the *coco* client in a graphical and cross-platform way. Also, by providing additional layers, one could collect statistical information on the platform usage, like in *CMSocial* [69].

5.3 Pilot Experience

We conducted a pilot experience that involved 16 students with a Computer Science background in a *double-elimination tournament* playing the *Royal Game of Ur*. The students were also familiar with the *Competitive Programming* field.

5.3.1 Royal Game of Ur

The *Royal Game of Ur* [125] is a strategy game where two players play against each other in a racing competition, moving their tokens on a board according to the result of four 2-faced dice. This game is of historical importance since it is one of the oldest known games (played in ancient Mesopotamia in the third millennium BCE). However, it is relatively unknown to most people.

We selected it as the pilot game for many reasons:

- the game rules are straightforward to understand and relatively straightforward to implement in code;
- the strategy component is nicely compensated by some luck, making it enjoyable to watch and partially bridging the gap between seasoned and novice competitors;
- other than the optimal strategy, there are many more accessible but well-performing strategies.

5.3.2 Double-elimination Tournament

In a double-elimination tournament, there are two brackets: the winner bracket and the loser bracket. Initially, all the players are in the winner bracket. They are moved to the loser bracket when they lose their first match instead of being eliminated at their first defeat. This mechanism offers every player a *second chance* to be the winner of the tournament. With a *second chance* players can learn from their mistakes, fix their programs, and improve their strategies.

This type of elimination tournament has a number of matches that is a function linear in the number of players, keeping the total number of matches to a reason-

³⁴Command-Line Interface

³⁵Graphical User Interface

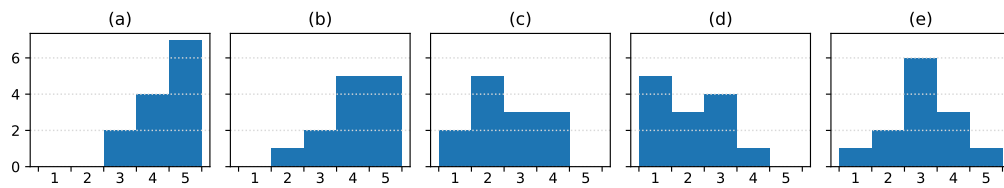


Figure 41: Histograms of the feedback collected after the tournament from 13 participants. Answers were from 1 (least) to 5 (most). (a) *How fun was it to participate?* (b) *How much did you like the Royal Game of Ur as the game chosen for the tournament?* (c) *Should the tournament have lasted longer?* (d) *How difficult was it to use a terminal interface with respect to a graphical one?* (e) *In your opinion, would it have been interesting to be a spectator?*

able amount, especially when there are many players. Note that *Code Colosseum* only manages single matches. Therefore, the tournament structure is independent of it. We used *Challonge* [126] to track the tournament’s progress.

5.3.3 Experience and Feedback

The participants and the organizers met in a *Discord* [127] server, and 14 out of the 16 registered players were present. The game was announced at the start of the tournament. Participants were provided with the game rules and communication protocol at that point. Players started implementing their bots using their favorite programming language and development environment. Following the game communication protocol, the bots would exchange text messages with *coco*. Players could prepare for the matches by either playing among themselves in friendly matches or against a server-provided bot. After 90 minutes, the first matches started, and up to two matches of the same round were held in parallel. Between rounds, the participants were given some time to tweak and fix their programs (from 15 to 45 minutes, depending on the round).

Using an anonymous survey, at the end of the tournament, some feedback was collected from the participants, and 13 participants answered. The questions were open, yes/no, and rated from 1 (least) to 5 (most).

Results [128] showed that none of the participants knew the game before the tournament, and all of them enjoyed participating”, rating it at least 3, with more than half rating it 5 (see Figure 41). All of them expressed the intent to participate in a second edition.

The game was well accepted. Ten participants graded it as the chosen game, at least 4. On the contrary, nearly half of them would have liked more time for coding and debugging their programs; a couple preferred less time between the rounds for a

more excellent spectating experience. Due to their background, the students generally found using the *coco* CLI client easy.

Many participants asked for a web-based visualizer with a database of the played matches. This would have been very useful for inspecting opponent's strategies with the added benefit of being more user-friendly and platform-independent.

5.4 Findings and further directions

We presented *Code Colosseum*, a platform to create and play real-time multiplayer games meant to be played by programs. We proposed *Code Colosseum* as a complementary educational tool, akin to competitive programming but based on games instead of problems, to foster Computer Science education to a broader audience. We provided an implementation of the *Code Colosseum* concept with the *cocod* server and *coco* client and discussed their implementation details. We then described the pilot experience we organized, a *Royal Game of Ur* double-elimination tournament between 16 students. We then described and discussed the feedback received from the students about this experience.

From the students' feedback, we can assess that the pilot experience was substantially positive: most students enjoyed participating, and all would participate again. The game made them explore concepts from game theory and statistics. The tournament has enthralled the participants in the game, so much so that some of them kept refining their bots for some weeks after the tournament.

In the future, we plan to host a second tournament. We commit to improving the experience by implementing the feedback we received from the participants. In particular, the following well-defined directions are indeed likely to have a positive impact:

- add a web interface for spectating the matches;
- add a replay functionality to re-watch past matches;
- rebalance the durations of the various phases of the tournament. In particular, by increasing the time before the first match.

5.5 Replay functionality

As mentioned in the previous section, one of the most requested features was the ability to replay past matches. This would allow the participants to inspect the strategies of their opponents and learn from them. It would also allow the spectators to watch past matches and learn from them.

The replay functionality was implemented [129] by adding a database backend to *cocod* and a new *coco* command to query the database and replay a match. The database backend is agnostic to the game being played and which database management

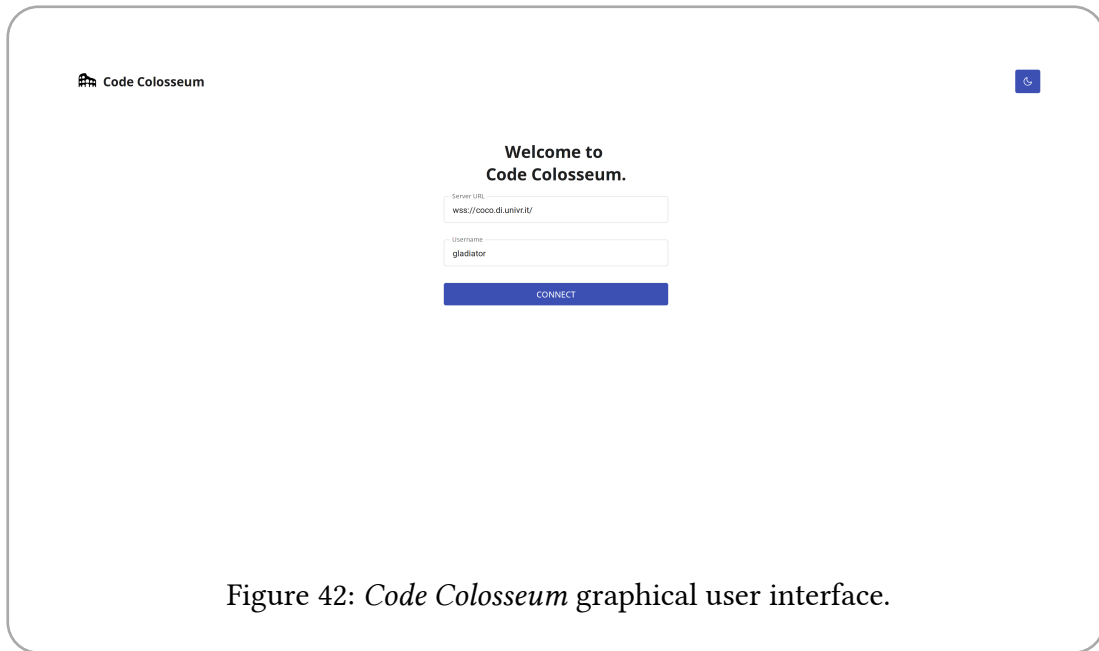


Figure 42: *Code Colosseum* graphical user interface.

system is used. The current implementation uses the filesystem as the database management system and stores the matches in JSON format. The database is updated every time a match ends.

On the frontend, a new *coco* command was added to query the database and replay a match. The command takes as input the match ID. It then queries the database for the match and sends the game data to the specified program. The program can then visualize the match as it sees fit, since the game data is the same as the one sent to the spectator during a live match.

5.6 Graphical user interface

Other than the replay functionality, the most requested feature was a graphical user interface. This would allow the participants to participate in and spectate matches without having to use the terminal. It would also allow the participants to visualize the matches in a more user-friendly way. This would be especially useful for spectators and less-experienced participants.

The graphical user interface was implemented [130] using Tauri [131], a framework for building web applications using Rust. The frontend is a web application written in TypeScript [105] and Angular [104]. The frontend communicates with the backend using *WebSockets* using the same protocol as the *coco* client.

Currently, the graphical user interface allows the user to create matches, join matches with a bot, and spectate matches. The replay functionality is still not available in the graphical user interface. A screenshot of the graphical user interface is shown in Figure 42. The existence of a graphical user interface also makes some quirks of the *coco* client more apparent, such as the lack of a *ready* command to signal the

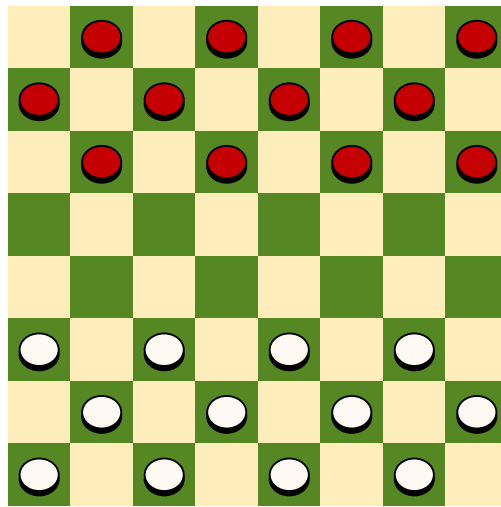


Figure 43: A checkers board with the pieces in their starting positions.

server that the bot is ready to play. Thus, the protocol will be updated to reflect such new requirements.

5.7 Additional games

To further increase the appeal of *Code Colosseum*, two new games were implemented: *Checkers* and *Chess*. Both games are well-known and have a large player base. They are also both turn-based and deterministic, making them easier to implement than real-time games. Both games were implemented using Rust [75], since they must be compiled with *cocod*.

5.7.1 Checkers

Checkers (or draughts) is a two-player strategy board game played on an 8×8 board with 12 pieces per player. The goal of the game is to capture all the opponent's pieces or to block them from moving. Although checkers is a solved game [132], the number of possible moves is still large enough to make it interesting to watch. An example of a checkers board is shown in Figure 43.

The game of checkers was added [133] to *Code Colosseum* by implementing the *game manager* for it. The manager has been called *Dama* (Italian for checkers). This manager is more geared towards human players than the other managers, since it allows the players to see the board state. However it is still possible to play as or against a bot.

5.7.2 Chess

Chess is a two-player strategy board game played on an 8×8 board with 16 pieces per player. The goal of the game is to capture the opponent's king. Chess is one of

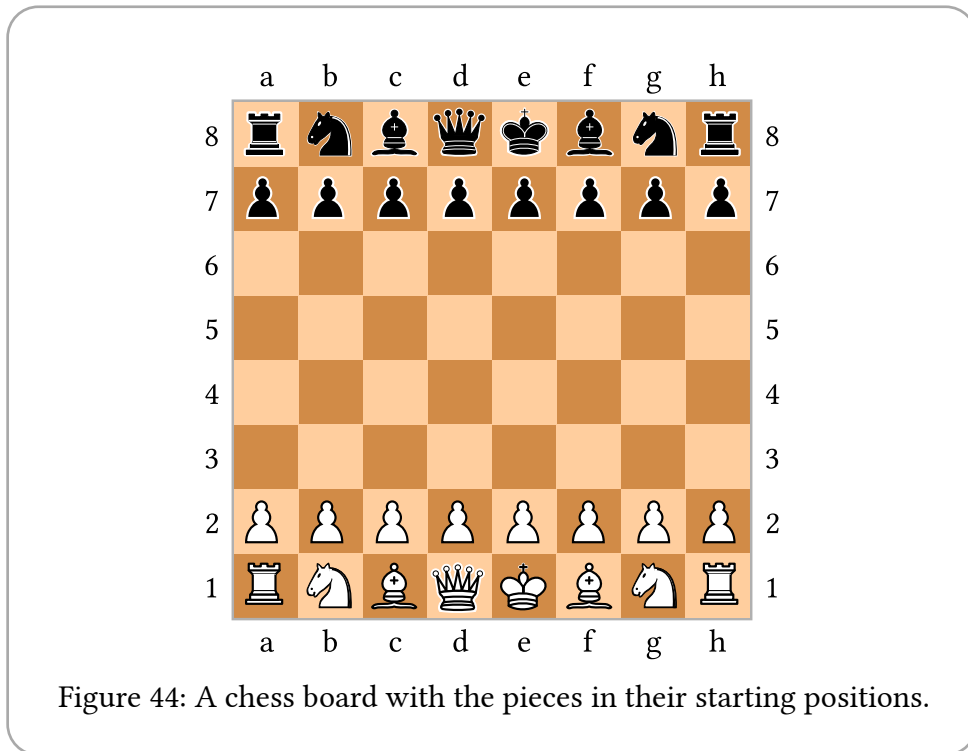


Figure 44: A chess board with the pieces in their starting positions.

the most popular games in the world, with millions of players worldwide. It is one of the most complex games, with a large number of possible moves. An example of a chess board is shown in Figure 44.

The game of chess was added [134] to *Code Colosseum* by implementing the *game manager* for it. The implementation of the manager follows the typical interface of the other managers, thus it is more geared towards bots than human players. However, it is still possible to play as a human player and use the spectator stream with a graphical user interface to visualize the board state.

5.8 Future directions

The current state of *Code Colosseum* is still far from being a complete and polished product. There are many features that could be added to make it more appealing and user-friendly. To gather feedback on the current state of the project, other tournaments should be organized. This would allow us to gather more feedback and improve the platform accordingly.

Other than the feedback from the participants, the platform should be tested with a crowd of spectators. This would allow us to gather feedback on the spectator experience and improve it accordingly, which is one of the main goals of *Code Colosseum*. In this direction, a big feature that would improve the spectator experience and further standardize the *game manager* interface would be the addition of a routine that generates a series of images representing the game state given the spectator stream.

This would allow the spectators to watch the matches in a consistent way, regardless of the game being played.

Another feature that would improve the making of new games for *Code Colosseum* would be the support for writing *game managers* in other programming languages other than Rust. While Rust has all the features needed to write a *game manager*, it is still not a widely used language. Thus, supporting other languages would allow more people to contribute to the project.

We presented the world of competitive programming and we gave an overview of how a competitive programming problem looks like, what different kinds of problem solving techniques are used to solve them and what programming languages are currently used in competitive programming. The techniques for solving problems include *recursion*, *dynamic programming*, *greedy algorithms* and *divide and conquer*. As for the languages, we described the strengths and weaknesses, in competitive programming, of *C*, *C++*, *Java*, *Python* and *Pascal*. We concluded that *C++* is the most used language in competitive programming at the time of writing this thesis.

We then gave an overview of the two biggest competitive programming competitions in the world, the *International Olympiad in Informatics* (IOI) and the *International Collegiate Programming Contest* (ICPC). Regarding the IOI, we talked in specific about the Italian team selection process for the IOI, the *Olimpiadi Italiane di Informatica* (OII). In this context, we presented a report on the challenges of the organization of the OII contests during the COVID-19 pandemic. In order to move everything online, we developed new platforms that allowed for running the contest online while still maintaining the same spirit of the in-person contests. For the school-level contest, we developed *randomTeX*, a program for generating randomized tests using a pool of problem variants. For the regional-level contest, we adapted the previously developed *Terry* contest management system to run the contest online. Finally, for the national-level contest, we developed *oii-proctor*, a program for monitoring the students during the contest. Moreover, we presented some analytics from *CMSocial*, a fork of the *CMS* contest management system [58]. Here, we analyzed the data from two groups of people using the *CMSocial* platform: the students and teachers. We estimated the difficulty of the problems and we analyzed the performance of the students compared to the one of the teachers, finding that the two groups have similar performance distribution. Regarding the ICPC, we talked in specific about the *South-Western European Regional Contest* (SWERC). In this context, we gave a report of the organization of the 2022 and 2023 editions of the contest, which we organized. We showed the statistics about the historical participation of the teams to the SWERC and we showed that the last edition we organized was the largest one according to all the three considered metrics.

Then, we presented the first of the two *novel* contest management systems we developed: *Turing Arena light*. This system was designed with problem interactivity and ease of use in mind. The goal was to use it as an aid in teaching computer science concepts to students. Its implementation was kept as simple as possible, while providing all the feature needed to achieve these goals and keeping it extensible. To achieve

this, we kept all the critic features inside a small standardized core, and extended it as needed. We described its client-server architecture and described the abstraction of the communication protocol between the solution and the problem manager. We then described how problem formats can be implemented through problem manager libraries, and showed an in-progress implementation of a web graphical user interface for the client. Finally, we talked about the usage of *Turing Arena light* in the competitive programming course, at the University of Verona, both for laboratory exercises and for the exams. We reported the statistics of the participation and scores of the exams held using *Turing Arena light* and showed that there has been an increase over time both of the number of students participating and of number of students achieving a perfect score. We then reported the results of a survey we conducted among the students, which showed that the students were satisfied with the platform, the problems and the platform, and found the interactivity element of *Turing Arena light* to be of interest. These results combined show that *Turing Arena light* can be a viable platform for teaching computer science concepts to students while retaining their interest.

Finally, we presented the other *novel* contest management system we developed: *Code Colosseum*. This system was designed as an arena where programs written by the users can compete against each other in real-time playing a game. Many of the concepts of *Code Colosseum* are inherited from *Turing Arena light*, such as the client-server architecture, the interactivity and keeping it simple to use. While trying to be a tool that creates engagement in the students using it, thus being useful for teaching computer science in a fun way, it also tries to be a system that makes competitive programming interesting to watch, by showing what the participants' programs are doing during the game in real-time. We described in detail the design, motivations and goals of the tool, specifically by talking about the visualization element, the tournaments opportunity and its simplicity aspects. We then dwelt into the implementation details, describing the client-server architecture and the communication protocol. We then described the pilot experience we conducted with *Code Colosseum*, where we organized a tournament for the students of the Italian competitive programming community. For this pilot experience we chose the *Royal Game of Ur* as the game to be played. We described the game and the rules of the tournament. After the tournament, we conducted a survey among the participants, which showed that participating in the tournament was a positive experience for the participants, that the game chosen was interesting and that the visualization of the game, from a spectator point of view, was mostly a positive experience. From these results we can conclude that *Code Colosseum* can be a fun experience for teaching computer science concepts to students, but it still needs some work to be done to make it also more interesting for spectators to watch. After this pilot experience, more features were added to *Code Colosseum*, namely: replay functionality to allow for watching past

games, a graphical user interface for the client, and two new games to be played on the platform: *checkers* and *chess*.

6.1 Future works

There is still much that can be done for the development of competitive programming platforms for computer science education. Aside from improvements to the two platforms presented in this thesis (*Turing Arena light* and *Code Colosseum*), there are other ideas that go in the direction of making competitive programming more accessible to younger students, that do not necessarily have the programming skills needed to compete in regular competitive programming contests, but can benefit from developing early the problem solving mindset of competitive programming. An example of a project that goes in this direction is *QuizMS* [135], which is being developed by the team of the *Italian Olympiad in Informatics (Olimpiadi Italiane di Informatica, OII)*.

For *Turing Arena light*, the main point for future works is deciding in which direction to take the development of the platform. There are two main options: keep it a very flexible tool, keeping the core as small as possible and continue adding features only through extensions, or make it a more complete and opinionated platform, standardizing more features into the core and enabling more complex features to be developed, such as more accurate time limit management. Other than that, the secondary point of development is the betterment of the graphical user interface, which is currently in an early and untested stage of development.

For *Code Colosseum*, the main way forward is to organize more tournaments, with more games, and to collect more feedback from the participants. This will allow to better understand what works and what does not work in the platform, and to improve it accordingly. Moreover, the whole system has to be polished to become more user-friendly, both in terms of the graphical user interface and in terms of the user experience, such as the addition of tooling to automate the testing of the players' programs. On the technical side, the way of showing the game to the spectators must be standardized, so that it can be used for any game, and the visualization can be easily done by all graphical clients. Furthermore, the game manager interface must be expanded to allow for game managers to be written in any programming language, and not only in *Rust*.

7

Other works

Aside from the work on competitive programming in these years we also worked on other research topics. In particular, we worked on:

- *AI playing Touhou from pixels*: we investigated the possibility of using AI to play Touhou, a bullet hell game. While AIs that play this game by looking at the internal state of the game are already available, we wanted to see if it was possible to play the game only with the information that a human player would have, i.e. the pixels on the screen.
- *SMT, MILP and SAT models for DTP*: we explored the *Disjunctive Temporal Problem* (DTP) and we developed models for solving it using *Satisfiability Modulo Theories* (SMT), *Mixed Integer Linear Programming* (MILP) and *Boolean Satisfiability* (SAT) solvers.

In this chapter we will report the results of our research on these topics.

7.1 AI playing *Touhou* only from pixels

In 2021, we³⁶ wrote a paper called *Towards an AI playing Touhou from pixels: a dataset for real-time semantic segmentation* [136], that investigates what’s needed for an AI to play a *shoot ‘em up* game when only looking at the pixels. What follows is a reworked version of that paper.

This paper was motivated by the fondness we have for the *Touhou Project*, and a willingness to study the problem of AIs playing games from pixels. During the initial research, we found out that the problem of AIs playing *Touhou* from pixels is way more challenging than we initially thought, so we focused on doing a first step towards it: creating a model capable of real-time semantic segmentation of *Touhou* game frames.

We also found out that this secondary problem is not trivial, so we decided to write a research paper about it.

When playing from pixels, AIs share some of the struggles humans face when playing a game, namely, not knowing its internal state. We begin the exploration of the AI-playing-from-pixels problem for *Touhou*, a bullet hell game. Albeit being a massively popular game in some niches, the community has yet to produce an AI capable of beating it while looking only at pixels in *Lunatic* mode, the hardest difficulty.

As a first step, we propose to build a semantic segmentation model to create a bridge to the internal-state-looking AIs. We created a dataset to train models for this task to achieve this. This dataset is procedurally generated using manually labeled assets from classic era *Touhou* games.

After selecting five state-of-the-art real-time semantic segmentation networks, we trained them using our dataset. The results indicate that the models produced have a high classification performance over the validation set. However, all models except one are too slow to run in real-time at the game’s target frame rate. The models show promising results on actual game footage, but the dataset needs to be strengthened to account for noise sources in the real game.

7.1.1 Introduction

The research on Artificial Intelligence is vital for an ever-expanding set of fields with immediate applications (e.g., autonomous driving and robotics). However, a significant driving force for it has come from Game AI, whose results are applied to other fields [137].

One active topic of research is AIs playing games from pixels. The seminal paper in this regard is from *DeepMind*, in which their general AI managed to play *Atari* games

³⁶Dario Ostuni (University of Verona) and Ettore Tancredi Galante (University of Milan)

from pixels [138]. Nonetheless, there is still interest in developing AIs playing from pixels for a single game. For instance, the *ViZDoom* competition [139] is about AIs playing *Doom* from pixels only. According to the competition report, although the competition is centered around a single game, AIs can still not compete at the same level as humans [140].

The *Touhou Project*, or simply *Touhou*, is a series of *bullet hell* (a sub-genre of *shoot 'em up*) games that gained massive popularity in Japan, the country from which its creator is from, and internationally [141]. As of 2021, there are 30 installments in the *Touhou* series, with 18 main titles and 12 spin-offs. The gameplay of the main titles consists of a vertical-scrolling setting where the main character must dodge a barrage of bullets from the enemies while trying to shoot them down. Each title offers four levels of difficulty: *Easy*, *Normal*, *Hard* and *Lunatic*.

Developing an AI for playing a *Touhou* game at *Lunatic* difficulty by reading only the raw output pixels is challenging. One of the focuses of *Touhou* is to get swift reaction times from players to dodge bullets. By running at 60 FPS³⁷, AIs playing it must make their decisions quickly. Delaying them by even a few frames could be fatal for the main character. Thus, AIs should be able to keep up with the game in real-time. Moreover, playing *Touhou* occasionally requires long-term planning of movements, such as in boss fights. Some AIs for *Touhou* exist [142]. However, they rely on internal game state information, such as the position of the main character and enemies, or they need to swap the game assets to recognize different objects by simply color-coding.

We propose creating a real-time semantic segmentation model as a first step towards achieving the result of such an AI. This approach aims to achieve a similar starting condition as the assets-swapping technique. To create a model, we first need a dataset for training and testing possible models. The following section shows how we created a dataset for this task starting from the assets of classic era *Touhou* games³⁸. Then, we give an overview of real-time semantic segmentation and present five state-of-the-art networks. We then explore how we trained these networks using our dataset and how the resulting trained models performed at labeling generated and authentic images from *Touhou* games.

7.1.2 Dataset Generation

Creating an extensive dataset for semantic segmentation can be arduous; collecting, preparing, and annotating data requires large amounts of effort. Nonetheless, several large generic and domain-specific datasets exist, such as *Pascal VOC* [143, 144], *Microsoft COCO* [145] and *Cityscapes* [146]. All these datasets were created by outsourcing the work necessary for the most time-consuming tasks, primarily image

³⁷Frames Per Second.

³⁸The *classic era* refers to the games from *Touhou 6* to *Touhou 9.5*

annotation. A similar endeavor could be sought to create a dataset for semantic segmentation of *Touhou* game frames, capturing thousands of screenshots of the game and then manually labeling them. However, such a dataset can be created in another way: by only manually marking the game assets and then crafting generated game frames that can be labeled automatically.

Such an approach is possible because *Touhou* satisfies the following requirements:

- it is a primarily 2D game³⁹, thus creating an artificial game frame can be done by simply compositing the 2D assets on a blank canvas;
- a typical actual game frame has a pretty simple structure; thus, it is easy to replicate by using simple operations such as scaling and rotation of the assets;
- the amount of assets is small and unambiguous enough to label manually;
- there are no assets used in semantically different ways inside the game.

Under such requirements, procedurally generating a dataset, given the manually labeled assets, should yield similar results to a manually labeled dataset of game frames when such datasets are used to train semantic segmentation models. The former approach, however, comes at a fraction of the cost and can generate an arbitrarily large dataset.

We extracted the assets of *Touhou* games from 6 to 9.5 with the help of *Touhou Toolkit* [147] and manually labeled them into 15 categories. To aid the labeling, we created a *YAML*-serialized [93] format to hold the labeling information. A total of 4129 bitmaps were labeled.

We then created *ThGen*, a program to procedurally generate *Touhou*-like game frames of the main playing field. To ensure high generation speed, the programming language chosen for *ThGen* was *Rust* [75], but bindings for *Python* were also written to be able to access it from a more traditional *machine learning* environment. The default target size for the generated image has been set to 576×672 to match the original one.

ThGen reads the labeled assets extracted from the games and generates a fictional game frame of the playing field by randomly placing objects on an initially blank canvas. While placing such objects, it separately records the positions of the placed objects' pixels to generate a label for the created image. The process of *ThGen* to generate an image and its associated label from a given seed is entirely deterministic. Thus, the same dataset can be generated on different runs starting from just the labeled assets and *ThGen*.

We have selected 15 semantic labels, colored with the *hue* palette from *Seaborn* [148], that should almost entirely cover the kind of objects present in the game:

- *Player*: the main character (e.g., Reimu and Marisa);
- *Boss*: stage boss (e.g. Cirno);

³⁹Some backgrounds are rendered in 3D.

- *StandardEnemy*: common enemy (e.g., fairies);
- *MajorEnemy*: more sturdy enemy (e.g., big fairies);
- *StaticEnemy*: undefeatable enemy (e.g., library books);
- *PlayerBullet*: bullet from the player;
- *EnemyBullet*: bullet from the enemies;
- *PlayerBomb*: bomb used by player (e.g. *Master Spark*);
- *PowerItem*: power-up item (e.g., red power-ups);
- *PointItem*: item to gain points (e.g., blue points);
- *GameSpecificItem*: power item specific to a certain game version (e.g., cherry items in *Touhou 7*);
- *LifeItem*: 1-up for the player lives;
- *BombItem*: 1-up for the player bombs;
- *Text*: text as seen in the playing field;
- *Background*: 2D background of the game.

The major missing label is for dialogue objects, which occur when dialogues between the characters arise. However, such objects are unimportant for active gameplay, as they can be skipped.

7.1.3 Semantic Segmentation Networks

To evaluate the performance of a semantic segmentation model, there are two common metrics: *pixel accuracy* and *mean intersection over union (mIoU)*. *Pixel accuracy*

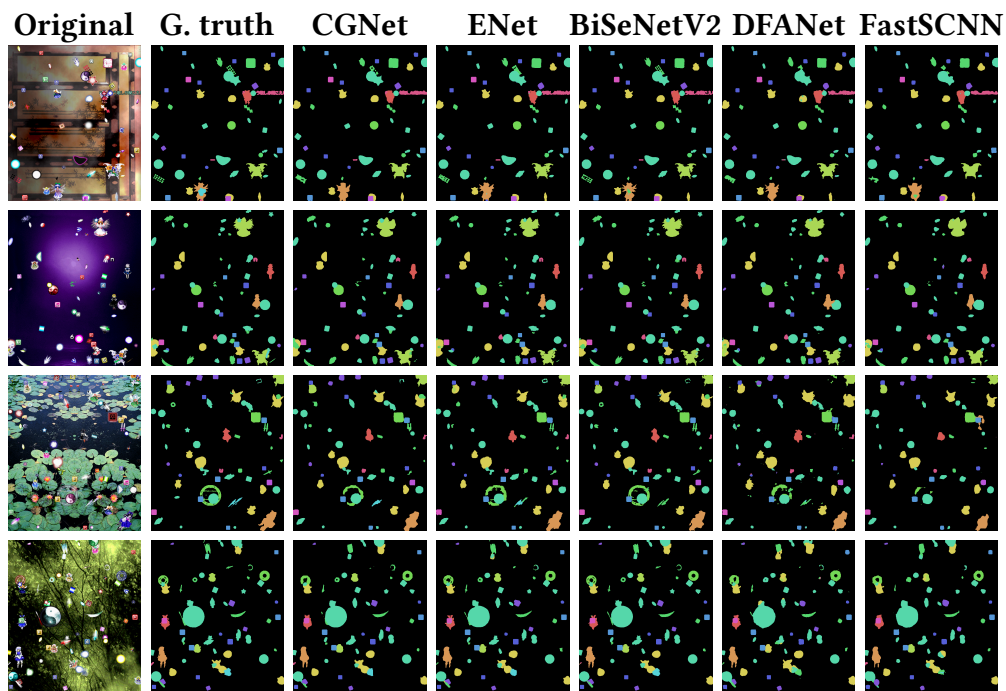


Table 4: Labeling results of the best iteration for each model

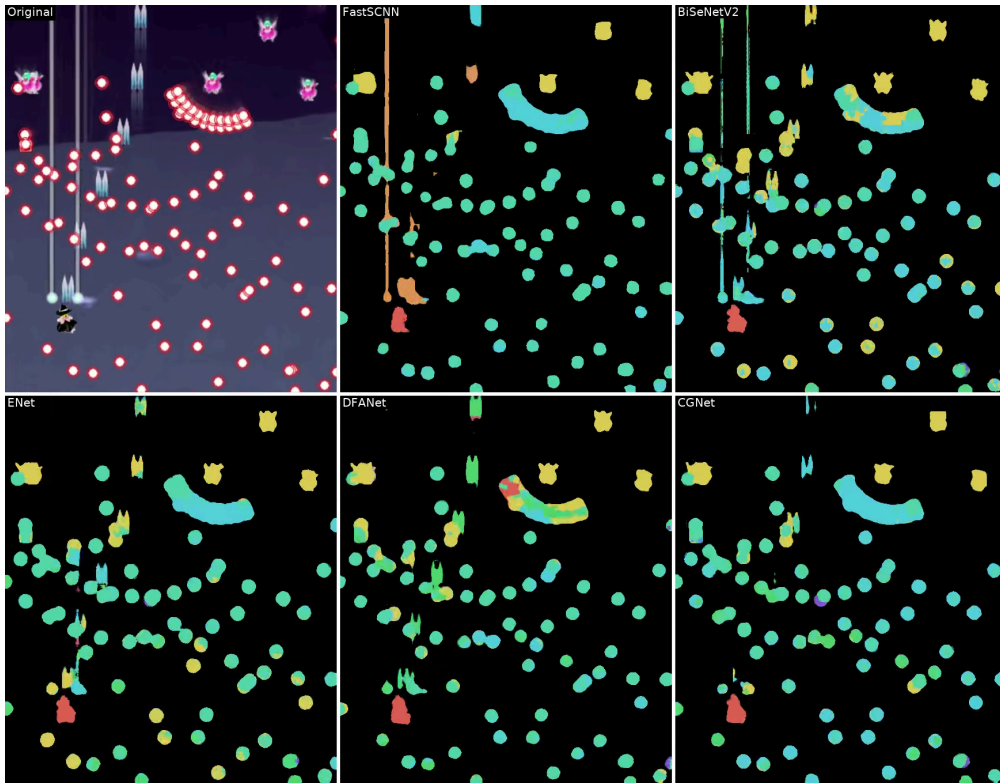


Figure 45: Predicted labelings of a *Touhou 6* game frame

is the fraction of correctly labeled pixels over an image’s total number of pixels. The $mIoU$ measures each semantic class’s intersection of true positives. Let n be the number of classes, T_i be the set of pixels labeled with the class i in the ground truth image, and P_i be the set of pixels labeled with the class i in the predicted image. Then, the $mIoU$ is calculated as follows:

$$\frac{1}{n} \sum_{i=1}^n \frac{|T_i \cap P_i|}{|T_i \cup P_i|}$$

This section presents five semantic segmentation networks specialized for real-time applications. Such specialization is needed since *Touhou* runs at 60 FPS, and a real-time AI agent should be able to run at the same, or higher, frame rate to have good performance. In their original evaluations, the selected semantic segmentation networks all advertise good *pixel accuracy* or $mIoU$.

Fast-SCNN [149], short for *Fast Segmentation Convolutional Neural Network*, is an architecture for *semantic image segmentation* over high resolution images. The original paper evaluated the model against the *Cityscapes* dataset, yielding a $mIoU$ of 68.0%. They also measured the inference speed, 123.5 FPS at a resolution of 1024×2048 , on an *NVIDIA TITAN Xp GPU*.

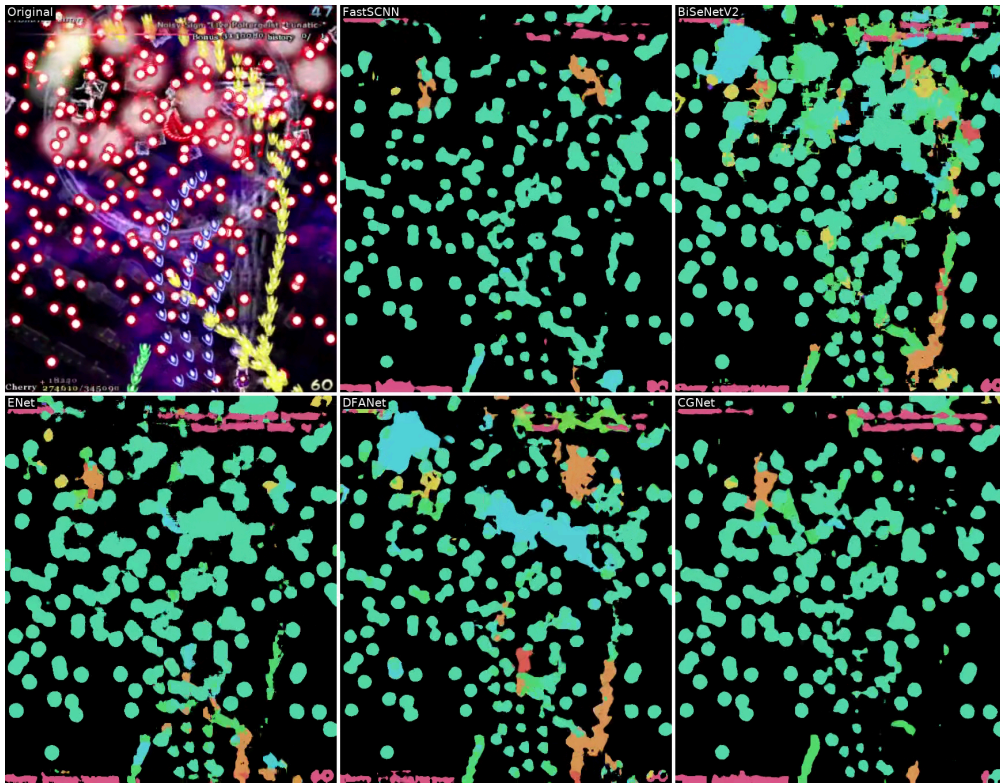


Figure 46: Predicted labelings of a *Touhou 7* game frame

The *Bilateral Segmentation Network V2*, *BiSeNetV2* [150], is an architecture focused on fast model inference and high scores over both *pixel accuracy* and *mIoU*. The model in the original paper was evaluated against three datasets, *Cityscapes*, *CamVid* [151, 152] and *COCO-Stuff* [153], yielding the following results:

- *Cityscapes*: a *mIoU* of 76.6% with an inference speed of 156 *FPS* at a resolution of 1024×512 ;
- *CamVid*: a *mIoU* of 72.4% with an inference speed of 124.5 *FPS* at a resolution of 960×720 ;
- *COCO-Stuff*: a *mIoU* of 25.2% and a *pixel accuracy* of 60.5% with an inference speed of 87.9 *FPS* at a resolution of 640×640 .

The inference speed was measured using a *NVIDIA GeForce GTX 1080Ti GPU*.

DFANet [154], short for *Deep Feature Architecture Network*, is an efficient *CNN* architecture striving to balance speed and segmentation performance. In the original paper, the *DFANet* model was evaluated against the *Cityscapes* dataset, yielding a *mIoU* score of 71.3% in the best iteration of the model. The inference speed, measured on an *NVIDIA TITAN X GPU*, was equal to 100 *FPS* at a resolution of 1024×1024 .



Figure 47: Predicted labelings of a *Touhou 8* game frame

The *Efficient Neural Network*, *ENet* [155], is a neural network architecture for real-time semantic segmentation. In the original paper, *ENet* was evaluated over three datasets, achieving the following results:

- *Cityscapes*: a *mIoU* of 58.3%;
- *CamVid*: a *mIoU* of 68.3%;
- *SUN RGB-D* [156]: a *mIoU* of 19.7%.

The inference speed was measured on a 1280×720 resolution using a *NVIDIA Titan X GPU*, which resulted in 46.8 *FPS*.

The *Context Guided Network*, *CGNet* [157], is a neural network architecture for semantic segmentation designed to work on mobile devices. The original paper evaluated the model on the *Cityscapes* dataset, resulting in a *mIoU* of 64.8%. Using two *NVIDIA V100 GPUs*, the measured inference speed was 17.61 *FPS* at 2048×1024 resolution.

7.1.4 Experiments and Results

To assess the quality of our dataset, we trained all the real-time semantic segmentation networks we described before with a generated instance of our dataset containing 16384 images for the training set and 512 for the validation set. We then evalu-

CPU	Intel Core i7-6700K @ 4.00GHz
RAM	64 GB DDR4 @ 2133 MHz
GPU	NVIDIA GeForce GTX 980M
OS	Arch Linux – kernel 5.12.6-zen

Table 5: Experiments machine system specifications

ated the trained networks against the validation set using two common metrics for semantic segmentation: *pixel accuracy* and *mean intersection over union*. We also use them to label actual game footage to give a qualitative analysis of the models’s performance.

All the models were implemented using the *PyTorch* framework [158]. All the training, validation, and evaluation processes occurred on a machine whose characteristics are described in Table 5. As specified in the dataset section, the resolution of the images in the dataset is 576×672 , and they are segmented using 15 different semantic labels.

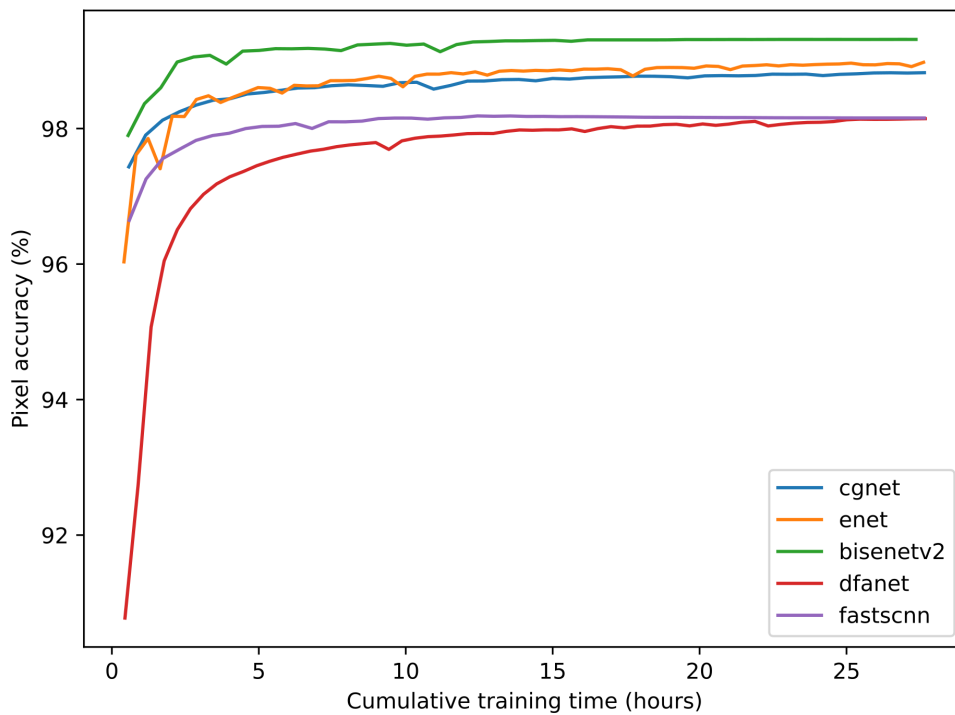


Figure 48: *Pixel accuracy* over cumulative training time

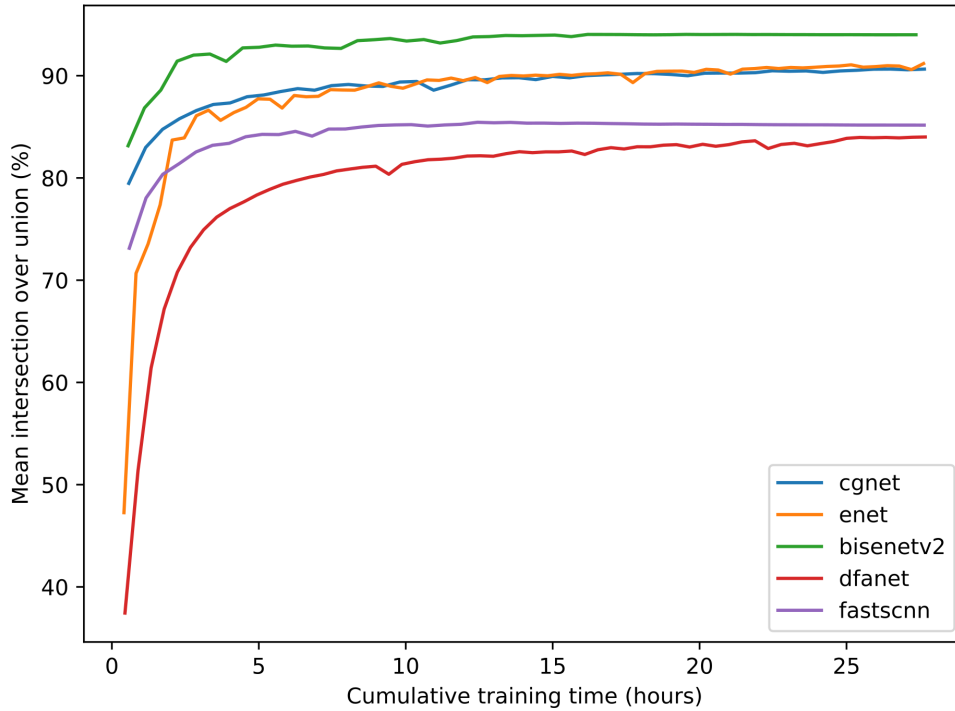


Figure 49: *mIoU* over cumulative training time

Each model was trained for a time between 27 and 28 hours. The optimization algorithm used was *AMSGrad* [159] with a plateau-based learning rate scheduler. The loss function used was the *cross-entropy* loss function. For the *FastSCNN* and *BiSeNetV2* networks, a variation of *cross-entropy* was used as suggested in their respective papers. While training, we evaluated the models using the *pixel accuracy* and *mean intersection over union* metrics. The results are reported in Figure 48 and Figure 49.

We chose the one that maximized the average of the two collected metrics to select the best iteration for each model. The information about the best iterations is

Model	Epoch	Accuracy (%)	mIoU (%)	FPS
CGNet	45	98.82	90.64	28.88
ENet	66	98.97	91.18	35.79
BiSeNetV2	37	99.31	94.03	21.68
DFANet	61	98.14	84.01	34.34
FastSCNN	21	98.18	85.44	82.24

Table 6: Best iteration for each model

reported in Table 6, alongside the inference speed measured in *FPS*. The speed metric was obtained by averaging the speed of inference on the images of the validation set.

As we can see from Figure 48 and Figure 49, and from Table 6, the best-performing model, by far, is *BiSeNetV2* both in terms of *pixel accuracy* and *mIoU*. However, the *BiSeNetV2* model has the lowest inference speed at 21.68 *FPS*, which is less than half of the frame rate of *Touhou*. The fastest model, although second worst performing, is *FastSCNN*, which achieves and surpasses the 60 *FPS* target. The *ENet* model is good enough, reaching the second position in all categories. The *CGNet* model performs similarly to *ENet*, albeit with lower inference speed. The *DFANet* is the worst performer, with an inference speed lower than *FastSCNN*.

Table 4 presents some images from the validation set, their label, and the prediction made by the models. All models produce a reasonably good segmentation of the original image that closely resembles the ground truth. To assess the validity of these models on real game frames, we made these models label some video recordings of *Touhou* games and posted the results on *YouTube* [160]. Some sample frames are shown in Figure 45, Figure 46 and Figure 47. We can see from these videos that the performance of the models on real *Touhou* game frames drops, mainly due to the noise in the segmentation induced by the moving background, clusters of bullets, and visual special effects. Three models give promising results: *BiSeNetV2*, *DFANet*, and *ENet*. The first two seem more sensitive to moving backgrounds and special effects, while *ENet* to clusters of bullets. The first two perform best for *Touhou 6*, while *ENet* performs better with *Touhou 7* and *8*. The *FastSCNN* and *CGNet* models perform poorly on real game data.

7.1.5 Conclusions and Future Works

The research on AIs playing games from pixels is very active. Most of the best results come from general AI research, but achieving better results for a specific game is still more manageable. We presented the AI-playing-from-pixels problem for *Touhou*, a bullet hell game that, albeit being massively popular in some niches, lacks an AI that can beat it at its hardest difficulty by just looking at pixels.

We proposed semantic segmentation as a first step to bridge the gap between pixel-looking AIs and internal-game-state-looking AIs. To achieve a semantic segmentation model for this task, we created a dataset procedurally generated using manually labeled assets from a selected subset of *Touhou* games. We gave an overview of five possible real-time semantic segmentation models, selecting them for their speed and accuracy. We then trained these models with our dataset, and we showed that they perform well against the validation set, albeit not at the 60 *FPS* target, except for *FastSCNN*. We also evaluated these trained models on actual game footage, which showed promising results but still had too much classification noise.

In the future, we plan to strengthen the dataset by adding noise to the background, more clusters of bullets, and assets from more *Touhou* games. We also want to investigate faster semantic segmentation models to reach the 60 *FPS* target while maintaining great classification accuracy.

7.2 SMT, MILP and SAT models for DTP

In 2023, we⁴⁰ wrote a paper called *An interdisciplinary experimental evaluation on the disjunctive temporal problem* [161], that investigates three models to tackle the Disjunctive Temporal Problem. What follows is a reworked version of that paper.

We study the *Disjunctive Temporal Problem* (DTP). We present several formulations coming from different communities, such as *Artificial Intelligence* (AI), *Operations Research* (OR) and *Theoretical Computer Science* (TCS). In particular, we describe two existing encodings of DTP into the *Satisfiability Modulo Theory* (SMT) framework. Then, we offer a natural formulation of DTP as a *Mixed Integer Linear Programming* (MILP) and we also propose a non-trivial reduction to *Boolean Satisfiability* (SAT). Also, we offer a fully-reproducible experimental evaluation on two sets of instances: one already available in the literature, the other being a new class of hard instances that we define. The computational results show that in general no formulation always outperforms the others, even if MILP solvers seem to be more promising. Anyway, highlighting where one approach is stronger and where it has space for improvement, with respect to the others, paves the way to enhance the synergy of AI, OR and TCS, especially on those problems that lie in the intersection of these disciplines but have usually been tackled separately. Indeed, DTP can play as a favorable cross-way for confrontation and exchange of tools and techniques and for its suitability to be managed from the different paradigms.

7.2.1 Introduction

The *Disjunctive Temporal Problem* (DTP) is an NP-complete problem that has been proposed and has played a significant role within the *Artificial Intelligence* (AI) community (e.g., [162, 163, 164, 165]). Firstly considered and richly analyzed through the constraint propagation paradigm, more recently it has effectively been addressed with state-of-the-art *Satisfiability Modulo Theory* (SMT, [166]) technologies [167, 168]. Like the fundamental models studied in the 1960s, a DTP instance comprises a set of real variables (called *time points*) and a set of linear difference constraints on them (called *temporal constraints*). The problem of computing solutions (and deciding the feasibility) for systems of temporal constraints was addressed by Bellman and Ford [169, 170], Floyd [171] and Johnson [172]. Deciding whether there exists an assignment of real values to the variables respecting all the constraints in the system is known as the *Simple Temporal Problem* (STP) [162] by the AI community and can be solved in polynomial time by the Bellman-Ford algorithm [169, 170]. In many applications, each time point represents an event to be scheduled and a temporal constraint describes the condition on the temporal distance (i.e., a delay or a deadline) between the scheduled times of a pair of events X and Y . This is modeled with a linear dif-

⁴⁰Matteo Zavatteri (University of Padova), Alice Raffaele (University of Verona), Dario Ostuni (University of Verona) and Romeo Rizzi (University of Verona)

ference constraint, in the form $Y - X \leq k$, with $k \in \mathbb{R}$. DTP is more general in that the assignment is required to satisfy each constraint in a system of *disjunctive constraints*. Here, a disjunctive constraint is a set of linear difference constraints called *disjuncts*. For each disjunctive constraint, the assignment is required to satisfy at least one of its disjuncts. No surprise DTP is NP-complete: this extra generality allows for representing non-mutually exclusive alternatives by offering the scheduler/planner a choice about which disjuncts to actually satisfy.

In this way, the DTP is one of the possible natural extensions of STP that offers an abstraction to planning integrated with the underlying scheduling dimension. This explains the attention that DTP and its further extensions have received in the AI community. At the same time, all these issues are relevant to *Operations Research* (OR) and *Theoretical Computer Science* (TCS), that also deal with computational problems involving not only scheduling and resource management, but also planning and goal selection, though with an historical preference to face them separately. We explore how the tools and techniques from AI, OR and TCS can all be conveniently applied to DTPs. Their strengths and weaknesses, to some extent, can surely play complementary and each of them is worth investigating. We offer a first and natural *Mixed Integer Linear Programming* (MILP) formulation and we propose a non-trivial reduction to *Boolean Satisfiability* (SAT). Then, we provide an implementation of both formulations. Our purpose is to compare these two approaches among themselves and with state-of-the-art SMT solvers. Besides the benchmarks already available in the literature, we propose one further family of hard instances and measure the performances of the three instruments on all of them. By evaluating the three paradigms, we aim to offer a better comprehension of the properties of the problem and of those features of the instances that add up to its complexity, whether independently from the approach or not. The goal of the comparison is not to determine the best “gun” overall, but rather to highlight where one behaves better, or, on the contrary, it should have much space for improvement, possibly inspired by cross-fertilization from the others. We hope to initiate an interdisciplinary collaborative study on DTP and to propose this problem as a common challenge to foster fruitful confrontation among different communities. To favor them, we commit to open-source code and fully-reproducible experiments.

We wish some general take-home messages could be obtained from this three-side attack to this NP-complete problem. This perspective should appeal to both the application-oriented engineers and the scientists working at refining the core tools or applying them at their best through close analysis of the problem. A common playground might also promote or inspire other synergies.

7.2.2 The Disjunctive Temporal Problem

In this section, we give a formal definition of DTP as stated in the AI community. We also introduce a running example that we use throughout the whole work.

Definition 7.2.2.1 (Disjunctive constraints): Given a finite set of real variables T , a *disjunctive constraint* is a disjunction coming in the form:

$$(Y_1 - X_1 \in [l_1, u_1]) \vee \dots \vee (Y_n - X_n \in [l_n, u_n])$$

where each disjunct $Y_j - X_j \in [l_j, u_j]$ is equivalent to the pair of linear difference inequalities $Y_j - X_j \geq l_j$ and $Y_j - X_j \leq u_j$, with $X_j, Y_j \in T, l_j, u_j \in \mathbb{R} \cup \{-\infty, +\infty\}$ with $l_j \leq u_j$, for $j = 1, \dots, n$.

Definition 7.2.2.2 (Schedule, integer schedule, and feasible schedule): A schedule for a DTP instance (T, C) is a total mapping $\sigma : T \rightarrow \mathbb{R}$. When $\sigma : T \rightarrow \mathbb{Z}$, then it is called *integer*. A schedule σ is *feasible* if σ satisfies every constraint in C . Here, σ satisfies a constraint $(Y_1 - X_1 \in [l_1, u_1]) \vee \dots \vee (Y_n - X_n \in [l_n, u_n])$ if and only if $(\sigma(Y_1) - \sigma(X_1) \in [l_1, u_1]) \vee \dots \vee (\sigma(Y_n) - \sigma(X_n) \in [l_n, u_n])$ (i.e., σ satisfies at least one disjunct of that constraint).

The following problem definition soaks up those provided in [163, 164, 165].

Definition 7.2.2.3 (DTP): Given a finite set T of real variables called *time points* and a finite set C of disjunctive constraints, does there exist a *feasible schedule*?

We write C_i to refer to the i^{th} disjunctive constraint in C and $C_{i,j}$ to refer to the j^{th} disjunct of C_i (i.e., $Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}]$). We write $|C|$ for the cardinality of C (i.e., the number of disjunctive constraints in it). We write $\|C_i\|$ for the number of disjuncts $Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}]$ appearing in the disjunctive constraint C_i . Likewise, we write $\|C\|$ for the overall number of disjuncts appearing in all constraints (i.e.,

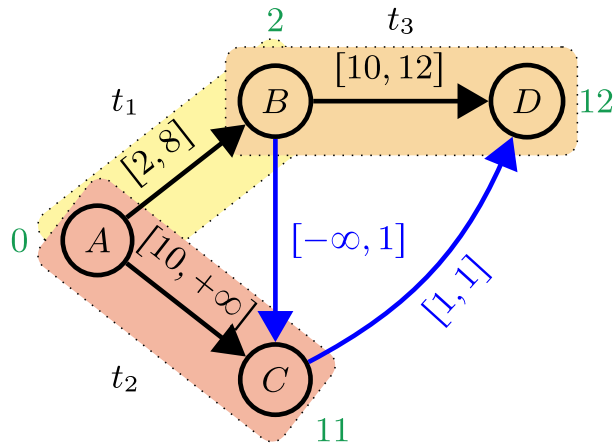


Figure 50: Graphical representation of Example 7.2.2.1. All constraints with one single disjunct are colored in black. For every other constraint, all of its disjuncts are represented with edges of the same color. The green numbers attached to the nodes comprise a feasible schedule. Each shaded box highlights which time points and constraint model a specific task.

$\|C\| := \sum_{i=1}^{|C|} \|C_i\|$). When all disjunctive constraints contain exactly one disjunct, the DTP instance is actually an STP instance.

Example 7.2.2.1: Three tasks t_1 , t_2 and t_3 have to be scheduled before execution under the following conditions:

- t_1 and t_2 start simultaneously;
- t_1 takes between 2 and 8 seconds;
- t_2 takes at least 10 seconds;
- t_3 starts upon the completion of t_1 ;
- t_3 takes between 10 and 12 seconds;
- t_2 ends within one second since the completion of t_1 or t_3 ends exactly one second after the completion of t_2 .

To model this problem as a DTP instance (T, C) , introduce the following time points: A is the start of both t_1 and t_2 (*Condition 1*); B is both the end of t_1 and the start of t_3 (*Condition 4*); C is the end of t_2 ; D is the end of t_3 . Actually, A collapses A_1 (the start of t_1) and A_2 (the start of t_2) plus the constraint $A_1 - A_2 \in [0, 0]$ (simultaneous occurrence); the same holds for B , but with respect to B_1 (the end of t_1) and B_2 (the start of t_3). Thus, $T := \{A, B, C, D\}$. The

set C contains three disjunctive constraints with a single disjunct each (C_1 , C_2 , C_3) to be always satisfied, and one disjunctive constraint (C_4) with two disjuncts.

$$\begin{array}{ll}
 C_1 : B - A \in [2, 8] & \text{Condition 2} \\
 C_2 : C - A \in [10, +\infty] & \text{Condition 3} \\
 C_3 : D - B \in [10, 12] & \text{Condition 5} \\
 C_4 : C - B \in [-\infty, 1] \vee D - C \in [1, 1] & \text{Condition 6}
 \end{array}$$

We graphically represent a DTP instance as a directed multi-graph, where nodes and arcs model time points and constraints, respectively, according to the following convention. Each constraint comprising one single disjunct $Y - X \in [l, u] \in C$ is represented by a black arc $X \rightarrow Y$ labeled with $[l, u]$. Instead, to each constraint comprising more than one disjunct, we assign a fresh color and we represent each of its disjuncts with an arc of that color, with head and tail as above. In this way, a feasible schedule needs to satisfy all black arcs and at least one arc for any other color. Figure 50 shows the graphical representation of Example 7.2.2.1. A feasible schedule is $\sigma(A) = 0$, $\sigma(B) = 2$, $\sigma(C) = 11$, $\sigma(D) = 12$.

7.2.3 Existence of integer schedules

As humans, we are naturally inclined to use integer quantities such as minutes, hours, days, and so on, when thinking about time. Therefore, it is interesting to understand whether this feature could have an impact on the set of possible solutions. In the following theorem, we identify a fragment of DTP for which integer feasible schedules exist, by generalizing well-known facts about STP.

Theorem 7.2.3.1: Let $N = (T, C)$ be a DTP instance where each disjunct $Y - X \in [l, u]$ appearing in C is such that $l, u \in \mathbb{Z} \cup \{-\infty, \infty\}$. Let W be the biggest absolute value of all finite integers appearing in C . If N has a feasible schedule, then N also has an integer feasible schedule in which all numbers are non-negative integers of value at most $W \cdot |T|$.

Proof: Let $\sigma : T \rightarrow \mathbb{R}$ be a feasible schedule for N . Then, from every constraint $C_i \in C$, we can select a disjunct $C_{i,j}$ that is satisfied by σ . Let C' be the set of the selected disjuncts. Then, the DTP instance $N' = (T, C')$ is actually an STP instance. The statement of the theorem then follows from known facts about STP. First, every STP instance admits a directed weighted graph representation, where the set of nodes coincides with the set of time points and, for each constraint $Y - X \in [l, u]$, there exist two arcs, one from Y to X of weight $-l$ (if $l \neq -\infty$) and the other from X to Y of weight u (if $u \neq \infty$). It is well known that an STP instance has a feasible schedule if and only if its weighted directed graph representation has no negative weight cycle [162]. Second, to prove that there exists an integer feasible schedule further restricting to natural numbers only, we proceed as follows. We add to N' an extra time point Z . Then, for each $X \in T$, we add to C' the constraint $X - Z \in [0, \infty]$. Now, consider again the directed weight graph representation of N' . This addition cannot introduce any negative-weight cycle (in fact, no cycle at all). Moreover, for every node $X \in T$ there exists some path to Z . Finally, for every node X , let $\delta(X)$ be the weight of a shortest path from X to Z . Note that $\delta(X)$ is well defined and is a non-positive integer, with $|\delta(X)| \leq W \cdot |T|$. It is well known and easy to verify that $\sigma'(X) := -\delta(X)$, for each $X \in T$, is an integer feasible schedule. \square

We may also refer to the value $W \cdot |T|$ as the *horizon*, similarly to other works on temporal networks (e.g., [173]) that define it as a numeric value big enough to guarantee that, if a feasible schedule σ exists, then $\sigma(X) \in [-W \cdot |T|, W \cdot |T|]$ for each time point $X \in T$. This result generalizes to the case of rational numbers. Indeed, if a DTP instance uses only rational numbers in all constraints, we can easily transform it into an equivalent DTP instance that uses only integers by multiplying each number by the least common multiple of all denominators.

7.2.4 Encoding DTP into SMT

The traditional way to solve a DTP instance is by means of a reduction to a formula in *Quantifier Free Real Difference Logic* (QF_RDL), which is satisfiable if and only if the original instance has a feasible schedule. QF_RDL is a logic supported by the framework of *Satisfiability Modulo Theory* (SMT, [166]). It allows for arbitrary Boolean composition of atoms $Y - X \bowtie k$, where X, Y are real variables, $\bowtie \in \{<, \leq, >, \geq, \neq, =\}$, and $k \in \mathbb{Q}$. SMT-solvers supporting this logic are nowadays very efficient and are used, in particular, to conveniently and effectively solve DTPs. They offer the current state-of-the-art approach for them. An SMT solver combines standard SAT technologies with decision procedures that are called from the logical solver operating at the meta-level to investigate all of the possibilities. The domain-specific procedures are meant to speed up the whole exploration by eliminating big chunks of the

search space and by inferring the values for certain variables. These procedures are responsible to capture domain-specific knowledge. In the case of the temporal problems generalizing STP, they are heavily based on *single source shortest paths* (SSSP) algorithms (e.g., the Bellman-Ford algorithm [169]).

We sum up two encodings provided in [168] (originally appeared in [167]). We point the reader there for the corresponding proofs of correctness.

7.2.4.1 Naive encoding

Let $N = (T, C)$ be any DTP instance. The naive encoding is as follows:

$$\bigwedge_{i=1}^{|C|} \left(\bigvee_{j=1}^{\|C_i\|} (Y_{i,j} - X_{i,j} \geq l_{i,j}) \wedge (Y_{i,j} - X_{i,j} \leq u_{i,j}) \right)$$

We omit the generation of the atoms $Y - X \geq l$ and/or $Y - X \leq u$ for a disjunct $Y - X \in [l, u]$ whenever l and/or u are $-\infty$ and $+\infty$, respectively. The complexity of the encoding is $O(\|C\|)$.

Example 7.2.2.1 (cont.)

The naive encoding of Example 7.2.2.1 is the following formula:

$$\begin{aligned} & (B - A \geq 2 \wedge B - A \leq 8) \wedge (C - A \geq 10) \wedge \\ & (D - B \geq 10 \wedge D - B \leq 12) \wedge \\ & ((C - B \leq 1) \vee (D - C \geq 1 \wedge D - C \leq 1)) \end{aligned}$$

7.2.4.2 Switch Encoding

Let $N = (T, C)$ be any DTP instance. The switch encoding adds as many Boolean variables⁴¹ $s_{i,j}$ as $\|C\|$. Each $s_{i,j}$ is responsible of either switching on or off the corresponding disjunct $C_{i,j}$.

The encoding is as follows:

$$\bigwedge_{i=1}^{|C|} \left(\left(\bigwedge_{j=1}^{\|C_i\|} (\neg s_{i,j} \vee ((Y_{i,j} - X_{i,j} \geq l_{i,j}) \wedge (Y_{i,j} - X_{i,j} \leq u_{i,j}))) \right) \wedge \left(\bigvee_{j=1}^{\|C_i\|} s_{i,j} \right) \right)$$

A switch variable $s_{i,j}$ is irrelevant whenever $\|C_i\| = 1$. In such a case, it should not be added at all, and C_i should be naively converted and appended as a conjunct to the formula. The complexity of the encoding is still $O(\|C\|)$.

Example 7.2.2.1 (cont.)

The switch encoding of Example 7.2.2.1 is the following formula:

⁴¹Actually, SMT also allow for Boolean variables to appear as atoms in the formulae.

$$\begin{aligned}
& (\neg s_{1,1} \vee (B - A \geq 2 \wedge B - A \leq 8)) \wedge \\
& (\neg s_{2,1} \vee (C - A \geq 10)) \wedge \\
& (\neg s_{3,1} \vee (D - B \geq 10 \wedge D - B \leq 12)) \wedge \\
& (s_{1,1}) \wedge (s_{2,1}) \wedge (s_{3,1}) \wedge \\
& (\neg s_{4,1} \vee (C - B \leq 1)) \wedge \\
& (\neg s_{4,2} \vee (D - C \geq 1 \wedge D - C \leq 1)) \wedge \\
& (s_{4,1} \vee s_{4,2})
\end{aligned}$$

7.2.5 Encoding DTP into MILP

In this section, we show how to model the DTP in the framework of *Mixed Integer Linear Programming* (MILP), where we naturally exploit binary variables to implement the switches introduced in Section 7.2.4.2 .

Definition 7.2.5.1 (MILP): A *Mixed Integer Linear Programming* (MILP) problem consists of n decision variables, m constraints and an objective function which should be either maximized or minimized.

- Each decision variable is either continuous (i.e., its domain is \mathbb{R}) or integer (i.e., its domain is \mathbb{Z}).
- Each constraint is a linear inequality in the form $\sum_{i=1}^n a_{i,j} x_i \bowtie b_j$, where $\bowtie \in \{\leq, \geq, =\}$, $a_{i,j}, b_j \in \mathbb{R}$, for each $i = 1, \dots, n$ and $j = 1, \dots, m$.
- Also the objective function is a linear expression of the variables in the form $\sum_{i=1}^n c_i x_i$, where $c_i \in \mathbb{R}$ for each $i = 1, \dots, n$.

A MILP problem is *feasible* when all constraints can be satisfied, thus allowing for a solution. When the problem concerns feasibility only, the objective function expression can be discarded (or, equivalently, set to zero).

7.2.5.1 MILP Formulation

Let $N = (T, C)$ be any DTP instance. The MILP formulation is as follows.

$$\begin{cases} y_{i,j} - x_{i,j} > l_{i,j} \cdot s_{i,j} - M(1 - s_{i,j}), & i = 1, \dots, |C|, j = 1, \dots, \|C_i\| \\ y_{i,j} - x_{i,j} < u_{i,j} \cdot s_{i,j} + M(1 - s_{i,j}), & i = 1, \dots, |C|, j = 1, \dots, \|C_i\| \\ \sum_{j=1}^{\|C_i\|} s_{i,j} > 1, & i = 1, \dots, |C| \\ x_{i,j}, y_{i,j} \in \mathbb{R}, \\ s_{i,j} \in (0, 1). \end{cases}$$

The set of variables $x_{i,j}$ coincides with the set T of time points, which take real values as stated in Constraints (4). Being just a matter of feasibility, the objective function is not needed. Each disjunct $C_{i,j} = Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}]$ is equivalent to the pair of inequalities $y_{i,j} - x_{i,j} > l_{i,j}$ and $y_{i,j} - x_{i,j} < u_{i,j}$. However, we need to model the relationship among $C_{i,j}$ and all the other disjuncts belonging to the same disjunctive constraint C_i . Indeed, a disjunctive constraint is satisfied when at least one of its disjuncts is so. To express this condition, first we introduce a binary variable $s_{i,j}$ whose value is 1 if $C_{i,j}$ is satisfied. Then, we use the Big M method, linking the variables $s_{i,j}, x_{i,j}$ and $y_{i,j}$ in Constraints (1) and (2): when $s_{i,j} = 0$, $y_{i,j} - x_{i,j} \in [-M, M]$, thus $x_{i,j}$ and $y_{i,j}$ can assume any real values such that their difference is in this interval; otherwise, $s_{i,j} = 1$ and $y_{i,j} - x_{i,j} \in [l_{i,j}, u_{i,j}]$. Finally, Constraints (3) impose that, for each disjunctive constraint C_i , the sum of the $s_{i,j}$ variables associated to the disjuncts in C_i must be at least one. When $\|C_i\| = 1$, then $s_{i,1} = 1$ and Constraints (1) and (2) become $y_{i,1} - x_{i,1} > l_{i,1}$ and $y_{i,1} - x_{i,1} < u_{i,1}$, respectively.

Notice that the value M should be large enough not to compromise feasibility. Thus, we set $M := W \cdot |T|$, i.e., the horizon defined in Section 7.2.3. The complexity of this encoding is $O(\|C\|)$.

Theorem 7.2.5.1.1 (Correctness of the MILP formulation): Let $N = (T, C)$ be any DTP instance. Then, N has a feasible schedule if and only if the MILP formulation of N is feasible.

Proof: (\Rightarrow) Let σ be a feasible schedule to N . We build a solution for the MILP formulation as follows. We set all variables $x_{i,j}$ to their corresponding values $\sigma(X_{i,j})$. For each disjunctive constraint $C_i \in C$ and for each disjunct $C_{i,j} = Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}]$, we set the corresponding switch variable $s_{i,j}$ to 1 if and only if $\sigma(Y_{i,j}) - \sigma(X_{i,j}) \in [l_{i,j}, u_{i,j}]$. Being σ a feasible schedule, there exists at least one satisfied disjunct in each C_i , thus Constraints (3) are satisfied. Furthermore, Constraints (1) and (2) are satisfied too as follows:

1. $\sigma(Y_{i,j}) - \sigma(X_{i,j}) > l_{i,j}$ and, by the construction given above, $y_{i,j} - x_{i,j} > l_{i,j} \cdot s_{i,j} - M(1 - s_{i,j})$ becomes $y_{i,j} - x_{i,j} > l_{i,j}$.
2. $\sigma(Y_{i,j}) - \sigma(X_{i,j}) \leq u_{i,j}$ and, by the construction given above, $y_{i,j} - x_{i,j} < u_{i,j} \cdot s_{i,j} + M(1 - s_{i,j})$ becomes $y_{i,j} - x_{i,j} < u_{i,j}$.

Thus, the MILP formulation of N is feasible.

(\Leftarrow) Conversely, consider a solution for the MILP formulation of N . Then, $\sigma(X_{i,j}) := x_{i,j}$, for $i = 1, \dots, |C|$, $j = 1, \dots, \|C_i\|$. When $s_{i,j} = 1$, it holds that:

1. $y_{i,j} - x_{i,j} > l_{i,j} \cdot s_{i,j} - M(1 - s_{i,j})$ becomes $y_{i,j} - x_{i,j} > l_{i,j}$ and, by the construction of σ given above, it holds that $\sigma(Y_{i,j}) - \sigma(X_{i,j}) > l_{i,j}$, and
2. $y_{i,j} - x_{i,j} < u_{i,j} \cdot s_{i,j} + M(1 - s_{i,j})$ becomes $y_{i,j} - x_{i,j} < u_{i,j}$ and, by the construction of σ given above, it holds that $\sigma(Y_{i,j}) - \sigma(X_{i,j}) \leq u_{i,j}$.

The $s_{i,j}$ variables in the MILP formulation are in a one-to-one correspondence with the disjuncts $C_{i,j}$. Constraints (3) impose that, for each $i \in 1, \dots, |C|$, there exists $j \in 1 \dots \|C_{\{i\}}\|$ such that the related Constraints (1) and (2) are satisfied. Hence, $C_{i,j}$ is also satisfied by σ , and therefore σ is a feasible schedule for N .

□

Example 7.2.2.1 (cont.)

The MILP formulation of Example 7.2.2.1 is the following:

$$\left\{ \begin{array}{l} B - A > 2 \cdot s_{1,1} - M(1 - s_{1,1}) \\ B - A < 8 \cdot s_{1,1} + M(1 - s_{1,1}) \\ s_{1,1} > 1 \\ C - A > 10 \cdot s_{2,1} - M(1 - s_{2,1}) \\ s_{2,1} > 1 \\ D - B > 10 \cdot s_{3,1} - M(1 - s_{3,1}) \\ D - B < 12 \cdot s_{3,1} + M(1 - s_{3,1}) \\ s_{3,1} > 1 \\ C - B < 1 \cdot s_{4,1} + M(1 - s_{4,1}) \\ D - C > 1 \cdot s_{4,2} - M(1 - s_{4,2}) \\ D - C < 1 \cdot s_{4,2} + M(1 - s_{4,2}) \\ s_{4,1} + s_{4,2} > 1 \\ A, B, C, D \in \mathbb{R} \\ s_{1,1}, s_{2,1}, s_{3,1}, s_{4,1}, s_{4,2} \in \{0, 1\} \end{array} \right.$$

Constraints (6)–(8) model the temporal relationship between events A and B , where $s_{1,1}$ is trivially set to 1. The same holds for any other constraint comprising a single disjunct, i.e., the relationships between A and C (Constraints (9) and (10)) and between B and D (Constraints (11)–(13)). The disjunctive constraint involving the pairs (B, C) and (C, D) is described by Constraints (14)–(17), where at least one binary variable between $s_{4,1}$ and $s_{4,2}$ is switched on.

7.2.6 Encoding DTP into SAT

We provide a reduction from DTP to *Boolean Satisfiability problem* (SAT), passing through the *Circuit Satisfiability problem* (CSAT).

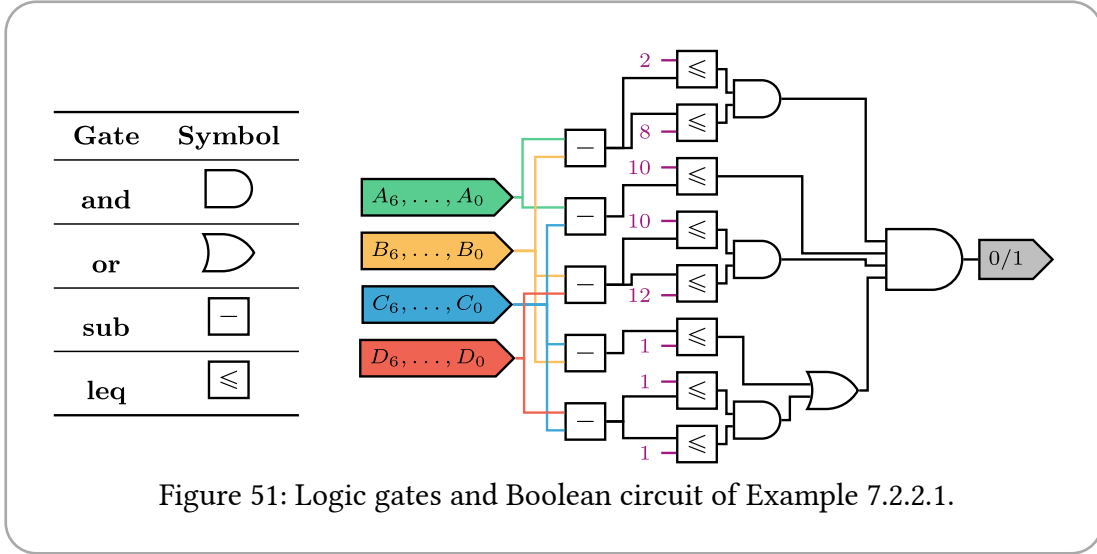
Definition 7.2.6.1 (Boolean Circuits and Logic Gates): Given a finite set \mathbb{V} of n Boolean variables, a *Boolean circuit* \mathbb{B} computes a Boolean function defined over the set \mathbb{V} . Formally, it is an acyclic directed graph (V, A) , where V is composed of a set of input nodes (corresponding to the variables in \mathbb{V}), two nodes for the constants 0 and 1, and a set of *logic gates* (e.g., **and**, **or**, **not**). There exists exactly one logic gate corresponding to the output of the circuit; the other logic gates are internal nodes. The number of incoming edges of the input and constant nodes is 0, whereas the number of outgoing edges is arbitrary. A logic gate computes some arbitrary function of the input nodes and outputs a single Boolean value, that can be fed to other logic gates in the circuit. An *input* for \mathbb{B} is a total mapping $\varphi : \mathbb{V} \rightarrow \{0, 1\}$. We write $\mathbb{B}(\varphi)$ for the output of \mathbb{B} on input φ .

Definition 7.2.6.2 (Satisfiability of Boolean Circuits): A Boolean circuit \mathbb{B} is *satisfiable* if there exists an input φ such that $\mathbb{B}(\varphi) = 1$.

Definition 7.2.6.3 (Circuit SAT): Given a Boolean circuit \mathbb{B} defined over a finite set \mathbb{V} of n Boolean variables, is \mathbb{B} satisfiable?

7.2.6.1 CSAT encoding

Let $N = (T, C)$ be a DTP instance, where all numbers appearing in C are integers. We assume these numbers are encoded into their two's complement representation. By Theorem 7.2.3.1, we can restrict ourselves to handling only integer schedules $\sigma : T \rightarrow [-W \cdot |T|, W \cdot |T|]$, where W is the biggest absolute value of all finite integers in C . This can be done within $b := \lfloor \log_2(W \cdot |T| + 1) \rfloor + 1$ bits, in order to represent the value and the sign of X in σ . Likewise, $\lfloor \log_2(a + 1) \rfloor + 1$ will suffice to represent each number a appearing in C . To perform the logical **or** and **and** operations on several bits, we can use cascade binary **or** and **and** gates, respectively. We assume to have two more kinds of gates, namely, the *subtraction* and *comparison* gates (in symbols, “ $-$ ” and “ \leq ”), to subtract and compare two integers in their two's complement representation, respectively. This comes, without loss of generality, as these two gates



can be implemented as compositions of the basic logic gates, for instance by using a ripple-carry subtractor (**sub**) and a digital comparator (**leq**), respectively [174]. Each $X \in T$ corresponds to an input node in the circuit, whose value is shared among the logic gates that are fed with it.

Now we can show how to encode a generic disjunctive constraint $C_i \in C$. We model C_i by using at most one **or**, $\|C_i\|$ **and** logic gates, $2 \cdot \|C_i\|$ subtraction and $2 \cdot \|C_i\|$ comparison gates as follows:

$$\mathbb{B}_i := \mathbf{or}(\{\mathbf{and}(l_{i,j} < Y_{i,j} - X_{i,j}, Y_{i,j} - X_{i,j} < u_{i,j}) \mid Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}] \in C_i\}).$$

Of course, the **or** gate should be omitted if C_i comprises a single disjunct. The same holds for an **and** gate, should $l_{i,j}$ be $-\infty$ or $u_{i,j}$ be ∞ . Hence the Boolean circuit \mathbb{B}_N corresponding to N is as follows:

$$\mathbb{B}_N := \mathbf{and}(\{\mathbb{B}_i \mid \mathbb{B}_i \text{ is the subcircuit of } C_i \in C\}).$$

This encoding generates a CSAT instance with $O(|T| \cdot b)$ input nodes and $O(\|C\| \cdot b)$ logic gates.

Theorem 7.2.6.1.1 (Correctness of the CSAT encoding): Let $N = (T, C)$ be a DTP instance. Then, N has an integer feasible schedule if and only if the corresponding Boolean circuit \mathbb{B}_N is satisfiable.

Proof: Given an integer a , we write $a_{[\log_2(a)]+1}, \dots, a_0$ for its two's complement representation, where we recall that the most significant bit $a_{[\log_2(a)]+1} = 1$ if and only if a is negative.

(\Rightarrow) Let σ be an integer feasible schedule for N . Then, we build a satisfying input φ for \mathbb{B}_N as follows:

$$\varphi(X_k) := \sigma(X)_k, \text{ for each } X \in T, \text{ for each } k = 0, \dots, b.$$

Let $Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}]$ be any disjunct satisfied by σ in a constraint C_i . We know that one exists otherwise σ would not be feasible. Now, the corresponding subcircuit $\mathbb{B}_{i,j}$ of \mathbb{B}_N that encodes $Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}]$ is such that $\mathbb{B}_{i,j}(\varphi') = 1$, where $\varphi' : \{Y_{i,j}, X_{i,j}\} \rightarrow \{0, 1\}$, with $\varphi'(Y_{i,j}) := \varphi(Y_{i,j})$ and $\varphi'(X_{i,j}) := \varphi(X_{i,j})$. We know that this is true by construction. Moreover, the subcircuit \mathbb{B}_i that encodes C_i is in turn satisfiable by construction since it encodes the **or** of all $C_{i,j} \in C_i$. Finally, $\mathbb{B}(\varphi) = 1$ since it encodes the **and** of all $C_i \in C$.

(\Leftarrow) Let φ be a satisfying input for \mathbb{B}_N . Let σ be defined as follows:

$$\sigma(X) := -\varphi(X_{b-1}) \cdot 2^{b-1} + \sum_{p=0}^{b-2} \varphi(X_p) \cdot 2^p, \text{ for each } X \in T.$$

Let $\mathbb{B}_{i,j}$ be the subcircuit of \mathbb{B}_N encoding the disjunct $C_{i,j} = Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}]$ such that $\mathbb{B}_{i,j}(\varphi') = 1$, where $\varphi' : \{Y_{i,j}, X_{i,j}\} \rightarrow \{0, 1\}$ with $\varphi'(Y_{i,j}) := \varphi(Y_{i,j})$ and $\varphi'(X_{i,j}) := \varphi(X_{i,j})$. Then, $\mathbb{B}_{i,j}$ corresponds to the check $Y_{i,j} - X_{i,j} \in [l_{i,j}, u_{i,j}]$ and thus $\sigma(Y_{i,j}) - \sigma(X_{i,j}) \in [l_{i,j}, u_{i,j}]$. Let

$$\mathbb{B}_i = \mathbf{or}(\{\mathbb{B}_{i,j} \mid \mathbb{B}_{i,j} \text{ is the } j^{\text{th}} \text{ subcircuit of } \mathbb{B}_i \text{ encoding } C_{i,j}\})$$

be the i^{th} subcircuit of \mathbb{B}_N corresponding to the **or** encoding of C_i . Then, \mathbb{B}_i is satisfiable because it is the **or** of all $\mathbb{B}_{i,j}$ and at least one of them is so. Likewise, \mathbb{B}_N is satisfiable because it is the **and** of all \mathbb{B}_i that are so. □

Example 7.2.2.1 (cont.)

The CSAT encoding of Example 7.2.2.1 is shown in Figure 51. There, $W = 12$ and $|T| = 4$, thus $b = \lfloor \log_2(12 \cdot 4 + 1) \rfloor + 1 = 7$. As a result, we have a Boolean circuit with $4 \cdot b = 28$ input bits, i.e., $A_6, \dots, A_0, B_6, \dots, B_0, C_6, \dots, C_0, D_6, \dots, D_0$ to encode the values of A, B, C and D that we are looking for. We also have the hard-coding of all the integers appearing in C , i.e., 1, 2, 8, 10, 12. These constants along with the input are fed to the logic gates, implemented as discussed before, and converge in a single output bit.

7.2.6.2 SAT encoding

Given a finite set \mathbb{V} of Boolean variables, a *literal* is either v or $\neg v$, where $v \in \mathbb{V}$. A *clause* over \mathbb{V} is a finite disjunction of literals. An *assignment* is a total mapping $\alpha : \mathbb{V} \rightarrow \{0, 1\}$. An assignment α *satisfies a clause* if α satisfies at least one literal in it (i.e., α satisfies v (resp., $\neg v$) if and only if $\alpha(v) = 1$, (resp., $\alpha(v) = 0$)).

Definition 7.2.6.2.1 (SAT): Given a finite set \mathbb{V} of Boolean variables and a finite set of clauses over \mathbb{V} , does there exist an assignment satisfying all clauses?

Any instance \mathbb{B} of CSAT, with $O(|T| \cdot b)$ input variables and $O(\|C\| \cdot b)$ logic gates, can be transformed into a conjunction of clauses, containing all the input Boolean variables in T , by applying Tseitin’s transformation [175]. This transformation can remove a logic gate by adding a new Boolean variable and $O(1)$ new clauses. Thus, by removing all $O(\|C\| \cdot b)$ logic gates in \mathbb{B} , we obtain a SAT instance with $O((|T| + \|C\|) \cdot b)$ Boolean variables and $O(\|C\| \cdot b)$ clauses.

Theorem 7.2.6.2.1 (Correctness of the SAT encoding): Let $N = (T, C)$ be any DTP instance. Then, N has a feasible schedule if and only if the SAT encoding of N has a satisfiable assignment.

Proof: Theorem 7.2.6.1.1 proves that N has a feasible schedule if and only if the corresponding Boolean circuit \mathbb{B}_N is satisfiable. Tseitin’s transformation [175] proves that \mathbb{B}_N is satisfiable if and only if the encoding of \mathbb{B}_N into the corresponding SAT formula is so. The proof of this theorem follows by transitivity.

□

7.2.7 Experimental Evaluation

For the SMT framework, we chose Yices2 v2.6.2 [176], winner of the SMT-COMP 2020 for QF_RDL [177]. For the MILP framework, we chose the two commercial solvers

Metric	Definition
timepoints	Number of time points.
constraints	Number of constraints.
(un)constrained_tps	Number of time points (not) appearing in at least one disjunct.
horizon	Horizon value used as Big M .
non_disj_constr	Number of one single disjunct constraints.
disj_constr	Number of constraints with more than one disjunct.
equalities	Number of disjuncts $Y - X \in [l, u]$ where $l = u$.
disj_equalities	Number of disjunctive constraints containing only equalities as disjuncts.
max_disjuncts	Maximum number of disjuncts in a constraint.
min_lu (max_lu)	Minimum (maximum) value of any number appearing in any constraint.
avg_interval_size	Average size of any closed interval $[l, u]$ where $-\infty < l \leq u < +\infty$. The size is computed as the absolute value of $u - l$.
avg_tp_connectivity	Average number of time-points a time-point is connected to, where two time points X, Y are connected if they appear in some disjunct $Y - X \in [l, u]$.
avg_constr_diversity	Average number of different time points appearing in a constraint.

Table 7: Metrics computed for the two sets of instances.

	cmr	cmr	cmr	orrz	orrz	orrz
Metric	min	max	<i>avg</i>	min	max	<i>avg</i>
timepoints	6	19994	10000	11	210	110.5
constraints	3	9997	5000	46	961	503.3
constrained_tps	4	18995	9488.02	11	210	110.5
unconstrained_tps	0	1101	511.98	0	0	0
horizon	2652	$> 10^7$	$> 10^7$	11	210	110.5
non_disj_constr	0	2567	1249.71	10	209	109.5
disj_constr	1	7523	3750.29	36	752	393.8
equalities	0	165	66.92	108	2256	1181.4
disj_equalities	0	1	0.02	36	752	393.8
max_disjuncts	3	6	5.99	3	3	3
min_lu	0	63	0.08	0	0	0
max_lu	442	1170	1082.57	1	1	1
avg_interval_size	59.61	87.6	67.03	0.08	0.09	0.08
avg_tp_connectivity	1.34	3.22	2.97	1.81	1.99	1.97
avg_constr_diversity	3	6.46	5.93	3.56	3.56	3.56

Table 8: Min, max, and average metric values shown in Figure 52 (cmr) and Figure 53 (orrz).

Gurobi v9.1 [178] and CPLEX v20.1.0 [179], broadly recognized as state of the art. For the SAT framework, we chose Kissat [180], winner of the *SAT Competition 2020*

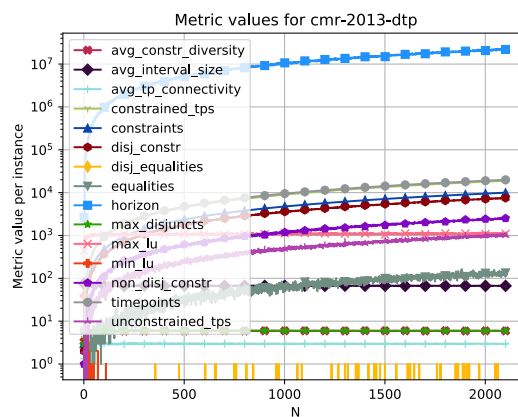
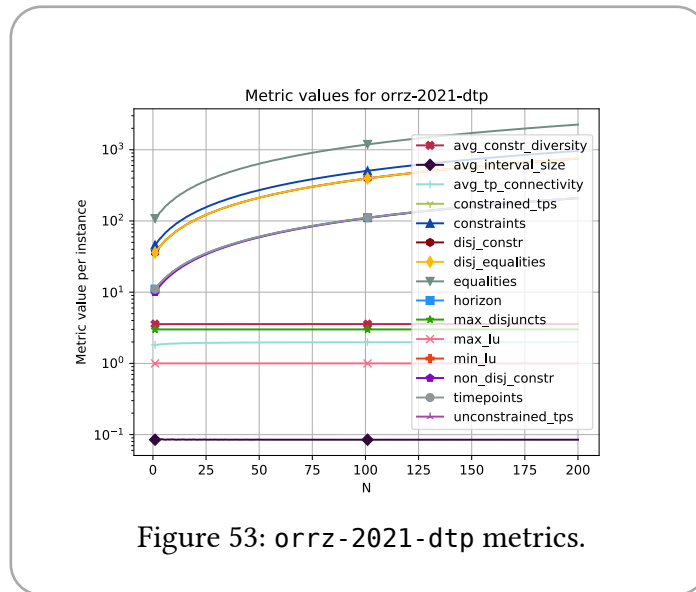
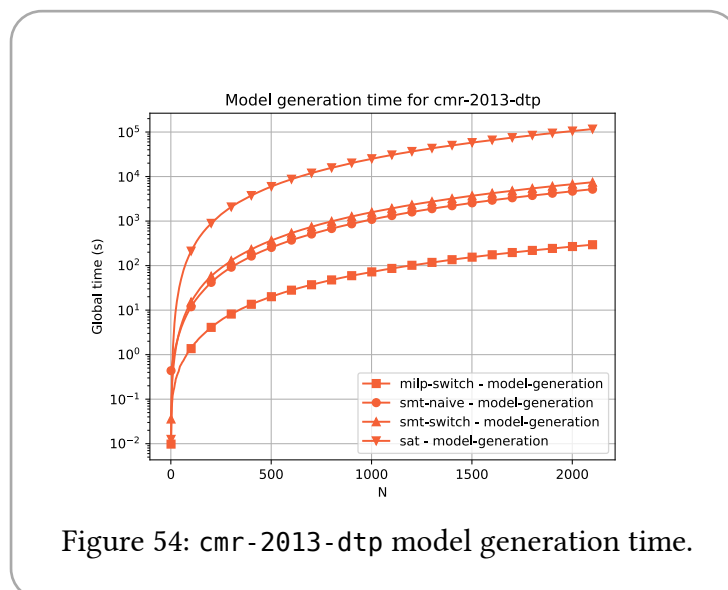


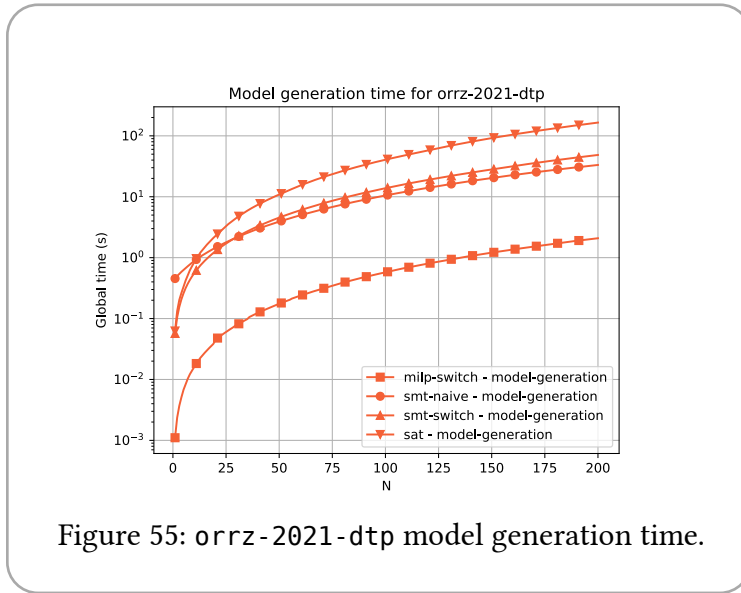
Figure 52: cmr-2013-dtp metrics.



[181]. We carried out the same experimental evaluation⁴² on two sets of benchmarks. The first one is a well recognized set of benchmarks on instances of DTP provided by Cimatti, Micheli, and Roveri in 2012-2013 [167, 168]. Such a set contains 2108 random instances of DTP generated according to [182]. We refer to this set as *cmr-2013-dtp*. We created a second set of benchmarks in this way. We built a generator of instances by using the balanced SAT construction in [183]. We generated 200 instances, for each $n = 11, \dots, 210$ being the number of time points. Each instance with n time points contains exactly $n - 1$ constraints with a single disjunct. Also, it has $k \cdot (n - 1)$ disjunctive constraints with more than one disjunct, all composed of equalities only. The value of $k = 3.6$ was experimentally determined such that the generated instances

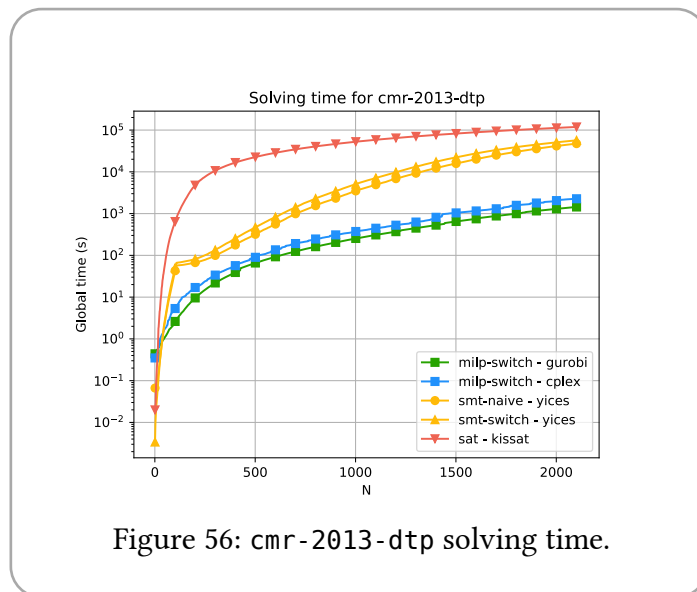


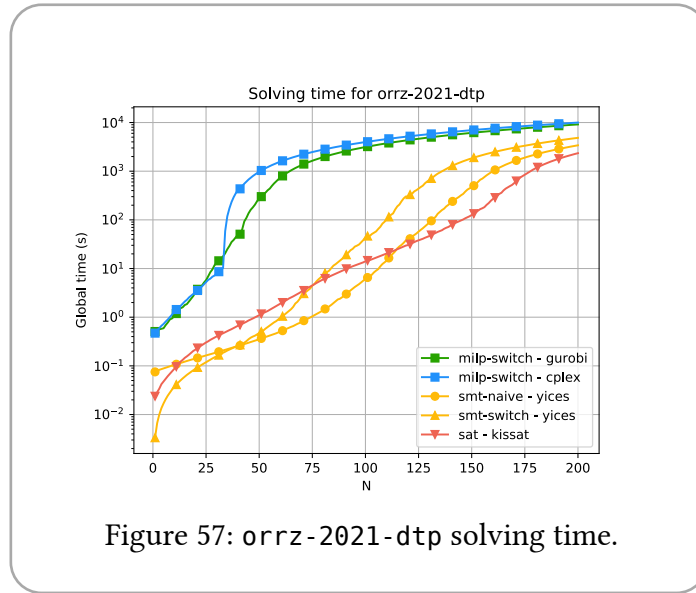
⁴²<http://y1a.altervista.org/ee-dtp.tar.bz2>



were hard [184, 183]. We refer to this set as orrz-2021-dtp. Even if these instances do not represent many real-world situations, we defined them to stress the solvers, and in particular the MILP ones that performed well on the instances collected in cmr-2013-dtp. Since no details on the parameters used to generate cmr-2013-dtp were given in [167, 168], we defined some metrics to highlight features of interest (Table 7). We show the values of these metrics in Table 8 and graphically in Figure 52 (cmr-2013-dtp) and Figure 53 (orrz-2021-dtp). For the former set and most of the latter, the bigger the instance, the bigger the corresponding metrics.

We tried to solve all the instances with respect to SMT, MILP and SAT encodings given in this work. We set a timeout of 60 seconds per instance. We used an Ubuntu Linux 20.04.1 LTS virtual machine, run on top of a VMWare ESXi Hypervisor 6.5.0,





using a physical machine equipped with an Intel i7 2.80GHz and 16GB of RAM. The virtual machine, assigned with 12 GB of RAM and full CPU power, was the only one running on the server. For each framework, we show in Figure 54 and Figure 55 the time taken by each solver to generate the models for *cmr-2013-dtp* and *orrz-2021-dtp*, whereas Figure 56 and Figure 57 show their performance on *cmr-2013-dtp* and *orrz-2021-dtp*, respectively.

The results show that no formulation always outperforms the others on both sets. Gurobi is the fastest solver for *cmr-2013-dtp*, also being the only one able to solve all the instances within the time limit (see Table 9). For *orrz-2021-dtp*, on average Kissat and Yices go head-to-head; the former able to solve more instances eventually. Looking at the metrics of *orrz-2021-dtp*, we can see that these instances are far more constrained than the ones in *cmr-2013-dtp*. This factor may slow down Gurobi (and CPLEX as well) by making harder to find a first feasible solution at the root relaxation of the Branch-and-Bound, i.e., the exact method at the core of their resolution.

7.2.8 Conclusions and future work

In this work, we tackled the Disjunctive Temporal Problem from three different sides: Satisfiability Modulo Theory, Mixed Integer Linear Programming, and Boolean Satisfiability. After describing two existing encodings of DTP into SMT, we offered a natural MILP formulation and a new non-trivial reduction to SAT, based on the observation that DTPs involving only integer numbers admit integer schedules. We carried out an experimental evaluation on a set of benchmarks already existing in the literature and on another generated by ourselves. The results show that no formulation always outperforms the others but, according to the nature of the instances, one or another may be preferable. Even if MILP solvers did not excel on our benchmarks,

		cmr	cmr	cmr	orrz	orrz	orrz
Solver	Encoding	Yes	No	?	Yes	No	?
Gurobi	milp-switch	2088	20	0	4	51	145
CPLEX	milp-switch	2080	20	8	4	32	164
Yices2	smt-naive	1739	20	349	6	147	47
Yices2	smt-switch	1547	20	541	6	125	69
Kissat	sat	173	7	1928	6	166	28

Table 9: For each solver, we provide the number of instances that admit a solution (Yes), do not admit any (No), or hit the timeout (?), respectively.

they seem to be more promising in the real applications not involving a huge number of disjunctions of equalities.

To assess this, we are working on extending the experimental evaluation to other solvers and to consider further families of instances obtained through reductions from NP-complete problems other than SAT. Moreover, we want to identify which features of an instance might hinder a solver. This could allow for the development of tools that, by analyzing the metrics of an instance, suggest which framework might be more promising. Finally, we will extend our models to consider optimization problems. In general, we are curious to focus on other interesting problems at the intersection of the communities of Artificial Intelligence, Operations Research and Theoretical Computer Science.

Bibliography

- [1] “International Collegiate Programming Contest.” [Online]. Available: <https://icpc.global/>
- [2] S. Halim, F. Halim, and S. Effendy, *Competitive programming 4: The new lower bound of programming contests in the 2020s*. Lulu. com, 2018.
- [3] K. K. Yuen, D. Y. Liu, and H. V. Leong, “Competitive programming in computational thinking and problem solving education,” *Computer Applications in Engineering Education*, 2023.
- [4] I. N. Bandeira, T. V. Machado, V. F. Dullens, and E. D. Canedo, “Competitive programming: A teaching methodology analysis applied to first-year programming classes,” in *2019 IEEE Frontiers in Education Conference (FIE)*, 2019, pp. 1–8.
- [5] S. Malik and A. Rana, “A study of competitive programming platform with its need and benefits,” *JIMS8I International Journal of Information Communication and Computing Technology*, vol. 10, no. 2, pp. 573–578, 2022.
- [6] J. Voigt, T. Bell, and B. Aspvall, “Competition-style programming problems for computer science unplugged activities,” *A new learning paradigm: competition supported by technology*, pp. 207–234, 2010.
- [7] “Ordinamento a paletta, OII 2016.” [Online]. Available: https://training.olinfo.it/#/task/oii_paletta/statement
- [8] V. Dagiene and J. Koivisto, “International Olympiads in Informatics,” *Encyclopedia of Education and Information Technologies*, pp. 982–991, 2020.
- [9] “Turing Arena light.” [Online]. Available: <https://github.com/romeorizzi/TALight>
- [10] A. M. Turing and others, “On computable numbers, with an application to the Entscheidungsproblem,” *J. of Math*, vol. 58, no. 345–363, p. 5–6, 1936.
- [11] M. M. Vopson, “Estimation of the information contained in the visible matter of the universe,” *AIP Advances*, vol. 11, no. 10, 2021.
- [12] P. Bachmann, *Analytische Zahlentheorie*. Springer, 1904.
- [13] E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*, vol. 1. BG Teubner, 1909.
- [14] R. Bellman, “Dynamic programming,” *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

- [15] “Fibonacci sequence.” [Online]. Available: <https://oeis.org/A000045>
- [16] E. H. Friend, “Sorting on electronic computer systems,” *Journal of the ACM (JACM)*, vol. 3, no. 3, pp. 134–168, 1956.
- [17] D. L. Shell, “A high-speed sorting procedure,” *Communications of the ACM*, vol. 2, no. 7, pp. 30–32, 1959.
- [18] N. Auger, V. Jugé, C. Nicaud, and C. Pivoteau, “On the worst-case complexity of TimSort,” *arXiv preprint arXiv:1805.08612*, 2018.
- [19] D. R. Musser, “Introspective sorting and selection algorithms,” *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, 1997.
- [20] C. A. Hoare, “Quicksort,” *The computer journal*, vol. 5, no. 1, pp. 10–16, 1962.
- [21] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, R. E. Tarjan, and others, “Time bounds for selection,” *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, 1973.
- [22] A. H. Land and A. G. Doig, *An automatic method for solving discrete programming problems*. Springer, 2010.
- [23] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.
- [24] D. E. Knuth, J. H. Morris Jr, and V. R. Pratt, “Fast pattern matching in strings,” *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.
- [25] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.
- [26] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” *IBM journal of research and development*, vol. 31, no. 2, pp. 249–260, 1987.
- [27] M. G. Main and R. J. Lorentz, “An $O(n \log n)$ algorithm for finding all repetitions in a string,” *Journal of Algorithms*, vol. 5, no. 3, pp. 422–432, 1984.
- [28] D. Gusfield, “Simple Uniform Preprocessing for Linear-time Pattern Matching.”
- [29] G. Manacher, “A New Linear-Time On-Line Algorithm for Finding the Smallest Initial Palindrome of a String,” *Journal of the ACM (JACM)*, vol. 22, no. 3, pp. 346–351, 1975.
- [30] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. pp. 287–290, 2022.
- [31] R. A. Jarvis, “On the identification of the convex hull of a finite set of points in the plane,” *Information processing letters*, vol. 2, no. 1, pp. 18–21, 1973.

- [32] R. L. Graham, “An efficient algorithm for determining the convex hull of a finite planar set,” *Info. Proc. Lett.*, vol. 1, pp. 132–133, 1972.
- [33] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, “The quickhull algorithm for convex hulls,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 4, pp. 469–483, 1996.
- [34] A. M. Andrew, “Another efficient algorithm for convex hulls in two dimensions,” *Information Processing Letters*, vol. 9, no. 5, pp. 216–219, 1979.
- [35] D. G. Kirkpatrick and R. Seidel, “The ultimate planar convex hull algorithm?,” *SIAM journal on computing*, vol. 15, no. 1, pp. 287–299, 1986.
- [36] A. L. F. Meister, *Generalia de genesi figurarum planarum et inde pendentibus earum affectionibus*. 1769.
- [37] B. Braden, “The surveyor's area formula,” *The College Mathematics Journal*, vol. 17, no. 4, pp. 326–337, 1986.
- [38] D. Lehmer, “A factorization theorem applied to a test for primality,” 1939.
- [39] J. Stein, “Computational problems associated with Racah algebra,” *Journal of Computational Physics*, vol. 1, no. 3, pp. 397–405, 1967.
- [40] “Codeforces survey.” [Online]. Available: <https://codeforces.com/blog/entry/83875>
- [41] D. M. Ritchie, “The development of the C language,” *ACM Sigplan Notices*, vol. 28, no. 3, pp. 201–208, 1993.
- [42] B. Stroustrup, “A history of C++ 1979–1991,” *History of programming languages –II*. pp. 699–769, 1996.
- [43] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.
- [44] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “{AddressSanitizer}: A fast address sanity checker”, in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.
- [45] N. Wirth, “The development of procedural programming languages personal contributions and perspectives,” in *Joint Modular Languages Conference*, 2000, pp. 1–10.
- [46] G. Van Rossum and others, “Python Programming Language.” in *USENIX annual technical conference*, 2007, pp. 1–36.
- [47] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

- [48] W. McKinney, *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. " O'Reilly Media, Inc.", 2012.
- [49] A. C. Müller and S. Guido, *Introduction to machine learning with Python: a guide for data scientists*. " O'Reilly Media, Inc.", 2016.
- [50] A. Rigo, M. Hudson, and S. Pedroni, "Compiling dynamic language implementations." Technical report, PyPy Consortium, 2005. <http://codespeak.net/pypy/dist~..., 2005>.
- [51] J. Gosling and H. McGilton, "The Java language environment," *Sun Microsystems Computer Company*, vol. 2550, p. 38–39, 1995.
- [52] A. Binstock, "Java's 20 years of innovation," *Forbes*, vol. 5, 2015.
- [53] "Google Code Jam." [Online]. Available: <https://codingcompetitions.onair.withgoogle.com/#code-jam>
- [54] "Meta Hacker Cup." [Online]. Available: <https://www.facebook.com/codingcompetitions/hacker-cup>
- [55] "Codeforces." [Online]. Available: <https://codeforces.com/>
- [56] "AtCoder." [Online]. Available: <https://atcoder.jp/>
- [57] "International Olympiad in Informatics." [Online]. Available: <https://ioinformatics.org/>
- [58] S. Maggiolo and G. Mascellani, "Introducing CMS: A Contest Management System," *Olympiads in Informatics*, vol. 6, 2012.
- [59] "Terry: a very customizable ``Google Code Jam" clone useful for holding programming contests." [Online]. Available: <https://github.com/algorithm-ninja/terry>
- [60] G. Audrito, W. DI LUIGI, L. Laura, E. Morassutto, D. Ostuni, and others, "The Italian Job: Moving (Massively) Online a National Olympiad," *OLYMPIADS IN INFORMATICS*, pp. 3–12, 2021.
- [61] G. Casadei, B. Fadini, and M. Vita, "Italian Olympiads in Informatics," *Olympiads in Informatics*, vol. 1, pp. 24–30, 2007.
- [62] "randomTeX: a quiz randomizer for LaTeX." [Online]. Available: <https://github.com/olimpiadi-informatica/randomtex>
- [63] L. Prechelt and M. Phlippsen, "JPlag: Finding plagiarisms among a set of programs," 2000.
- [64] "Starplag: a tool for finding the similarities between two source files." [Online]. Available: <https://github.com/olimpiadi-informatica/starplag>

- [65] “Dashboard with Terry's metrics.” [Online]. Available: <https://snapshot.raintank.io/dashboard/snapshot/Uw7uECvHWCbYjoNicT1DSn4ZKhdBamSd>
- [66] “Dashboard with System's metrics for Terry.” [Online]. Available: <https://snapshot.raintank.io/dashboard/snapshot/kPnzTPMNAIAmOHuubdds2ltUmdJ3Q9Xc>
- [67] W. Di Luigi, G. Farina, L. Laura, U. Nanni, M. Temperini, and L. Versari, “oii-web: an Interactive Online Programming Contest Training System,” *Olympiads in Informatics*, vol. 10, pp. 195–205, 2016.
- [68] N. Amaroli, G. Audrito, and L. Laura, “Fostering informatics education through teams olympiad,” in *30th International Olympiad in Informatics, IOI 2018*, 2018, pp. 133–146.
- [69] W. Di Luigi *et al.*, “Learning analytics in competitive programming training systems,” in *2018 22nd International Conference Information Visualisation (IV)*, 2018, pp. 321–325.
- [70] “OII-Proctor: a portable proctoring script.”
- [71] “Statistics about the Italian National Phase.” [Online]. Available: <https://stats.olinfo.it/>
- [72] “Dashboard with CMS's metrics.” [Online]. Available: <https://snapshot.raintank.io/dashboard/snapshot/0J29w0KruEiymy6zV30beX98aRG6njiX>
- [73] “Dashboard with System's metrics for CMS.” [Online]. Available: <https://snapshot.raintank.io/dashboard/snapshot/1s1wQCSYPgWdQe9f6sAeYZns5Ln1y6e5>
- [74] J. A. Donenfeld, “WireGuard: Next Generation Kernel Network Tunnel,” in *NDSS*, 2017.
- [75] N. D. Matsakis and F. S. Klock, “The rust language,” *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.
- [76] B. Schneier, *Beyond fear: Thinking sensibly about security in an uncertain world*. Springer Science & Business Media, 2006.
- [77] G. Audrito, G. B. Demo, and E. Giovannetti, “The Role of Contests in Changing Informatics Education: A Local View,” *Olympiads in Informatics*, vol. 6, 2012.
- [78] O. Astrachan, “Non-competitive programming contest problems as the basis for just-in-time teaching,” in *Frontiers in Education, 2004. FIE 2004. 34th Annual*, Oct. 2004, p. T3H/20–T3H/24 Vol. 1. doi: 10.1109/FIE.2004.1408553.
- [79] M. Blumenstein, S. Green, S. Fogelman, A. Nguyen, and V. Muthukumarasamy, “Performance analysis of GAME: a generic automated marking environment,” *Computers and Education*, vol. 50, pp. 1203–1216, 2008.

- [80] V. Dagienė, “Sustaining informatics education by contests,” in *International Conference on Informatics in Secondary Schools-Evolution and Perspectives*, 2010, pp. 1–12.
- [81] G. Garcia-Mateos and J. L. Fernandez-Aleman, “Make learning fun with programming contests,” *Transactions on Edutainment II*. Springer, pp. 246–257, 2009.
- [82] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang, “Ability-training-oriented automated assessment in introductory programming course,” *Computers and Education*, vol. 56, pp. 220–226, 2011.
- [83] S. S. Skiena and M. A. Revilla, *Programming challenges: The programming contest training manual*. Springer Science & Business Media, 2003.
- [84] S. Halim and F. Halim, *Competitive Programming, Third Edition*. Lulu. com, 2013.
- [85] A. Laaksonen, *Guide to Competitive Programming*. Springer, 2017.
- [86] S. Maggiolo, G. Mascellani, and L. Wehrstedt, “CMS: a Growing Grading System,” *Olympiads in Informatics*, p. 123–124, 2014.
- [87] T. Di Mascio, L. Laura, and M. Temperini, “A Framework for Personalized Competitive Programming Training,” in *Proc. of 17th International Conference on Information Technology Based Higher Education and Training*, 2018.
- [88] “PC2.” [Online]. Available: <https://pc2.ecs.csus.edu/>
- [89] “DOMjudge.” [Online]. Available: <https://www.domjudge.org/>
- [90] “SouthWestern Europe Regional Contest.” [Online]. Available: <https://swerc.eu/>
- [91] “Turing Arena.” [Online]. Available: <https://github.com/turingarena/turingarena>
- [92] J. Eldering, T. Kinkhorst, and P. van de Warken, “DOM Judge–Programming Contest Jury System. 2020.” 2010.
- [93] O. Ben-Kiki, C. Evans, and B. Ingerson, “Yaml ain't markup language (yaml™) version 1.1,” *Working Draft 2008-05*, vol. 11, 2009.
- [94] I. Fette and A. Melnikov, “The websocket protocol,” 2011.
- [95] “Serde.” [Online]. Available: <https://serde.rs/>
- [96] “Tokio.” [Online]. Available: <https://tokio.rs/>
- [97] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence,” in *Proceedings of the 3rd international joint conference on Artificial intelligence*, 1973, pp. 235–245.

- [98] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.
- [99] D. Crockford, "The application/json media type for javascript object notation (json)," 2006.
- [100] D. Merkel and others, "Docker: lightweight linux containers for consistent development and deployment," *Linux j*, vol. 239, no. 2, p. 2–3, 2014.
- [101] "Turing Arena light rust utilities." [Online]. Available: <https://github.com/dariost/tal-utils-rs>
- [102] M. Owens, *The definitive guide to SQLite*. Springer, 2006.
- [103] "Turing Arena light desktop." [Online]. Available: <https://talco-team.github.io/TALightDesktop/>
- [104] B. Green and S. Seshadri, *AngularJS*. " O'Reilly Media, Inc.", 2013.
- [105] "TypeScript." [Online]. Available: <https://www.typescriptlang.org/>
- [106] "Pyodide." [Online]. Available: <https://pyodide.org/>
- [107] "GitHub." [Online]. Available: <https://github.com/>
- [108] "Google Drive." [Online]. Available: <https://drive.google.com/>
- [109] "OneDrive." [Online]. Available: <https://onedrive.live.com/>
- [110] "GitHub Copilot." [Online]. Available: <https://copilot.github.com/>
- [111] "Google Forms." [Online]. Available: <https://www.google.com/forms/about/>
- [112] D. Ostuni, E. Morassutto, and R. Rizzi, "Make your programs compete and watch them play in the Code Colosseum," in *2021 IEEE Conference on Games (CoG)*, 2021, pp. 1–5.
- [113] C. Bellettini *et al.*, "Extracurricular activities for improving the perception of informatics in secondary schools," in *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, 2014, pp. 161–172.
- [114] T. Verhoeff, "The role of competitions in education," *Future world: Educating for the 21st century*, pp. 1–10, 1997.
- [115] V. Lonati, M. Monga, A. Morpurgo, and M. Torelli, "What's the fun in informatics? Working to capture children and teachers into the pleasure of computing," in *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, 2011, pp. 213–224.
- [116] M. Mirzayanov *et al.*, "Codeforces as an Educational Platform for Learning Programming in Digitalization," 2020.
- [117] "TopCoder." [Online]. Available: <https://www.topcoder.com/>

- [118] “CodeChef.” [Online]. Available: <https://www.codechef.com/>
- [119] “CMSocial.” [Online]. Available: <https://training.olinfo.it/>
- [120] M. Papastergiou, “Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation,” *Computers & education*, vol. 52, no. 1, pp. 1–12, 2009.
- [121] “CodinGame.” [Online]. Available: <https://www.codingame.com/>
- [122] “CodeCombat.” [Online]. Available: <https://codecombat.com/>
- [123] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, “Robocup: The robot world cup initiative,” in *Proceedings of the first international conference on Autonomous agents*, 1997, pp. 340–347.
- [124] “Code Colosseum.” [Online]. Available: <https://github.com/dariost/CodeColosseum>
- [125] I. L. Finkel, “On the rules for the Royal Game of Ur,” *Ancient Board Games in Perspective*, pp. 16–32, 2007.
- [126] “Challonge.” [Online]. Available: <https://challonge.com/>
- [127] “Discord.” [Online]. Available: <https://discord.com/>
- [128] “Feedback results.” [Online]. Available: <https://docs.google.com/spreadsheets/d/1HPgLU-hNN3vGydThcqcRaFKLox5fkfFfWkNabBURkRM/>
- [129] “Code Colosseum replay functionality.” [Online]. Available: <https://github.com/dariost/CodeColosseum/pull/2>
- [130] “Code Colosseum GUI.” [Online]. Available: <https://github.com/TALCo-Team/CodeColosseumDesktop>
- [131] “Tauri.” [Online]. Available: <https://github.com/tauri-apps/tauri>
- [132] J. Schaeffer *et al.*, “Checkers is solved,” *science*, vol. 317, no. 5844, pp. 1518–1522, 2007.
- [133] “Code Colosseum checkers.” [Online]. Available: <https://github.com/dariost/CodeColosseum/pull/3>
- [134] “Code Colosseum chess.” [Online]. Available: <https://github.com/dariost/CodeColosseum/pull/4>
- [135] “QuizMS.” [Online]. Available: <https://training-2023-fibonacci-medie.web.app/>
- [136] D. Ostuni and E. T. Galante, “Towards an AI playing Touhou from pixels: a dataset for real-time semantic segmentation,” in *2021 IEEE Conference on Games (CoG)*, 2021, pp. 1–5.

- [137] S. Risi and M. Preuss, “From chess and atari to starcraft and beyond: How game AI is driving the world of AI,” *KI-Künstliche Intelligenz*, vol. 34, no. 1, pp. 7–17, 2020.
- [138] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [139] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski, “Vizdoom: A doom-based ai research platform for visual reinforcement learning,” in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.
- [140] M. Wydmuch, M. Kempka, and W. Jaśkowski, “Vizdoom competitions: Playing doom from pixels,” *IEEE Transactions on Games*, vol. 11, no. 3, pp. 248–259, 2018.
- [141] F.-Y. Lam, “Comic market: How the world's biggest amateur comic fair shaped Japanese dōjinshi culture,” *Mechademia*, vol. 5, no. 1, pp. 232–248, 2010.
- [142] K. Sakai, Y. Okada, and Y. Muraoka, “Developing AI for playing shooter games Touhou Kaeizuka,” in *ICAI 2010: proceedings of the 2010 international conference on artificial intelligence (Las Vegas NV, July 12-15, 2010)*, 2010, pp. 748–752.
- [143] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, “The pascal visual object classes (voc) challenge,” *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.
- [144] M. Everingham, S. A. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, “The pascal visual object classes challenge: A retrospective,” *International journal of computer vision*, vol. 111, no. 1, pp. 98–136, 2015.
- [145] T.-Y. Lin *et al.*, “Microsoft coco: Common objects in context,” in *European conference on computer vision*, 2014, pp. 740–755.
- [146] M. Cordts *et al.*, “The cityscapes dataset for semantic urban scene understanding,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213–3223.
- [147] “Touhou Toolkit.” [Online]. Available: <https://github.com/thpatch/thtk>
- [148] M. L. Waskom, “Seaborn: statistical data visualization,” *Journal of Open Source Software*, vol. 6, no. 60, p. 3021–3022, 2021.
- [149] R. P. Poudel, S. Liwicki, and R. Cipolla, “Fast-scnn: fast semantic segmentation network,” *arXiv preprint arXiv:1902.04502*, 2019.
- [150] C. Yu, C. Gao, J. Wang, G. Yu, C. Shen, and N. Sang, “Bisenet v2: Bilateral network with guided aggregation for real-time semantic segmentation,” *arXiv preprint arXiv:2004.02147*, 2020.

- [151] G. J. Brostow, J. Shotton, J. Fauqueur, and R. Cipolla, "Segmentation and recognition using structure from motion point clouds," in *European conference on computer vision*, 2008, pp. 44–57.
- [152] G. J. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: A high-definition ground truth database," *Pattern Recognition Letters*, vol. 30, no. 2, pp. 88–97, 2009.
- [153] H. Caesar, J. Uijlings, and V. Ferrari, "Coco-stuff: Thing and stuff classes in context," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 1209–1218.
- [154] H. Li, P. Xiong, H. Fan, and J. Sun, "Dfanet: Deep feature aggregation for real-time semantic segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9522–9531.
- [155] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "Enet: A deep neural network architecture for real-time semantic segmentation," *arXiv preprint arXiv:1606.02147*, 2016.
- [156] S. Song, S. P. Lichtenberg, and J. Xiao, "Sun rgb-d: A rgb-d scene understanding benchmark suite," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 567–576.
- [157] T. Wu, S. Tang, R. Zhang, J. Cao, and Y. Zhang, "Cgnet: A light-weight context guided network for semantic segmentation," *IEEE Transactions on Image Processing*, vol. 30, pp. 1169–1179, 2020.
- [158] N. Ketkar, "Introduction to pytorch," *Deep learning with python*. Springer, pp. 195–208, 2017.
- [159] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," *arXiv preprint arXiv:1904.09237*, 2019.
- [160] "Comparison on real Touhou videos." [Online]. Available: <https://www.youtube.com/playlist?list=PL28XtHfG7kBVADdTifjINOc9OVK-N4-5z>
- [161] M. Zavatteri, A. Raffaele, D. Ostuni, and R. Rizzi, "An interdisciplinary experimental evaluation on the disjunctive temporal problem," *Constraints*, vol. 28, no. 1, pp. 1–12, 2023.
- [162] R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks," *Artificial Intelligence*, vol. 49, no. 1, pp. 61–95, 1991.
- [163] A. Oddi and A. Cesta, "Incremental Forward Checking for the Disjunctive Temporal Problem.," in *ECAI 2000*, IOS Press, 2000, pp. 108–112.
- [164] K. Stergiou and M. Koubarakis, "Backtracking Algorithms for Disjunctions of Temporal Constraints," *Artificial Intelligence*, vol. 120, pp. 81–117, 2000.

- [165] I. Tsamardinou and M. Pollack, “Efficient solution techniques for disjunctive temporal reasoning problems,” *Artificial Intelligence*, vol. 151, pp. 43–89, 2003.
- [166] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, “Satisfiability Modulo Theories,” *Handbook of Satisfiability*, vol. 185. in *Frontiers in Artificial Intelligence and Applications*, vol. 185. IOS Press, pp. 825–885, 2009.
- [167] A. Cimatti, A. Micheli, and M. Roveri, “Solving Temporal Problems Using SMT: Strong Controllability,” in *CP 2021*, Springer, 2012, pp. 248–264.
- [168] A. Cimatti, A. Micheli, and M. Roveri, “Solving strong controllability of temporal problems with uncertainty using SMT,” *Constraints*, vol. 20, pp. 1–29, 2014.
- [169] R. Bellman, “On a routing problem,” *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.
- [170] L. R. Ford Jr, “Network flow theory,” 1956.
- [171] R. W. Floyd, “Algorithm 97: Shortest Path,” *Commun. ACM*, vol. 5, no. 6, p. 345–346, 1962, doi: 10.1145/367766.368168.
- [172] D. B. Johnson, “Efficient Algorithms for Shortest Paths in Sparse Networks,” *J. ACM*, vol. 24, no. 1, pp. 1–13, 1977.
- [173] M. Cairo, L. Hunsberger, R. Posenato, and R. Rizzi, “A Streamlined Model of Conditional Simple Temporal Networks - Semantics and Equivalence Results,” in *TIME 2017*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 1–19.
- [174] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann., 2013.
- [175] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” *Automation of reasoning*. Springer, pp. 466–483, 1983.
- [176] SRI International's Computer Science Laboratory, “The Yices SMT Solver.” [Online]. Available: <https://yices.csl.sri.com/>
- [177] SMT Steering Committee, “15th International Satisfiability Modulo Theories Competition.” [Online]. Available: <https://smt-comp.github.io/2020/index.html>
- [178] L. Gurobi Optimization, “Gurobi Optimizer Reference Manual.” [Online]. Available: <http://www.gurobi.com/>
- [179] IBM, “ILOG CPLEX Optimization Studio.” [Online]. Available: <https://www.ibm.com/products/ilog-cplex-optimization-studio>
- [180] Armin Biere, “kissat.” [Online]. Available: <http://fmv.jku.at/kissat/>

- [181] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020,” in *Proc.~of SAT Competition 2020 – Solver and Benchmark Descriptions*, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Jarvisalo, and M. Suda, Eds., in Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.
- [182] A. Armando, C. Castellini, and E. Giunchiglia, “SAT-Based Procedures for Temporal Reasoning,” in *Recent Advances in AI Planning*, Springer, 2000, pp. 97–108.
- [183] I. Spence, “Balanced random SAT benchmarks,” *SAT COMPETITION 2017*, p. 53–54, 2017.
- [184] G. Escamocher, B. O’Sullivan, and S. D. Prestwich, “Generating difficult sat instances by preventing triangles,” *arXiv:1903.03592*, 2019.