Computer Science and Information Systems 21(1):117–142 https://doi.org/10.2298/CSIS230615004P

Comparing Semantic Graph Representations of Source Code: The Case of Automatic Feedback on Programming Assignments

José Carlos Paiva[1,2], José Paulo Leal[1,2], and Álvaro Figueira[1,2]

[1] CRACS – INESC TEC
Porto, Portugal
[2] DCC – FCUP
Porto, Portugal
jose.c.paiva@inesctec.pt
zp@dcc.fc.up.pt
arfiguei@fc.up.pt

Abstract. Static source code analysis techniques are gaining relevance in automated assessment of programming assignments as they can provide less rigorous evaluation and more comprehensive and formative feedback. These techniques focus on source code aspects rather than requiring effective code execution. To this end, syntactic and semantic information encoded in textual data is typically represented internally as graphs, after parsing and other preprocessing stages. Static automated assessment techniques, therefore, draw inferences from intermediate representations to determine the correctness of a solution and derive feedback. Consequently, achieving the most effective semantic graph representation of source code for the specific task is critical, impacting both techniques' accuracy, outcome, and execution time. This paper aims to provide a thorough comparison of the most widespread semantic graph representations for the automated assessment of programming assignments, including usage examples, facets, and costs for each of these representations. A benchmark has been conducted to assess their cost using the Abstract Syntax Tree (AST) as a baseline. The results demonstrate that the Code Property Graph (CPG) is the most feature-rich representation, but also the largest and most space-consuming (about 33% more than AST).

Keywords: semantic representation, source code, graph, source code analysis, automated assessment, programming.

1. Introduction

Reasoning over graph representations of source code has innumerous applications in a wide variety of areas, ranging from software security (e.g., for detecting bugs [48, 15] and vulnerabilities [25, 80], and de-obfuscating code [52, 4, 64]) to coding assistance and computer science education (e.g., predicting variable and method names [1, 4], inferring types [36, 47], detecting similar code [74, 34, 68, 50], inferring specifications [13], and automated program repairing [14, 24]). Automated assessment has also been driving efforts into such techniques [43], as "looking" directly at the source code can enable less rigid assessment as well as richer and more formative-driven feedback than considering the result of its execution.

Achieving an effective representation for inference is a critical step in these techniques because it is an important determinant of accuracy, performance, and cost. On the one hand, encoding too much information leads to higher complexity. On the other hand, less information can reduce the accuracy of the technique. As far as the authors are aware, there is no comparison of these representations in the literature from a practical point of view, let alone for our purpose of automatically generating feedback for programming tasks. The goal of this work is to provide a detailed comparison of the most relevant semantic graph representations of source code to find out which is the most adequate for automatic feedback generation on programming assignments. Such a comparison should not only reveal which aspects of the source code each representation captures and which is the simpler representation that answers the questions at hand, but also suggest possible new ways to explore some semantic graph representations for the purpose here intended.

This work constitutes an intermediate step to building ground to the decision of which representation(s) to use in each phase of our novel automated feedback mechanism, that given an incorrect attempt (1) selects the closest correct solution from the dataset of past solutions and (2) identifies the differences between the wrong attempt and the correct one selected. To this end, we briefly describe each representation and present some recent work that uses them. We also perform feature coverage and a practical comparison of the presented semantic graph representations. The results are shown and discussed.

The remainder of this paper is organized as follows. Section 2 presents the most relevant semantic graph representations of source code. Section 3 compares the presented representations according to several aspects, using some typical program samples found in introductory programming classes. Section 4 reviews and discusses results. Finally, Section 5 summarizes the main contributions of this study. Relevant literature work is combined with the presentation of the representations and the discussion of results.

2. Intermediate Representations of Source Code

Research in the fields of program analysis and compiler design has developed a number of semantic graph representations of source code to reason about different aspects of programs. This section presents the most relevant semantic graph representations of source code for assessing submissions to programming assignments. Firstly, it introduces the AST, i.e., the "mother" of all representations. After the AST, five flow-based representations are introduced, namely the Control Flow Graph (CFG) and its interprocedural variant, the Evaluation Order Graph (EOG), the Data Flow Graph (DFG), and the Call Graph (CG). The two following subsections present two dependency-based representations, namely the Program Dependence Graph (PDG) and the System Dependence Graph (SDG). Finally, the Code Property Graph (CPG), which combines syntax, flow, and dependence information, is described.

Fig. 1 depicts the relationships between the selected semantic graph representations. The nodes illustrate the semantic graph representations, whereas the edges denote the relationships among them. There is a relationship between two representations if one is "based" (i.e., captures information from) on the other. The edge direction indicates the source of information. Syntax-based, flow-based, dependence-based, and mixed information representations are highlighted in different grey levels.

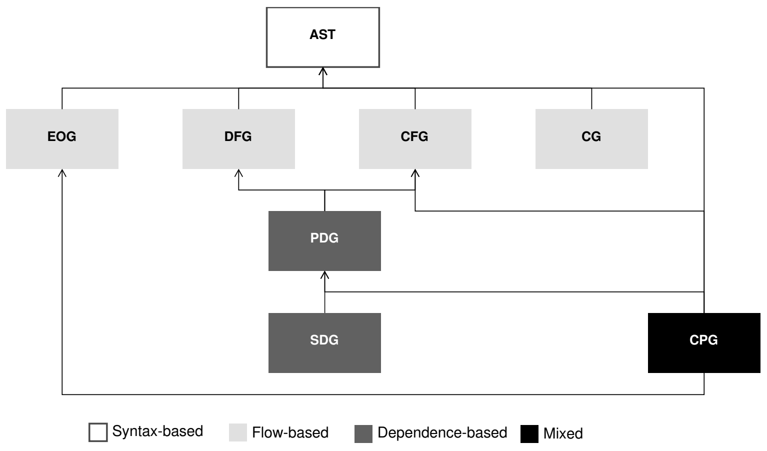

Fig. 1. Diagram view of the relationships between semantic graph representations

During the introduction of the representations, some exemplary graphs extracted from the Python programs presented in Listing 1.1, 1.2, and 1.3 are depicted. These programs are solutions to the following tasks

T1: enumerate even numbers from 1 to X ($X = 10$),
T2: calculate the factorial of Y using a recursive approach ($Y = 10$),
T3: sort an unordered array of Z numbers using QuickSort algorithm ($Z = 10$).

Listing 1.1. Python program to print even numbers from 1 to 10 (task T1)

```python
X = 10
for i in range(1, X + 1):
  if (i % 2) == 0:
    print(i, end=" ")
print()
```

2.1. Abstract Syntax Tree (AST)

An AST is a finite, labeled, directed tree that models the syntactical structure of a program. Its internal nodes correspond to operators, whereas leaves are the operands (symbols) of the parent operator (e.g., variables, arguments, and constants). The AST preserves information regarding types and locations of variable declarations, order and components of statements, terms, operators, and their positioning in expressions, and the names and assigned values of identifiers.

Differently from parse trees, the AST omits nodes and edges from syntax rules that have no effect on the semantics of the program. For instance, grouping parentheses are

Listing 1.2. Python program to calculate the factorial of 10 using a recursive approach (task S2)

```python
def factorial(Y):
  if Y == 0:
    return 1
  return Y * factorial(Y - 1)

Y = 10
factorial(Y)
```

Listing 1.3. Python program to sort an unordered array of 10 numbers using Quicksort algorithm (task S3)

```python
def partition(arr, l, h):
  pivot = arr[h]
  i = l - 1
  for j in range(l, h):
    if arr[j] <= pivot:
      i = i + 1
      (arr[i], arr[j]) = (arr[j], arr[i])
  (arr[i + 1], arr[h]) = (arr[h], arr[i + 1])
  return i + 1

def quickSort(arr, l, h):
  if l < h:
    pi = partition(arr, l, h)
    quickSort(arr, l, pi - 1)
    quickSort(arr, pi + 1, h)

data = [8, 7, 6, 1, 0, 9, 2, 11, 10, 3]
Z = len(data)
quickSort(data, 0, Z - 1)
```

not part of the AST as their grouping is already implicit in the structure of the AST. Therefore, one way to build the AST is by performing a post-pass over the parse tree, removing nodes and edges which do not belong to the abstract syntax. An alternative way for programming languages described by context free grammars is to create the AST in the parser. In this regard, rules that contribute to the AST insert a node in the AST, whereas the symbols of the rule form edges. Figure 2 presents an AST extracted from the Python program in Listing 1.2 (note that no particular order, other than the authors' opinion, has been adopted while using the code listings in examples).

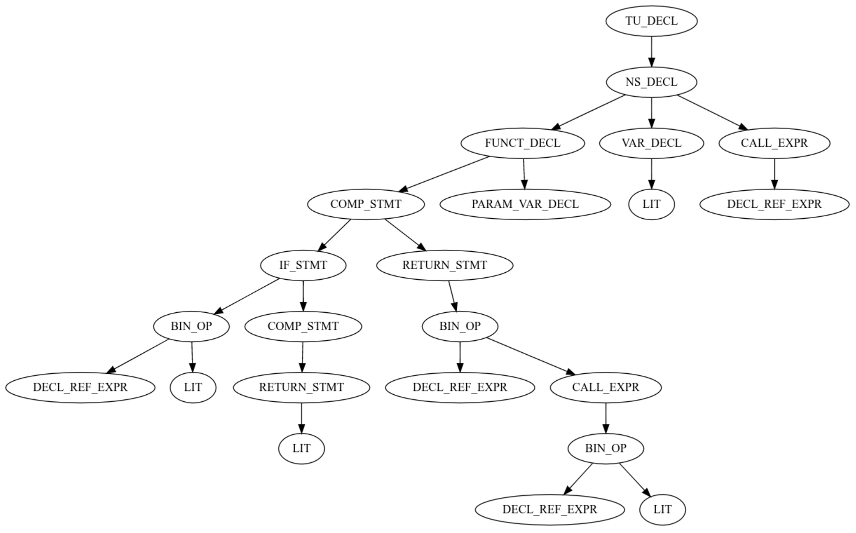

Fig. 2. Abstract syntax tree of the Python program in Listing 1.2

As ASTs are capable of maintaining a great amount of syntactic information from the original source-code, they are ideal for tasks requiring regeneration of source code, such as programming language translation [77], automatic parallelization tools [6], and program rewriting [67].

2.2. Control Flow Graph (CFG)

The Control Flow Graph (CFG) [3] models the flow of control between the basic blocks (i.e., maximal-length sequences of branch-free instructions) of a program. It consists of a directed graph, whose nodes correspond to basic blocks and edges represent control dependencies. The graph has two additional nodes, namely the entry block $ENTRY$ – which is the node through which the control enters the graph – and the exit block $EXIT$ – which is the node through which the control exits the graph –, such that each node has at most two successors. For instance, Figure 3 depicts the CFG extracted from the Python program in Listing 1.1.

Furthermore, two important concepts for a number of compiler optimizations and detection techniques are related to the CFG, namely post-domination and control de-

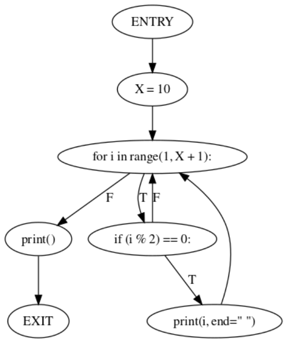

Fig. 3. Control flow graph of the Python program in Listing 1.1

pendency. The former is defined in Statement 1. For instance, in the CFG of Figure 3, node `for i in range(1, X + 1)` and node `X = 10` are post-dominated by node `print()`.

Statement 1 *Consider the CFG G of a program. Let v_1 and v_2 be two nodes of G. Then, v_1 is post-dominated by a node v_2 in G, if and only if every directed path from v_1 to $EXIT$ contains v_2.*

The latter is defined in terms of post-domination relationship as in Statement 2. For instance, in the CFG of Figure 3, node `print(i, end=``''` is control-dependent on node `if (i % 2) == 0`.

Statement 2 *Consider the CFG G of a program and the nodes $v_3, v_4 \in G$. Then, v_4 is control-dependent on v_3 if and only if*

1. *there is a directed path P from v_3 to v_4 with any v_5 in P (excluding v_3 and v_4) post-dominated by v_4 and*
2. *v_3 is not post-dominated by v_4.*

Therefore, the CFG is an essential structure for many optimization techniques and static analysis tools, particularly because it facilitates locating inaccessible code and syntactic structures of a program such as loops. There are many recent examples of such techniques/tools in various areas of Computer Science, ranging from bug localization and vulnerability detection [12, 18, 29] to code optimization [64].

2.3. Interprocedural Control Flow Graph (ICFG)

A CFG can handle only a single procedure. For investigating the control-flow in complete programs, the Interprocedural Control Flow Graph (ICFG) is the representation to use. The ICFG connects the multiple CFGs of a program into a single graph as follows.

– For every call instruction to a function F, insert a new edge from the call to the first instruction of F, and

– from every return statement of F, insert a new edge from the return to the instruction following the call to F.

Rather than gathering information from each predecessor as usual during the analysis, the first statement of a procedure loses all data-flow information about local variables from the caller and adds details for parameters in the callee, initializing their values according to the arguments provided in the call. For instructions immediately after a call, they inherit the data-flow information from the previous instruction regarding local variables. Additionally, global variables are taken from the return instruction of the called function, whereas the variable to which the result of the function call was assigned comes from the data-flow about the returned value.

The ICFG is particularly useful for interprocedural optimizations, such as constant folding, caller-site inlining, or elimination of dead code [53, 9, 40].

2.4. Evaluation Order Graph (EOG)

An Evaluation Order Graph (EOG) – primarily introduced in Fraunhofer's implementation of the Code Property Graph (CPG) [21] – is an intraprocedural directed graph whose vertices correspond to nodes from the AST and edges connect them in the order they would be executed when running the program, similar to the CFG. The EOG is a variant of the CFG with some subtle differences: (1) a procedure header is a node in the EOG, while the CFG only considers the first executable statement of the procedure as a node; (2) the beginning of a compound statement is a separate node in the EOG, while the CFG does not consider it; (3) the EOG represents the "if" keyword and the condition as separate nodes, while the CFG uses a single node; and (4) the EOG always ends in one (virtual) or multiple return statements (i.e., a virtual return statement is created, if the actual source code does not have an explicit one).

Therefore, comparing to the CFG in semantic terms, the EOG models the evaluation order within expressions and has operator-level precision, which enables finer granularity (more precision) in analysis. Figure 4 presents the EOG extracted from the Python program in Listing 1.2. The interprocedural version of the EOG also follows a similar approach to that of the ICFG (see Subsection 2.3), but for every call instruction to a function F, insert a new edge from the call to the F function declaration rather than the first instruction of F.

2.5. Data Flow Graph (DFG)

A Data Flow Graph (DFG) is a directed labeled graph that models the program without conditionals, i.e., using basic blocks. Edges of the graph represent paths over which data is passed, whereas nodes are the producers and consumers of such data. Dynamically, a node of the DFG accepts multiple data items from its inputs, performs some computation with them, and passes the resulting items to its outputs. For instance, Figure 5 depicts the DFG extracted from the Python program in Listing 1.1.

The DFG allows several analysis such as dead code detection, liveness analysis, and variable misuse identification [2, 26] as well as performance optimizations through parallelization [33].

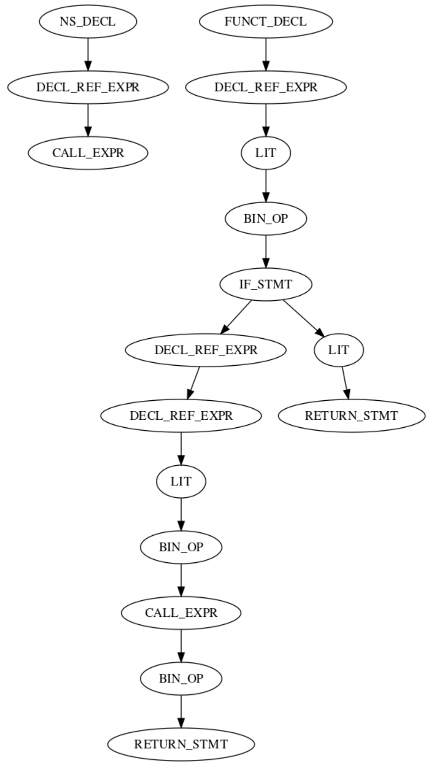

Fig. 4. Evaluation order graph of the Python program in Listing 1.2

2.6. Call Graph (CG)

The Call Graph (CG) [57] is a directed graph that connects nodes representing procedure calls to procedure start nodes. The set of nodes consists of procedure calls and starts only, whereas the edges are established between them from the start node of each routine to the call nodes. Formally, a CG can be defined, from the set of control flow graphs corresponding to the procedures of a program, as follows.

Statement 3 *Consider the call graph $G = (V, E)$ and the set of control flow graphs F (so, V_f is the set of vertices of $f \in F$ and E_f its set of edges) of a program. Then, $V = V_C \cup V_S$, where V_C is the set of* `call` *instructions and V_S is the set of procedure starts in the source code, and the edge set $E \subseteq V \times V$ is defined as*

$$E = \{(c, s) : c \in V_C, s \in V_S(c)\} \cup \bigcup_{(V_f, E_f) \in F} \{(s, c) : s \in V_S, c \in V_C : \exists s \to c \in V_f\} \tag{1}$$

Assuming also that $V_S(c)$ is the procedure called by $c \in V_C$. The call nodes should have exactly one incoming edge in the CFG (i.e., from the procedure it belongs to), which can be achieved by adding an empty node for nodes not meeting this requirement. Hence,

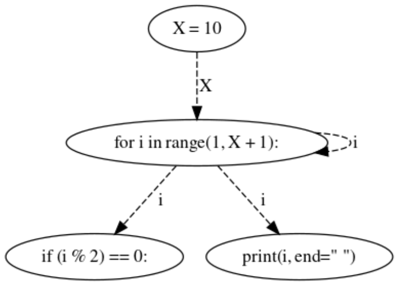

Fig. 5. Data flow graph of the Python program in Listing 1.1

each call node also has only one outgoing edge in the CFG. Concerning the CG, a call node also has exactly one incoming edge (from the start node). An example CG for the Python program in Listing 1.3 is presented in Figure 6.

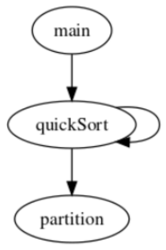

Fig. 6. Call graph of the Python program in Listing 1.3

Call graphs are widely explored in the literature as they encode symbolic and topological features of software that can be important in several different areas such as malware detection [59, 17, 23], measuring software complexity [58, 49], bug detection [69, 61], and unreachable code detection [56].

2.7. Program Dependence Graph (PDG)

A Program Dependence Graph (PDG) is a directed attributed graph that approximates program semantics, explicitly encoding control and data dependence information of the program [20]. Each node of the graph indicates a statement of the program such as an assignment, an assertion, a method call, or a return, whereas an edge represents a dependence, either regarding control or data, between two statements. For instance, given two statements $s1$ and $s2$, there is a control dependence edge from $s1$ to $s2$ if and only if the execution of $s2$ depends upon $s1$, and a data dependence edge if and only if a component assigned at $s1$ is used in the execution of $s2$. Figure 7 depicts the PDG extracted from the Python program in Listing 1.1. Dashed edges represent data dependencies, whereas solid edges encode control dependencies.

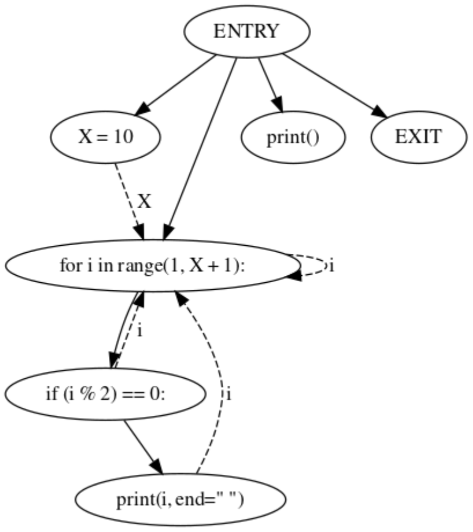

Fig. 7. Program Dependence Graph of the Python program in Listing 1.1

These graphs resort to control- and data-flow analysis to determine control and data dependencies, respectively. The former are defined in terms of the CFG (see Subsection 2.2), while the latter can be formally defined in terms of the DFG as:

Statement 4 *Consider G, a DFG of a program, then nodes $s_1, s_2 \in G$ are data-dependent if and only if there is a variable v such that*

- *s_1 is an assignment to v,*
- *s_2 uses v, and*
- *there is a path between s_1 and s_2 without an assignment to v.*

In PDGs, dependencies connect the computationally relevant parts of the program without requiring the unnecessary control sequencing present in the CFGs (see Subsection 2.2), which means they are immune to syntactic modifications such as renaming variables, reordering statements, among others. Hence, these graphs are adequate representations to measure semantic similarity between two codes and detect semantic clones [28, 46, 38].

2.8. System Dependence Graph (SDG)

A System Dependence Graph (SDG) [27] is an extension to the PDG (see Subsection 2.7) to handle programs with multiple procedures and interprocedural calls. It models a language in which a program consists of a main procedure and a set of auxiliary procedures, to which parameters are passed by value. More formally, the SDG is a directed graph $G = (N, E)$, where N contains the nodes representing statements and predicates of the program, and E is the set of edges depicting the dependencies between them.

The SDG can be built from the PDGs of the several procedures, by connecting them at *call sites*. This consists of linking the call node c to the entry node e of the called procedure, through an edge from c to e, if there are no procedure parameters and returned values. Procedure parameters and returned values, as well as potential side effects of the called procedure, are modeled through artificial parameter nodes and edges created as follows:

– For every passed parameter p_i there exist two nodes, actual-in a_i and formal-in f_i, connected via an edge $a_i \rightarrow_{p_i} f_i$.
– For every modified parameter and returned value p_o there exist two nodes, an actual-out a_o and formal-out f_o, connected via an edge $f_o \rightarrow_{p_o} a_o$.
– Nodes f_i and f_o nodes are control dependent on the entry node e of the called procedure.
– Nodes a_i and a_o nodes are control dependent on the call node c.

In this way, it guarantees that any interprocedural effect of a procedure is propagated through the *call sites*, enabling the computation of summary information about the impact of all procedures directly or indirectly invoked. Figure 8 presents the SDG extracted from the Python program in Listing 1.2.

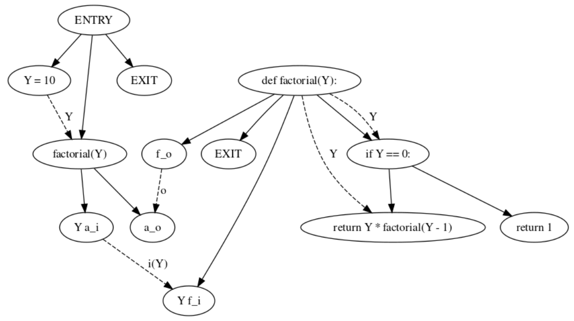

Fig. 8. System Dependence Graph of the Python program in Listing 1.2

SDGs are the basis for several applications in program analysis, ranging from program understanding [63, 45] and testing [31] to program slicing [79, 22].

2.9. Code Property Graph (CPG)

Code Property Graphs (CPGs) [76] combine three other intermediate representations, particularly ASTs, CFGs, and PDGs, into a joint data structure using the concept of property graphs – a type of graph in which edges not only connect nodes, but also carry information (e.g., a name, type, or other properties). To this end, each of the three graphs is, firstly, converted into a property graph and then combined. The AST, which provides a detailed decomposition of source code into language constructs, is expressed as a property graph $G_A = (V_A, E_A, \lambda_A, \mu_A)$, where V_A is the set of AST nodes, E_A are the corresponding edges of the tree which are labelled as such (i.e., AST nodes) by the labelling function λ_A, and μ_A which assigns two properties, `code` and `order`, to each node. The value of the former consist of the operator or operand the node represents, whereas the latter is the index of the node in the ordered form of the tree.

Following the AST, we need to depict the CFG as a property graph $G_C = (V_C, E_C, \lambda_C, \cdot)$, where V_C relates to statements and predicates in the AST (i.e., all nodes in V_A whose property `code` corresponds to either a statement or a predicate). Moreover, the edge labeling function λ_C assigns to each edge of the property graph a label from the set $\sum_C = \{true, false, \epsilon\}$, according to the condition of the source node.

The next step is to transform the PDG into a property graph. As described in Subsection 2.7, the PDG represents data and control dependencies between statements and predicates and, thus, its nodes are the same as those from the CFG. In this sense, the PDG can be depicted as a property graph $G_P = (V_C, E_P, \lambda_P, \mu_P)$, where E_P is a new set of edges and $\lambda_P : E_P \longrightarrow \{C, D\}$ is the corresponding edge labeling function that assigns either label C or D if it matches control and data dependencies, respectively. The former dependencies have an additional property `condition` indicating the state of the original predicate (i.e., `true` or `false`), whereas the latter dependencies have an additional property `symbol` holding the corresponding symbol.

Finally, the three property graphs are merged into a new graph $G = (V, E, \lambda, \mu)$. The set of nodes V has the same elements of V_A, whereas the set of edges E includes all edges in the three graphs. Similarly, the labeling and the property function, λ and μ respectively, are the combination of the homonymous functions of the three graphs (if they exist there). As an example, Figure 9 presents the CPG extracted from the Python program in Listing 1.2. Edges of the AST are traced in black, data dependencies are dashed, and control dependencies are in red. Property edges are omitted.

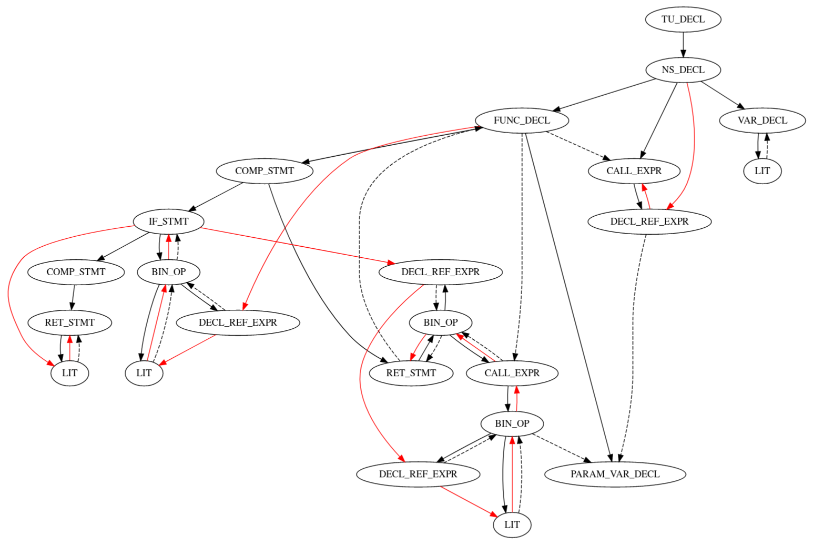

Fig. 9. Code property graph of the Python program in Listing 1.2.

The CPG was initially designed for finding vulnerabilities and detecting performance and efficiency weaknesses in source code, having already been widely and successfully applied in these tasks [76, 75, 80, 16]. The unique characteristics of this graph such as the ability to encode both syntactic and semantic value while being language-agnostic, make it attractive to other tasks (e.g., predicting names and repairing code). However, primarily due to its recency, the potential of this representation in other scenarios is yet to be explored.

3. Comparison

Determining the most effective semantic graph representation of source code for a specific task is a critical step of designing any source code analysis-dependent technique. On the one hand, the more information is encoded, the more rich and accurate can be the approach. On the other hand, too much information means higher costs, in particular, regarding execution time and memory usage. Finding the right balance between expressiveness and cost is the challenge. To this end, both the expressiveness and cost of the various semantic graph representations of source code have been compared. The following subsections present the result of each comparison.

Table 1. Comparative analysis of the expressiveness of semantic graph representations of source code.

Facet	Syntax	Flow					Dependence		Mixed
	AST	CFG	ICFG	DFG	EOG	CG	PDG	SDG	CPG
Control Dependency	Y	Y	Y	N	Y	N	Y	Y	Y
Data Dependency	Y	N	N	Y	N	N	Y	Y	Y
Multiple Procedures	Y	N	Y	Y	N	Y	N	Y	Y
Interprocedural Calls	Y	N	Y	N	N	Y	N	Y	Y
Multiple Types of Edges	N	N	N	N	N	N	Y	Y	Y
Slicing	Y	N	N	Y	N	N	Y	Y	Y
Flow-Sensitive	Y	Y	Y	Y	Y	Y	Y	N	Y
Context-Sensitive	Y	N	N	N	N	N	N	Y	Y
Inheritance & Polymorphism	Y	N	N	N	N	N	N	Y** [51]	Y
Exception Handling	Y	N	N	N	N	N	Y	Y	Y
Parameter Passing	Y	N	Y	N	N	N	N	Y	Y
Reversibility	Y*	N	N	N	N	N	N	N	Y*

* - cannot reverse to exact original source code

** - extended version

3.1. Expressiveness

From the literature and Section 2, it has been made clear that each semantic graph representation of source code described has unique capabilities. To better understand them, Table 1 summarizes the expressiveness of the previously presented representations, including: whether it encodes control and data dependencies, if it is limited to a single procedure or captures multiple procedures, whether it can represent interprocedural calls, whether it supports multiple types of edges, whether program slicing (i.e., a reduction technique for capturing the statements that could affect the value of a variable at a specific point of interest) and flow- and context-sensitive analysis techniques can be applied, whether it can describe inheritance and polymorphism, whether it correctly deals with exception-handling constructs, whether parameter-passing information is represented, and if the source code can be recovered.

Naturally, syntax-based representations are the the most feature rich, as Table 1 demonstrates for the covered facets. They represent the program with all original language constructs as well as the order between statements, typically losing only ambiguous grammar (e.g., white spaces, semicolons, trailing commas, etc) and comments. Therefore, they can even be reverted to a source code equivalent (not exactly equal) to the original. Concerning flow-based representations, these are sensitive to flow changes. They focus on either the flow of control (e.g., CFG and EOG), in which case they contain information that allows to infer control dependencies (except for the CG, which only captures procedures and interprocedural calls), or the data flow, in which case they encode details about data dependencies. Dependence-based representations, such as the PDG and the SDG, encode program dependencies, both control, and data. The SDG can represent multiple procedures, interprocedural calls, parameter passing, inheritance, and polymorphism. Both the PDG and the SDG support program slicing and exception-handling constructs. While the SDG is sensitive to the context, the PDG is flow-sensitive.

3.2. Benchmark

The extraction of semantic graph representations has been evaluated against the three sample Python programs introduced in Section 2. To this end, an existing library – Fraunhofer's CPG library [21] –, initially designed to extract the Code Property Graph (CPG) out of source code in C/C++ (C17) and Java (Java 13), with experimental support for several other programming languages such as Python, Golang, and TypeScript, has been adapted. The changes were twofold: (1) extend the analysis with multiple passes to build the different representations that were missing (e.g., CG and CFG); and (2) develop functionality to export the representations into Comma-Separated Values (CSV), JavaScript Object Notation (JSON), and DOT [65] formats.

The JSON serialization encodes all data into a single `.json` file, containing a list of nodes, each having an ID, a list of their outgoing edges, and a key-value pair for each node property. CSV serialization stores edge and node data in separate `.csv` files. The CSV file with edges has a source and a target column, plus a column per edge property. The node descriptor file contains an ID column and an additional column for each node property being stored. DOT has been specifically designed to describe graphs and, thus, its serialization adheres to the language. For the sake of this analysis, we have used the CSV-based serialization.

Table 2 presents the results of the comparison of the semantic graph representations of source code presented in Section 2 (ICFG has been omitted as it is easily obtained from the CFG). It includes aspects such as the number of nodes, edges, and connected components, the size of the export (in kilobytes), the build complexity (n-pass means n iterations over the AST), and whether it has edge properties.

Table 2. Results of the comparison of semantic graph representations of source code.

	Nodes			Edges			Components			Export Size (kB)			Build	Edge
IR	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3	Compl.	Prop.
AST	30	23	99	29	22	98				1.7	1.5	7.1	-	0
DFG	4	5	27	4	3	42		2	3	0.2	0.2	7.1	1-pass	1
CFG	7	8	14	8	7	14		2	3	0.3	0.3	0.3	1-pass	1
EOG	20	17	67	21	15	67		2	3	0.7	0.6	2.7	1-pass	1
CG	1	2	3	0	2	3				0.1	0.1	0.1	1-pass	0
PDG	7	9	27	10	10	72		2	3	2.1	1.6	6.6	2-pass	1
SDG	7	13	43	10	16	96				2.2	1.7	6.9	2-pass	1
CPG	30	23	99	66	52	226				2.3	2.0	9.2	3-pass	4

The most expensive representation is the CPG, as Table 2 demonstrates. In the worst case, the CPG has, in most implementations and in the one used here, the same amount of nodes as the AST, which is roughly one intermediate node for each used programming language construct plus the leaf nodes containing any parameters of these constructs. The number of edges is always less or equal to the sum of the edges in the AST, the EOG (or CFG, as in its original implementation), and the control and data dependencies between the AST nodes.

4. Discussion

There are not many works published in the literature comparing the different intermediate representations of source code. Nevertheless, a few exceptions exist. Arora et al. [7] provides a comparison of flow and dependence graphs based on features such as procedural call, slicing, sensitivity, exception handling, and test case generation. For instance, they conclude that the SDG is a feature rich representation comparing to a flow graph, supporting features like control, data and transitive dependence, single and multiple procedure, inter- and intra-procedural calls, multiple types of edges, slicing, context-sensitivity, inheritance and polymorphism, test case generation and parameter passing, whereas the flow graph is insufficient in representing data and transitive dependence, multiple procedures, inter- and intra-procedural calls, multiple types of edges, slicing, context-sensitivity, inheritance and polymorphism.

In this work, we conduct a more practical-driven comparison of various semantic graph representations of source code, measuring a number of aspects of the produced graphs. From the results presented in Section 3, it is clear that encoding more information has great impact on the graph size and, consequently, export size. On the one hand, the CPG, which combines syntactic and semantic information consumes on average 1.33 times the disk space of the AST and has about 2.3 edges for each edge of the AST, while having the same number of nodes. On the other hand, the CG is a very tiny graph whose number of nodes matches the number of subroutines of the program plus the main routine and the number of edges between any two nodes is at most 1.

The CPG is the only graph representation larger and heavier than the AST. Nevertheless, it has a single connected component, similarly to the SDG, whereas the others can have multiple connected components, typically one per procedure. Furthermore, the CPG is also the only representation (besides the AST) which can be transformed back to the original (or a similarly working) source code, as all the others lose important syntactic meaning for this transformation (e.g., EOG loses data relationships, SDG ignores instructions order, etc).

Concerning the build time (in terms of build complexity), the CPG is again the representation with the highest cost, requiring three passes over the AST: two needed to obtain the PDG (and CFG) and another to "combine" the three intermediate representations, particularly AST, CFG, and PDG. The SDG is the second in this matter, requiring a connection step after the two passes required to build the PDGs. All other representations (except the mentioned PDG) are built from the AST in a single pass.

Furthermore, we have evaluated the impact of such representations using real student submissions in multiple programming languages to programming exercises of different complexities. These submissions, taken from the PROGpedia dataset [44], refer to several years within the 2003-2020 timespan of an undergraduate Computer Science course. We have selected three correct submissions to each of four randomly chosen assignments from the dataset (6, 16, 18, and 19). The first assignment involves array manipulation. Assignment 16 requires string comparison. Assignment 18 challenges students to find the minimum cost path in a graph. Finally, assignment 19 involves depth-first search in graphs. The selected submissions for a given programming assignment adopt a similar algorithmic strategy.

Table 3 presents the export sizes of the semantic graph representations for such submissions, including the lines of code (LoC) and size of the original source code. The results support the previous conclusions regarding export sizes, i.e., the CPG has the highest cost (approximately 1.5 times more disk space than the AST), while the CG has the lowest. Regarding programming languages, the syntactic weight of Java is noticeable in most examples through a typically heavier AST. The maximum build time registered in the processed samples was 5.5 seconds for the CPG of the Java submission to programming assignment 18. However, the extraction took 4.6 seconds per submission on average.

4.1. Use Case: Dead Code Analysis

Dead code encompasses any code that has no effect on the program's behavior, including both unused and unreachable code [55]. It is considered a harmful code smell that hinders source code comprehension [37] and downgrades its maintainability [19] and performance [66]. Examples of this anti-pattern include but are not limited to: (1) code after a return statement, in the same block; (2) variable assigned but never used; (3) methods defined but never called; and (4) code inside a conditional block whose condition is always false.

Detecting instances of the first example requires a representation that captures the possible flows of execution of the program. Therefore, the CFG (i.e., the ICFG version) is the most minimal of the presented representations that could be used. Regarding example (2), a representation that could support its detection needs to encode variables' definition/usage lifecycle, which is exactly what DFG does. The CG, which represents the function declarations and calls between them, is the lighter representation that can

Table 3. Export sizes of semantic graph representations on PROGpedia dataset.

ID	Prog. Lang.	LoC	Size (kB)	Export Size (kB)							
				AST	DFG	CFG	EOG	CG	PDG	SDG	CPG
6	C	33	0.3	6.4	0.6	1.4	4.7	0.1	6.5	6.8	10.4
	JAVA	25	0.5	7.9	0.7	1.9	5.0	0.1	6.4	6.7	12.4
	PYTHON	20	0.4	7.0	0.6	1.3	4.6	0.1	6.1	6.3	11.2
16	C	46	0.6	7.8	0.6	1.9	5.5	0.2	5.9	5.9	11.9
	JAVA	38	0.6	8.9	0.7	2.4	5.4	0.1	5.5	5.6	13.8
	PYTHON	17	0.2	5.8	0.5	1.1	4.0	0.1	4.8	5.3	9.1
18	C	73	1.5	28.0	2.9	3.3	20.3	0.3	22.1	22.9	43.4
	JAVA	107	2.1	37.6	3.0	3.9	24.2	0.5	26.5	27.8	57.4
	PYTHON	47	1.4	27.3	1.9	3.1	17.7	0.3	20.8	21.5	37.6
19	C	98	1.9	39.3	4.4	4.8	25.7	0.2	28.2	28.4	63.9
	JAVA	87	2.1	33.0	3.0	4.6	21.4	0.5	26.1	26.7	51.9
	PYTHON	70	2.4	40.9	2.5	4.5	20.5	0.3	27.8	28.0	56.5

accurately identify instances of example (3) [42]. In this case, methods defined but never called would not be connected to the main component. Finally, instances of the fourth example demand knowledge about the flows of execution of the program and, in a few cases, tracing usage/definition of variables used in the condition (note that the condition is not evaluated, thus only obvious instances will be detected). The lighter representation fulfilling these needs is the SDG, as the PDG cannot represent multiple procedures [8]. Furthermore, the SDG is able to capture all enumerated examples of dead code.

4.2. Use Case: Measuring Program Similarity

Measuring program similarity is a common approach in automated assessment with diverse applications. On the one hand, clustering programs facilitates the generation of feedback, as the same feedback given to a previous solution in the group likely fits the new program. This is especially used in semi-automated assessment tools to reduce the number of programs that the instructors need to analyze by reusing feedback given to a prior solution to those marked as identical. Other techniques use a correct solution from the same cluster as the attempt to match against and compute the differences to generate feedback on correctness, possibly with fixes. On the other hand, plagiarism detection, which is a must-have add-on to any automated assessment system, is also solved through similarity measuring.

The similarity of programs is typically measured on intermediate representations of source code. In particular, several recent works rely on semantic graph representations due to their many advantages over textual representations, such as the ability to capture programs' strategy (e.g., iterative or recursive approach). This is the case of program clustering, where it is typically intended to group programs adopting the same strategy and/or at an identical phase of development. This is well-captured in the control flow of a program. For instance, Naudé et al. [41] clusters programs by system dependence graph similarity, with the goal of selecting the most similar model solution. This solution will be compared against the attempt to measure program correctness. Zougari et al. [81] and the prototype tool eGrader [5] follow the same approach, after a xUnit-based testing step that ensures incorrect submissions are not marked as correct, but using a control flow graph and a combination of control and system dependence graphs as the graph representations, respectively. LAV [71] also combines such approach based on the control flow graph representation with bug finding and traditional *blackbox* testing techniques to estimate a composed grade. Nevertheless, suggesting a correction to the program code as feedback to the student, requires knowledge about the syntax [24].

In the case of plagiarism detection, these representations offer resilience against most code obfuscation techniques such as renaming variables and classes, changing order of independent instructions, among others [60, 10]. Even though plagiarism can be checked using any of these representations, resilience to a specific obfuscation method as well as the accuracy of the identified plagiarism pairs depend on the chosen representation. Therefore, several semantic representations have been successfully explored such as control flow graph [10], program dependence graph [60, 35], and system dependence graph [73].

4.3. Use Case: Automated Feedback on Correctness for Programming Assignments

Automated feedback on students' solutions to programming assignments is crucial for them to develop practical programming skills. Even though the most widely adopted assessment technique consists of executing the program with a specific input and comparing the obtained output against the expected output [43], the feedback that can be generated from such a technique is too limited for formative purposes [32]. Consequently, several static analysis approaches have been introduced that reason over semantic representations of the program's source code to derive feedback to help learners understand and improve their programs [43].

Evaluating the correctness of source code automatically through semantic representations cannot be done accurately, as any small modification (e.g., changing the order of operands in a binary comparison) can affect the result of the program. Only the AST is capable of representing such minor details, but classifying a program as correct or wrong based on its AST requires the set of all correct AST solutions (infinite) to be known. However, the logic behind correct solutions, greatly captured through semantic representations, is typically common to a group of solutions. The number of such groups can be limited, e.g., solutions to calculate the factorial belong to one of two groups: recursive and iterative. Solutions of the same group share certain characteristics such as the number of loops, number of possible execution flows, or the variable usage/definition dependencies.

Based on that, several approaches measure the similarity of a submitted program's representation to that of known solutions, i.e., the more similar the program is, the better grade it receives. In this sense, any of the representations can be used, but the CFG [70, 82] - which focuses on the structure of a program - is the most widely used. The PDG (and its extension, the SDG) are also commonly used representations for similarity-based grading [72], as besides structure information they also capture data dependencies. Even though being fully similar to a solution does not guarantee the program's correctness, if differences are found they may contain good hints on improving both the program's correctness and quality (e.g., there is a nested loop, between X and Y, which could be a simple loop).

Furthermore, after knowing the closest working solution, the ASTs of both the submitted program and the referred solution can be compared to find the minimal set of transformations that convert the submitted program into the correct solution [24]. Moreover, the AST has also been used to learn program transformations that fixed students' incorrect programs in the past, aiming to suggest fixes as feedback to students with similar faults [54]. The CPG, even though compiles all information of the other representations used, has not been explored yet in generating feedback on correctness for programming assignments, due to its recency.

4.4. Summary

As these use cases demonstrate, representations may have different applications within the same task. Hence, it is essential to thoroughly define the task requirements to choose the representation that satisfies them at minimal cost. Table 4 summarizes the fulfillment of the use cases' requirements by representations. We have considered 3 levels of fulfillment: complete (•), partial (○), and none.

Table 4. Summary of the results of use case analysis

	AST	DFG	CFG	EOG	CG	PDG	SDG	CPG
Dead Code Analysis	•	○	○	○	○	○	•	•
Measuring Program Similarity	Any can be applied, depends on what one wants to compare.							
Automated Feedback on Correctness	•		○	○	○	○	○	•

5. Conclusions

Reasoning about semantic representations of source code is gaining popularity as a technique in automated assessment of programming assignments, mainly for generating feedback. Such representations are easily manageable by a computer, encoding a set of features of the source code without unnecessary "noise". As feedback provided by automated assessment tools must be real-time and meaningful, selecting or deriving an appropriate representation that optimizes both the quality of generated feedback and resource consumption is crucial. However, there is no study that summarizes what each representation offers and compares their practical costs, much less towards our goal.

This paper provides a brief summary of how most semantic graph representations are constructed from the original source code, some of their recent applications reported in the literature, and examples. In addition, a benchmark of the different representations has been conducted to evaluate their cost, in terms of size and compilation complexity, using the AST as a baseline. The results show that, on the one hand, the CPG is the most feature-rich representation, but also the largest, i.e., it requires more edges, more properties on the edges, and more space. On the other hand, the CG is the lightest representation, but it only represents the calling relationships between the subroutines of the program.

We examined three use cases to determine which representations are more appropriate for each. For the most common examples of dead code, the PDG is sufficient. As for measuring source code similarity, we can do on any representation, but to achieve a good balance between accuracy and resilience to obfuscation, it is common to use representations that encode control flow (CFG or EOG). The last use case refers to our main target, i.e., the automatic generation of correctness feedback for programming assignments. Typically, this requires both syntactic and structural information. There are approaches that use CFG, PDG, SDG, and AST, as well as some that combine multiple representations in the same or separate phases. Therefore, we propose that our automated feedback mechanism should use (1) a control flow-based representation (e.g., CFG or EOG) to select the closest correct solution from the dataset of past solutions and (2) a representation combining syntactic and structural details to find out the meaningful differences between the wrong attempt and the correct one selected. For this, we can explore the CPG (which remains unexplored behind security topics) as the EOG is already part of it and it combines syntactic and structural data, having a cost not significantly greater than the sum of any two other representations providing such information. Furthermore, since it is a queryable graph, instructors can query it for bad coding patterns and identify common errors, among other things.

With this in mind, we have already collected and published a dataset – PROGpedia [44] – composed of the CPGs of the source code submitted on an automated assessment system to 16 programming exercises proposed in multiple years within the 2003-2020 time span to undergraduate Computer Science students. Considering the goal of our project (i.e., build a novel automated feedback mechanism), this dataset (the first of its type) is useful as it concentrates on the semantic and syntactic value of the source code of all previous attempts to solve a set of programming problems. Therefore, for any new attempt we can: (1) match it with the group of solutions following the same/similar strategy; (2) identify the closest correct solution; and (3) derive feedback from the differences that transform the current CPG into a correct one. Furthermore, these data can also help understanding the students' program development process, the relationship between the different problem-solving strategies and difficulty in achieving a correct solution, among other research problems.

Admittedly, some semantic graph representations, such as the Program-derived Semantics Graph (PSG) [30] and the Context-Aware Parse Tree (CAPT) [78], were not considered in this study due to a lack of both space and time to implement their extraction algorithm. These representations, as well as those that encode the dynamic behavior of a program, such as the Program Interaction Dependency Graph (PIDG) [11], are definitely worthy of a new study.

Several platforms for programming education could make use of these findings. These include not only automated assessment tools such as those described in the latest literature review of the area [43], but also intelligent tutoring systems [39], program visualization tools [62], among others. The improvements made to the existing CPG tool [21] are available upon request to the first author's email.

References

1. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. p. 38–49. ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA (2015), https://doi.org/10.1145/2786805.2786849

2. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs (2018)
3. Allen, F.E.: Control flow analysis. In: Proceedings of a Symposium on Compiler Optimization. p. 1–19. Association for Computing Machinery, New York, NY, USA (1970), https://doi.org/10.1145/800028.808479
4. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: A general path-based representation for predicting program properties. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 404–419. PLDI 2018, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3192366.3192412
5. AlShamsi, F., Elnagar, A.: An automated assessment and reporting tool for introductory java programs. In: 2011 International Conference on Innovations in Information Technology. pp. 324–329. IEEE, Abu Dhabi, United Arab Emirates (April 2011)
6. Arabnejad, H., Bispo, J., Cardoso, J.M.P., Barbosa, J.G.: Source-to-source compilation targeting OpenMP-based automatic parallelization of c applications. The Journal of Supercomputing 76(9), 6753–6785 (Dec 2019), https://doi.org/10.1007/s11227-019-03109-9
7. Arora, V., Bhatia, R.K., Singh, M.P.: Evaluation of flow graph and dependence graphs for program representation. International Journal of Computer Applications 56, 18–23 (2012)
8. Binkley, D.: Interprocedural constant propagation using dependence graphs and a data-flow model. In: Fritzson, P.A. (ed.) Compiler Construction. pp. 374–388. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
9. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 62–73. POPL '03, Association for Computing Machinery, New York, NY, USA (2003), https://doi.org/10.1145/604131.604137
10. Chae, D.K., Ha, J., Kim, S.W., Kang, B., Im, E.G.: Software plagiarism detection: A graph-based approach. In: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management. p. 1577–1580. CIKM '13, Association for Computing Machinery, New York, NY, USA (2013), https://doi.org/10.1145/2505515.2507848
11. Cheers, H., Lin, Y.: A novel graph-based program representation for java code plagiarism detection. In: Proceedings of the 3rd International Conference on Software Engineering and Information Management. p. 115–122. ICSIM '20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3378936.3378960
12. Chen, P., Liu, J., Chen, H.: Matryoshka: Fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 499–513. CCS '19, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3319535.3363225
13. DeFreez, D., Thakur, A.V., Rubio-González, C.: Path-based function embedding and its application to error-handling specification mining. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 423–433. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3236024.3236059
14. Devlin, J., Uesato, J., Singh, R., Kohli, P.: Semantic code repair using neuro-symbolic transformation networks (2017)
15. Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., Wang, K.: Hoppity: Learning graph transformations to detect and fix bugs in programs. In: 8th International Conference on Learning Representations. OpenReview.net, Addis Ababa, Ethiopia (April 2020), https://openreview.net/forum?id=SJeqs6EFvB
16. Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., Jiang, Y.: Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In: Proceedings of the 41st International Conference on Software Engineering. p. 60–71. ICSE '19, IEEE Press, Montreal, Quebec, Canada (2019), https://doi.org/10.1109/ICSE.2019.00024

17. Du, Y., Wang, J., Li, Q.: An android malware detection approach using community structures of weighted function call graphs. IEEE Access 5, 17478–17486 (Jun 2017)
18. Duan, Y., Li, X., Wang, J., Yin, H.: DeepBinDiff: Learning program-wide code representations for binary diffing. In: Proceedings 2020 Network and Distributed System Security Symposium. Internet Society, San Diego, CA, USA (2020), https://doi.org/10.14722/ndss.2020.24311
19. Eder, S., Junker, M., Jürgens, E., Hauptmann, B., Vaas, R., Prommer, K.H.: How much does unused code matter for maintenance? In: 2012 34th International Conference on Software Engineering (ICSE). pp. 1102–1111. IEEE, Zurich, Switzerland (June 2012)
20. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (Jul 1987), https://doi.org/10.1145/24039.24041
21. Fraunhofer AISEC: Code property graph. https://github.com/Fraunhofer-AISEC/cpg (2022)
22. Galindo, C., Pérez, S., Silva, J.: Slicing unconditional jumps with unnecessary control dependencies. In: Fernández, M. (ed.) Logic-Based Program Synthesis and Transformation. pp. 293–308. Springer International Publishing, Cham (2021)
23. Ge, X., Pan, Y., Fan, Y., Fang, C.: Amdroid: Android malware detection using function call graphs. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 71–77. IEEE, Sofia, Bulgaria (July 2019)
24. Gulwani, S., Radiček, I., Zuleger, F.: Automated clustering and program repair for introductory programming assignments. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 465–480. PLDI 2018, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3192366.3192387
25. Harer, J.A., Kim, L.Y., Russell, R.L., Ozdemir, O., Kosta, L.R., Rangamani, A., Hamilton, L.H., Centeno, G.I., Key, J.R., Ellingwood, P.M., McConley, M.W., Opper, J.M., Chin, S.P., Lazovich, T.: Automated software vulnerability detection with machine learning. CoRR abs/1803.04497 (2018), http://arxiv.org/abs/1803.04497
26. Hellendoorn, V.J., Sutton, C., Singh, R., Maniatis, P., Bieber, D.: Global relational models of source code. In: 8th International Conference on Learning Representations. OpenReview.net, Addis Ababa, Ethiopia (Apr 2020), https://openreview.net/forum?id=B1lnbRNtwr
27. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. p. 35–46. PLDI '88, Association for Computing Machinery, New York, NY, USA (1988), https://doi.org/10.1145/53990.53994
28. Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. p. 234–245. PLDI '90, Association for Computing Machinery, New York, NY, USA (1990), https://doi.org/10.1145/93542.93574
29. Huo, X., Li, M., Zhou, Z.H.: Control flow graph embedding based on multi-instance decomposition for bug localization. Proceedings of the AAAI Conference on Artificial Intelligence 34(04), 4223–4230 (Apr 2020), https://ojs.aaai.org/index.php/AAAI/article/view/5844
30. Iyer, R.G., Sun, Y., Wang, W., Gottschlich, J.: Software language comprehension using a program-derived semantics graph. CoRR abs/2004.00768 (2020), https://arxiv.org/abs/2004.00768
31. vahab karuthedath, A., Vijayan, S., S, V.K.K.: System dependence graph based test case generation for object oriented programs. In: 2020 International Conference on Power, Instrumentation, Control and Computing (PICC). pp. 1–6. IEEE, Thrissur, India (Dec 2020), https://doi.org/10.1109/picc51425.2020.9362460
32. Kyrilov, A., Noelle, D.C.: Binary instant feedback on programming exercises can reduce student engagement and promote cheating. In: Proceedings of the 15th Koli Calling Conference on Computing Education Research. p. 122–126. Koli Calling '15, Association for Computing Machinery, New York, NY, USA (2015), https://doi.org/10.1145/2828959.2828968

33. Lévai, T., Németh, F., Raghavan, B., Rétvári, G.: Batchy: Batch-scheduling data flow graphs with service-level objectives. In: Bhagwan, R., Porter, G. (eds.) 17th USENIX Symposium on Networked Systems Design and Implementation. pp. 633–649. USENIX Association, Santa Clara, CA, USA (Feb 2020), https://www.usenix.org/conference/nsdi20/presentation/levai
34. Li, W., Saidi, H., Sanchez, H., Schäf, M., Schweitzer, P.: Detecting similar programs via the weisfeiler-leman graph kernel. In: Kapitsaki, G.M., Santana de Almeida, E. (eds.) Software Reuse: Bridging with Social-Awareness. pp. 315–330. Springer International Publishing, Cham (2016)
35. Liu, C., Chen, C., Han, J., Yu, P.S.: Gplag: Detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 872–881. KDD '06, Association for Computing Machinery, New York, NY, USA (2006), https://doi.org/10.1145/1150402.1150522
36. Malik, R.S., Patra, J., Pradel, M.: Nl2type: Inferring javascript function types from natural language information. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 304–315. IEEE, Montreal, QC, Canada (May 2019)
37. Mantyla, M., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. pp. 381–384. IEEE, Amsterdam, Netherlands (Sep 2003)
38. Mehrotra, N., Agarwal, N., Gupta, P., Anand, S., Lo, D., Purandare, R.: Modeling functional similarity in source code with graph-based siamese networks (2020)
39. Mousavinasab, E., Zarifsanaiey, N., Kalhori, S.R.N., Rakhshan, M., Keikha, L., Saeedi, M.G.: Intelligent tutoring systems: a systematic review of characteristics, applications, and evaluation methods. Interactive Learning Environments 29(1), 142–163 (2021), https://doi.org/10.1080/10494820.2018.1558257
40. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 330–341. POPL '04, Association for Computing Machinery, New York, NY, USA (2004), https://doi.org/10.1145/964001.964029
41. Naudé, K.A., Greyling, J.H., Vogts, D.: Marking student programs using graph similarity. Computers & Education 54(2), 545 – 561 (2010), http://www.sciencedirect.com/science/article/pii/S0360131509002450
42. Obbink, N.G., Malavolta, I., Scoccia, G.L., Lago, P.: An extensible approach for taming the challenges of javascript dead code elimination. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 291–401. IEEE, Campobasso, Italy (March 2018)
43. Paiva, J.C., Leal, J.P., Figueira, Á.: Automated assessment in computer science education: A state-of-the-art review. ACM Trans. Comput. Educ. (jan 2022), https://doi.org/10.1145/3513140, just Accepted
44. Paiva, J.C., Leal, J.P., Figueira, Á.: Progpedia: Collection of source-code submitted to introductory programming assignments. Data in Brief 46, 108887 (2023), https://www.sciencedirect.com/science/article/pii/S2352340923000057
45. Pereira, N., Pereira, M.J.V., Henriques, P.R.: Comment-based Concept Location over System Dependency Graphs. In: Pereira, M.J.V., Leal, J.P., Simões, A. (eds.) 3rd Symposium on Languages, Applications and Technologies. OpenAccess Series in Informatics (OASIcs), vol. 38, pp. 51–58. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014), http://drops.dagstuhl.de/opus/volltexte/2014/4558
46. Podgurski, A., Clarke, L.: The implications of program dependencies for software testing, debugging, and maintenance. In: Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification. p. 168–178. TAV3, Association for Computing Machinery, New York, NY, USA (1989), https://doi.org/10.1145/75308.75328

47. Pradel, M., Gousios, G., Liu, J., Chandra, S.: Typewriter: Neural type prediction with search-based validation. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 209–220. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3368089.3409715
48. Pradel, M., Sen, K.: Deepbugs: A learning approach to name-based bug detection. Proc. ACM Program. Lang. 2(OOPSLA) (Oct 2018), https://doi.org/10.1145/3276517
49. Qingfeng, D., Kun, S., Kanglin, Y., Juan, Q.: Metrics analysis based on call graph of class methods. In: 2017 International Conference on Progress in Informatics and Computing (PIC). pp. 18–24. IEEE, Nanjing, China (Dec 2017)
50. Ragkhitwetsagul, C., Krinke, J.: Siamese: Scalable and incremental code clone search via multiple code representations. Empirical Softw. Engg. 24(4), 2236–2284 (Aug 2019), https://doi.org/10.1007/s10664-019-09697-7
51. Ray, M., lal Kumawat, K., Mohapatra, D.P.: Source code prioritization using forward slicing for exposing critical elements in a program. Journal of Computer Science and Technology 26(2), 314–327 (Mar 2011), https://doi.org/10.1007/s11390-011-9438-1
52. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from "big code". In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 111–124. POPL '15, Association for Computing Machinery, New York, NY, USA (2015), https://doi.org/10.1145/2676726.2677009
53. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming 58(1), 206–263 (2005), https://www.sciencedirect.com/science/article/pii/S0167642305000493, special Issue on the Static Analysis Symposium 2003
54. Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., Hartmann, B.: Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering. p. 404–415. ICSE '17, IEEE Press, Buenos Aires, Argentina (2017), https://doi.org/10.1109/ICSE.2017.44
55. Romano, S.: Dead code. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 737–742. IEEE, Madrid, Spain (Sep 2018)
56. Romano, S., Scanniello, G., Sartiani, C., Risi, M.: A graph-based approach to detect unreachable methods in java software. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. p. 1538–1541. SAC '16, Association for Computing Machinery, New York, NY, USA (2016), https://doi.org/10.1145/2851613.2851968
57. Ryder, B.G.: Constructing the call graph of a program. IEEE Transactions on Software Engineering SE-5(3), 216–226 (May 1979)
58. Sawadpong, P., Allen, E.B.: Software defect prediction using exception handling call graphs: A case study. In: 2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE). pp. 55–62. IEEE, Orlando, FL, USA (Jan 2016)
59. Shang, S., Zheng, N., Xu, J., Xu, M., Zhang, H.: Detecting malware variants via function-call graph similarity. In: 2010 5th International Conference on Malicious and Unwanted Software. pp. 113–120. IEEE, Nancy, France (Oct 2010)
60. Silva, C.D.S., Ferreira da Costa, L., Rocha, L.S., Viana, G.V.R.: Knn applied to pdg for source code similarity classification. In: Intelligent Systems: 9th Brazilian Conference, BRACIS 2020, Rio Grande, Brazil, October 20–23, 2020, Proceedings, Part II. p. 471–482. Springer-Verlag, Berlin, Heidelberg (2020), https://doi.org/10.1007/978-3-030-61380-8_32
61. Singh, P., Batra, S.: A novel technique for call graph reduction for bug localization. International Journal of Computer Applications 47(15), 1–5 (2012)
62. Sorva, J., Karavirta, V., Malmi, L.: A review of generic program visualization systems for introductory programming education. ACM Trans. Comput. Educ. 13(4) (nov 2013), https://doi.org/10.1145/2490822

63. Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 112–122. PLDI '07, Association for Computing Machinery, New York, NY, USA (2007), https://doi.org/10.1145/1250734.1250748
64. Suk, J.H., Lee, Y.B., Lee, D.H.: Score: Source code optimization & reconstruction. IEEE Access 8, 129478–129496 (2020)
65. The Graphviz Project: Dot language. https://graphviz.org/doc/info/lang.html (2022)
66. Theodoridis, T., Rigger, M., Su, Z.: Finding missed optimizations through the lens of dead code elimination. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 697–709. ASPLOS 2022, Association for Computing Machinery, New York, NY, USA (2022), https://doi.org/10.1145/3503222.3507764
67. van Tonder, R., Le Goues, C.: Lightweight multi-language syntax transformation with parser parser combinators. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 363–378. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3314221.3314589
68. Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D.: Deep learning similarities from different representations of source code. In: Proceedings of the 15th International Conference on Mining Software Repositories. p. 542–553. MSR '18, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3196398.3196431
69. Turhan, B., Kocak, G., Bener, A.: Data mining source code for locating software bugs: A case study in telecommunication industry. Expert Systems with Applications 36(6), 9986–9990 (2009), https://www.sciencedirect.com/science/article/pii/S0957417408009275
70. Vujošević-Janičić, M., Nikolić, M., Tošić, D., Kuncak, V.: Software verification and graph similarity for automated evaluation of students' assignments. Inf. Softw. Technol. 55(6), 1004–1016 (jun 2013), https://doi.org/10.1016/j.infsof.2012.12.005
71. Vujošević-Janičić, M., Nikolić, M., Tošić, D., Kuncak, V.: Software verification and graph similarity for automated evaluation of students' assignments. Information and Software Technology 55(6), 1004–1016 (Jun 2013), https://linkinghub.elsevier.com/retrieve/pii/S0950584912002406
72. Wang, T., Su, X., Wang, Y., Ma, P.: Semantic similarity-based grading of student programs. Inf. Softw. Technol. 49(2), 99–107 (feb 2007), https://doi.org/10.1016/j.infsof.2006.03.001
73. Wang, T., Wang, K., Su, X., Ma, P.: Detection of semantically similar code. Frontiers of Computer Science 8(6), 996–1011 (Dec 2014), https://doi.org/10.1007/s11704-014-3430-1
74. White, M., Tufano, M., Vendome, C., Poshyvanyk, D.: Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. p. 87–98. ASE 2016, Association for Computing Machinery, New York, NY, USA (2016), https://doi.org/10.1145/2970276.2970326
75. Xiaomeng, W., Tao, Z., Runpu, W., Wei, X., Changyu, H.: Cpgva: Code property graph based vulnerability analysis by deep learning. In: 2018 10th International Conference on Advanced Infocomm Technology (ICAIT). pp. 184–188. IEEE, Stockholm, Sweden (Aug 2018)
76. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. pp. 590–604. IEEE, Berkeley, CA, USA (May 2014)
77. Yan, Z., Qian, W., Yang, Z., Zeng, W., Yang, X., Li, A.: Tffv: Translator from eos smart contracts to formal verification language. In: Sun, X., Wang, J., Bertino, E. (eds.) Artificial Intelligence and Security. pp. 652–663. Springer Singapore, Singapore (2020)
78. Ye, F., Zhou, S., Venkat, A., Marcus, R., Petersen, P., Tithi, J.J., Mattson, T., Kraska, T., Dubey, P., Sarkar, V., Gottschlich, J.: Context-aware parse trees. CoRR abs/2003.11118 (2020), https://arxiv.org/abs/2003.11118

79. Zhang, Y., Fu, W., Qian, X., Chen, W.: Program slicing based buffer overflow detection. Journal of Software Engineering and Applications 03(10), 965–971 (2010), https://doi.org/10.4236/jsea.2010.310113
80. Zhou, Y., Liu, S., Siow, J.K., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. CoRR abs/1909.03496 (2019), http://arxiv.org/abs/1909.03496
81. Zougari, S., Tanana, M., Lyhyaoui, A.: Hybrid assessment method for programming assignments. In: 2016 4th IEEE International Colloquium on Information Science and Technology (CiSt). pp. 564–569. IEEE, Tangier, Morocco (Oct 2016), http://ieeexplore.ieee.org/document/7805112/
82. Zougari, S., Tanana, M., Lyhyaoui, A.: Towards an automatic assessment system in introductory programming courses. In: 2016 International Conference on Electrical and Information Technologies (ICEIT). pp. 496–499. IEEE, Tangiers, Morocco (May 2016)

José Carlos Paiva graduated and took his MSc in Computer Science, at the Faculty of Sciences of the University of Porto. He is currently working on his PhD in Computer Science from the same institution, entitled *Reasoning on Semantic Representations of Source Code to Support Programming Education*. He is also a research assistant at the Center for Research in Advanced Computing Systems (CRACS), a R&D unit of INESC TEC Research Laboratory, since 2014. His main research interests are automated assessment of programming tasks, gamification, e-learning and web-based learning, and machine learning.

José Paulo Leal graduated in Mathematics at the Faculty of Sciences of the University of Porto, and has a PhD in Computer Science from the same institution. His current position is auxiliary professor at the Computer Science department of the Faculty of Sciences of the University of Porto. He is also affiliated with the Center for Research in Advanced Computing Systems (CRACS), a R&D unit of INESC TEC Research Laboratory, where he is an effective member. His main research interests are technology enhanced learning, web adaptability and semantic web.

Álvaro Figueira graduated in Mathematics Applied to Computer Science, from Faculty of Sciences of the University of Porto in 1995, and took his MSc in Foundations of Advanced Information Technology, from Imperial College, London, in 1997. In 2004, he concluded his PhD in Computer Science in concurrent and distributed programming. He is currently an Assistant Professor, with tenure, at the Faculty of Sciences in University of Porto. He is also a researcher in the CRACS Research Unit where he has been leading international projects involving University of Porto, University of Texas at Austin, University of Coimbra and University of Aveiro, regarding the automatic detection of relevance in social networks. His main research interests are in the areas of text and web mining, community detection, e-learning and web-based learning and standards in education.

Received: June 15, 2023; Accepted: November 15, 2023.