

AStrId: An Interpretable Code Representation and Distance for Classifying Student Solution Strategies

Guillaume Steveny
ICTEAM institute
Université catholique de Louvain
UCLouvain
Louvain-la-Neuve, Belgium
guillaume.steveny@uclouvain.be

Kim Mens
ICTEAM institute
Université catholique de Louvain
UCLouvain
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Siegfried Nijssen
DTAI, Department of Computer Science
Katholieke Universiteit Leuven
KU Leuven
Leuven, Belgium
siegfried.nijssen@kuleuven.be

Abstract—To provide more effective feedback, instructors can benefit from automatically classifying student programs according to underlying solution strategies. We propose an automated classification approach based on **ASTrId**, a novel abstract code representation, combined with a tree edit distance that enables structured comparison between student programs and representative strategy implementations. **ASTrId** balances *classification performance*, *instructor effort*, and *interpretability* by capturing the essential structural complexity of programs while abstracting away *Semantic-Preserving Variations*. We evaluate our approach against two alternative representations based on JPlag and ASTs, using three training set configurations with varying labelling effort. The evaluation is conducted on 1454 programs from seven exercises in an introductory Python programming course. Our results show that **ASTrId** achieves strong classification performance (F1-scores exceeding 0.9), even with reduced training data, thereby limiting instructor effort. At the same time, it preserves interpretability by providing insights into how and why student programs are matched to specific solution strategies.

Index Terms—Program analysis, Python, Abstract program representations, Tree edit distance, Programming education, Programming strategies, Classification, Artificial intelligence.

I. INTRODUCTION

A. Context and Motivation

Introductory programming courses, including the ones taught at our host institution, often rely on project-based learning and frequent hands-on coding exercises. A single programming exercise can typically be solved using multiple distinct solution patterns [1]. They represent the various constructs and strategies [2], [3] or schemas [4] a student can pick to solve the exercise. Students select such strategies based on their prior knowledge and experience. E.g., whereas one student may implement a recursive strategy, another might prefer using an iterative loop. Since each strategy exhibits distinct structural and behavioural characteristics, this gives rise to different classes of mistakes and requires different forms of feedback. For instance, recursive solutions may suffer from missing base cases, whereas iterative ones must be checked for off-by-one errors.

Identifying which strategy a student used is essential for instructors to provide accurate, targeted feedback [5]. Most

existing assessment and tutoring systems [6] evaluate only correctness or test compliance. They lack the ability to infer the underlying solution strategy chosen by a student. Without this information, instructors cannot distinguish general mistakes from pattern-specific misconceptions, limiting the precision and usefulness of the feedback they can provide. Some prior work has attempted to cluster student programs into solution patterns for retrospective analysis [1], [7], but does not support real-time, strategy-aware classification during automated grading.

B. Solution Idea and Approach

To address this gap, we study an approach to automatically identify the solution strategy in students' programs, in the context of an undergraduate introductory Programming course in Python. Novice students are likely to adopt suboptimal behaviours in their coding practice and can exhibit a wide variety of different solution patterns. We do make a simplifying assumption, however, that each student's program corresponds to one particular solution strategy at a time. In our study setting, exercises are relatively small, often requiring less than 20 lines of code, and cover only basic language concepts and algorithmic principles. This restricted context arguably makes the definition, detection and classification of possible solution patterns easier.

Our approach is driven by three central requirements: high classification performance (in terms of precision and recall), strong interpretability and reduced instructor effort. An instructor should be able to understand easily why a given student's program gets classified in a certain manner; moreover, the effort required from the instructor to build a classifier should be small. To meet these requirements, we propose a structural code representation and distance-based classification approach, where new student programs are compared to stored programs, for which the solution patterns are known a priori, based on the computed distance. Within this approach, two independent technical dimensions will be explored.

[D1] Which code representation and associated distance metric to use to compare programs? Choosing a code representation at the right abstraction level is important for comparing program structures, since a *too detailed* representation

can make it hard to distinguish strategies. For instance, while we may wish to distinguish the presence of recursive calls from iterative control flow, we may not wish to distinguish different variants of looping structures. We propose `AStrId`, an acronym for “**AST** for **Strategy Identification**”, as an abstract code representation based on generalised Abstract Syntax Trees. This abstraction aims at finding the sweet spot between *interpretability*, by remaining close to the student’s original code, and structure abstraction, by focusing on a program’s “*essential complexity*” while ignoring its “*accidental complexity*” [8]. In particular, we envision `AStrId` to be tolerant to specific “*Semantic-Preserving Variations*” (SPVs) [7], [9]. Section III-A will describe `AStrId`’s design choices and discuss which SPVs it inherently supports (see Table I). To validate our choices, we will compare `AStrId` against two alternatives: a JPlag-based [10] and a standard AST-based approach [11]. JPlag, a state-of-the-art system for abstract code-structure similarity computation, is often used by instructors in programming courses to flag plagiarism. Raw ASTs are used by computer systems to describe the structure of a program without preserving information irrelevant for its actual execution. Section III-B will discuss the differences between these approaches and `AStrId`.

[D2] Which programs should be stored to perform the classification of new student programs? Using the abstract code representations described in dimension **D1** and their associated distance metric, we will classify new student programs by comparing them to stored programs. We wish to minimise the effort that instructors have to spend to label stored programs. This second dimension **D2** explores 3 different approaches for determining the labelled reference programs to be stored:

Full-corpus: The first approach based on k -Nearest Neighbours (k -NN) [12] assumes that a large number of student programs are available and have already been manually labelled by an expert. We then use the distances to all these labelled programs to classify new student programs.

k -medoids: A second approach clusters student programs using k -medoid clustering [13], yielding a subset of instances representative of all student programs. These medoids are subsequently labelled by an expert as reference solutions for specific strategies. Only the labelled medoids are then used to classify student programs, again using k -NN as in the full-corpus approach.

Instructor-written: In the third approach, an expert instructor manually writes reference programs that represent each of the strategies expected to be detected.

Section III-C will describe each of these approaches, their benefits and limitations, in more detail.

C. Evaluation Criteria

We will evaluate and compare each combination of dimensions **D1** (`AStrId`, JPlag, AST-based) and **D2** (full-corpus, k -medoids, and instructor-written) on 3 main criteria:

[C1] Classification Performance. Based on the F1-scores obtained by each combination of **D1** and **D2**, when classifying

large amounts of labelled programs, we can evaluate the effectiveness of how programs are matched to classes representing different solution patterns.

[C2] Labelling Effort. The various approaches described in dimension **D2** come with decreasing labelling effort. The instructor-written approach even allows classification without pre-collecting actual student programs as a reference set. We need to evaluate, however, whether the advantages obtained by reducing labelling effort do not come at a cost of decreased classification performance (**C1**).

[C3] Classification Interpretability. We strive for interpretability of the classification process [**C3a**], obtaining clear reasons why a program gets classified in a certain way, as well as understandability of the mapping [**C3b**] between a program to be classified and its representative solution program.

We aim to find the best combination of **D1** and **D2** based on these 3 criteria. From our evaluation in Section IV, we observe that `AStrId` enables efficient classification with the configuration that requires the least labelling effort (i.e. “instructor-written”) and is most interpretable, thus reaching the best tradeoff regarding criteria **C1**, **C2** and **C3a**. In addition, `AStrId` obtains results comparable to JPlag and ASTs for the other configurations but with improved interpretability levels, thus being optimal regarding criterion **C3b**.

D. Paper Structure

The remainder of this paper is structured as follows. Section II presents work related to code similarity and solution pattern identification within datasets of student programs. Section III describes the `AStrId` representation and its associated structure-matching metric, and compares it with the JPlag and AST-based approaches (**D1**). This section also covers the three approaches for classification based on k -NN (**D2**). Section IV describes the evaluation we conducted and describes its results in terms of the three evaluation criteria we defined. Section V revisits our key findings, present limitations of our experiments and paths for future work.

II. RELATED WORK

A. Code Similarity Measures

To measure similarity between code fragments, one can rely on token-based approaches, like JPlag [10], MOSS [14], CP-Miner [15], or SCSDS [16]. Systems in this category transform a program’s statements into token streams and compare these according to their respective token types. However, relying solely on token types can still miss essential information. We will compare our `AStrId` approach in detail with JPlag [10] as a prominent open-sourced representative of this category [17].

Alternatively, tree-based approaches such as Out-of-step [18], Deckard [19], or those of Song *et al.* [11] or Wen *et al.* [20] represent programs as abstract syntax trees (ASTs), enabling fine-grained structural comparison. Using tree edit distances on ASTs naturally captures insertions, deletions, and moves of syntactic constructs, making them well-suited for structural similarity metrics. Our own `AStrId` representation

builds directly on this idea. We will compare `AStrId` in particular with the AST representation of Song *et al.* [11].

Ragkhitwetsagul *et al.* [21] and Zakeri-Nasrabadi *et al.* [22] present multiple alternative approaches, like metrics-based and text-based. We did not consider these two classes of approaches since we want to compare programs at a slightly higher level of abstraction, where superficial syntax plays a lesser role, and the focus lies on capturing the essence of the underlying solution strategy.

Zhang *et al.* [23] provide a recent systematic review of machine-learning-based similarity approaches. Despite their prevalence, such embedding or model-based methods tend to obscure the reasoning behind similarity judgments, limiting interpretability (C3) and making it difficult to identify which concrete program differences drive model decisions.

B. Identification of Solution Strategies in Student Programs

Prior work has focused on identifying, from existing datasets of student programs, variations in how students approach and solve programming exercises: OverCode [1], Codewebs [7], and those of Artser *et al.* [24], and Keuning *et al.* [3]. However, these tools are designed for a posteriori data exploration and depend on the availability of a substantial dataset of student programs beforehand [1], [7], [24].

Whereas we treat normalisation as an abstract structural representation, Keuning *et al.* [3] apply the *Semantic-Preserving Variations* of Xu *et al.* [9] as transformation rules, repeatedly applying them until no further normalisation is possible. Like OverCode [1], they treat similarity as a binary relation: programs are either fully equivalent in their normalised form or entirely dissimilar. Code pattern matchers [25]–[28] also suffer from this binary relation issue, since outputted elements have to be perfect matches. `AStrId` overcomes this problem by relying on a tree edit distance to offer insights into what makes programs distinct or nearly similar. Finally, unlike Keuning *et al.* [3] who also rely on code execution, `AStrId` only expects code to be parsable into an AST.

III. APPROACH

In this section we first introduce the `AStrId`¹ abstract representation and corresponding structure-matching metric (§III-A) we use to compare programs (D1). We also discuss how it compares to the two other representations and structure-matching metrics (§III-B) we use during our experiments (IV): JPlag [10] and AST-based [11]. We then describe (§III-C) the classification mechanisms employed in our experiments (D2).

Several of our design choices were guided by characteristics of our target setting: undergraduate students taking a large introductory Python programming course. In our context, each exercise can yield hundreds to thousands of submissions, programs are relatively short and make use of simple language constructs, and instructors can identify a representative set of reference solutions for each exercise. These properties influenced both the abstractions we adopted and the scalability and interpretability requirements of our approach.

Table I: Currently supported *Semantic-Preserving Variations* [9] and their `AStrId` equivalence.

Name	<code>AStrId</code> 's support
<i>SPV2</i> : Source code format	AST ignores code formatting details
<i>SPV3</i> : Different syntax forms	Unified representation in AST
<i>SPV5</i> : Algebraic expression forms	EXPR nodes ignore operators
<i>SPV6</i> : Different control structures	Considered <i>compatible</i> (cf. §III-A4)
<i>SPV7</i> : Boolean expression forms	Idem as for <i>SPV5</i> .
<i>SPV10</i> : Different statement orders	Supported by <i>unordered subtrees</i>
<i>SPV11</i> : Different variable names	Assigned variables are normalised

A. `AStrId`'s code representation and distance metric

1) *Semantic-Preserving Variations*: To illustrate `AStrId`'s code representation, consider the Python programs shown in Listings 1 and 2. Despite their surface-level differences, both programs implement variations of the same underlying solution strategy, i.e. a double loop with inner accumulation.

The main differences in these programs are essentially semantic-preserving variations [9] such as reordered statements (*SPV10*), renamed variables (*SPV11*), operator substitutions (*SPV5*), or the transformation of a `for` loop into a `while` loop (*SPV6*). Normalising the programs to ignore such variations [3], [7], [24] would require defining and maintaining a potentially large set of program transformation rules. Instead, we aim for a program representation that abstracts away such accidental implementation details.

`AStrId`'s code representation, illustrated in Listings 3 and 4, is designed to be abstract yet understandable, like a pseudocode version of a student's program. The code is represented in the form of a tree, where nodes can have different types. Depending on the node type either ordered or unordered matching is used. `AStrId`'s representation inherently ignores *Semantic-Preserving Variations*, and its nodes focus on the essential components highlighting variations in solution strategies. Among the 13 *SPVs* described by Xu *et al.* [9], Table I presents the 7 currently supported by `AStrId`.

2) *Code representation*: `AStrId` trees contain two types of nodes: ordered and unordered ones. We can see an `AStrId` structure as an ordered tree-structured forest of *unordered subtrees*. Each ordered node has an ordered list of children, as shown indented below the nodes in Listings 3 and 4. Additionally, each ordered node can have one *unordered subtree*, shown in parentheses in the example listings. These support semantically similar expressions, like `n == 1` and `1 == n`.

For example, the `WHILE` node on line 5 in Listing 4 is an ordered node with two child nodes: a `WHILE` and an `ASSIGNS`. It is also linked to an *unordered subtree* `EXPR(#3, n)` where #3 is the loop Variable. Fig. 1 depicts the structure of these nodes. To manipulate the *unordered subtree* associated to an ordered node, an *unordered subroot* node is created that has a set of unordered nodes as children.

To construct an `AStrId` tree, we first transform the source program into its corresponding AST. Due to the target context of our approach, `AStrId` currently only supports Python. However, we believe our abstract representation to be suffi-

¹Source code: <https://github.com/StevenGuy-research/AStrId> [29]

Listing 1: Example Python program of a function that computes the number of prime factors of $n \geq 1$.

```

1 def factors(n):
2     if n == 1:
3         return 0
4     else:
5         count = 0
6         value = n
7         for i in range(2, n+1):
8             while value % i == 0:
9                 value = value // i
10                count += 1
11            return count

```

Listing 3: AStrId representation of the code in Listing 1.

```

MODULE()
1  DEF(factors , n)
2    IF(EXPR(n, 1))
3      OUT(0)
4    ELSE()
5-6    ASSIGNS(0, n)
7    FOR(2, EXPR(n, 1))
8      WHILE(EXPR(#2, #F1, 0))
9-10     ASSIGNS(EXPR(#2, #F1), EXPR(#1, 1))
11     OUT(#1)

```

Listing 2: A different Python program representing a semantic-equivalent variation of the program in Listing 1.

```

1 def factors(n):
2     val = n
3     c = 0
4     i = 2
5     while i <= n:
6         while val % i <= 0:
7             c = c + 1
8             new_value = value / i
9             val = new_value
10            i += 1
11    return c

```

Listing 4: AStrId representation of the code in Listing 2.

```

MODULE()
1  DEF(factors , n)
2-4  ASSIGNS(n, 0, 2)
5    WHILE(EXPR(#3, n))
6      WHILE(EXPR(#1, #3, 0))
7-9    ASSIGNS(EXPR(#2, 1), EXPR(#1, #3), #4)
10    ASSIGNS(EXPR(#3, 1))
11    OUT(#2)

```

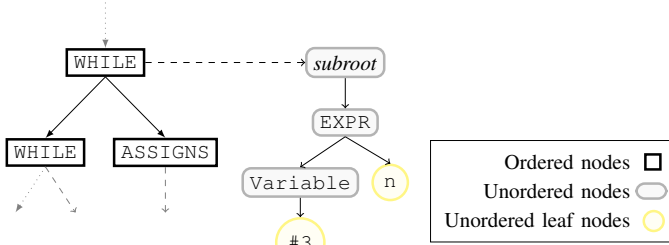


Figure 1: Tree representation of the first WHILE node in Listing 4.

ciently generic to be extendable to other languages, as long as an AST is available and a mapping to AStrId nodes is defined. Our overall conversion process through AST traversal is language-independent. For each node of the AST, we retrieve its AStrId equivalent (WhileNode gives WHILE, DefNode gives DEF, ...). We then check whether the AST node has instance-specific data, like for example, IfNode and WhileNode containing a condition. These node-specific elements are further processed, through AST traversal of these elements, to constitute the *unordered subtree* of the currently mapped AStrId node. The child nodes of the new AStrId structure are constructed from the AST nodes in the body of the currently processed AST node, if any.

To normalise the representation, specific AST source nodes are merged when consecutive or nested in the input code. This is the case for nodes representing assignments and expressions. Since the *subtrees* are unordered, such a merging operation supports commutativity of consecutive instructions and of

expressions' operands. These two node types are also further simplified: assignments lose their target variable (to support SPV10) and expressions their operators (to support SPV5). Further simplifications also impact for loops, which lose their call to range (SPV3), and variables, which are normalised according to their first occurrence in the code (SPV10 and SPV11). While #X represents the X-th variable created by one of the assignments, for loop variables include an additional F to highlight their specific role (cf. #F1 in Listing 3).

Lastly, we discard some AST nodes when creating the mapping into AStrId equivalents. We focus on those nodes that are the most common within student programs in our context of an introductory Python programming course. Discarded AST nodes are mapped to an UNKNOWN node, which contains the type of the original AST node as a single node in its *unordered subtree*. Adding the source type in the node permits saving this information and using it later for the edit-distance computation. For the programs considered in our experiments (Subsection IV-A), only 307 among the 50895 mapped nodes received an UNKNOWN label, representing only 0.6% of the total number of nodes; they represented constructs like match-cases or list comprehensions.

3) *Formalism*: To define the distance computation between two trees, we formalize AStrId trees as a tuple $(\mathcal{N}, \mathcal{V}, \mathcal{L}_{\mathcal{N}}, \mathcal{L}_{\mathcal{V}}, C_{\mathcal{N}}, C_{\mathcal{V}}, U, L)$, where $\mathcal{N}$ is the set of all ordered nodes and $\mathcal{V}$ are the unordered nodes. We denote their union as $\mathcal{NV} = \mathcal{N} \cup \mathcal{V}$. Each set of nodes has its own set of possible labels, $\mathcal{L}_{\mathcal{N}}$, $\mathcal{L}_{\mathcal{V}}$ and $\mathcal{L}_{\mathcal{NV}} = \mathcal{L}_{\mathcal{N}} \cup \mathcal{L}_{\mathcal{V}}$. For each ordered node $n \in \mathcal{N}$, $C_{\mathcal{N}}(n)$ denotes its ordered list of children; for each unordered node $v \in \mathcal{V}$, $C_{\mathcal{V}}(v)$ is its unordered set of children. The mapping between an ordered

node $n \in \mathcal{N}$ and its associated *unordered subtree* is given by $U(n) : \mathcal{N} \rightarrow \mathcal{V}$. Lastly, the label of any node can be retrieved by the function $L(n) : \mathcal{N} \rightarrow \mathcal{L}_{\mathcal{N}\mathcal{V}}$.

Within `ASTrId` trees, we indicate as *subroots* $\mathcal{S} \subseteq \mathcal{V}$ (cf. Fig. 1) the nodes that are in the image of function $U(n)$. These nodes cannot be the child of another unordered node. The set of unordered nodes $\mathcal{V}$ has two other particular subsets: the set of all unordered leaf nodes $\mathcal{F} = \{v \in \mathcal{V} \mid C_{\mathcal{V}}(v) = \emptyset\}$, and the set of all leaf nodes that are the child of a node labelled ‘Variable’: $Var = \{v \in \mathcal{F} \mid \exists v' \in \mathcal{V} : L(v') = \text{‘Variable’}, v \in C_{\mathcal{V}}(v')\}$. We will use these two subsets to define the edit-cost computation.

4) *Distance Computation*: Except for the *unordered subtrees*, we consider the main structure of `ASTrId` trees as ordered. Thus, as a state-of-the-art ordered-tree edit-distance we use *AP-TED* (All-Path Tree Edit Distance) [30], [31], a time- and memory-efficient algorithm for computing edit distances between ordered trees. *UTED*, an Unordered Tree Edit Distance algorithm [32], will constitute the edit-distance algorithm for the *unordered subtrees*. To compute the size of an *unordered subtree*, we use the function $size(x) : \mathcal{V} \rightarrow \mathbb{N} : v \mapsto \sum_{c \in C_{\mathcal{V}}(v)} (1 + size(c))$.

To support semantic equivalence between statements, some matchings of statements can be considered as *compatible comparisons*, even when they have different labels. For example, since both `FOR` and `WHILE` nodes represent loops, we consider them *compatible*, at a small cost penalty to account for the label change operation. Which node labels are *compatible* is defined by a function $compatible(L_n, L_m) : \mathcal{L}_{\mathcal{N}\mathcal{V}} \times \mathcal{L}_{\mathcal{N}\mathcal{V}} \rightarrow \text{Boolean}$. In case of compatibility, the associated penalty between nodes $n, m \in \mathcal{N}\mathcal{V}$ is given by $penalty(n, m) = 0.5$ if $L(n) \neq L(m)$ and 0 otherwise. Other than (`for` and `while`) loops and nodes having the same label, other *compatible comparison* groups are calls (to function or methods) and conditions (in `if` and `while` statements).

AP-TED defines three basic operations: inserting, deleting or renaming a node (zero-cost operation if the nodes are the same), each with a corresponding cost defined:

$$insert_A(n) = delete_A(n) = 1 + size(U(n)) \quad (1a)$$

$$rename_A(n, m) = \begin{cases} penalty(n, m) + UTED(U(n), U(m)) & \text{if } compatible(L(n), L(m)) \\ delete_A(n) + insert_A(m) & \text{otherwise} \end{cases} \quad (1b)$$

UTED uses the following three cost operations:

$$insert_U(v) = delete_U(v) = 1 \quad (2a)$$

$$rename_U(v, w) = \begin{cases} 0.5 & \text{if } v, w \in Var \wedge L(v) \neq L(w) \\ 1 & \text{if } L(v) \neq L(w) \\ 0 & \text{otherwise} \end{cases} \quad (2b)$$

Note that the *rename* cost function (2b) restricts the cost for renaming nodes representing variables with different labels to

0.5, to limit the impact of assignment reorderings; otherwise, the rename cost for non-matching labels is 1.

B. Comparison with the selected structure-based metrics

1) *JPlag’s Similarity Metric*: JPlag is primarily used to flag potential cases of plagiarism by using a similarity score computed between programs represented as “token strings”. Listing 5 shows the token string for the program in Listing 1. By relying solely on structural information, JPlag intentionally ignores superficial code features that students might modify in an attempt to circumvent plagiarism detection.

Unlike `ASTrId`, which stores part of the information in *unordered subtrees*, the tokens that JPlag uses pose limitations for interpretability. Two matching `ASSIGN` tokens could correspond to fundamentally different assignment statements in the original programs. For example, `a+=["x"]` and `b=2` are mapped to the same `ASSIGN` token, while for `ASTrId` we have `ASSIGNS(EXPR(#1, SEQ("x")))` and `ASSIGNS(2)`. Comparing the two statements, JPlag gives a similarity score of 100%, which is not desirable in this case, while `ASTrId` indicates that 4 edit operations are required to go from one statement to the other. More generally, this observation highlights the weakness of similarity scores over edit-cost metrics, as for `ASTrId`. The former gives only a rough estimation of the similarity of programs, while the latter naturally reflects the actual sequence of operations required to transform one program into another.

2) *AST-Based Metric*: To obtain a more detailed view of the code than with JPlag’s tokens, we can directly use the program’s AST, instead of further abstracting it into its `ASTrId` equivalent. On such a structure, we can also leverage tree edit operations, rather than a single similarity score. However, raw ASTs still preserve syntactic details that may obscure the underlying solution strategy of a program. Minor, strategy-irrelevant variations in a program, e.g. variable’s names, can lead to disproportionately large structural differences in the AST representation. To address this, we adopt an additional abstraction layer on top of the AST nodes: the representation introduced by Song *et al.* [11] for `TSED`.

Listing 6 shows a fragment of the AST derived from the program in Listing 1. As opposed to the authors’ original implementation, which uses *AP-TED* as we did for the ordered nodes of `ASTrId`, we use *UTED* to compute the tree edit-distance. This choice has been made due to the impact of fully ordered trees in our context: reordering independent statements does not affect program semantics (SPV10), yet an ordered-tree metric does not support it, while *UTED* will.

Unlike `ASTrId`, abstract nodes such as `identifier` still hide the concrete code elements they originate from, making it harder to understand why two nodes match. The matching process may even align nodes that are syntactically or semantically unrelated but share the same abstract label. An `identifier` node within an assignment statement could match any other such `identifier` node, regardless of whether it plays a similar role in the program’s solution. For `ASTrId`, (ordered) nodes will only be compared if they

Listing 5: JPlag representation of the program in Listing 1. Indentation is merely for readability purposes.

```
METHOD_BEGIN,
  IF_BEGIN,
    RETURN,
    ASSIGN,
    ASSIGN,
    FOR_BEGIN,
      APPLY,
      WHILE_BEGIN,
        ASSIGN,
        ASSIGN,
      WHILE_END,
    FOR_END,
    RETURN,
  IF_END,
METHOD_END,
FILE_END
```

Listing 6: Slice of the abstracted AST generated from Listing 1, covering the beginning of the first loop.

```
...
  (assignment
    left: (identifier)
    right: (identifier))
  (for_statement
    left: (identifier)
    right: (call
      function: (identifier)
      arguments: (argument_list
        (integer)
        (binary_operator
          left: (identifier)
          right: (integer))))
    body: (block
      (while_statement
        condition: ...
```

are considered as ‘compatible’, in which case they can make use of the more detailed information available in the node’s *subtrees* for computing similarity.

C. Classification Mechanisms

Having described the structure-based metrics (**D1**), we now turn to the classification approaches (**D2**). Each such approach can be characterised with respect to the **Labelling Effort (C2)** and **Classification Process Interpretability (C3a)** criteria (see I-C), independently of dimension **D1**.

Full-corpus: This approach relies on pre-collected students’ programs to automatically classify the implemented strategies. An expert instructor is expected to have manually identified the different strategies in an entire dataset. A k -NN classifier [12] is subsequently used for classifying future submissions, with $k \geq 1$, using the corpus as a training set. This approach, even though more robust due to the diversity contained in the training set, suffers from two weaknesses. Results are less interpretable when $k > 1$, as multiple codes have to be analysed to understand a prediction, which is bad for **C3a**. Moreover, the cumbersome manual labelling process makes it hard for instructors to specify additional non-represented strategies (bad for **C2**).

k -medoids: Here, it is assumed that a dataset is not fully labelled by an instructor. This approach relies on the k -medoid clustering algorithm [13] to find k representative unlabelled codes, which are then labelled by the instructor. This reduces the labelling effort to k instances highly representative of groups of student submissions. These instances will constitute the training set of a k' -NN classifier with $k' = 1$. With this approach, the predictions can be interpreted more easily, as they are based on a smaller number of labelled programs (positive for **C2**). However, it still requires an already available dataset of students’ programs, which are not always pedagogically meaningful representatives of their underlying solution strategy.

Instructor-written: In this last approach, an instructor writes one reference program for each specific solution pattern to

be detected, like in the approach of Keuning *et al.* [3]. These programs form a training set for a k -NN classifier with $k = 1$. Relying on this configuration further reduces the labelling effort (positive for **C2**) as model solutions are already implemented at exercise creation for comparison with student-produced programs. Still it assumes that instructors are able to determine a large set of possible strategies.

A pre-collected dataset of student submissions is not required to perform classification, enabling the use of the classification on never-tried or newly-implemented exercises. This approach also provides the instructor with freedom to tune the process to detect underrepresented strategies by simply extending the set of already implemented reference solution patterns. Lastly, due to the 1-NN classifier’s nature, as with the k -medoids approach, results are more interpretable as each student’s submission is associated with a single highly representative and pedagogically meaningful implementation of a solution pattern (positive for **C3a**).

In conclusion, for dimension **D2**, based on criteria **C2** and **C3a**, we should prefer the k -medoids and instructor-written methods to construct the reference set for classification. However, since they limit the training set for k -NN, these could negatively impact **Classification Performance (C1)** when considered along the dimension **D1**. The next section will evaluate if this negative impact is observed in practice.

IV. EXPERIMENTS

A. Data Collection and Labelling

To conduct our experiments, we collected 1454 programs that students submitted across seven programming exercises from a final exam of an introductory Python course. Each exercise is estimated to take 30 minutes and requires writing a single Python function covering basic programming constructs, like strings, loops, lists, files, or dictionaries. To evaluate the submissions, each assignment benefits from highly robust unit tests. Even though we can classify incorrect programs using their closeness to specific solution patterns, since they are not

solutions, defining their ground truth label for classification performance measurements is difficult. To address this, the 1454 programs selected are all successful submissions. Table II summarises the exercises and the classes found.

Regarding the use of student-produced programs, students were prompted, at the start of the academic year, to agree to have their anonymised programs used for research purposes. Such use has been acknowledged and validated by our institution in collaboration with our ethics committee. Due to confidentiality requirements, the dataset cannot be made entirely public; we can share it upon request under specific conditions.

For each of the exercises, we manually inspected all submitted programs and defined solution-strategy classes during the labelling process. Among the common solution strategies students used, some classes were not representative enough (containing less than 2% of the instances of an exercise) or too idiosyncratic, typically representing intricate and uncommon solution strategies devised by only one or a few students. For example, one student produced an unusual Fibonacci solution that splits the computation into separate procedures for odd and even values. Such cases can be considered outliers and are better discarded. Among these, we also have instances labelled as `side_function`: students who used a variety of auxiliary functions with complex behaviours as part of their solution. Except for these unusual solution strategies (representing 3.4% of the instances in all our datasets), for which it was difficult to identify a single representative solution pattern, we created one manually crafted reference solution for each remaining class. These reference solutions were designed to reflect the label of the class and to capture the essential strategy that characterises most programs that could belong to it.

B. JPlag configuration

To use JPlag for our experiments, we set all its parameters to their default values [33], except for the `minTokens` parameter, for which its value has been set to 2. Using too high values for this parameter leads to programs being ignored by JPlag, as JPlag discards programs if they have fewer tokens than `minTokens`. After multiple tries with values between 2 and 12 (default) for this parameter, we observed that 2 was the best performing in our experiments. For example, in the full-corpus experiment on the **encoding** exercise, the classifier achieved an average F1-score of 0.272 with `minTokens` set to its default value, while it improved to 0.617 with this parameter set to 2.

C. Configuration of Dimension D2 Variants

Full-corpus: We classify the instances with a k -NN classifier with $k = 5$. This value has been selected after comparison with other possible values within the range from 3 to 9. For the three metrics, the impact of k on the full-corpus classification was not significant, oscillating with values of ± 0.05 around the scores found with $k = 5$.

k -medoids: In this configuration, the training sets are the medoids found by clustering the datasets based on the distance values obtained by each **D1** metric. We set $k = 20$ for

the k -medoid clustering to ensure that multiple medoids are selected to cover each class. Then, a 1-NN classifier, using the same metric as for clustering, will classify instances based on the selected medoids. Listing 7 presents one of the medoids extracted from the **fibonacci** exercise dataset.

Instructor-written: In this experiment, the training sets of the 1-NN classifier contain only instructor-written programs, each representing a single class. Each of these representative programs focuses on the “essential complexity” that a program should depict to be considered as belonging to that strategy class. Listings 8 and 9 present two examples of instructor-written representative programs for the **fibonacci** exercise.

D. Validation Process

For the full-corpus and k -medoids configurations, we use a k -fold cross-validation approach as a validation process. We split each exercise dataset into 10 training-test set pairs, where each test set contains 10% of its original dataset. We also ensure that for k -medoids, the clustering is only applied on the training set to extract the representative instances in order to avoid indirect bias where the test instances would influence the clustering process. For the instructor-written configuration, we do not use k -fold validation since the instances in the actual dataset of students’ programs (the test set) are considered as “unseen” by the classifiers.

E. Results

Figures 2a, 2b and 2c present the classification results of each **D1** metric (JPlag, AST and AStrId) for each **D2** classifier configuration, respectively, full-corpus, k -medoids and instructor-written. We compare these results using the criteria defined in § I-C.

1) **Classification Performance (C1):** As a global trend for all **D2** configurations, we can highlight that structure-matching metrics do allow to successfully classify solution strategies within students’ programs. By relying on a k -NN classifier with a complete dataset of programs labelled beforehand (full-corpus, Fig. 2a), one can obtain the highest F1-score values, up to > 0.9 . In the full-corpus setting, we cannot pinpoint a specific metric from **D1** as better than the others; in general, the scores are relatively close, i.e. for full-corpus, **C1** is independent of **D1**.

2) **Exercise-Dependent Performance:** Across all configurations, we observe a lesser performance for the **encoding** and **search** exercises. This effect seems to be caused by the diversity of student-written programs for those exercises, which are more likely to encounter variations between programs with the same strategy. For easier exercises, where variations between different programs matching a same strategy are more limited, like **fibonacci**, the classifiers obtain better scores. The more a strategy has multiple possible implementation variations, the less accurate the classification with our structure-matching-based metrics becomes.

3) **Impact of Labelling Effort (C2) on C1:** When focusing only on **Classification Performance (C1)**, the full-corpus approach is slightly better than the other **D2** configurations.

Table II: List of exercises with the number of successful submissions compared to the total number of submissions, the success rate (in %), the number of identified classes and their coverage, and a short description of the exercise statement.

Exercise	Submissions	% Success	Classes	Description
fibonacci	100 / 103	97.09	4 (95.0%)	Compute the n -th term of the Fibonacci sequence.
zorglanguage	91 / 527	17.27	3 (98.9%)	Reverse each word in a sentence while keeping relative order of the words in the sentence.
neighbours	285 / 498	57.23	8 (97.9%)	Transform a list of tuples into a dictionary.
encoding	143 / 554	25.81	6 (96.5%)	Compute the run length encoding of a string.
search	159 / 514	30.93	7 (96.9%)	Search for words inside a text file.
syracuse	330 / 593	55.65	8 (98.5%)	Compute maximal total stopping time of the Syracuse sequence up to $n \geq 1$.
primes	346 / 571	60.60	7 (93.6%)	Compute the number of prime factors of a number $n \geq 1$.

Listing 7: Medoid extracted from the dataset of **fibonacci** using the `loop_tmp_var` solution pattern.

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    i = 2
    b = 1
    a = 0

    if n != 1 and n != 0:
        while i <= n:
            c = a
            a = b
            b = a + c
            i = i+1
        return b
```

Listing 8: Instructor-written representative of the solution pattern depicted in Listing 7.

```
def fibonacci(n):
    a = 0
    b = 1
    for i in range(n):
        t = a + b
        a = b
        b = t
    return a
```

Listing 9: Instructor-written representative of the recursive solution pattern for **fibonacci**.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

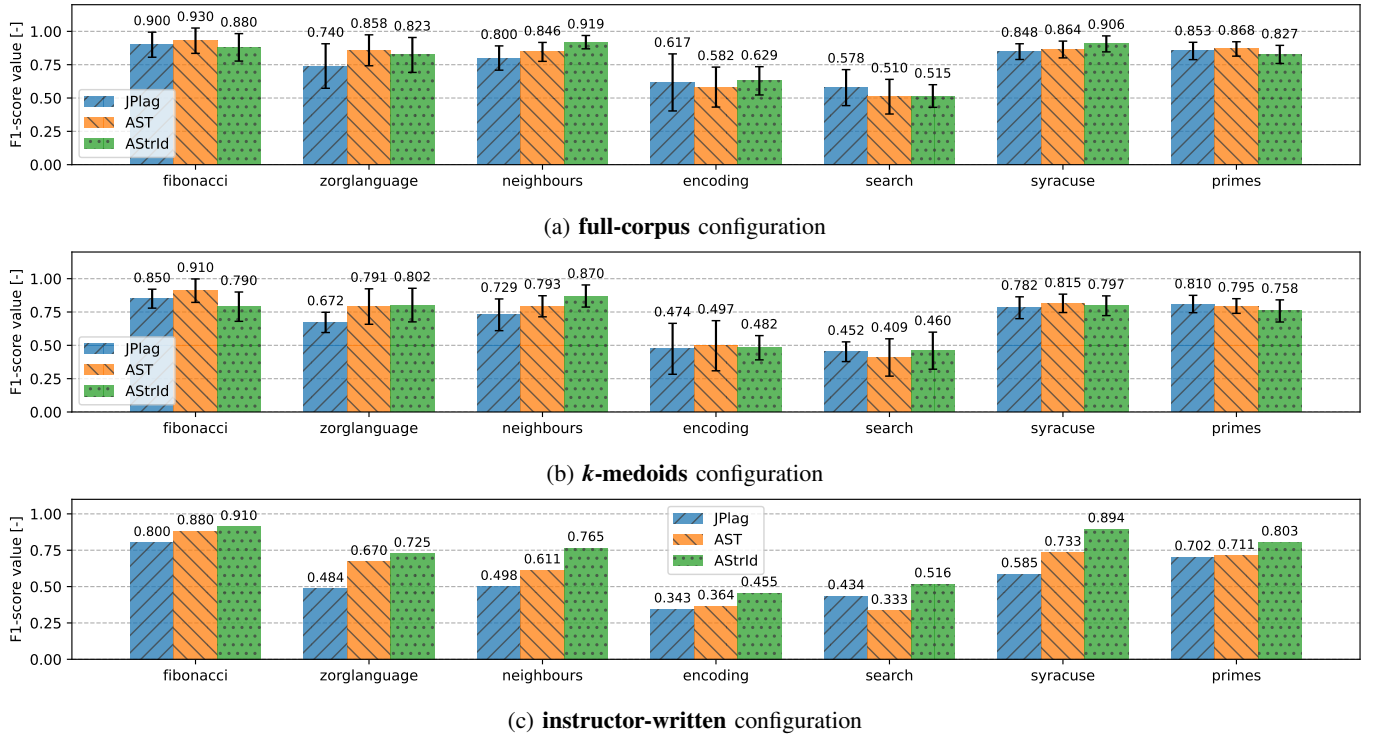


Figure 2: F1-scores on the difference exercises of the **D2** configurations over the **D1** representations.

However, the results of the k -medoids approach (Fig. 2b) show that using a small set of reference programs as a training set does not considerably impact the results compared to the full-corpus approach. The classifiers are less performant, but only slightly, since using medoids forces the classifiers to rely solely on a smaller set of programs, while the full-corpus approach can leverage all the information contained in the dataset. For example, for **neighbours** with our `ASTrId` representation, the F1-score decreased from 0.919 for full-corpus to 0.870 with medoids, but with a training set reduced from 257 to 20 instances. Using 1285% more instances as a training set hence only provides an additional F1-score of 0.049. From these observations, we conclude that the k -medoids approach offers an acceptable tradeoff between **C1**, with a slight negative impact, and **C2**, for which this approach is already better than the full-corpus one.

4) *Remark on Classification Interpretability (C3a)*: We concluded previously that the k -medoids approach should be preferred regarding criterion **C3a** (see III-C). However, the student-written programs used as a reference set for this approach are likely to contain bad practices and useless statements with respect to their underlying solution strategy. A medoid represents an “average” implementation but is not necessarily a pedagogically meaningful representative of a solution strategy; it does not focus on the essential complexity. Listing 7 presents one of the extracted medoids for the **fibonacci** exercise, while Listing 8 is its associated instructor-written solution pattern. Pedagogically speaking, an instructor would prefer using the cleaner implementation (Listing 8) than the student implementation that uses unnecessary condition checks. We can conclude from this that, even if the k -medoids approach is already good regarding criterion **C3a**, in a pedagogical context, the instructor-written approach could be preferred.

5) *Best Performance-Effort Tradeoff*: Lastly, as presented by Fig. 2c, `ASTrId`, in addition to exhibiting a classification performance comparable to JPlag and AST for full-corpus and k -medoids configurations, obtains the best scores (**C1**) when classifying programs based on instructor-written instances. The reduced labelling effort (**C2**) of this configuration does not significantly impact `ASTrId`’s performance, because of its focus on essential complexity, thus being more tolerant to accidental deviations in student programs from the instructor’s representative solution.

From this last observation (IV-E5) and the discussion above (IV-E4), we can conclude that, in our context, the combination of `ASTrId` for **D1** and the instructor-written approach for **D2** reaches the best tradeoff regarding **Classification Performance (C1)**, **Labelling Effort (C2)** and **Classification Process Interpretability (C3a)**.

F. Interpretability of the Predictions

A last criterion to discuss is the interpretability of the code representations (**D1**) and the understandability of the matching produced by the associated metrics, i.e. criterion **C3b**. This subsection qualitatively illustrates the interpretability of the

classifiers’ predictions, using the example code in Listing 7 and the instructor-written program in Listing 8.

First of all, regarding the metrics, JPlag outputs a similarity score of 0.32, while AST and `ASTrId` yield an edit distance of 35 and 29, respectively. Whereas AST and `ASTrId`’s tree edit-distances enable interpretability of the classification, because they directly reflect the number of edit operations required to transform one program into the other, understanding the meaning of JPlag’s similarity score is conceptually harder.

Comparing the mapping each approach produces, we can observe that, as expected, JPlag’s tokens are too generic to effectively capture the behaviour of the code (cf. Listing 10). `[ASSIGN]` tokens are mapped between each program, even if their underlying statements are unrelated, as depicted by the red color. Furthermore, no mapping was identified between the `while` and the `for` loop, missing a key aspect of the student’s implementation. As for the mapping identified by the AST-based approach (cf. Listing 11), in addition to being particularly long, we lose all the information about the nesting level of the nodes. `identifier` nodes are grouped together without checking their parent nodes, and unrelated nodes, like `block` and `identifier`, are considered as nodes that can be mapped to one another, even if this does not make sense in practice. Lastly, for `ASTrId`, benefiting from the nodes’ readability and informativeness (including the transformation cost), the operations to transform one program to another can be projected directly on the student’s program, like reordering the assignments or transforming the `while` into a `for` loop. Unlike the AST-based approach, only related nodes are mapped to each other, and `ASTrId` still allows, like for the loops, nodes to be considered close even if actually different, unlike with JPlag’s tokens. We can thus conclude that `ASTrId` is the best choice regarding criterion **C3b**. In the context of the introductory programming course that this work considered, `ASTrId` provided the right balance between good classification performance using instructor-written canonical solution patterns and representation expressiveness. The combination of the two, in addition to a metric corresponding to a number of edit operations, permits one to interpret the classifiers’ predictions and understand what makes an instance closer to a solution pattern than another.

V. CONCLUSION

In this paper, we explored the combination of two orthogonal dimensions for classifying student programs based on their underlying solution strategies: structure-matching metrics (**D1**) and reference set creation for classifiers using k -NN (**D2**). For the first dimension, we introduced `ASTrId`, a novel code abstraction based on ASTs, that focuses on the “essential complexity” in students’ program. When comparing programs, this representation inherently supports a set of *Semantic-Preserving Variations*. The other abstractions we explored were based on JPlag and standard ASTs. The second dimension focused on determining which programs to store as representatives to compare student’s programs with for classification.

Listing 10: Matching tiles found by JPlag when comparing the programs in Listings 7 and 8. The blue color is for matching tokens, and red the unrelated information matched by JPlag.

```
Mapping:
Match #1:
> Code A = [7 -> 9]
> Code B = [5 -> 7]
    [ASSIGN] i = 2 | [ASSIGN] t = a + b
    [ASSIGN] b = 1 | [ASSIGN] a = b
    [ASSIGN] a = 0 | [ASSIGN] b = t
Match #2:
> Code A = [12 -> 13]
> Code B = [1 -> 2]
    [ASSIGN] c = a | [ASSIGN] a = 0
    [ASSIGN] a = b | [ASSIGN] b = 1
```

Listing 11: Slice of the edit-mapping found using the AST-based approach when comparing the programs in Listings 7 and 8. Blue indicates zero-cost operations, red the others.

```
Mapping (Tree A | Tree B):
... (5 lines)
(cost 0.0) > block | block
(cost 1.0) > consequence | identifier
(cost 0.0) > identifier | identifier
(cost 1.0) > integer | identifier
(cost 1.0) > block | identifier
(cost 1.0) > consequence | body
(cost 1.0) > integer | identifier
(cost 1.0) > right | return_statement
(cost 1.0) > right | arguments
... (47 lines)
```

Listing 12: Edit-mapping found using AStrId when comparing the programs in Listings 7 and 8. Insert and delete operations have been filtered from this representation. The red color represents operations with unit cost, orange *compatible comparisons* and blue perfect matches.

```
Mapping (Tree A | Tree B):
(cost 0.0) > MODULE() | MODULE()
(cost 0.0) > DEF(fibonacci, n) | DEF(fibonacci, n)
... (8 lines)
(cost 1.0) > ASSIGNS(2, 1, 0) | ASSIGNS(0, 1)
(cost 2.5) > WHILE(EXPR(#1, n)) | FOR(n)
(cost 4.0) > ASSIGNS(#3, #2, EXPR(#3, #4), EXPR(#1, 1)) | ASSIGNS(EXPR(#1, #2), #2, #3)
(cost 0.5) > OUT(#2) | OUT(#1)
```

We compared each combination of both dimensions according to three criteria: **C1 (Classification Performance)**, **C2 (Labelling Effort)** and **C3 (Classification Interpretability)**. We highlighted that all studied abstract code representations and their corresponding structure-matching metrics can support strategy classification (**C1**) using k -NN over corpora of labelled student programs. However, the observed performance of this classification is exercise-dependent. For exercises allowing for a broader space of implementation-level variation, performance drops due to the increased structural noise and higher diversity within a same strategy class.

We also observed that using even a small number of highly representative instances can preserve most of the system’s predictive performance, meaning that a full corpus of pre-labelled student programs is not strictly necessary for reliable classification (**C2**).

Furthermore, we identified that the combination of AStrId (**D1**) with instructor-written reference sets for creating the classification (**D2**) leads to the best tradeoff with respect to our three criteria. Under this setting, which positively reduced the labelling effort for instructors (**C2**), owing to its deliberate abstraction from accidental complexity, AStrId was markedly more performant (**C1**) than the other abstractions considered (JPlag and AST-based), demonstrating AStrId’s ability to capture key structural elements characteristic of a strategy. AStrId also had a positive impact on **C3** as compared to JPlag and ASTs by making it possible to understand the solution strategy implemented and to reconstruct the edit operations that map one program to another.

Limitations, Threats to Validity and Future Work: Our current validation only focused on an introductory Python programming course, where exercise solutions are relatively short and require simple language constructs only. In a context targeting more advanced concepts, the essential complexity of a strategy could be harder to define, as we observed during our experiments with the **encoding** and **search** exercises. Further experiments could be conducted to evaluate AStrId on a higher number of exercises, submissions and possible solution patterns, as well as within multiple programming courses and with the participation of a larger set of instructor perspectives. We will also extend our representation to be language independent and include control and data flow analysis.

Even though we designed AStrId to be tolerant to *Semantic-Preserving Variations*, these are not yet all fully supported by our representation. For example, temporary variables (SPV8) are subject to the limits of *compatible comparisons*, and sequences of *if-elif* conditions should permit unordered comparisons (SPV6 and SPV7). Further work is planned to make AStrId more resilient to such variations.

Even though we performed preliminary experiments with variations of the parameter choices, such as the parameters of JPlag, the cost model of AStrId or the k values for the k -NN and k -medoids algorithms, and that these did not reveal a significant impact for our choices, a more detailed study could still be conducted to make this picture more complete. We will also consider comparing AStrId to other tools, such as the ones in Section II (e.g. Keuning *et al.* [3], Out-of-step [18] or machine-learning-based approaches).

REFERENCES

- [1] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, "OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale," *ACM Trans. Comput. Hum. Interact.*, vol. 22, no. 2, pp. 7:1–7:35, mar 2015. [Online]. Available: <https://doi.org/10.1145/2699751>
- [2] B. Heeren, J. Jeuring, and A. Gerdes, "Specifying Rewrite Strategies for Interactive Exercises," *Mathematics in Computer Science*, vol. 3, no. 3, pp. 349–370, may 2010. [Online]. Available: <https://doi.org/10.1007/s11786-010-0027-4>
- [3] H. Keuning, B. Heeren, and J. Jeuring, "Strategy-based feedback in a programming tutor," in *Proceedings of the Computer Science Education Research Conference, CSERC 2014, Berlin, Germany, November 5-6, 2014*, ser. CSERC '14, E. Barendsen and V. Dagienė, Eds. New York, NY, USA: ACM, nov 2014, pp. 43–54. [Online]. Available: <https://doi.org/10.1145/2691352.2691356>
- [4] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, pp. 595–609, sep 1984. [Online]. Available: <https://doi.org/10.1109/TSE.1984.5010283>
- [5] J. Hattie and H. Timperley, "The Power of Feedback," *Review of Educational Research*, vol. 77, no. 1, pp. 81–112, mar 2007, publisher: American Educational Research Association. [Online]. Available: <https://doi.org/10.3102/003465430298487>
- [6] J. C. Paiva, J. P. Leal, and Á. Figueira, "Automated Assessment in Computer Science Education: A State-of-the-Art Review," *ACM Trans. Comput. Educ.*, vol. 22, no. 3, pp. 34:1–34:40, jun 2022. [Online]. Available: <https://doi.org/10.1145/3513140>
- [7] A. Nguyen, C. Piech, J. Huang, and L. J. Guibas, "Codewebs: scalable homework search for massive open online programming courses," in *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, ser. WWW '14, C. Chung, A. Z. Broder, K. Shim, and T. Suel, Eds. New York, NY, USA: ACM, apr 2014, pp. 491–502. [Online]. Available: <https://doi.org/10.1145/2566486.2568023>
- [8] F. P. B. Jr., "No Silver Bullet Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, pp. 10–19, apr 1987. [Online]. Available: <https://doi.org/10.1109/MC.1987.1663532>
- [9] S. Xu and Y. S. Chee, "Transformation-based diagnosis of student programs for programming tutoring systems," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 360–384, apr 2003. [Online]. Available: <https://doi.org/10.1109/TSE.2003.1191799>
- [10] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *Journal of Universal Computer Science*, vol. 8, no. 11, p. 1016, 03 2002. [Online]. Available: <https://doi.org/10.3217/jucs-008-11-1016>
- [11] Y. Song, C. Lohrithz, X. Tang, T. F. Bissyandé, and J. Klein, "Revisiting code similarity evaluation with abstract syntax tree edit distance," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024 - Short Papers, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, aug 2024, pp. 38–46. [Online]. Available: <https://doi.org/10.18653/v1/2024.acl-short.3>
- [12] T. M. Cover and P. E. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, jan 1967. [Online]. Available: <https://doi.org/10.1109/TIT.1967.1053964>
- [13] L. Kaufman and P. J. Rousseeuw, *Partitioning Around Medoids (Program PAM)*. John Wiley & Sons, Ltd, 1990, pp. 68–125, section: 2. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470316801.ch2>. [Online]. Available: <https://doi.org/10.1002/9780470316801.ch2>
- [14] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, ser. SIGMOD '03, A. Y. Halevy, Z. G. Ives, and A. Doan, Eds. New York, NY, USA: ACM, 2003, pp. 76–85. [Online]. Available: <https://doi.org/10.1145/872757.872770>
- [15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, mar 2006. [Online]. Available: <https://doi.org/10.1109/TSE.2006.28>
- [16] Z. Duric and D. Gasevic, "A Source Code Similarity System for Plagiarism Detection," *The Computer Journal*, vol. 56, no. 1, pp. 70–86, jan 2013. [Online]. Available: <https://doi.org/10.1093/comjnl/bxs018>
- [17] M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia: A systematic review," *ACM Trans. Comput. Educ.*, vol. 19, no. 3, pp. 27:1–27:37, may 2019. [Online]. Available: <https://doi.org/10.1145/3313290>
- [18] S. Jimenez, G. Rakić, S. Takahashi, and N. Cardozo, "Out of step: Code clone detection for mobile apps across different language codebases," *Science of Computer Programming*, vol. 236, p. 103112, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167642324000352>
- [19] L. Jiang, G. Mishnerghi, Z. Su, and S. Glondou, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, may 2007, pp. 96–105, ISSN: 1558-1225. [Online]. Available: <https://doi.org/10.1109/ICSE.2007.30>
- [20] W. Wen, X. Xue, Y. Li, P. Gu, and a. J. Xu, "Code Similarity Detection using AST and Textual Information," *International Journal of Performability Engineering*, vol. 15, no. 10, p. 2683, oct 2019. [Online]. Available: <https://www.ijpe-online.com/EN/10.23940/ijpe.19.10.p14.26832691>
- [21] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2464–2519, aug 2018. [Online]. Available: <https://doi.org/10.1007/s10664-017-9564-7>
- [22] M. Z. Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," *Journal of Systems and Software*, vol. 204, p. 111796, oct 2023. [Online]. Available: <https://doi.org/10.1016/j.jss.2023.111796>
- [23] Z. Zhang and T. Saber, "Machine Learning Approaches to Code Similarity Measurement: A Systematic Review," *IEEE Access*, vol. 13, pp. 51 729–51 764, 2025. [Online]. Available: <https://doi.org/10.1109/ACCESS.2025.3553392>
- [24] E. Artser, A. Birillo, Y. Golubev, M. Tigina, H. Keuning, N. Vyahhi, and T. Bryksin, "Clustering mooc programming solutions to diversify their presentation to students," in *Proceedings of the 24th Koli Calling International Conference on Computing Education Research, Koli Calling 2024, KoliFinland, November 12-17, 2024*, ser. Koli Calling '24, J. Leinonen and A. Mühling, Eds. New York, NY, USA: ACM, 2024, pp. 19:1–19:8. [Online]. Available: <https://doi.org/10.1145/3699538.3699548>
- [25] J. Liénard, K. Mens, and S. Nijssen, "Pyttern: a Python-Based Program Query Language," in *CEUR Workshop Proceedings, 2024*, ISSN: 1613-0073. [Online]. Available: <https://hdl.handle.net/2078.5/256590>
- [26] K. Mens, I. Michiels, and R. Wuyts, "Supporting software development through declaratively codified programming patterns," *Expert Systems with Applications*, vol. 23, no. 4, pp. 405–413, 2002. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0957417402000763>
- [27] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-01078532/document>
- [28] J. Smits, T. Hartman, and J. Cockx, "Optimising first-class pattern matching," in *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 74–83. [Online]. Available: <https://doi.org/10.1145/3567512.3567519>
- [29] G. Steveny, K. Mens, and S. Nijssen, "AStrId code package," Aug. 2026. [Online]. Available: <https://doi.org/10.5281/zenodo.2182182>
- [30] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," *ACM Trans. Database Syst.*, vol. 40, no. 1, pp. 3:1–3:40, mar 2015. [Online]. Available: <https://doi.org/10.1145/2699485>
- [31] —, "Tree edit distance: Robust and memory-efficient," *Information Systems*, vol. 56, pp. 157–173, mar 2016. [Online]. Available: <https://doi.org/10.1016/j.is.2015.08.004>
- [32] K. Zhang, "A constrained edit distance between unordered labeled trees," *Algorithmica*, vol. 15, no. 3, pp. 205–222, mar 1996. [Online]. Available: <https://doi.org/10.1007/BF01975866>
- [33] JPlag. Jplag wiki. [Online]. Available: <https://github.com/jplag/JPlag/wiki>