Proceedings of the 19th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 331–342
DOI: 10.15439/2024F8479
ISSN 2300-5963 ACSIS, Vol. 39

Goliath, a Programming Exercises Generator Supported by AI

Tiago Carvalho Freitas
ALGORITMI Research Centre / LASI, DI
University of Minho, Braga, Portugal
Email: tiago10cf@hotmail.com

Alvaro Costa Neto
0000-0003-1861-3545
ALGORITMI Research Centre / LASI, DI
University of Minho, Braga, Portugal
Instituto Federal de Educação,
Ciência e Tecnologia de São Paulo
Barretos, Brazil
Email: alvaro@ifsp.edu.br

Maria João Varanda Pereira
0000-0001-6323-0071
Research Centre in Digitalization and Intelligent Robotics (CeDRI)
Laboratório para a Sustentabilidade e Tecnologia
em Regiões de Montanha (SusTEC)
Instituto Politécnico de Bragança
Campus de Santa Apolónia, 5300-253 Bragança, Portugal
Email: mjoao@ipb.pt

Pedro Rangel Henriques
0000-0002-3208-0207
ALGORITMI Research Centre / LASI, DI
University of Minho, Braga, Portugal
Email: prh@di.uminho.pt

***Abstract*—The teaching-learning process is complex in nature, requiring many tasks and skills to achieve success in the construction of knowledge. As per any particular kind of cognitive development, teaching and learning Computer Programming is no different in this regard: tasks must be executed, sometimes repeatedly, and skills must be developed. Despite different approaches and methodologies, exercising what has been studied is proven to be effective in pretty much any teaching-learning process. Many tools have been developed throughout time to aid in the execution of this important task, sometimes approaching the problem from the students' perspective, sometimes from the teachers'. This paper presents Goliath, a semi-automatic generator of Computer Programming exercises, whose functionality is based on Artificial Intelligence (AI) models, a Domain-Specific Language (DSL), and an online application that binds them together. Goliath's goals are directed towards teachers (and indirectly, students) by aiming to lower the burden of repeatedly constructing exercises. This is achieved through the use of templates that allow for automatic variations of an exercise to be created instantly, while relying on a common foundation. Goliath is meant to be a facilitator, raising availability of exercise lists, while avoiding repetition and the common mistakes that accompany their construction.**

***Index Terms*—Computer Programming, Programming Education, Artificial Intelligence, Domain-Specific Languages, Programming Exercises**

I. Introduction

TEACHING and learning Computer Programming is an advanced, arduous and complex process, both for teachers and students. Several challenges present themselves continuously, ranging from technical to personal [1], [2]. Among the many strategies to aid in this endeavour that have been research for decades [3], [4], [5], exercising is paramount. Not only it is an opportunity for the student to grow and validate his or her lessons, but also serves as a guiding metric for the teacher, who can evaluate how to better pace the course content, and which points have been better cemented in the students' minds. In this regard, repetition is essential for a continuous and uneventful evolution in students' growth.

In order to achieve consistency in repetition, students should be able to access, answer and obtain response feedback from exercises as fast, and as frequent, as possible. In this regard, dedicated applications to aid both teachers and students in achieving the ultimate goal of teaching and learning how to program are invaluable, and have been constructed for many years. Goliath is an application that aims to leverage AI and a dedicated DSL to support teachers in creating exercises that offer students variation, availability and consistency in their training time.

This paper is further divided into six more sections. Section II contextualizes resources for practicing computer programming. Section III presents how exercises are structurally constructed, based on foundational research. Section IV compares AI-supported methods for automatic and semi-automatic generation of Computer Programming exercises. Section V details structural and functional aspects of Goliath. Section VI presents the tests, results and feedback obtained for Goliath. Finally, Section VII concludes the paper with final regards on Goliath's goals and achievements, and suggests future derivations and improvements within the scope of this research.

This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.
The work of Maria João was supported by national funds through FCT/MCTES (PIDDAC): CeDRI, UIDB/05757/2020 (DOI: 10.54499/UIDB/05757/2020) and UIDP/05757/2020 (DOI: 10.54499/UIDP/05757/2020); SusTEC, LA/P/0007/2020 (DOI: 10.54499/LA/P/0007/2020).

II. Resources for Practicing Programming

Practicing is paramount to learn and construct knowledge. It is so intrinsic to the educational process that several approaches have been developed and applied to teach and learn.

When it comes to computer programming, practice support assumes a wide range of implementations. Lists of exercises (written and printed by teachers or tutors) and problems in textbooks are classical offerings in educational contexts. More modern tools, such as online guides to programming learning [6], online courses, program animation applications [7], and automatic evaluation tools [8].

Nonetheless, creating exercises is still a challenge. Many factors demand consideration in order to create clear, useful ones: the approach to the topic under practice, the difficulty it will present to a diverse student population, the correct and unambiguous wording of the problem statement to avoid misdirection and general confusion, *etc*. Once all of these challenges have been surpassed and a collection of quality exercises is reached, creating new ones, or even variations of those that already exist, is not trivial. Besides being a time-consuming task, it becomes ever more prone to errors as repetition allows for lost of focus to creep in, resulting in typos, missed information, and incoherence. Systems to automatize these tasks have been implemented, such as SIETTE [9], which allows the creation and management of exercises repositories, and R/exams [10], a package for the R language that provides mechanisms to create both HTML and LaTeX versions of parameterized exercise lists. Although successful in their own ways, the initial generation of the exercise components (more on that on Section III) is done directly by the user.

Goliath aims to aid in the assignment of programming exercises and their answering, but more importantly, it supports their actual construction with the use of two Large Language Models (LLMs) and a DSL-based template system, avoiding those typical mistakes and improving the possibility of variations in already proven exercises.

III. Structure of Programming Exercises

Understanding how programming exercises are designed and structured was paramount to automate parts of their construction—which is, ultimately, Goliath's central objective. Generally speaking, these exercises were classified into different *types* that are commonly found in tests, lists, websites, *etc*. Furthermore, any of these types can be segmented into three main *components* (the statement, the code, and the answer area), each with its own responsibility in communicating the exercise's intent to the student.

A. Exercise Types

From many definitions of exercise types available in the literature [11], [12], [13], Goliath relied on those published in [14], given their resemblance to how it handles their construction and which types it is capable of generating.

In total, seven types were collected:

1) **Code from scratch:** students must write down the complete solution to a problem from scratch (as the name implies). No support code (or template) is provided, only a dedicated empty space for the answer;
2) **Code completion:** In order to solve this type of exercise, students must fill blanks that have been strategically positioned in a provided excerpt of code;
3) **Code improvement:** after being provided with a complete snippet of code that solves a given problem, students are asked to improve it. The modifications may require improvements to the performance, reduction of lines of code, use of specific constructs, *etc.*
4) **Bug finding:** as the name suggests, students must identify bugs (and their characteristics) in a excerpt of code, without actually correcting them;
5) **Debugging and fixing:** consists in a mix of the last two types, in which students are to write a correct version of a source code that contains bugs;
6) **Code interpretation:** students are required to interpret a given snippet of code and report on its behaviour, goals, evolution of a variable's value throughout execution, *etc*.
7) **Output or state prediction:** students are asked to find out either the output of a source code's execution, or the value of a variable throughout its lifetime.

These types are commonplace in programming courses and are implemented either physically (via printed paper, for example), or digitally. Several adaptations are also possible, including the option to transform the answer format from open to multiple choices.

Given certain implementation requirements (which will be presented and discussed later in the article), of all seven types, three were chosen and adapted to be generated by Goliath: *code from scratch*, *code completion*, and *output or state prediction*.

B. Exercise Components

There are usually three main components to consider in a typical exercise (of any of the seven types): the problem *statement*, the accompanying *code*, and the *answer area* (see Fig. 1).

The problem *statement* contains text that is presented to the student explaining the context and parameters of the problem, the type of answer that is expected, and other pertinent details about the exercise. An excerpt of *code* usually follows, containing entire programs, snippets with blanks to be filled, or concurrent versions to be compared, analysed or fixed. It supports the problem *statement* to establish a basis for solving the exercise. Finally, the *answer area* contains wither a blank space or the distribution of possible options for the student's answer.

The definition of these three components were required to design the semi-automatic generation of exercises. Goliath based its generative mechanisms on a divide-and-conquer strategy: each component, albeit semantically connected, could be created independently, as long as the reasoning behind the problem was consistently maintained. The next step in implementing Goliath's semi-automatic generation mechanism

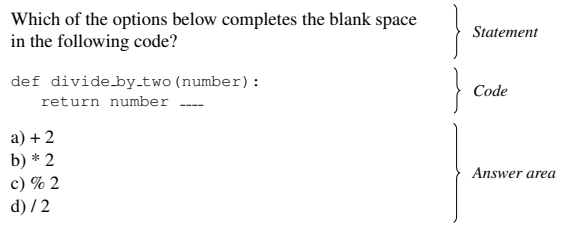

Fig. 1. Typical components of a programming exercise.

was to evaluate and choose AI models that could generate one or more of these components.

IV. AI-supported Exercise Generation

Commonly applied to create chat-bots, translators and similar tools, Natural Language Generation (NLG) (a sub-field of Natural Language Processing (NLP)) contains theories and techniques that allow for the production of coherent and useful text in multiple languages [15], [16], including complex articles and stories [17]. The process involves taking input data, such as keywords, a set of facts or a starting piece of text, and transforming (or complementing) it into meaningful output text.

NLG is based on tasks [16] and architectures [15] that are specifically designed to produce useful text. Their features and functionalities are implemented into *models* that are trained to perform text generation, realising the foundation on which AI-supported systems are built.

A. Models

Models are pre-trained pieces of software that can recognise patterns or generate expected outputs given a set of inputs. Specifically in the case of text generation, AI models are trained to generate text based on an input of some form (keywords, sentences, questions *etc*.) [18].

The implementation techniques for AI models are varied, and have been evolving in a fast pace. Artificial Neural Networks (ANNs), a subset of Machine Learning (ML) technologies, are currently the *de facto* choice for implementing models. Their structures have been inspired by the human brain and operate in a similar fashion, offering results that are human-like [19]. Architectures of ANN that have been commonly and effectively used for text generation include:

- **Recurrent Neural Network (RNN):** suited for processing sequential data, such as text, incorporates feedback connections that take into account previous time steps and observations [20]. Implementations include Long Short-Term Memory (LSTM) networks [21], [22], and Seq2seq models [23], [24];
- **Transformers:** a ANN architecture that is capable of modelling long-range dependencies between words in a piece of text. Includes self-attention mechanisms to weight the relevance of words (or tokens) on the input. Transformers are well suited for applications that involve comprehension of context or semantics [25], [26]. Implementations include Generative Pre-training Transformer (GPT) [27], [28], [29], [30], Bidirectional Encoder Representations from Transformers (BERT) [31], [32], and others;
- **Generative Adversarial Network (GAN):** consists of two networks linked together (a generator and a discriminator) [33], but trained individually. The primer generates text that is then verified by the later. TextGAN [34] implements and improves a GAN to generate coherent text samples;
- **Variational Autoencoder (VAE):** can be used to several goals, including text generation. It consists of two networks (an encoder and a decoder) and a latent space [35]. These three components work by trying to minimise differences between the input and a reconstructed counterpart. Bowman *et al* [36] implemented a VAE-based model to generate natural language sentences based on comparisons in two different languages.

These architectures are implemented through several techniques, languages and libraries. Examples include: OpenAI's libraries [37], TensorFlow [38], Keras [39], PyTorch [40], Hugging Face [41] and TextGenrnn [42].

B. AI Models to Generate Exercises

In order to determine which AI models would be used for the generation of the exercise components, four candidates were tested and evaluated: GPT-3.5[1], GPT-2, KeyToText [43] and CodeT5 [44]. The overall testing methodology, results and conclusions have been published in greater detail in a previous study [45]. Table I presents a resumed version of the models' features, as evaluated for the purposes of generating full programming exercises (statement, code and answer).

OpenAI's GPT-3.5 was the first model to be tested (see Listing 1 for an example). Despite returning accurate results, this version had a few shortcomings that made it less valuable for use in Goliath:

- It was free to use only within a gracing period;
- The input prompt had to contain a complete description of the exercise and how it should be generate, contradicting one of Goliath's main goals: to automate the generation of the exercise's statement;
- The output contained both the problem statement and the answer, which would require some interpretation and post-processing in order to establish an internal representation of the exercise's components;
- Even if OpenAI released it publicly (which they don't), it would be unfeasible to use it locally given its size. This means Goliath would be dependent on OpenAI's servers and consequent communication.

The second tested model was a previous version of OpenAI's offerings, GPT-2. Since it has a more manageable size, the whole model was downloaded and fine-tuned with a dataset of 300 basic Python exercises [46]. The prompt input consisted in a formatted request for the code of a specific exercise. It used Application Programming Interfaces (APIs)

[1]GPT-3.5 was the newest version at the time of the study.

TABLE I
COMPARISON BETWEEN DIFFERENT AI MODELS AND THEIR FEATURES FOR PROGRAMMING EXERCISES GENERATION.

Model	Cost	Hosting	Input	Output
GPT-3.5	Paid after gracing period	OpenAI Servers	Full request and definitions	Statement, code and answer
GPT-2	Free	Local	Formatted text and statement	Statement and code
KeysToText	Free	Local	Keywords	Statement
CodeT5	Free	Local	Function statement	Code

Listing 1 Example of a programming exercise generation using GPT-3.5.

Source:

```
prompt = "Create a programming exercise in Python
    about string manipulation and write the answer."
exercise = openai.Completion.create(engine="text-
    davinci-003", prompt=prompt, temperature=0.7,
    max_tokens=500, top_p=1, frequency_penalty=0,
    presence_penalty=0)
```

Output:

```
Write a program to print out the last three
    characters of a string.

#Answer:
string = input("Enter a string: ")
length = len(string)
last_three_chars = string[length - 3:]
print(last_three_chars)
```

from *aitextgen* [47], a free library that interfaces with GPT-2 (see Listing 2). The output contained both the statement and the code for the exercise. It resulted in less accurate output than GPT-3.5, and some of the same shortcomings were also present (necessity to provide most of the exercise as input, and full length output that would require post-processing).

Finally, two other models were tested in tandem: KeyToText and CodeT5. KeyToText is based on Text-to-Text Transfer Transformer (T5) [48] and was fine-tuned with the Mostly Basic Python Problems (MBPP) dataset from Google Research [49], containing 1000 entry-level programming problems. The model, used with the support of the Natural Language Toolkit (NLTK) [50] platform, was able to generate accurate (albeit simple) exercise statements from a minimum of three keywords (see Listing 3 for an example). This solution was unique when compared to the GPT models, since it was able to automate the generation of the exercise statement. This statement, in turn, could be adapted and fed into the CodeT5 model to generate its accompanying source code (see Listing 4 for an example). CodeT5 also implements a T5 model, trained with the CodeSearchNet [51] dataset collection. It was able to accurately generate Python source code coherent with the exercise statement.

After approximately 100 tests were conducted, 80% rate of success was achieved in generating comprehensible and coherent exercises via KeyToText and CodeT5. Goliath implemented both models in a sequence of execution (further

Listing 2 Example of a programming exercise generation using the fine-tuned GPT-2 model.

Source:

```
prompt = """Exercise

Instructions:
Write a Python program to reverse a string.

Code:"""

model.generate(prompt=prompt, max_length=500,
    temperature=0.3, top_p=0.9)
```

Output:

```
Exercise

Instructions:
Write a Python program to reverse the binary
    representation of a given integer.

Code:
def test(n):
  return int(bin(n)[::-1][:-2], 2)
```

Listing 3 Example of an exercise statement generation using the fine-tuned *Keys-To-Text* model.

Source:

```
model.predict(["remove", "list", "odd numbers"])
```

Output:

```
Write a function to remove odd numbers from a list.
```

Listing 4 Example of code generation using *CodeT5*.

Source:

```
p = "Function to remove odd numbers from a list."

ids  = tokenizer(p, return_tensors="pt").input_ids
code = model.generate(ids, max_length=128)
tokenizer.decode(code[0], skip_special_tokens=True)
```

Output:

```
def remove_odd_numbers(nums):
    return [n for n in nums if n % 2 == 0]
```

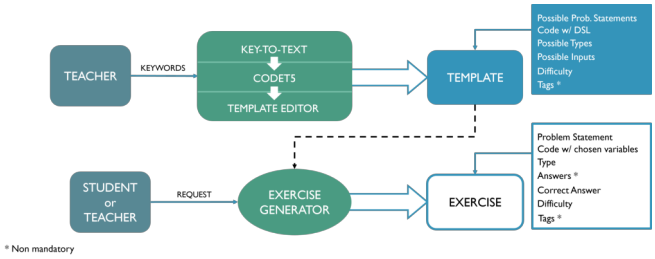

Fig. 2. Goliath's functionality and main features.

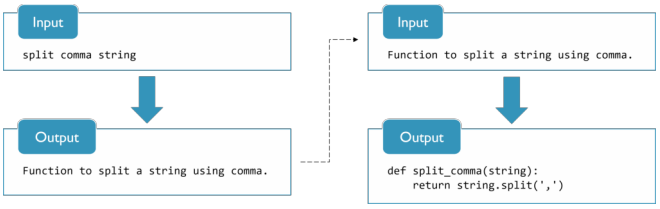

Fig. 3. AI-supported statement and code generation.

explained in Subsection V-B) to generate initial templates for Python programming exercises. This resulted in the effective use of the NLG models to aid in the construction of the whole system.

V. Goliath

Goliath is an online application[2] for semi-automatic generation of programming exercises[3]. It is based on two AI models (KeyToText and CodeT5), a template system, a DSL, and other supporting functionalities. It allows teachers to create a repository of templates that are used to generate programming exercises. These are then assigned to students to aid in their programming practices.

Fig. 2 shows the general design of Goliath's functionality and its main features. There two basic workflows: the creation of templates (top half of Fig. 2) and the generation of exercises (bottom half).

The template creation is based on a sequence of operations:

1) The teacher provides Goliath with (at least three) keywords to generate an exercise statement using KeyToText (left side of Fig. 3);
2) The statement, after reviewed by the teacher, is fed into CodeT5 to generate its accompanying code (right side of Fig. 3);
3) Both statement and code are presented to the teacher in a Template Editor;
4) The teacher embeds parameters into the template using commands from a DSL, specifying variations on the exercise;
5) The template is stored in a repository, along with a few other settings that the teacher can define (explained later in the article).

Given that the teacher may not want to use the AI-supported functionalities, both steps 1 and 2 can be skipped, as long as he or she writes the statement and the code from scratch. Further details about the template system and the DSL are presented in Subsection V-A.

[2]Accessible at: https://goliath.epl.di.uminho.pt/

[3]Only Python is currently supported as the programming language for the exercises. This technical limitation was implemented due to two reasons: the AI models have been fine-tuned with datasets of Python source code, and internal verification mechanisms also assume Python as the language of choice for the exercises.

Creating a template does not generate an exercise in itself. This only happens after a request is sent to the Exercise Generator by the student. The generation process is straightforward:

1) The student requests an exercise that was assigned by the teacher;
2) The corresponding template is fetch from the repository;
3) A version of the exercise is automatically generated by choosing randomly one of the pre-defined variations that the template specifies;
4) The answer alternatives are created *ad hoc*, based on the exercise version;
5) The exercise (statement, code and answer alternatives) is presented to the student who can answer it;
6) The correct answer is shown as feedback.

This division of workflows is intentional: by delaying the generation of the exercise, Goliath is able to add a layer of controlled randomisation to the process. This fact contributes to its replayability and provides a sense of discovery to the students. Further explanations on the exercise generation are given in Subsection V-B.

A. Goliath's DSL and the Template System

Goliath's template system allows educators to create *replayability*[4], in which one template can automatically generate different versions of an exercise.

Versions, in this context, represent variations of what is requested from students. As an example, a simple exercise statement could read "*Write a function that removes all odd numbers from a list of integers*". In order to change the request from *remove odd numbers* to *remove even numbers*, this statement needs only to be minutely changed. Furthermore, the statement could also be modified to request the removal of all positive numbers, all prime numbers, or any equivalent variation. Thus, if given the possibility of automatically generating these versions (odd, even, positive, prime, *etc.*) from one template, students could practice multiple times, while educators needed only to construct and parameterise the template once.

The template consists in the statement, the code, and a few added settings (explained in Subection V-C). The answer area is the only component that is not directly included in the template, as it is automatically created when the exercise is generated. For demonstration purposes, consider Listing 5 as a basis for a template that was suggested by the AI models.

[4]Replaybility indicates the possibility of using the system multiples with a lower chance of encountering the same state with frequency.

Listing 5 Basis for a template created from the keywords *list*, *remove*, and *numbers*.

Statement:

```
Function that removes all odd numbers from a list of
     integers.
```

Code:

```
def remove_odd_numbers(nums):
    return [n for n in nums if n % 2 == 0]
```

Listing 6 Grammar for the DSL that allows templates to generate several versions of an exercise. Note that the commands can not be recursively nested.

```
start       : declaration* commands

declaration : ID "=" STRING ("," STRING)+
commands    : (alternatives | conditional)*

alternatives: STRING ("," STRING)* fail?
conditional : "case" vars ":" actions else? fail?
vars        : ID ("," ID)*
actions     : action (";" action)*
action      : STRING ("," STRING)* "=>" STRING

else        : ";" "else" ":" STRING
fail        : ";" "fail" ":" STRING ("," STRING)*
```

If left unchanged, this template would only generate one version of the exercise ("*Function that removes all odd numbers from a list of integers*"). In order to make it more flexible, it is necessary to parameterise it. A DSL was designed specifically for this purpose. Its grammar is presented in Listing 6, which was used with Lark [52] to develop an interpreter. This DSL is intended to be used in the same fashion as a markup language, whose commands must be interspersed into the text.

The commands of parameterisation are placeholders delimited by double curly brackets ({{ and }}). They are substituted by a value (generally speaking, a piece of text) when the exercise is generated. There are three types of commands: *key declarations*, *conditional placements*, and *simple alternatives*.

1) Key declaration: Multiple keys can be declared in the exercise statement, representing variations on what is requested from the student. Each key defines a list of possible values. For example, in order to parameterise the template of Listing 5 and allow the generation of both *odd* and *even* versions of its exercise, the statement should be rewritten as:

```
Function that removes all {{ x = 'odd', 'even' }}
    numbers from a list of integers.
```

The key `x` creates the variation for *odd* and *even*. When an exercise is generated from this template, its statement will read either "*...function that removes* odd *numbers...*" or "*...function that removes* even *numbers...*". Listing 7 shows the template with the parameterised statement.

Although unnecessary for this example, there could be multiple key declarations. The following statement would be able to generate four different versions of the exercise (remove *odd* from *a list*, remove *odd* from *an array*, remove *even* from *a list*, and remove *even* from *an array*):

Listing 7 Template with a key declaration in the statement.

Statement:

```
Function that removes all {{ x = 'odd', 'even' }}
    numbers from a list of integers.
```

Code:

```
def remove_odd_numbers(nums):
    return [n for n in nums if n % 2 == 0]
```

```
Function that removes all {{ x = 'odd', 'even' }}
    numbers from {{ y = 'a list', 'an array' }} of
    integers.
```

Finally, key declarations must only occur in the statement, as they establish variations on what the exercise will ask from the student. They would not make sense in the code, given that its logic is supposed to be derived from the statement, not the other way around.

2) Conditional placements: Assuming that an exercise must be entirely coherent, if there are possible variations on what is asked from student (via *key declarations* in the statement), a fixed code would probably be wrong. *Conditional placements* allow the code to adapt to the variations of the keys when the exercise is generated.

Conditional placements always start with the keyword `case`, followed by the list of keys that should be considered. For each combination of values, a result is specified using the *hash-rocket* notation (=>). It is similar to a `switch-case` in a conventional programming language.

Going back to the example in Listing 7, if the key `x` assumes the value `odd` in the exercise, the second line of code must read:

```
return [n for n in nums if n % 2 == 0]
```

This guarantees that only even numbers are left in the list, effectively removing odd numbers. On the other hand, if `even` is chosen for `x`, the line must change to:

```
return [n for n in nums if n % 2 == 1]
```

The template can be parameterised for this variation through a conditional placement on `x`:

```
return [n for n in nums if n % 2 == {{ case x: 'odd'
    => '0'; 'even' => '1' }}]
```

This conditional placement results[5] in `0` if `x` is `odd`, and `1` if `x` is `even`. Since `x` has only two possible values, the command could also be written using an `else` clause:

```
return [n for n in nums if n % 2 == {{ case x: 'odd'
    => '0'; else: '1' }}]
```

[5]As previously explained, all commands from the DSL are placeholders. The result of a command refers to the value that will substituted it in the exercise.

This clause acts as a complementary branch in the conditional placement. It is similar to the `default` branch in a `switch-case`.

Finally, the `fail` clause can be used to specify invalid options for the placement:

```
return [n for n in nums if n % 2 == {{ case x: 'odd'
     => '0'; else: '1'; fail: '2', '3' }}]
```

The definition of invalid options using the `fail` clause is important as Goliath is unable to automatically come up with wrong answers (further explanations in Subsection V-B).

Finally, a conditional placement can take multiple keys into consideration:

```
{{ case x, y: 'odd',  'a list'   => '0';
              'even', 'an array' => '1';
        else: '-1';
        fail: '2','3' }}
```

3) Simple alternatives: Unlike conditional placements, *simple alternatives* do not take keys into consideration. They simply create variations, with the added possibility of specifying invalid options:

```
return [n for n in nums if n {{ '% 2'; fail: '/ 2',
    '* 2', '+ 2' }} == 0]
```

The correct option for that portion of code is `% 2`, while the others (defined by the `fail` clause) are invalid and will generate incorrect alternatives for the answer in the exercise.

Simple alternatives can also be used to create equivalent variations in the code:

```
h = {{ 'n / 2.0', 'n * 0.5' }}
```

Both variations of the code above are equivalent in their objective (assigning the half of `n` to `h`) and would result in two different, albeit valid, versions.

4) Final remarks on the DSL: The three types of commands (key declarations, conditional placements and simple alternatives) allow for the definition of both valid and invalid variations. Both are equally important for the generation of the exercise, given that Goliath is not able to self-determine which changes in the code would result in correct answers and which would not. In order to that, Goliath would be required to not only perceive what is asked by the statement—meaning, the actual problem that the exercise entails—but also to check if the code's logic is coherent with it. The Halting Problem guarantees this impossibility.

Typical error checking is done during the interpretation of the DSL commands, including the use of undeclared keys, syntactic mistakes and invalid commands. After all commands have been successfully interpreted, an internal JSON-like representation of the keys and their variations is created, and stored with the template (see Listing 8).

Listing 9 shows the complete version of the exercise template, including all three types of commands.

B. Exercise Generation

As previously mentioned, a template acts as a blueprint that generates different versions of an exercise. The generative process begins with a request from a student, when he or she accesses an assignment created by the teacher[6]. In this context, the real exercise can be considered an instantiation of the assignment's template.

Listing 8 Internal representation of a template's keys and variations.

```
"keys": {
    "x": ["odd", "even"]
},

"variables": {
    "alt1": {
        "correct": [
            { "value": "% 2" }
        ],
        "wrong": ["/ 2", "* 2", "+ 2"]
    },
    "alt2": {
        "correct": [
            {
                "value": "0",
                "conditions": [
                    { "key": "x", "index": 0 }
                ]
            },
            {
                "value": "1",
                "conditions": [
                    { "key": "x", "index": 1 }
                ]
            }
        ],
        "wrong": ["2", "3"]
    }
}
```

A few important considerations must be made in order to understand the exercise generation:

- There are three types of exercises available in Goliath: *code selection* (adapted from *code from scratch*), *input/output*, and *code completion* (also adapted)[7]. This limitation was imposed by the mechanisms implemented in the exercise generation. Additional types would require more commands in the DSL and more settings in the template, which was unfeasible for this version of Goliath;
- Answers are presented in multiple choice format for the *code selection* and *code completion* types[8]. Implementing an open answer format for these types would either negate the immediate feedback to students, as they waited for the teacher to correct the answers, or require the implementation of an extremely accurate NLP model. *input/output*, on the other hand, is presented in open answer format, since it can be trivially verified by executing the code of the exercise itself;
- Both the statement and the code in the template need to follow specific discourses. Statements must start with

[6]Teachers may create an assignment at any time, as long as the template has already been created and stored in the repository.

[7]See Subsection III-A for the description of each type.

[8]This is the reason Goliath implements adapted versions of the original exercise types.

Listing 9 Template containing all three constructs.

Statement:

```
Function that removes all {{ x = 'odd', 'even' }} numbers from a list of integers.
```

Code:

```
def remove_{{ x }}_numbers(nums):
    return [n for n in nums if n {{ '% 2'; fail: '/ 2', '* 2', '+ 2' }} == {{ case x: 'odd' => '0';
                                                                                 else: '1';
                                                                                 fail: '2', '3' }}]
```

"*Function to...*", "*Function that...*", *etc*. while the code should always contain a single function definition. This requirement exists because Goliath needs to adapt them to the exercise type when the generation occurs, which requires complementing the statement text and running the code's function.

Goliath follows a simple routine to generate an exercise:

1) The type of the exercise is randomly picked within the range of possible options;
2) The value for each key in the statement is randomly chosen;
3) The code is adjusted to the values of the keys;
4) The answer alternatives are calculated;
5) The exercise is constructed from the three components (statement, code and answer alternatives).

The calculation of the answer alternatives demands further explanation. The template carries more than just the statement and the code. It also contains other settings that are used to determine the possible exercise types, the answer alternatives, its correctness, difficulty and category (more on this in Subsection V-C). One of these settings is a list of valid inputs for the function defined in the code. This list is used to generate exercises of the *input/output* type, which read "*What is the output of the following function when the input is ...?*". It is also used to check the student's answer by comparing it to the output of the function when fed with the valid input.

The incorrect alternatives for the other two types of exercises are constructed using the values of the `fail` clauses in the code. In the end, the presence or absence of valid inputs and `fail` clauses determine which exercise types can be generated. Listing 10 shows one example for each type of exercise based on the template of Listing 9. In these examples, `even` was chosen for the key `x` of the statement.

Listing 10 Three exercise types generated from the same template.

Code selection:

```
Which of these options is a function that removes
    all even numbers from a list of integers.

a) def remove_even_numbers(nums):
       return [n for n in nums if n % 2 == 0]
b) def remove_even_numbers(nums):
       return [n for n in nums if n % 2 == 1]
c) def remove_even_numbers(nums):
       return [n for n in nums if n * 2 == 0]
d) def remove_even_numbers(nums):x
       return [n for n in nums if n % 2 == 3]
```

Input/output:

```
What is the output of the following function when
    the input is [1, 2, 3, 4, 5].

def remove_even_numbers(nums):
    return [n for n in nums if n % 2 == 1]

Answer: _______
```

Code completion:

```
Which of these options complete the following
    function that removes all even numbers from a
    list of integers.

def remove_even_numbers(nums):
    return [n for n in nums if n % 2 == ___]

a) 1
b) 0
c) 2
d) 3
```

C. Goliath's Interface

Goliath was implemented as an online application using a mix of technologies and languages (a Programming Cocktail) containing Python, Flask, Lark and MongoDB. Its interface follows the general workflows presented in Fig. 2, with a few added pages to manage users, control access to the several parts of the application, define assignments, and manage templates and exercises.

The main pages of the application are the *AI suggestion page*, the *template edit form*, the *exercises management page* and the *exercise page*.

The AI suggestion page (Fig. 4) is the starting point to the creation of a new template in which the teacher inputs keywords for the KeyToText model to process and suggest an statement. After review (the lower input field), the teacher can send this statement for the CodeT5 to generate the associated code and fill the next page, the template edit form.

The template edit form (Fig. 5) allows the teacher to edit

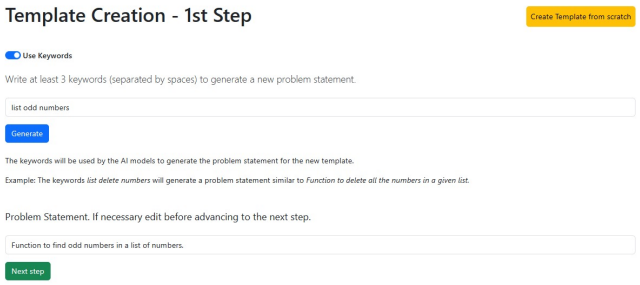

Fig. 4. Keyword input for the AI model.

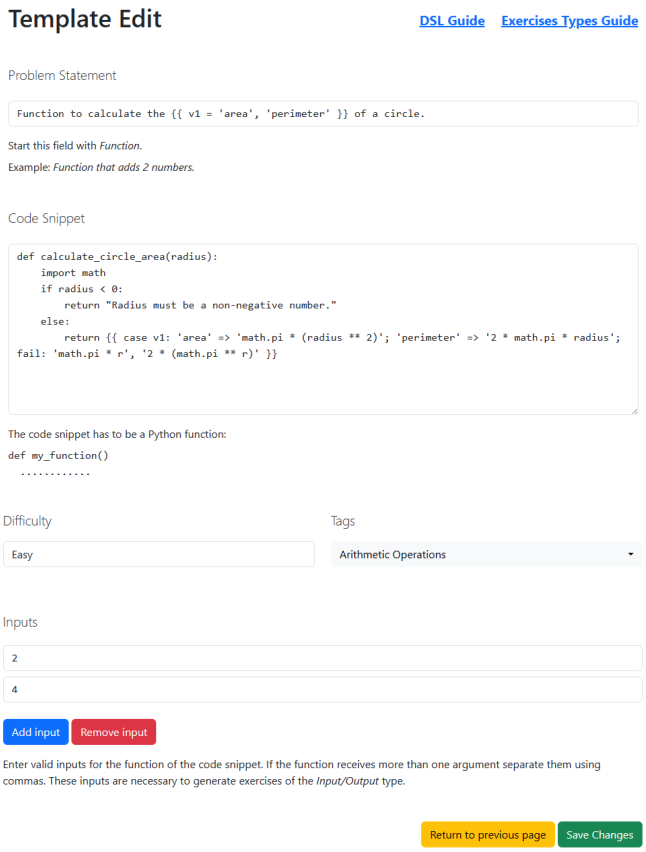

Fig. 5. Template edit form.

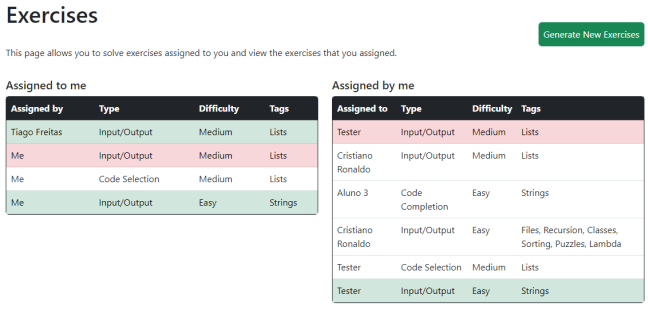

Fig. 6. Exercises management page.

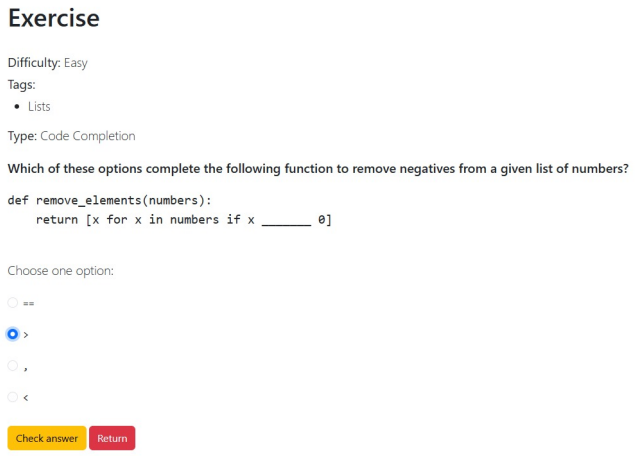

Fig. 7. Exercise page.

the statement, the code and the other settings for the template. It comes after the AI models have made their suggestions (or if they were skipped entirely). The settings for the template are:

- **Difficulty:** a difficulty level from easy to hard designed to guide students in approaching the easier exercises first. Defined entirely by the teacher;
- **Tags:** a form of categorization for the template. Specifies the general programming concepts that the exercises of this template will tackle;
- **Inputs:** a list of valid inputs for the function defined in the code. As explained at the end of Subsection V-B, these inputs will be used to generate and verify the answer to exercises of the *input/output* type.

The exercises management page (Fig. 6) has two points-of-view: the teacher has access to the right table and the *Generate New Exercise* button, while the student only sees and interacts with the left table.

The right table lists exercises that have been assigned to students and it is only visible to teachers. Each entry in the table represents one assignment and its background color indicates if the student was correct (green), incorrect (red) or still have not answered (white). The button labeled *Generate New Exercises* is used to create the assignments.

The left table is only visible for users with the student role, as they represent exercises that have been assigned to them by the teachers. In fact, the exercise requests explained in Subsection V-B effectively occurs when a student first opens entries from this table.

Finally, the exercise page (Fig. 7) allows the student to answer an exercise and obtain immediate feedback of his or her response. The image shows an exercise of type *code completion* begin answered.

VI. Tests and Feedback

In order to evaluate Goliath's functionality, a survey was conducted with teachers from Computer Programming back-

ground. The survey consisted in ten questions: two for establishing the respondent's background, seven to evaluate Goliath's features and usability—in a scale from 1 (terrible) to 5 (excellent)—and a final open question for additional remarks and feedback.

The questions, in order of appearance, were:

1) Programming experience (in years).
2) Experience in programming teaching (in years).
3) Regarding general usability, how do you rate the ease of navigation and interaction with the application?
4) How do you rate the ease of generating text (instructions and code) from the AI models?
5) How do you rate the quality of the text generation results (instructions and code) of the AI models?
6) How do you rate the way the templates are structured?
7) How do you rate the influence of the DSL in generating different exercises from the same template?
8) How do you rate the ease of using the DSL?
9) How do you rate the quality of the generated exercises?
10) In this section, you are asked to comment on any aspects not covered by the questions and to report errors/bugs that have appeared while using the application.

The survey was answered by 10 people and the results produced the charts in Fig. 8.

The first two questions revealed that the survey was answered by people with different levels of experience, including those that have never taught programming. This result indicates that the evaluation for the next seven questions are not skewed towards teachers, but a general overview of the application.

The results were mostly positive, especially in usability and ease of use of the main functionalities. The question with the lowest rating is the fifth (*Quality of the text generated by the AI models*), which indicates that the AI models have not been completely effective. Although, some respondents also indicated that the lower grades where due to instability of the models. The processes that run the models hung up the application a few times when the server was under heavier load from other sources. Nevertheless, the evaluations show that, despite a few inefficiencies, their preparation and the detailed processing of the input and output texts, resulted in average to good results.

The feedback obtained from the last question mainly centered around suggestions of features that would complement the application, such as the possibility to create entire tests inside the application, and even a layout suitable for printing. Small bugs were also reported, which were addressed, making the application as consistent as possible.

VII. Conclusion

This paper presented Goliath, an online application aimed at supporting teachers and students in the practice of Computer Programming. It is based on two AI models to kickstart the construction of programming exercise templates. The templates are parameterised to generate different versions of an exercise. This is done through commands of a DSL that was developed specifically for Goliath. Exercises are generated *ad hoc* when a students request them via assignments created by their teachers. Also, the exercises may be requested by the teachers and used indirectly in their classes or to compose offline lists.

Through a testing period and a survey, results showed that Goliath is already in a working state, capable of supporting teachers in their educational endeavours. A few facets of the User Experience can be improved in order to allow for more efficiency and efficacy in the resulting exercises. Overall, Goliath fulfilled its foundational goal of using AI models in a supportive way, while also providing teachers with high level flexibility and control in the entire process.

Suggestions for future works include both new features and improvements to the existing ones. Among the new features, new modules to apply Goliath to tests and other practice-oriented situations would be beneficial for teachers, supervisors and tutors. Also, some mechanisms to provide greater independence to students would be of great value, such as the automatic generation of complete lists of exercises based on their history and background. This features would also free teachers from the assignment task, which could stimulate Goliath's adoption in the educational setting. On the improvement side, the quality of both the statement and the code generated by the AI models should be improved, in order to obtain more variety and complexity in their suggestions. Finally, reliability and efficiency in the communication between Goliath's internal parts and the models could also be improved.

References

[1] A. Gomes and A. J. Mendes, "Learning to program: Difficulties and solutions," Proceedings of the 2007 International Conference on Engineering and Education (ICEE). International Network on Engineering Education and Research, 2007, pp. 283–287. [Online]. Available: http://icee2007.dei.uc.pt/proceedings/papers/411.pdf

[2] J. Figueiredo and F. J. García-Peñalvo, "Building skills in introductory programming," F. J. García-Peñalvo, Ed., Proceedings of the Sixth International Conference on Technological Ecosystems for Enhancing Multiculturality. New York: ACM, 10 2018. doi: 10.1145/3284179. ISBN 9781450365185 p. 46–50. [Online]. Available: https://dl.acm.org/doi/10.1145/3284179.3284190

[3] M. J. V. Pereira and P. R. Henriques, "Visualization/animation of programs in alma: Obtaining different results," in *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments*, 2003. doi: 10.1109/HCC.2003.1260242 pp. 260–262. [Online]. Available: https://ieeexplore.ieee.org/document/1260242

[4] R. R. Fenichel, J. Weizenbaum, and J. C. Yochelson, "A program to teach programming," *Communications of the ACM*, vol. 13, pp. 141–146, 03 1970. doi: 10.1145/362052.362053. [Online]. Available: https://dl.acm.org/doi/10.1145/362052.362053

[5] S. A. Robertson and M. P. Lee, "The application of second natural language acquisition pedagogy to the teaching of programming languages: a research agenda," *ACM SIGCSE Bulletin*, vol. 27, no. 4, p. 9–12, 12 1995. doi: 10.1145/216511. [Online]. Available: https://dl.acm.org/doi/10.1145/216511.216517

[6] M. V. P. Almeida, L. M. Alves, M. J. V. Pereira, and G. A. R. Barbosa, "Easycoding: Methodology to support programming learning," R. Queirós, F. Portela, M. Pinto, and A. Simões, Eds., vol. 81, Open Access Series in Informatics (OASIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 06 2020. doi: 10.4230/OASIcs.ICPEC.2020.1. ISBN 978-3-95977-153-5. ISSN 2190-6807 pp. 1–8. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12288

[7] Python Tutor, "Python tutor." [Online]. Available: https://pythontutor.com

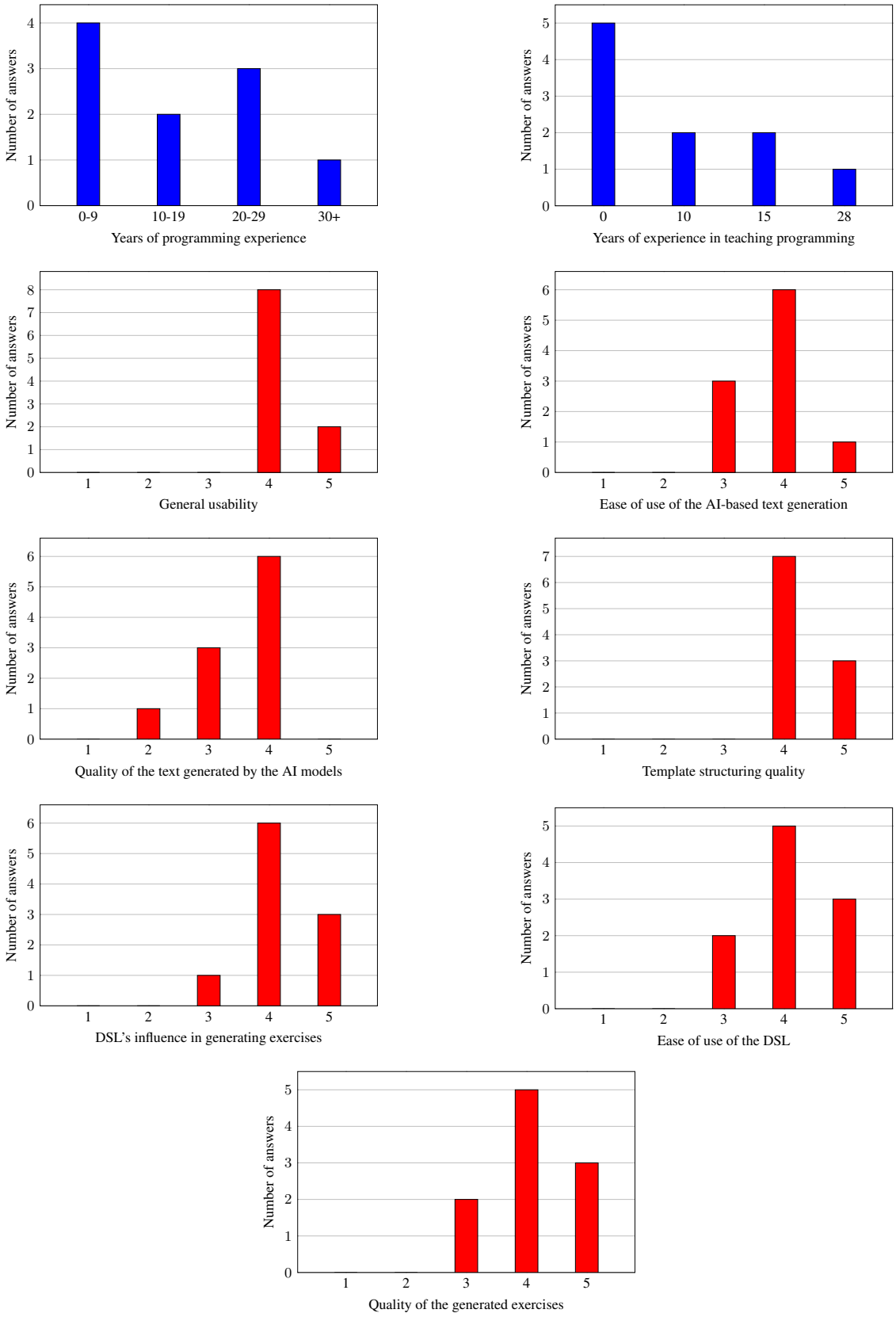

Fig. 8. Results from the survey.

[8] beecrowd, "beecrowd," 2024. [Online]. Available: https://beecrowd.com
[9] A. Rios, J. L. Pérez de la Cruz, and R. Conejo, "Siette: Intelligent evaluation system using tests for teleeducation," in *WWW-Based Tutoring Workshop at 4th International Conference on Intelligent Tutoring Systems*, 1998. [Online]. Available: https://www.siette.org
[10] A. Zeileis, "R/exams." [Online]. Available: https://www.r-exams.org
[11] M. Bower, "A taxonomy of task types in computing," *SIGCSE Bull.*, vol. 40, no. 3, p. 281–285, jun 2008. doi: 10.1145/1597849.1384346. [Online]. Available: https://doi.org/10.1145/1597849.1384346
[12] A. Ruf, M. Berges, and P. Hubwieser, "Classification of programming tasks according to required skills and knowledge representation," vol. 9378, 09 2015. doi: 10.1007/978-3-319-25396-1_6. ISBN 978-3-319-25395-4
[13] N. Ragonis, "Type of questions - the case of computer science," *Olympiads in Informatics*, vol. 6, pp. 115–132, 01 2012.
[14] A. Simões and R. Queirós, "On the Nature of Programming Exercises," in *First International Computer Programming Education Conference (ICPEC 2020)*, ser. OpenAccess Series in Informatics (OASIcs), R. Queirós, F. Portela, M. Pinto, and A. Simões, Eds., vol. 81. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/OASIcs.ICPEC.2020.24. ISBN 978-3-95977-153-5. ISSN 2190-6807 pp. 24:1–24:9. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12311
[15] E. Reiter and R. Dale, *Building Natural Language Generation Systems*. Cambridge University Press, 2000.
[16] ——, "Building applied natural language generation systems," *Natural Language Engineering*, vol. 3, 03 2002.
[17] A. Celikyilmaz, E. Clark, and J. Gao, "Evaluation of text generation: A survey," *CoRR*, vol. abs/2006.14799, 2020. [Online]. Available: https://arxiv.org/abs/2006.14799
[18] IBM, "What is an AI model?" 2024. [Online]. Available: https://www.ibm.com/topics/ai-model
[19] ——, "What are Neural Networks?" https://www.ibm.com/topics/neural-networks, 2023.
[20] ——, "What are Recurrent Neural Networks?" https://www.ibm.com/topics/recurrent-neural-networks, 2023.
[21] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850
[22] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks," 2015. [Online]. Available: http://karpathy.github.io/2015/05/21/rnneffectiveness/
[23] P. Dugar, "Attention — seq2seq models," https://towardsdatascience.com/day-1-2-attention-seq2seq-models-65df3f49e263, 2019.
[24] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 3104–3112.
[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017. ISBN 9781510860964 p. 6000–6010.
[26] IBM, "What is Unsupervised Learning?" https://www.ibm.com/topics/unsupervised-learning, 2023.
[27] OpenAI, "OpenAI," https://www.openai.com/product, 2023.
[28] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165
[29] OpenAI, "Gpt-4 technical report," 2023.
[30] ——, "ChatGPT," https://openai.com/blog/chatgpt, 2023.
[31] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019. doi: 10.18653/v1/n19-1423 pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423
[32] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," 2020.
[33] S. Ali, D. DiPaola, and C. Breazeal, "What are gans?: Introducing generative adversarial networks to middle school students," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 17, 2021. [Online]. Available: https://par.nsf.gov/biblio/10252915
[34] Y. Zhang, Z. Gan, K. Fan, Z. Chen, R. Henao, D. Shen, and L. Carin, "Adversarial feature matching for text generation," 2017. [Online]. Available: https://arxiv.org/abs/1706.03850
[35] T. Cemgil, S. Ghaisas, K. Dvijotham, S. Gowal, and P. Kohli, "The autoencoding variational autoencoder," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 15 077–15 087. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/ac10ff1941c540cd87c107330996f4f6-Paper.pdf
[36] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Józefowicz, and S. Bengio, "Generating sentences from a continuous space," *CoRR*, vol. abs/1511.06349, 2015. [Online]. Available: http://arxiv.org/abs/1511.06349
[37] PyPI, "OpenAI," https://pypi.org/project/openai/, 2023.
[38] TensorFlow, "Why TensorFlow," https://www.tensorflow.org/about, 2023.
[39] Keras, "Keras documentation: About Keras," https://keras.io/about/, 2023.
[40] J. Terra, "Pytorch Vs Tensorflow vs Keras," https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article, 2023.
[41] HuggingFace, "Transformers," https://huggingface.co/docs/transformers/index, 2023.
[42] M. Wolf, "Textgenrnn," https://github.com/minimaxir/textgenrnn, 2020.
[43] G. Bhatia, "keytotext." [Online]. Available: https://github.com/gagan3012/keytotext
[44] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685
[45] T. C. Freitas, A. Costa Neto, M. J. a. V. Pereira, and P. R. Henriques, "NLP/AI Based Techniques for Programming Exercises Generation," in *4th International Computer Programming Education Conference (ICPEC 2023)*, ser. Open Access Series in Informatics (OASIcs), R. A. Peixoto de Queirós and M. P. Teixeira Pinto, Eds., vol. 112. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/OASIcs.ICPEC.2023.9. ISBN 978-3-95977-290-7. ISSN 2190-6807 pp. 9:1–9:12. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2023/18505
[46] w3resource, "Python exercises, practice, solution," https://www.w3resource.com/python-exercises/, 2023.
[47] M. Woolf, "aitextgen," https://docs.aitextgen.io/, 2021.
[48] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, jan 2020.
[49] J. Austin, A. Odena, M. Nye *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.
[50] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc., 2009.
[51] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436
[52] E. Shinan, "Lark." [Online]. Available: https://lark-parser.readthedocs.io/en/stable/