MISIM: A Novel Code Similarity System

Fangke Ye [*1 2] **Shengtian Zhou** [*1] **Anand Venkat** [1] **Ryan Marcus** [1 3] **Nesime Tatbul** [1 3] **Jesmin Jahan Tithi** [1]
Niranjan Hasabnis [1] **Paul Petersen** [4] **Timothy Mattson** [1] **Tim Kraska** [3] **Pradeep Dubey** [1] **Vivek Sarkar** [2]
Justin Gottschlich [1 5]

Abstract

Semantic code similarity can be used for a range of applications such as code recommendation and automated software defect correction. Yet, code semantic similarity systems are still in their infancy in terms of accuracy for general purpose code. To help address this, we present *Machine Inferred Code Similarity* (MISIM), a novel end-to-end code similarity system that consists of two core components. First, MISIM uses a novel *context-aware semantic structure*, which is designed to aid in lifting semantic meaning from code syntax. Second, MISIM provides an open-ended neural code similarity scoring algorithm, which can be implemented with various neural network architectures with learned parameters. We compare MISIM to four state-of-the-art systems across 328k programs (18+ million lines of code) and show it has up to $43.4\times$ better accuracy.

1. Introduction

The field of *machine programming* (MP) is concerned with the automation of software development (Gottschlich et al., 2018). In recent years, there has been an emergence of many MP systems, due, in part, to advances in machine learning, formal methods, data availability, and computing efficiency (Allamanis et al., 2018a; Alon et al., 2018; 2019b;a; Ben-Nun et al., 2018; Cosentino et al., 2017; Li et al., 2017; Luan et al., 2019; Odena & Sutton, 2020; Finkel & Laguna, 2020; Tufano et al., 2018; Wei & Li, 2017; Zhang et al., 2019; Zhao & Huang, 2018). An open challenge in MP is the construction of accurate *code similarity* systems. *Code similarity* is the problem of determining if two or more code snippets have some degree of semantic similarity (or equivalence), sometimes even in the presence of syntactic dissimilarity. At the highest level, code similarity systems aim to determine if two or more code snippets are solving a similar problem, even if the implementations they use differ (e.g., various algorithms of `sort()` (Cormen et al., 2009)).

Code semantic similarity systems can be used to improve programmer productivity with tools such as code recommendation, automated bug detection, and language-to-language transformation for small kernels (e.g., stencils), to name a few (Allamanis et al., 2018b; Ahmad et al., 2019; Bader et al., 2019; Barman et al., 2016; Bhatia et al., 2018; Dinella et al., 2020; Kamil et al., 2016; Luan et al., 2019; Pradel & Sen, 2018). Yet, state-of-the-art systems lack the accuracy to be used for general-purpose code. Without largely accurate code similarity systems, which can be leveraged to automate parts of software development (e.g., architecture-to-architecture code transformation), the growth of heterogeneous software and hardware may become untenable due to the shortage of software developers (Ahmad et al., 2019; Batra et al., 2018; Bogdan et al., 2019; Chen et al., 2020; Deng et al., 2020; Hannigan et al., 2019) . Yet some of the most fundamental questions in code similarity remain open. One open question is regarding the proper structural representation and approach to learn code semantics for similarity analysis (Alam et al., 2019; Allamanis et al., 2018b; Becker & Gottschlich, 2017; Ben-Nun et al., 2018; Dinella et al., 2020; Iyer et al., 2020; Luan et al., 2019). In this paper, we principally focus on this problem.

While prior work has explored some structural representations of code in the space of code similarity and understanding, these explorations are still in their early stages. For example, the abstract syntax tree (AST) is used in the code2vec and code2seq systems (Alon et al., 2019b;a), while two novel structures called the conteXtual flow graph (XFG) and the simplified parse tree (SPT) are used in Neural Code Comprehension (NCC) (Ben-Nun et al., 2018) and

[*]Equal contribution [1]Intel Labs. [2]Georgia Institute of Technology. [3]Massachusetts Institute of Technology. [4]Intel. [5]University of Pennsylvania.. Correspondence to: Fangke Ye <yefangke@gatech.edu>, Shengtian Zhou <shengtian.zhou@intel.com>, Anand Venkat <anand.venkat@intel.com>, Ryan Marcus <raynmarcus@csail.mit.edu>, Nesime Tatbul <tatbul@csail.mit.edu>, Jesmin Jahan Tithi <jesmin.jahan.tithi@intel.com>, Niranjan Hasabnis <niranjan.hasabnis@intel.com>, Paul Petersen <paul.petersen@intel.com>, Timothy Mattson <timothy.g.mattson@intel.com>, Tim Kraska <kraska@mit.edu>, Pradeep Dubey <pradeep.dubey@intel.com>, Vivek Sarkar <vsarkar@gatech.edu>, Justin Gottschlich <justin.gottschlich@intel.com>.

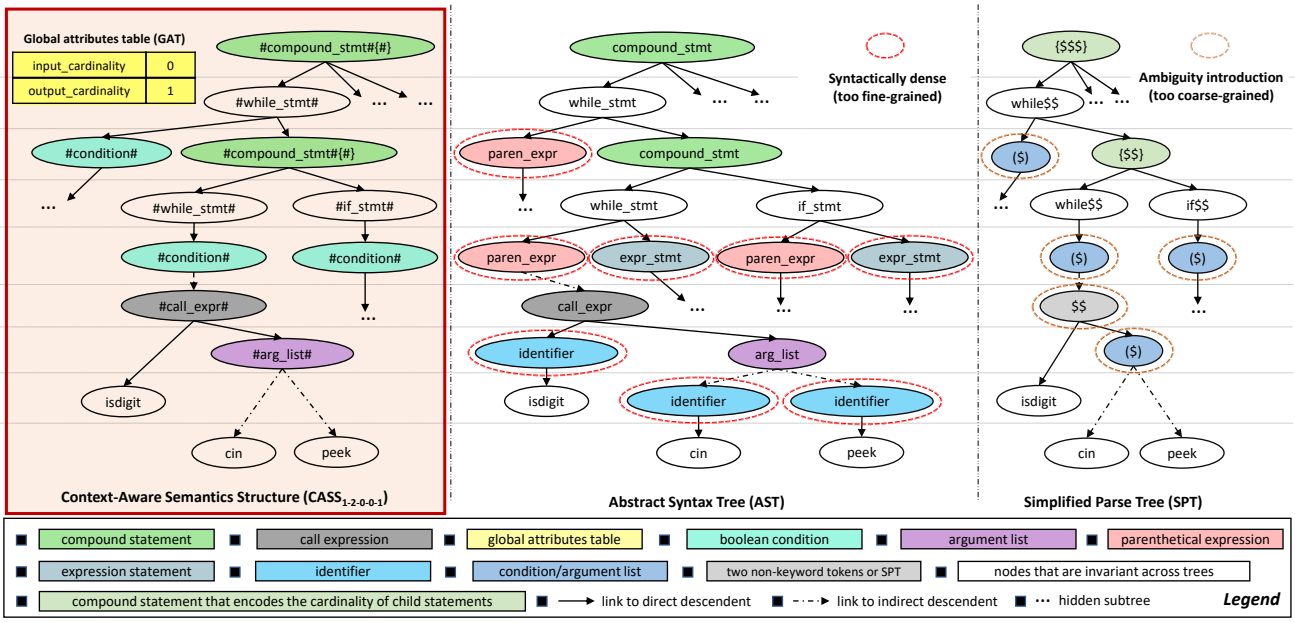


Figure 1. Context-aware semantic structure (CASS), abstract syntax tree (AST), and simplified parse tree (SPT) for Program A.

Aroma (Luan et al., 2019), respectively. While each of these representations has benefits in certain contexts, they possess one or more limitations when considered more broadly. For example, the AST – while having a notable historical importance for optimizing compilers – can be syntactically dense (see Figure 1). Such syntax density can often mislead code similarity systems into memorizing syntax, rather than learning semantics (i.e., the meaning behind the code). Alternatively, the XFG can capture important data dependencies, but is obtained from an intermediate representation (IR) that requires compilation. This restriction can limit its application, especially in interactive developer environments. Finally, the SPT is structurally driven, which enables it to lift certain semantic meaning from code, yet, it does not always resolve syntactic ambiguities. Instead, it can sometimes introduce them unnecessarily, due to its coarse-grain approach (see Figure 1).

Learning from these observations, we attempt to address some of the open questions around code similarity with our novel end-to-end code similarity system called *Machine Inferred Code Similarity* (MISIM). In this paper, we principally focus on two main novelties of MISIM and how they may improve code similarity analysis: *(i)* its structural representation of code, called the *context-aware semantic structure* (CASS), and *(ii)* its neural-based *learned* code similarity scoring algorithm. These components can be used individually or together as we have chosen to do.

This paper makes the following technical contributions:

- We present MISIM's *context-aware semantic structure* (CASS), a configurable representation of code designed to *(i)* lift semantic meaning from code syntax and *(ii)* provide an extensible representation that can be augmented as needed.
- We present MISIM's flexible deep neural network (DNN) similarity scoring framework for a given code corpus and show its efficacy across three DNN topologies: *(i)* bag-of-features, *(ii)* a recurrent neural network (RNN), and *(iii)* a graph neural network (GNN).
- We compare MISIM to four state-of-the-art code similarity systems: *(i)* code2vec, *(ii)* code2seq, *(iii)* Neural Code Comprehension, and *(iv)* Aroma. Our experimental evaluation, across 328,155 C/C++ programs comprising of over 18 million lines of code, illustrates that MISIM is more accurate than them, ranging from $1.5\times$ to $43.4\times$.

2. Background & Motivation

```
Program A
int main() {
  int a;
  while (!cin.eof()) {
    while (!cin.eof() && !isdigit(cin.peek())) // !digit
      cin.get(); // ignore
    if (cin >> a) cout << a << endl;
  }
  return 0;
}
```

Code representation is a core component of semantic code similarity analysis. Existing code representations can be generally categorized into two emerging spaces: syntax-based representations (e.g., abstract syntax trees) and semantic-based representations (e.g., conteXtual flow

graph). In this section, we provide a brief anecdotal analysis of these representations and discuss their strengths and weaknesses for semantic code similarity.

To ground the discussion, we analyze two simple programs (A and B) from POJ-104 dataset (Mou et al., 2016) that solve the same problem (number 88), which emits digits from input strings. While the implementations of programs A and B are syntactically dissimilar, they are semantically equivalent. That is, they both correctly solve the same problem. Code of program A is shown above[1]. MISIM's CASS, the AST, and the SPT representations of program A are shown in Figure 1.

Syntax-based representations. Compilers have successfully used syntax-based representations of programs, such as parse trees and abstract syntax trees (ASTs), for several decades (Baxter et al., 1998). More recently, the code understanding systems `code2vec` and `code2seq` have utilized ASTs as their basic representation of code for their program semantic analysis (Alon et al., 2019b;a).

Parse trees are typically built by a language-specific parser during the compilation process and faithfully capture every syntactic detail of the source program. In this regard, they are also called concrete syntax trees. ASTs, on the other hand, abstract away certain syntactic details (e.g., parentheses), which can simplify program analysis. However, ASTs can still be syntactically dense (see Figure 1), which can mislead code similarity systems into memorizing syntax, rather than learning semantics. For instance, the AST in Figure 1 represents the parenthesis of `while` statements in program A as `paren_expr` nodes, while missing the semantic binding they have to the condition expression of the `while` statements. Syntax-based representations also seem conceptually ill-fit for semantic similarity across different programming languages (PLs) due to potential structural and grammatical PL differences.

Semantic-based representations. Semantics-based representations capture the semantics of various program constructs instead of syntax. ConteXtual flow graph (XFG) used by NCC and simplified parse tree (SPT) used by Aroma are two such representations. NCC relies on the hypothesis that program statements that operate in similar contexts are semantically-similar, where the context of a program statement is defined as the statements surrounding that statement with direct data- and control-flow dependencies. They capture these dependencies for program statements at IR level by defining XFG representation. `inst2vec` embeddings are then trained for IR instructions by deriving insights from `word2vec` (Mikolov et al., 2013a) and skip-gram model (Mikolov et al., 2013b). Overall, NCC views instructions as semantically similar, if their embedding vectors are closer. Yet, NCC's dependency on a compiler IR restricts XFG to compilable code.

Aroma uses SPT as a program representation to enable code search and recommendation using machine learning. SPT is different than AST as SPT only consists of program tokens and does not use any special language-specific rule names. SPT is thus language-agnostic and enables uniform handling of different languages. In a nutshell, rather than looking for string-level or token-level matches for the program snippet used as a search query, Aroma builds SPT of the query program, featurizes it using manually-selected features, and performs dot product of the query's feature vector and that of the candidates to find similar code snippets.

Our analysis of SPT for code similarity between program A and B, however, reveals some limitations. Although SPT is structurally-driven (intentionally against the syntax-driven AST (Luan et al., 2019)), it may unintentionally inject semantic ambiguity. For instance, SPT in Figure 1 does not distinguish between the argument list of a function call (e.g., "`( cin.peek() )`") and the condition node of an `if` statement (e.g., "`( cin >> a )`"), and represents both of them by "`($)`". Moreover, it also seems to capture program details that intuitively seem irrelevant for the code similarity task. For instance, it captures the number of program statements in `main` in the root node (e.g., as `($$$)` for program A, and `($$$$$$$$)` for program B), yet while both of these programs have different number of statements they are semantically equivalent. Therefore featurizing on this metric could mislead and mistrain a machine learning system to infer such information is semantically meaningful.

We developed the *context-aware semantic structure* (CASS) to improve upon the limitations of the existing representations. At the highest level, CASS is a semantic-based representation that is configurable, which enables it to represent various existing representations (more details in Section 3). Figure 1 shows the CASS for program A, obtained from one of its 216 different configurations. In this example, CASS resolves ambiguity introduced by SPT by representing argument list and condition node of an `if` and `while` statement differently, while simultaneously eliding away unnecessary syntactic density produced by the AST (e.g., `identifier` nodes). These modifications help in improving CASS's accuracy, which, in this particular case, outperforms both the AST and SPT by more than a 4.3% margin. These accuracy improvements grow in magnitude as semantically similar programs become more complex and more syntactically divergent. We discuss these general results, leading to upwards of a $43.4\times$ accuracy improvement, in Section 4.

[1]Program B's code is in Appendix C.1 due to space restrictions.

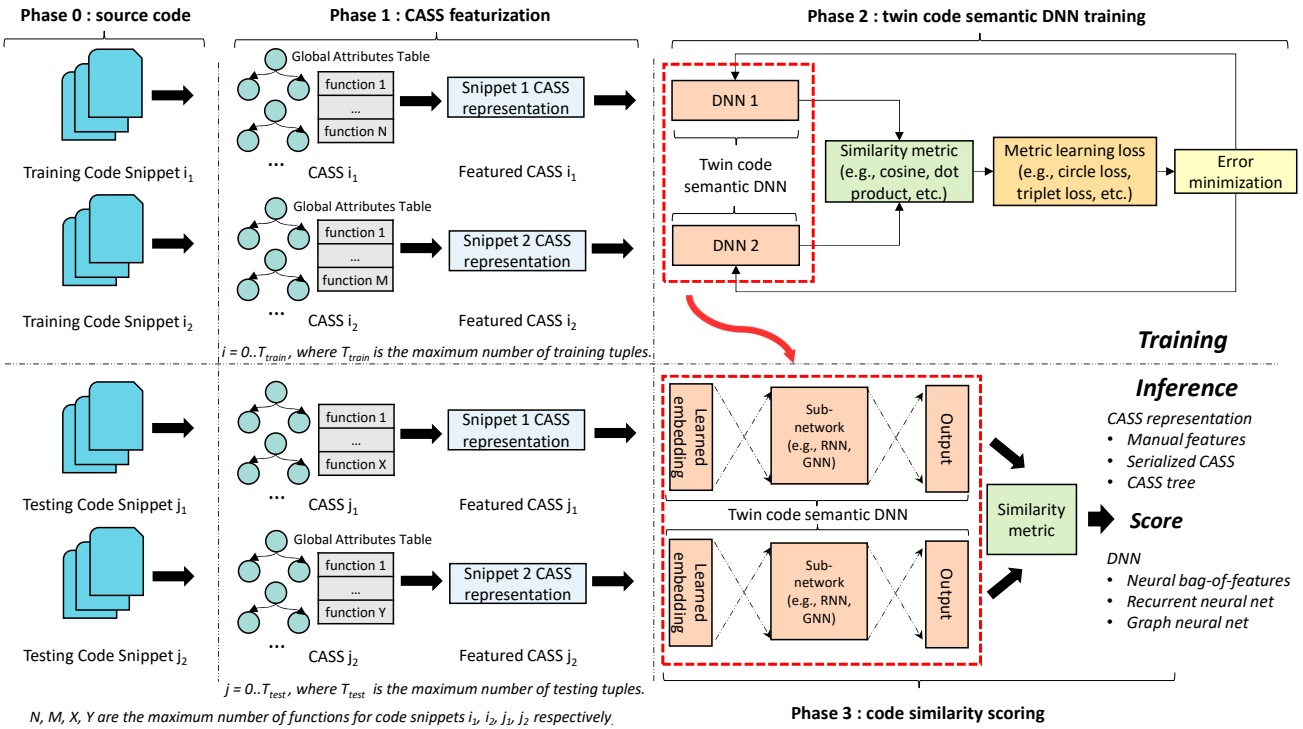


Figure 2. Overview of the MISIM System.

3. MISIM System

In Figure 2, we provide an overview of MISIM's system diagram. A core component of MISIM is the novel *context-aware semantic structure* (CASS), which aims to capture semantically salient properties of the input code. Moreover, CASS is designed to be *context-aware*. That is, it can capture information that describes the context of the code (e.g., parenthetical operator disambiguation between a function call, mathematical operator precedence, Boolean logic ordering, etc.) that may otherwise be ambiguous without such context-sensitivity. Once these CASSes are constructed, they are vectorized and used as input to a neural network, which produces a feature vector for a corresponding CASS. Once a feature vector is generated, a code similarity measurement (e.g., cosine similarity (Baeza-Yates & Ribeiro-Neto, 1999)) calculates the similarity score.

3.1. Context-Aware Semantic Structure (CASS)

We have designed CASS with the following guiding principles: ***(i)*** it should not require compilation, ***(ii)*** it should be a flexible representation that captures code semantics, and ***(iii)*** it should be capable of resolving code ambiguities in both its context sensitivity to the code and its environment. The first principle ***(i)*** originates from the observation that unlike programs written in higher-level scripting languages (e.g., Python (Van Rossum & Drake, 2009), JavaScript (Flanagan, 2006)), C/C++ programs found "in the wild" may not be well-formed (e.g., due to specialized compiler dependencies) or exhaustively include all of their dependencies (e.g., due to assumptions about library availability) and therefore may not compile. Moreover, for code recommendation systems that are expected to function in a live setting, requiring compilation may severely constrain their practical application. We address this by introducing a structure such that it does not require compilation (Section 3.1.1). The second ***(ii)*** and third ***(iii)*** principles originate from the observation that different scenarios may require attention to different semantics (e.g., embedded memory-bound systems may prefer to use algorithms that do not use recursion due to a potential call stack overflow) and that programming languages (PLs) evolve and new PLs continue to emerge. We attempt to address these issues with CASS's configuration categories (Section 3.1.2).

3.1.1. CASS TREE AND GLOBAL ATTRIBUTES TABLE

Here we provide an informal definition of CASS (a formal definition is in Appendix A). The CASS consists of one or more CASS trees and an optional global attributes table (GAT). A CASS tree is a tree, in which the root node represents the entire span of the code snippet. During the construction of a CASS tree, the program tokens are mapped to their corresponding node labels using the grammar of the high-level programming language. A CASS's GAT contains

exactly one entry per unique function definition in the code snippet. A GAT entry currently includes only the input and output cardinality values for each corresponding function, but can be extended as new global attributes are needed.

3.1.2. CASS CONFIGURATION CATEGORIES

In general, CASS configurations can be broadly classified into two categories: *language-specific* and *language-agnostic*. Exact values of the options for each of the configuration categories are described in Appendix A.1 in Table 3. Below we provide an intuitive description of the categories and their values.

Language-specific configurations (LSCs). Language-specific configurations are meant to capture semantic meaning by resolving syntactic ambiguities that could be present in the concrete syntax trees. It also introduces specificity related to the high-level programming language. For example, the parentheses operator is overloaded in many programming languages to enforce an order of evaluation of operands as well as to enclose a list of function arguments, amongst other things. CASS disambiguates these by explicitly embedding the semantic contextual information in the CASS tree nodes using the *node prefix label* (defined in Appendix A).

(A) Node Prefix Label. The configuration options for node prefix labels[2] correspond to various levels of semantic to syntactic information. In Table 3, option 0 corresponds to the extreme case of a concrete syntax embedding, option 1 corresponds to eliminating irrelevant syntax, and option 2 is principally equivalent to option 1, except it applies only to parentheticals, which we have identified – through empirical evaluation – to often have notably divergent semantic meaning based on context.

Language-agnostic configurations (LACs). LACs can improve code similarity analysis by unbinding overly-specific semantics that may be present in the original concrete syntax tree structure.

(B) Compound Statements. The *compound statements* configuration option enables the user to control how much non-terminal node information is incorporated into the CASS. Option 0 is equivalent to Aroma's SPT, option 1 omits separate features for compound statements altogether, and option 2 does not discriminate between compound statements of different lengths and specifies a special label to denote the presence of a compound statement.

(C) Global Variables. The *global variables* configuration specifies the degree of global variable-specific information contained in a CASS. In other words, it provides the user with the ability to control the level of abstraction – essentially binding or unbinding global variable names as needed. If all code similarity analysis will be performed against the same software program, retaining global variable names may help elicit deeper semantic meaning. If not, unbinding global variable names may improve semantic meaning.

(D) Global Functions. The *global functions* configuration serves the dual purpose of *(i)* controlling the amount of function-specific information to featurize and *(ii)* to explicitly disambiguate between the usage of global functions and global variables (a feature that is absent in Aroma's SPT design).

(E) Function I/O Cardinality. The *function I/O cardinality* configuration aims to abstract the semantics of certain groups of functions through input and output cardinality (i.e., embedded semantic information that can be implicitly derived by analyzing the number of input and output parameters of a function).

We have found that the specific context in which code similarity is performed seems to provide some indication of the optimal specificity of the CASS configuration. In other words, one specific CASS configuration is unlikely to work in all scenarios. To address this, CASS provides a number of options to control the language-specific and language-agnostic configurations. We discuss this in greater detail in Appendix A.2.

3.2. Neural Scoring Algorithm

MISIM's neural scoring algorithm aims to compute the similarity score of two input programs. The algorithm consists of two phases. The first phase involves a neural network model that maps a featurized CASS to a real-valued code vector. The second phase generates a similarity score between a pair of code vectors using a similarity metric.[3] We describe the details of the scoring model, its training strategy, and other neural network model choices in this section.

3.2.1. MODEL

We investigated three neural network approaches for MISIM's scoring algorithm: *(i)* a graph neural network (GNN), *(ii)* a recurrent neural network (RNN), and *(iii)* a bag of manual features (BoF) neural network. We name these models MISIM-GNN, MISIM-RNN, and MISIM-BoF respectively. The graphical nature of CASS, as well as the recent success in applying GNNs in the program domain (Allamanis et al., 2018b; Brockschmidt et al., 2019; Dinella et al., 2020; Wei et al., 2020), leads us to design the MISIM-GNN model that directly encodes the graphical structure of

[2] Analytically deriving the optimal selection of node prefix labels across all C/C++ code may be untenable. To accommodate this, we currently provide two levels of granularity for C/C++ node prefix labels in CASS.

[3] For this work, we have chosen cosine similarity as the similarity metric used within MISIM.

CASS. MISIM-RNN is based on (Hu et al., 2018), which serializes a CASS into a sequence and uses an RNN to encode the structure. Unlike the two aforementioned models, MISIM-BoF takes in not the CASS but a bag of manual features extracted from it, and uses a feed-forward network to encode them into a vector. We compared the three models in our experiments and observed that MISIM-GNN performed the best overall. Therefore, we describe it in detail in this section. Appendix B has details of the MISIM-RNN and MISIM-BoF models.

MISIM-GNN. In the MISIM-GNN model, an input program's CASS representation is transformed into a graph. Then, each node in the graph is embedded into a trainable vector, serving as the node's initial state. Next, a GNN is used to update each node's state iteratively. Finally, a global readout function is applied to extract a vector representation of the entire graph from the final states of the nodes. We describe each of these steps in more detail below.

Input Graph Construction. We represent each program as a single CASS instance. Each instance can contain one or more CASS trees, where each tree corresponds to a unique function of the program. The CASS instance is converted into a single graph representation to serve as the input to the model. The graph is constructed by first transforming each CASS tree and its GAT entry into an individual graph. These graphs are then merged into a single (disjoint) graph. For a CASS consisting of a CASS tree $T = (V, E)$ and a GAT entry a, we transform it into a directed graph $G = (V', E', R)$, where V' is the set of graph nodes, R is the set of edge types, and $E' = \{(v, u, r) \mid v, u \in V', r \in R\}$ is the set of graph edges. The graph is constructed as follows:

$$V' = V \cup \{a\}, \qquad R = \{p, c\},$$
$$E' = \{(v, u, p) \mid (v, u) \in E\} \cup \{(v, u, c) \mid (u, v) \in E\}.$$

The two edge types, p and c, represent edges from CASS tree nodes to their parent and children nodes, respectively.

Graph Neural Network. MISIM embeds each node $v \in V'$ in the input graph G into a vector by assigning a trainable vector to each unique node label (with the optional prefix) and GAT attribute. The node embeddings are then used as node initial states ($\mathbf{h}_v^{(0)}$) by a relational graph convolutional network (R-GCN (Schlichtkrull et al., 2018)) specified as the following:

$$\mathbf{h}_v^{(l)} = \mathrm{ReLU}\left(\frac{1}{\sum_{r\in R}|\mathcal{N}_v^r|}\sum_{r\in R}\sum_{u\in\mathcal{N}_v^r}\mathbf{W}_r^{(l)}\mathbf{h}_u^{(l-1)} + \mathbf{W}_0^{(l)}\mathbf{h}_v^{(l-1)}\right) \qquad v \in V', l \in [1, L],$$

where L is the number of GNN layers, $\mathcal{N}_v^r = \{u \mid (u, v, r) \in E'\}$ is the set of neighbors of v that connect to v through an edge of type $r \in R$, and $\mathbf{W}_r^{(l)}$, $\mathbf{W}_0^{(l)}$ are weight matrices to be learned.

Code Vector Generation. To obtain a code vector $\mathbf{c}$ that represents the entire input graph, we apply a graph-level readout function as specified below:

$$\mathbf{c} = \mathrm{FC}\left(\left[\begin{array}{c}\mathrm{AvgPool}\left(\left\{\mathbf{h}_v^{(L)} \,\middle|\, v \in V'\right\}\right)\\ \mathrm{MaxPool}\left(\left\{\mathbf{h}_v^{(L)} \,\middle|\, v \in V'\right\}\right)\end{array}\right]\right).$$

The output vectors of average pooling and max pooling on the nodes' final states are concatenated and fed into a fully-connected layer, yielding the code vector for the entire input program.

3.2.2. Training

We train the neural network model following the setting of metric learning (Schroff et al., 2015; Hermans et al., 2017; Musgrave et al., 2020; Sun et al., 2020), which tries to map input data to a vector space where, under a distance (or similarity) metric, similar data points are close together (or have large similarity scores) and dissimilar data points are far apart (or have small similarity scores). The metric we use is the cosine similarity in the code vector space. As shown in the lower half of Figure 2, we use pair-wise labels to train the model. Each pair of input programs are mapped to two code vectors by the model, from which a similarity score is computed and optimized using a metric learning loss function.

4. Experimental Evaluation

In this section, we analyze the performance of MISIM compared to code2vec, code2seq, NCC, and Aroma on two datasets containing a total of more than 328,000 programs.[4] Overall, we find that MISIM has improved performance than these systems across three metrics. We also perform an abbreviated analysis of two MISIM variants, each trained with a different CASS configuration, to provide insight into when different configurations may be better fit for different code corpora.

Table 1. Dataset statistics.

Split	GCJ		POJ-104	
	#Problems	#Programs	#Problems	#Programs
Training	237	223,171	64	28,137
Validation	29	36,409	16	7,193
Test	31	22,795	24	10,450
Total	297	282,375	104	45,780

Datasets. Our experiments are conducted on two datasets:

[4] Although other code similarity systems exist, we were not able to compare to them due to the differences in target languages, problem settings, and lack of open-source availability.

the Google Code Jam (GCJ) dataset (Ullah et al., 2019) and the POJ-104 dataset (Mou et al., 2016). The GCJ dataset consists of solutions to programming problems in Google's Code Jam coding competitions. We use a subset of it that consisting of C/C++ programs that solve 297 problems. The POJ-104 dataset consists of student-written C/C++ programs solving 104 problems. For both datasets, we label two programs as similar if they are solutions to the same problem. After a filtering step, which removes unparsable/non-compilable programs, we split each dataset by problem into three subsets for *training*, *validation*, and *testing*. Detailed statistics of the dataset partitioning are shown in Table 1.

Training. Unless otherwise specified, we use the same training procedure in all experiments. The models are built and trained using PyTorch (Paszke et al., 2019). To train the models, we use the Circle loss (Sun et al., 2020), a state-of-the-art metric learning loss function that has been tested effective in various similarity learning tasks. Following the P-K sampling strategy (Hermans et al., 2017), we construct a batch of programs by first randomly sampling 16 different problems, and then randomly sampling at most 5 different solutions for each problem. The loss function takes the similarity scores of all intra-batch pairs and their pair-wise labels as input. Further details about the training procedure and hyperparameters are discussed in Appendix C.2.

Evaluation Metrics. The accuracy metrics we use for evaluation are Mean Average Precision at R (MAP@R) (Musgrave et al., 2020), Average Precision (AP) (Baeza-Yates & Ribeiro-Neto, 1999), and Area Under Precision-Recall-Gain Curve (AUPRG) (Flach & Kull, 2015). Since these metrics are already defined, we do not detail them here, but would refer readers to Appendix C.4.

Configuration Identifier. In the following sections, we refer to a configuration of CASS by its unique identifier (ID). A configuration ID is formatted as A-B-C-D-E. Each of the five letters corresponds to a configuration type in the second column of Table 3, and will be replaced by an option number specified in the third column of the table. Configuration 0-0-0-0-0 corresponds to Aroma's SPT.

Results. Figure 3 shows the accuracy of MISIM, code2vec, code2seq, NCC, and Aroma[5]. The blue bars show the results of the MISIM system variants trained using the baseline CASS configuration of 0-0-0-0-0. The orange bars show the results of code2vec, code2seq, NCC, and Aroma. We observe that MISIM-GNN results in the best performance for MAP@R, yielding 1.5× to 43.4× improvements over the other systems. In some cases, MISIM-BoF achieves the best AP and AUPRG scores. In summary, MISIM system has better accuracy than the other systems we compared against across all three metrics.

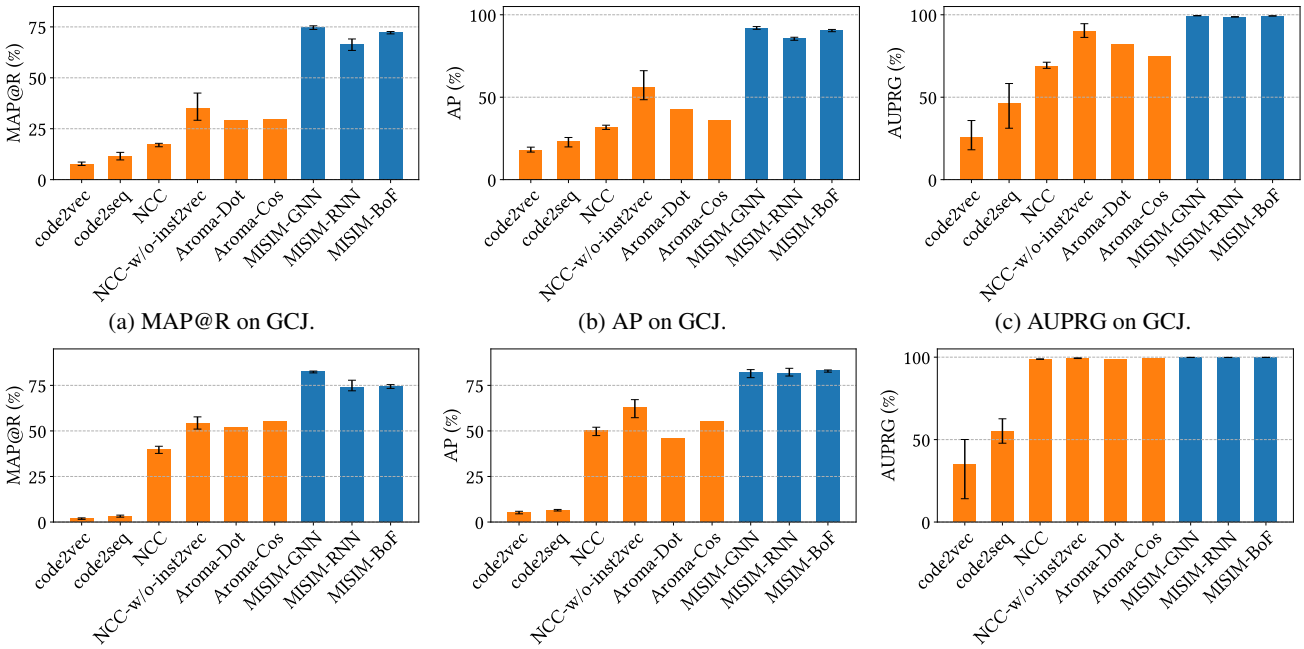
[5]A table for these results can be found in Appendix C.5.

Results Analysis. The code2vec and code2seq systems use paths in an abstract syntax tree (AST) as the input to its neural network. We speculate that such representation may *(i)* keep excessive fine-grained syntactical details while *(ii)* omitting structural (i.e., semantic) information. This may explain why code2vec and code2seq have smaller accuracy in our experiments. The Aroma system employs manual features derived from the simplified parse tree and computes the number of overlapping features from two programs as their similarity score. The selection of manual features appears to be heuristic-based and might potentially result in a loss in semantic information. NCC tries to learn code semantics from LLVM IR, a low-level code representation designed for compilers. The lowering process from source code to LLVM IR may discard some semantic-relevant information such as identifier names and syntactic patterns, which is usually not utilized by compilers, but might be useful for inferring code semantics. The absence of such information from NCC's input may limit its code similarity accuracy.

4.1. Specialized Experiment: CASS Configurations

Here we provide early anecdotal evidence indicating that perhaps no CASS configuration is invariably the best for all code. Instead, the configurations may need to be chosen based on the characteristics of code that MISIM will be trained on and, eventually, used for. We conducted a series of experiments that train MISIM-GNN models with two CASS configurations on several randomly sampled sub-training sets and compared their test accuracy. The two configurations used were C_1, which is the CASS baseline configuration of 0-0-0-0-0 and C_2, which is a non-baseline configuration of 2-2-3-2-1 that provides a higher abstraction over the source code than C_1 (e.g., replace global variable names with a unified string, etc.). Table 2 shows the results from four selected sub-training sets, named T_A, T_B, T_C, and T_D from POJ-104. It can be seen that when trained on T_A or T_B, the system using configuration C_2 performs better than the baseline configuration in all three accuracy metrics. However, using the training sets T_C or T_D, the results are inverted.

To better understand this divergence, we compared the semantic features of $T_A \cap T_B$ to $T_C \cap T_D$. We observed that some CASS-defined semantically salient features (e.g., global variables – see Appendix A) that C_2 had been customized to extract, occurred less frequently in $T_A \cap T_B$ than in $T_C \cap T_D$. We speculate that, in the context of the POJ-104 dataset, when global variables are used more frequently, they are more likely to have consistent meaning across different programs. As a result, abstracting them away as C_2 does for T_C, T_D, leads to a loss in semantic information salient to code similarity. Conversely, when global variables are not frequently used, there is an increased likelihood that



(a) MAP@R on GCJ. (b) AP on GCJ. (c) AUPRG on GCJ.

(d) MAP@R on POJ-104. (e) AP on POJ-104. (f) AUPRG on POJ-104.

Figure 3. Summarized accuracy results on the test sets for code2vec, code2seq, NCC, and Aroma and MISIM. Bar heights are the averages of the measurements over 3 runs, and error bars are bounded by the minimum and the maximum of measured values.

Table 2. Test Accuracy of MISIM-GNN Trained on Different Subsets of the Training Set. The results are shown as the average and min/max values relative to the average over 3 runs.

Sub-Training Set	Configuration	MAP@R (%)	AP (%)	AUPRG (%)
T_A	C_1	69.78 (-0.42/+0.21)	76.39 (-1.68/+1.51)	99.78 (-0.03/+0.03)
	C_2	**71.99 (-0.26/+0.45)**	**79.89 (-1.20/+0.71)**	**99.83 (-0.02/+0.01)**
T_B	C_1	63.45 (-1.58/+1.92)	68.58 (-2.51/+2.85)	99.63 (-0.06/+0.06)
	C_2	**67.40 (-1.85/+1.23)**	**69.86 (-3.34/+1.79)**	**99.65 (-0.10/+0.05)**
T_C	C_1	**63.53 (-1.08/+1.53)**	**72.47 (-0.95/+1.24)**	**99.70 (-0.04/+0.03)**
	C_2	61.23 (-2.04/+1.57)	69.83 (-1.03/+1.60)	99.65 (-0.03/+0.03)
T_D	C_1	**61.78 (-0.46/+0.47)**	**66.86 (-2.31/+2.81)**	**99.56 (-0.06/+0.07)**
	C_2	60.86 (-1.59/+0.90)	63.86 (-3.06/+3.43)	99.46 (-0.14/+0.11)

the semantics they extract are specific to a single program's embodiment. As such, retaining their names in a CASS, may increase syntactic noise, thereby reducing model performance. Therefore, when C_2 eliminates them for T_A, T_B, there is an improvement in accuracy.

5. Related Work

There is a growing body of work on code comprehension that is not directly intended for code similarity, but may provide indirect value to it. For example, a body of work has studied applying machine learning to learn from the AST for completing various tasks on code (Alon et al., 2019a; Chen et al., 2018; Hu et al., 2018; Li et al., 2018; Mou et al., 2016). Odena & Sutton (2020) represent a program as property signatures inferred from input-output pairs, which may be used to improve program synthesizers, amongst other things. There has also been work exploring graph representations, such as the following. For bug detection and code generation, Allamanis et al. (2018b); Brockschmidt et al. (2019) represent a program as a graph with a backbone AST and additional edges representing lexical ordering and semantic relations between the nodes. Dinella et al. (2020) also use an AST-backboned graph representation of programs to learn bug fixing through graph transformation. Hellendoorn et al. (2020) introduce a simplified graph containing only AST leaf nodes for program repair. Wei et al. (2020) extract type dependency graphs from JavaScript programs for probabilistic type inference.

6. Conclusion

In this paper, we presented MISIM, an end-to-end code similarity system. MISIM has two core novelties. The first is the *context-aware semantics structure* (CASS) designed specifically to lift semantic meaning from code syntax. The

second is a neural-based code similarity scoring algorithm for learning code similarity scoring using CASS. Our experimental evaluation showed that MISIM outperforms four state-of-the-art code similarity systems usually by a large factor (up to 43.4×). We also provided anecdotal evidence illustrating that there may not be one universally optimal CASS configuration. An open research question for MISIM is in how to automatically derive the proper configuration of its various components for a given code corpus, specifically the CASS and neural scoring algorithms, which we plan to explore in future work.

References

Ahmad, M. B. S., Ragan-Kelley, J., Cheung, A., and Kamil, S. Automatically Translating Image Processing Libraries to Halide. *ACM Trans. Graph.*, 38(6), November 2019. ISSN 0730-0301. doi: 10.1145/3355089.3356549. URL https://doi.org/10.1145/3355089.3356549.

Alam, M., Gottschlich, J., Tatbul, N., Turek, J. S., Mattson, T., and Muzahid, A. A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions. In Wallach, H., Larochelle, H., Beygelzimer, A., dAlchBuc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, NeurIPS 2019, pp. 11623–11635. Curran Associates, Inc., 2019. URL http://papers.nips.cc/paper/9337-a-zero-positive-learning-approach-for-diagnosing-software-performance-regressions.pdf.

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 51(4), September 2018a.

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*, 2018b. URL https://openreview.net/forum?id=BJOFETxR-.

Alon, U., Zilberstein, M., Levy, O., and 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*, PLDI 2018, pp. 404–419, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192412. URL https://doi.org/10.1145/3192366.3192412.

Alon, U., Levy, O., and Yahav, E. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*, 2019a. URL https://openreview.net/forum?id=H1gKYo09tX.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019b. ISSN 2475-1421. doi: 10.1145/3290353. URL http://doi.acm.org/10.1145/3290353.

Bader, J., Scott, A., Pradel, M., and Chandra, S. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360585. URL https://doi.org/10.1145/3360585.

Baeza-Yates, R. A. and Ribeiro-Neto, B. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN 020139829X.

Barman, S., Chasins, S., Bodik, R., and Gulwani, S. Ringer: Web Automation by Demonstration. *SIGPLAN Not.*, 51 (10):748–764, October 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984020. URL https://doi.org/10.1145/3022671.2984020.

Batra, G., Jacobson, Z., Madhav, S., Queirolo, A., and Santhanam, N. Artificial-Intelligence Hardware: New Opportunities for Semiconductor Companies, 2018. URL https://www.mckinsey.com/~/media/McKinsey/Industries/Semiconductors/Our%20Insights/Artificial%20intelligence%20hardware%20New%20opportunities%20for%20semiconductor%20companies/Artificial-intelligence-hardware.pdf.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, 1998.

Becker, K. and Gottschlich, J. AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms. *CoRR*, abs/1709.05703, 2017. URL http://arxiv.org/abs/1709.05703.

Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. Neural Code Comprehension: A Learnable Representation of Code Semantics. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 3585–3597. Curran Associates, Inc., 2018.

Bhatia, S., Kohli, P., and Singh, R. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International*

Conference on Software Engineering, ICSE '18, pp. 60–70, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180219. URL https://doi.org/10.1145/3180155.3180219.

Bogdan, P., Chen, F., Deshwal, A., Doppa, J. R., Joardar, B. K., Li, H. H., Nazarian, S., Song, L., and Xiao, Y. Taming Extreme Heterogeneity via Machine Learning Based Design of Autonomous Manycore Systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis Companion*, CODES/ISSS '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369237. doi: 10.1145/3349567.3357376. URL https://doi.org/10.1145/3349567.3357376.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative Code Modeling with Graphs. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bke4KsA5FX.

Chen, X., Liu, C., and Song, D. Tree-to-tree Neural Networks for Program Translation. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 2547–2557. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7521-tree-to-tree-neural-networks-for-program-translation.pdf.

Chen, Y., Xie, Y., Song, L., Chen, F., and Tang, T. A Survey of Accelerator Architectures for Deep Neural Networks. *Engineering*, 6(3):264 – 274, 2020. ISSN 2095-8099. doi: https://doi.org/10.1016/j.eng.2020.01.007. URL http://www.sciencedirect.com/science/article/pii/S2095809919306356.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL https://www.aclweb.org/anthology/D14-1179.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

Cosentino, V., Cánovas Izquierdo, J. L., and Cabot, J. A Systematic Mapping Study of Software Development With GitHub. *IEEE Access*, 5:7173–7192, 2017. ISSN 2169-3536. doi: 10.1109/ACCESS.2017.2682323.

Deng, S., Zhao, H., Fang, W., Yin, J., Dustdar, S., and Zomaya, A. Y. Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020.

Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJeqs6EFvB.

Feitelson, D., Mizrahi, A., Noy, N., Ben Shabat, A., Eliyahu, O., and Sheffer, R. How Developers Choose Names. *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. ISSN 2326-3881. doi: 10.1109/TSE.2020.2976920.

Finkel, H. and Laguna, I. Program Synthesis for Scientific Computing. https://www.anl.gov/cels/program-synthesis-for-scientific-computing-report, August 2020.

Flach, P. A. and Kull, M. Precision-recall-gain curves: Pr analysis done right. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pp. 838–846, Cambridge, MA, USA, 2015. MIT Press.

Flanagan, D. *JavaScript: The Definitive Guide*. "O'Reilly Media, Inc.", 2006.

Gellenbeck, E. M. and Cook, C. R. An Investigation of Procedure and Variable Names as Beacons During Program Comprehension. In *Empirical studies of programmers: Fourth workshop*, pp. 65–81. Ablex Publishing, Norwood, NJ, 1991.

Gottschlich, J., Solar-Lezama, A., Tatbul, N., Carbin, M., Rinard, M., Barzilay, R., Amarasinghe, S., Tenenbaum, J. B., and Mattson, T. The Three Pillars of Machine Programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pp. 69–80, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5834-7. doi: 10.1145/3211346.3211355. URL http://doi.acm.org/10.1145/3211346.3211355.

Hannigan, E., Burkacky, O., Kenevan, P., Mahindroo, A., Johnson, R., Rivait, J., Byer, H., Simcock, V., Brown, E., Draper, R., Herbein, G., Norton, P., Petriwsky, K., Rice, C., Sanchez, J. C., Sand, D., Vats, S., Yadav, P., Yu, B., Rahilly, L., Borruso, M. T., Javetski, B., Staples, M., and Communications, L. McKinsey on Semiconductors, 2019. URL https://www.mckinsey.com/~/media/

McKinsey/Industries/Semiconductors/Our%20Insights/McKinsey%20on%20Semiconductors%20Issue%207/McK_Semiconductors_Oct2019-Full%20Book-V12-RGB.pdf.

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global Relational Models of Source Code. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1lnbRNtwr.

Hermans, A., Beyer, L., and Leibe, B. In Defense of the Triplet Loss for Person Re-Identification. *CoRR*, abs/1703.07737, 2017. URL http://arxiv.org/abs/1703.07737.

Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. Deep Code Comment Generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pp. 200–210, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357142. doi: 10.1145/3196321.3196334. URL https://doi.org/10.1145/3196321.3196334.

Iyer, R. G., Sun, Y., Wang, W., and Gottschlich, J. Software language comprehension using a program-derived semantic graph, 2020.

Kamil, S., Cheung, A., Itzhaky, S., and Solar-Lezama, A. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pp. 711–726, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908117. URL http://doi.acm.org/10.1145/2908080.2908117.

Li, J., Wang, Y., Lyu, M. R., and King, I. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, pp. 4159–25. AAAI Press, 2018. ISBN 9780999241127.

Li, L., Feng, H., Zhuang, W., Meng, N., and Ryder, B. Cclearner: A deep learning-based clone detection approach. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 249–260, 2017.

Liu, T.-Y. Learning to Rank for Information Retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, March 2009. ISSN 1554-0669. doi: 10.1561/1500000016. URL https://doi.org/10.1561/1500000016.

Loshchilov, I. and Hutter, F. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Luan, S., Yang, D., Barnaby, C., Sen, K., and Chandra, S. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.*, 3(OOPSLA):152:1–152:28, October 2019. ISSN 2475-1421. doi: 10.1145/3360578. URL http://doi.acm.org/10.1145/3360578.

Mikolov, T., Chen, K., Corrado, G. S., and Dean, J. Efficient estimation of word representations in vector space, 2013a. URL http://arxiv.org/abs/1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pp. 3111–3119, Red Hook, NY, USA, 2013b. Curran Associates Inc.

Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI '16, pp. 1287–1293. AAAI Press, 2016.

Musgrave, K., Belongie, S., and Lim, S.-N. A Metric Learning Reality Check, 2020.

Odena, A. and Sutton, C. Learning to Represent Programs with Property Signatures. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rylHspEKPr.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Pradel, M. and Sen, K. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi: 10.1145/3276517. URL https://doi.org/10.1145/3276517.

Schlichtkrull, M. S., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In Gangemi, A., Navigli, R., Vidal, M., Hitzler, P., Troncy, R., Hollink, L., Tordai, A., and Alam, M. (eds.), *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece,*

June 3-7, 2018, Proceedings, volume 10843 of *Lecture Notes in Computer Science*, pp. 593–607. Springer, 2018. doi: 10.1007/978-3-319-93417-4_38. URL https://doi.org/10.1007/978-3-319-93417-4_38.

Schroff, F., Kalenichenko, D., and Philbin, J. FaceNet: A Unified Embedding for Face Recognition and Clustering. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2015.

Sun, Y., Cheng, C., Zhang, Y., Zhang, C., Zheng, L., Wang, Z., and Wei, Y. Circle Loss: A Unified Perspective of Pair Similarity Optimization. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., and Poshyvanyk, D. Deep Learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pp. 542–553, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196431. URL https://doi.org/10.1145/3196398.3196431.

Ullah, F., Naeem, H., Jabbar, S., Khalid, S., Latif, M. A., Al-turjman, F., and Mostarda, L. Cyber Security Threats Detection in Internet of Things Using Deep Learning Approach. *IEEE Access*, 7:124379–124389, 2019.

Van Rossum, G. and Drake, F. L. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

Wei, H. and Li, M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 3034–3040, 2017. doi: 10.24963/ijcai.2017/423. URL https://doi.org/10.24963/ijcai.2017/423.

Wei, J., Goyal, M., Durrett, G., and Dillig, I. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=Hkx6hANtwH.

Wulf, W. and Shaw, M. Global Variable Considered Harmful. *SIGPLAN Not.*, 8(2):28–34, February 1973. ISSN 0362-1340. doi: 10.1145/953353.953355. URL https://doi.org/10.1145/953353.953355.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pp. 783–794. IEEE Press, 2019. doi: 10.1109/ICSE.2019.00086. URL https://doi.org/10.1109/ICSE.2019.00086.

Zhao, G. and Huang, J. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pp. 141–151, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236068. URL https://doi.org/10.1145/3236024.3236068.

A. Context-Aware Semantics Structure Details

The following is the formal definition of CASS.

Definition 1 (Context-aware semantic structure (CASS)) *A CASS consists of one or more CASS trees and an optional global attributes table (GAT). A CASS tree, T, is a collection of nodes, $V = \{v_1, v_2, \ldots, v_{|V|}\}$, and edges, $E = \{e_1, e_2, \ldots, e_{|E|}\}$, denoted as $T = (V, E)$. Each edge is directed from a parent node, v_p to a child node, v_c, or $e_k = (v_p, v_c)$ where $e_k \in E$ and $v_p, v_c \in V$. The root node, v_r, of the tree signifies the beginning of the code snippet and has no parent node, i.e., $\nexists v_p, (v_p, v_r) \in E$. A child node is either an internal node or a leaf node. An internal node has at least one child node while a leaf node has no child nodes. A CASS tree can be empty, in which it has no nodes. The CASS GAT contains exactly one entry per unique function definition in the code snippet. A GAT entry includes the input and output cardinality values for the corresponding function.*

Definition 2 (Node labels) *Every CASS node has an associated label, l_v. During the construction of a CASS tree, the program tokens at each node, t_v are mapped to its corresponding label or $l_v = f(t_v)$. This is depicted with an expression grammar for node labels and the function mapping tokens to labels below.*[6]

⟨*bin-op*⟩ ::= '+' | '−' | '*' | '/' | ...

⟨*unary-op*⟩ ::= '++' | '−−' | ...

⟨*leaf-node-label*⟩ ::= `LITERAL` | `IDENT` | '`#VAR`' | '`#GVAR`' | '`#EXFUNC`' | '`#LIT`' | ...

⟨*exp*⟩ ::= '`$`' | '`$`' ⟨*bin-op*⟩ '`$`' | ⟨*unary-op*⟩ '`$`' | ...

⟨*internal-node-label*⟩ ::= '`for`' '`(`' ⟨*exp*⟩ '`;`' ⟨*exp*⟩ '`;`' ⟨*exp*⟩ '`)`' ⟨*exp*⟩ '`;`'
| '`int`' ⟨*exp*⟩ '`;`'
| '`return`' ⟨*exp*⟩ '`;`'
| ⟨*exp*⟩
| ...

$$l_v = f(t_v) = \begin{cases} \langle \textit{leaf-node-label} \rangle & \text{if } v \text{ is a leaf node} \\ \langle \textit{internal-node-label} \rangle & \text{otherwise} \end{cases}$$

Definition 3 (Node prefix label) *A node prefix label is a string prefixed to a node label. A node prefix label may or may not be present.*

[6]Note: the expression grammar we provide is non-exhaustive due to space limitations. The complete set of standard C/C++ tokens or binary and unary operators is collectively denoted in shorthand as '...'.

A.1. CASS configuration options

Table 3. CASS Configuration Options.

Language-specific	
Type	Option
Node Prefix Label	0. No change (Aroma's original configuration) 1. Add a prefix to each internal node label 2. Add a prefix to parenthesis node label (C/C++ Specific)
Language-agnostic	
Type	Option
Compound Statements	0. No change (Aroma's original configuration) 1. Drop all features relevant to compound statements 2. Replace with '{#}'
Global Variables	0. No change (Aroma's original configuration) 1. Drop all features relevant to global vars 2. Replace with '#GVAR' 3. Replace with '#VAR' (label for local vars)
Global Functions	0. No change (Aroma's original configuration) 1. Drop all features relevant to global functions 2. Drop function identifier and replace with '#EXFUNC'
Function I/O Cardinality	0. No change 1. Include the input and output cardinality per function

Table 3 specifies the exact configuration options and their values that we used in our experiments.

A.2. Discussion

We believe there is no silver bullet solution for code similarity for all programs and programming languages. Based on this belief, a key intuition of CASS's design is to provide a structure that is semantically rich based on structure, with inspiration from Aroma's SPT, while simultaneously providing a range of customizable parameters to accommodate a wide variety of scenarios. CASS's language-agnostic and language-specific configurations and their associated options serve for exploration of a series of tree variants, each differing in their granularity of detail of abstractions.

For instance, the *compound statements* configuration provides three levels of abstraction. Option 0 is Aroma's baseline configuration and is the finest level of abstraction, as it featurizes the number of constituents in a compound statement node. Option 2 reduces compound statements to a single token and represents a slightly higher level of abstraction. Option 1 eliminates all features related to compound statements and is the coarsest level of abstraction. The same trend applies to the *global variables* and *global functions* configurations. It is our belief, based on early evidence, that the appropriate level of abstraction in CASS is likely based on many factors such as *(i)* code similarity purpose, *(ii)* programming language expressiveness, and *(iii)* application domain.

Aroma's original SPT seems to work well for a common code base where global variables have consistent semantics and global functions are standard API calls also with consistent semantics (e.g., a single code-base). However, for cases outside of such spaces, some question about applicability arise. For example, assumptions about consistent semantics for global variables and functions may not hold in cases of non-common code-bases or non-standardized global function names (Wulf & Shaw, 1973; Gellenbeck & Cook, 1991; Feitelson et al., 2020). Having the capability to differentiate between these cases, and others, is a key motivation for CASS.

We do not believe that CASS's current structure is exhaustive. With this in mind, we have designed CASS to be extensible, enabling a seamless mechanism to add new configurations and options. Our intention with this paper is to present initial findings in exploring CASS's structure. Based on our early experimental analysis, presented in Section C.7, CASS seems to be a promising research direction for code similarity.

An Important Weakness. While CAST provides added flexibility over SPT, such flexibility may be misused. With CAST, system developers are free to add or remove as much syntactic differentiation detail they choose for a given language or given code body. Such overspecification (or underspecification), may result in syntactic overload (or underload) which may cause reduced code similarity accuracy over the original SPT design, as we illustrate in Section C.7.

B. Models

In this section, we describe the models evaluated in our experiments other than MISIM-GNN, and discuss the details of the experimental procedure.

B.1. Model: MISIM-BoF

The MISIM-BoF model takes a set of manual features extracted from a CASS as its input. The features include the ones extracted from CASS trees, using the same procedure described in Aroma (Luan et al., 2019), as well as the entries in CASS GATs. The MISIM-BoF model is specified as below:

$$\mathbf{c} = \mathrm{FC}\left(\mathrm{AvgPool}\left(\{\mathbf{e}_x \mid x \in S\}\right)\right),$$

where S is the feature set of the input program and $\mathbf{e}_x$ is the embedding vector of feature x. The output code vector is computed by performing average pooling on the feature embeddings and projecting its result into the code vector space with a fully connected layer.

B.2. Model: MISIM-RNN

The input to the MISIM-RNN model is a serialized representation of a CASS. Each CASS tree, representing a function in the program, is converted to a sequence using the technique proposed in (Hu et al., 2018). The GAT entry associated with a CASS tree is both prepended and appended to the tree's sequence, forming the sequence of the corresponding function. The model architecture can be expressed as:

$$\mathbf{h}_f = \mathrm{biGRU}\left(\bar{\mathbf{e}}_f\right),$$
$$\mathbf{c} = \mathrm{FC}\left(\begin{bmatrix} \mathrm{AvgPool}\left(\{\mathbf{h}_f \mid f \in F\}\right) \\ \mathrm{MaxPool}\left(\{\mathbf{h}_f \mid f \in F\}\right) \end{bmatrix}\right),$$

where F is the set of functions in the input program and $\bar{\mathbf{e}}_f$ is the sequence of embedding vectors for the serialized CASS of function f. Each function's sequence first has its tokens embedded, and then gets summarized to a function-level vector by a bidirectional GRU layer (Cho et al., 2014). The code vector for the entire program is subsequently computed by performing average and max pooling on the function-level vectors, concatenating the resulting vectors, and passing it through a fully connected layer.

C. Experimental Details

C.1. Program B used in Section 2

We did not specify program B that we used in our similarity analysis of AST, SPT, and CASS from Section 2 for space reason. We specify this program below.

```
// Program B
int main() {
  char *p, *head, c;
  p = (char *) malloc(sizeof(char) * 30);
  head = p;
  scanf("%c", p);
  while (*p != '\n') {
    p++; *p = getchar();
  }
  *p = '\0';
  p = head;
  for (; *p != '\0'; p++) {
    if (*p <= '9' && *p >= '0')
      printf("%c",*p);
    else if (*(p+1) < 58 && *(p+1) > 47)
      putchar('\n');
  }
}
```

C.2. Training Procedure and Hyperparameters

We use the AdamW optimizer with a learning rate of 10^{-3} (Loshchilov & Hutter, 2019). The training runs for 100 epochs, each containing 1,000 iterations, and the model that gives the best validation accuracy is used for testing.[7]

[7] We have observed that the validation accuracy stops to increase before the 100$^{\text{th}}$ epoch in all experiments.

The hyperparameters used for the Circle loss are $\gamma = 80$ and $m = 0.4$. For all of our MISIM models, we use 128-dimensional embedding vectors, hidden states, and code vectors. We also apply dropout with a probability of 0.5 to the embedding vectors. To handle rare or unknown tokens, a token that appears less than 5 times in the training set is replaced with a special `UNKNOWN` token.

C.3. Modifications to code2vec, code2seq, NCC, and Aroma

To compare with code2vec, code2seq, NCC, and Aroma, we adapt them to our experimental setting in the following ways. The original code2vec takes a function as an input, extracts its AST paths to form the input to its neural network, and trains the network using the function name prediction task. In our experiments, we feed the AST paths from all function(s) in a program into the neural network and train it using the metric learning task described in Section 3.2.2. We make similar adaptions to code2seq by combining AST paths from the whole program as one input sample. Additionally, we replace the sequence decoder of code2vec with an attention-based path aggregator used in code2vec. NCC contains a pre-training phase, named `inst2vec`, on a large code corpus for generating instruction embeddings, and a subsequent phase that trains an RNN for a downstream task using the pre-trained embeddings. We train the downstream RNN model on our metric learning task in two ways. The first uses the pre-trained embeddings (labeled as NCC in our results). The second trains the embeddings from scratch on our task in an end-to-end fashion (labeled as NCC-w/o-inst2vec). For both code2vec and NCC, we use the same model architectures and embedding/hidden sizes suggested in their papers and open-sourced implementations. The dimension of their output vectors (i.e., code vectors) is set to the same as our MISIM models. Aroma extracts manual features from the code and computes the similarity score of two programs by taking the dot product of their binary feature vectors. We experiment with both its original scoring mechanism (labeled: Aroma-Dot) and a variant that uses the cosine similarity (labeled: Aroma-Cos).

C.4. Evaluation Metrics

MAP@R measures how accurately a model can retrieve similar (or relevant) items from a database given a query. MAP@R rewards a ranking system (e.g., a search engine, a code recommendation engine, etc.) for correctly ranking relevant items with an order where more relevant items are ranked higher than less relevant items. It is defined as the mean of average precision scores, each of which is evaluated for retrieving R most similar samples given a query. In our case, the set of queries is the set of all test programs. For a program, R is the number of other programs in the same class (i.e., a POJ-104 problem). MAP@R is applied to both validation and testing. We use AP and AUPRG to measure the performance in a binary classification setting, in which the models are viewed as binary classifiers that determine whether a pair of programs are similar by comparing their similarity score with a threshold. AP and AUPRG are only used for testing. They are computed from the similarity scores of all program pairs in the test set, as well as their pair-wise labels. For the systems that require training (i.e., systems with ML learned similarity scoring), we train and evaluate them three times with different random seeds.

C.5. MISIM Accuracy Results (in tabular form)

Table 4 shows the results of MISIM in comparison to other systems. Same results are presented in the graphical form in Figure 3.

C.6. CASS vs AST

Some recent research on code representation uses the AST-based representation (Dinella et al., 2020) or AST paths (Alon et al., 2019b;a). In this subsection, we explore how AST and CASS perform on the task of code semantic representation described here.

We compared the code similarity performance of ASTs and CASSes on the test set of POJ-104, as shown in table 1, by transforming both kinds of representations into feature vectors. using the same method described in (Luan et al., 2019) and compute the similarity scores using dot or cosine similarity. For each program in the dataset, we extracted its CASS under three different configurations: 0-0-0-0-0 [8], the base configuration, and 2-1-3-1-1/1-2-1-0-0, the best/worst performing configuration according to our preliminary evaluation of CASS (see Appendix C.7 for details). We also extracted the ASTs of function bodies in a program. Each syntax node in the AST is labeled by its node type, and an identifier (or literal) node also gets a single child labeled by the corresponding identifier name (or literal text).

As shown in Table 5, CASS configurations show an improvement in accuracy over the AST up to $1.67\times$ across three evaluation metrics described in Appendix C.4. To better understand the performance difference, we investigated a few solutions for the same problems from the POJ-104 dataset. One of the interesting observations we found is that for the same problem, a solution may have a different naming convention for local variables than that of another solution (e.g., English vs Mandarin description of variables), but the resulting different variable names may carry the same semantic meaning. AST uses variable names in its structure, but CASS has the option to not use variable names. Thus

[8] Configuration 0-0-0-0-0 is the duplicate of SPT. As shown in Table 5, configuration 2-1-3-1-1 shows better accuracy than configuration 0-0-0-0-0.

Table 4. Code similarity system accuracy. Results are shown as the average and min/max values, relative to the average, over 3 runs. We had to make a few modifications to adapt code2vec, code2seq, NCC and Aroma to our experimental settings. Please refer to Appendix C.3 for details.

Method	GCJ			POJ-104		
	MAP@R (%)	AP (%)	AUPRG (%)	MAP@R (%)	AP (%)	AUPRG (%)
code2vec	7.76 (-0.79/+0.88)	17.95 (-1.24/+1.76)	25.48 (-7.37/+10.37)	1.90 (-0.43/+0.38)	5.30 (-0.80/+0.60)	34.97 (-20.83/+15.10)
code2seq	11.67 (-1.98/+1.73)	23.09 (-3.24/+2.49)	46.29 (-15.10/+12.02)	3.12 (-0.45/+0.67)	6.43 (-0.37/+0.48)	54.97 (-7.15/+7.65)
NCC	17.26 (-1.11/+0.57)	31.56 (-1.11/+1.46)	68.76 (-1.25/+2.46)	39.95 (-2.29/+1.64)	50.42 (-2.98/+1.61)	98.86 (-0.20/+0.10)
NCC-w/o-inst2vec	34.88 (-5.72/+7.63)	56.12 (-7.63/+9.96)	90.10 (-3.83/+4.49)	54.19 (-3.18/+3.52)	62.75 (-5.49/+4.42)	99.39 (-0.22/+0.17)
Aroma-Dot	29.08	42.47	82.03	52.09	45.99	98.42
Aroma-Cos	29.67	36.21	75.09	55.12	55.40	99.07
MISIM-GNN	**74.90 (-1.15/+0.64)**	**92.15 (-0.97/+0.7)**	**99.46 (-0.08/+0.05)**	**82.45 (-0.61/+0.40)**	82.00 (-2.77/+1.65)	99.86 (-0.04/+0.03)
MISIM-RNN	66.38 (-2.93/+2.68)	85.72 (-1.19/+0.65)	98.78 (-0.18/+0.1)	74.01 (-2.00/+3.81)	81.64 (-1.52/+2.72)	99.84 (-0.03/+0.04)
MISIM-BoF	72.21 (-0.64/+0.52)	90.54 (-0.81/+0.56)	99.33 (-0.07/+0.05)	74.38 (-1.04/+1.04)	**82.95 (-0.70/+0.50)**	**99.87 (-0.01/+0.01)**

Table 5. Test Accuracy for AST and CASS configurations on POJ-104.

Method	MAP@R (%)	AP (%)	AUPRG (%)
AST-Dot	45.12	35.98	97.29
AST-Cos	47.39	45.31	98.41
SPT-Dot	52.09	45.99	98.42
SPT-Cos	55.12	55.4	99.07
CASS (2-1-3-1-1)-Dot	55.59	48.31	98.62
CASS (2-1-3-1-1)-Cos	**60.78**	**60.42**	**99.31**
CASS (1-2-1-0-0)-Dot	52.74	40.73	97.87
CASS (1-2-1-0-0)-Cos	57.99	54.75	99.06

the erasure of local variable names in CASS might help in discovering the semantic similarity between code with different variable names. This might explain some of the performance differences between AST and CASS in this experiment.

C.7. Experimental Results of Various CASS configurations

In this section, we discuss our experimental setup and analyze the performance of CASS compared to Aroma's simplified parse tree (SPT). In Section C.7.1, we explain the dataset grouping and enumeration for our experiments. We also discuss the metrics used to quantitatively rank the different CASS configurations and those chosen for the evaluation of code similarity. Section C.7.2 demonstrates that, a code similarity system built using CASS *(i)* has a greater frequency of improved accuracy for the total number of problems and *(ii)* is, on average, more accurate than SPT. For completeness, we also include cases where CASS configurations perform poorly.

C.7.1. Experimental Setup

In this section, we describe our experimental setup. At the highest level, we compare the performance of various configurations of CASS to Aroma's SPT. The list of possible CASS configurations is shown in Table 3.

Dataset. The experiments use the same POJ-104 dataset introduced in Section 4.

Problem Group Selection. Given that POJ-104 consists of 104 unique problems and nearly 50,000 programs, depending on how we analyze the data, we might face intractability problems in both computational and combinatorial complexity. With this in mind, our initial approach is to construct 1000 sets of five unique, pseudo-randomly selected problems for code similarity analysis. Using this approach, we evaluate every configuration of CASS and Aroma's original SPT on each pair of solutions for each problem set. We then aggregate the results across all the groups to estimate their overall performance. While this approach is not exhaustive of possible combinations (in set size or set combinations), we aim for it to be a reasonable starting point. As our research with CASS matures, we plan to explore a broader variety of set sizes and a more exhaustive number of combinations.

Code Similarity Performance Evaluation. For each problem group, we exhaustively calculate code similarity scores for all unique solution pairs, including pairs constructed from the same program solution (i.e., program A compared to program A). We use G to refer to the set of groups and g to indicate a particular group in G. We denote $|G|$ as the number of groups in G (i.e. cardinality) and |g| as the number of solutions in group g. For $g = G_i$, where $i = \{1, 2, \ldots, 1000\}$, the total unique program pairs (denoted by g_P) in G_i is $|g_P| = \frac{1}{2}|g|(|g| + 1)$.

To compute the similarity score of a solution pair, we use Aroma's approach. This includes calculating the dot product of two feature vectors (i.e., a program pair), each of which is generated from a CASS or SPT structure. The larger the magnitude of the dot product, the greater the similarity.

We evaluate the quality of the recommendation based on *average precision*. *Precision* is the ratio of true positives to the sum of true positives and false positives. Here, true

positives denote solution pairs correctly classified as similar and false positives refer to solution pairs incorrectly classified as similar. *Recall* is the ratio of true positives to the sum of true positives and false negatives, where false negatives are solution pairs incorrectly classified as different. As we monotonically increase the threshold from the minimum value to the maximum value, precision generally increases while recall generally decreases. The *average precision* (AP) summarizes the performance of a binary classifier under different thresholds for categorizing whether the solutions are from the same equivalence class (i.e., the same POJ-104 problem) (Liu, 2009). AP is calculated using the following formula over all thresholds.

1. All unique values from the M similarity scores, corresponding to the solution pairs, are gathered and sorted in descending order. Let N be the number of unique scores and $s_1, s_2, \ldots, s_N$ be the sorted list of such scores.
2. For i in $\{1, 2, \ldots, N\}$, the precision p_i and recall r_i for the classifier with the threshold being s_i is computed.
3. Let $r_0 = 0$. The average precision is computed as:

$$AP = \sum_{i=1}^{N} (r_i - r_{i-1}) p_i$$

C.7.2. Results

Figure 4a depicts the number of problem groups where a particular CASS variant performed better (blue) or worse (orange) than SPT. For example, the CASS configuration 2-0-0-0-1 outperformed SPT in 859 of 1000 problem groups, and underperformed in 141 problem groups. This equates to a 71.8% accuracy improvement of CASS over SPT. Figure 4a shows the two best (2-0-0-0-1 and 0-0-0-0-1), the median (2-2-3-0-0), and the two worst (1-0-1-0-0 and 1-2-1-0-0) configurations with respect to SPT. Although we have seen certain configurations that perform better than SPT, there are also configurations that perform worse. We observed that the configurations with better performance have function I/O cardinality option as 1. We also observed that the configurations with worse performance have function I/O cardinality option as 0. These observations indicate that function I/O cardinality seems to improve code similarity accuracy, at least, for the data we are considering. We speculate that these configuration results may vary based on programming language, problem domain, and other constraints.

Figure 4b shows the group containing the problems for which CASS achieved the best performance relative to SPT, among all 1000 problem groups. In other words, Figure 4b shows the performance of SPT and CASS for the single problem group with the greatest difference between a CASS configuration and SPT. In this single group, CASS achieves the maximum improvement of more than 30% over SPT for this problem group on two of its configurations. We note that, since we tested 216 CASS configurations across 1000 different problem groups, there is a reasonable chance of observing such a large difference *even if CASS performed identically to SPT in expectation.* We do not intend for this result to demonstrate statistical significance, but simply to illustrate the outcome of our experiments.

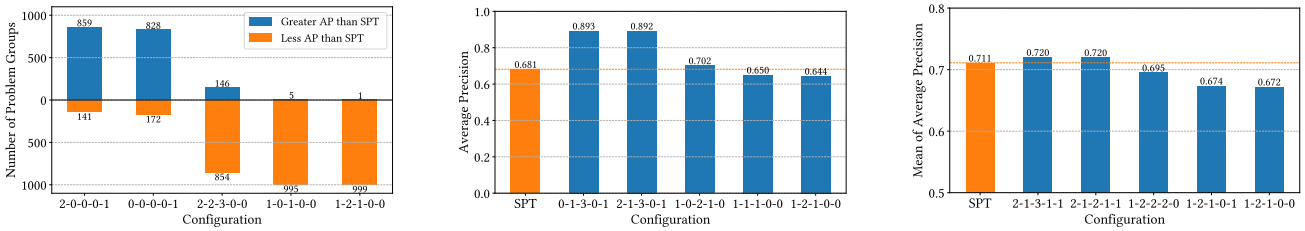
Figure 4c compares the mean of AP over all 1000 problem groups. In it, the blue bars, moving left to right, depict the CASS configurations that are *(i)* the two best, *(ii)* the median, and *(iii)* the two worst in terms of average precision. Aroma's baseline SPT configuration is highlighted in orange. The best two CASS configurations show an average improvement of more than 1% over SPT, while the others degraded performance relative to the baseline SPT configuration.

These results illustrate that certain CASS configurations can outperform the SPT on average by a small margin, and can outperform the SPT on specific problem groups by a large margin. However, we also note that choosing a good CASS configuration for a domain is essential. We leave automating this configuration selection to future work.

C.7.3. Analysis of Configurations

Figures 5a-5e serve to illustrate the performance variation for individual configurations. Figure 5a shows the effect of varying the options for the *node prefix label* configuration. Applying the node prefix label for the parentheses operator (option 2) results in the best overall performance while annotating every internal node (option 1) results in a concrete syntax tree and the worst overall performance. This underscores the trade-offs in incorporating syntax-binding transformations in CASS. In Figure 5b we observe that removing all features relevant to *compound statements* (option 1) leads to the best overall performance when compared with other options. This indicates that adding separate features for compound statements obscures the code's intended semantics when the constituent statements are also individually featurized.

Figure 5c shows that removing all features relevant to *global variables* (option 1) degrades performance. We also observe that eliminating the global variable identifiers and assigning a label to signal their presence (option 2) performs best overall, possibly because global variables appearing in similar contexts may not use the same variable identifiers. Further, option 2 performs better than the case where global variables are indistinguishable from local variables (option 3). Figure 5d indicates that removing features relevant to identifiers of *global functions*, but flagging their presence with

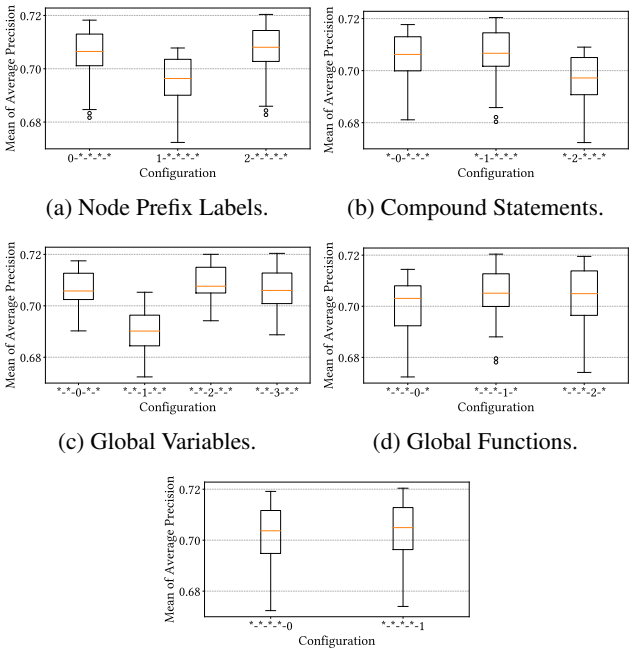


(a) Breakdown of the Number of Groups with AP Greater or Less than SPT.

(b) Average Precision for the Group Containing the Best Case.

(c) Mean of Average Precision Over All Program Groups.

Figure 4. Comparison of CASS and SPT. The blue bars in (a) and (b), and all the bars in (c), from left to right, correspond to the best two, the median, and the worst two CASS configurations, ranked by the metric displayed in each subfigure.



(a) Node Prefix Labels.

(b) Compound Statements.

(c) Global Variables.

(d) Global Functions.

(e) function I/O cardinality.

Figure 5. The Distributions of Performance for Configurations with a Fixed Option Type.

a special label as done in option 2, generally gives the best performance. This result is consistent with the intuitions for eliminating features of function identifiers in CASS as discussed in Section A.2. Figure 5e shows that capturing the input and output cardinality improves the average performance. This aligns with our assumption that function I/O cardinality may abstract the semantics of certain groups of functions.

A Subtle Observation. A more nuanced and subtle observation is that our results seem to indicate that for each CASS configuration the optimal granularity of abstraction detail is different. For *compound statements*, the best option seems to correspond to the coarsest level of abstraction detail, while for *node prefix label*, *global variables*, and *global functions* the best option seems to corresponds to one of the intermediate levels of abstraction detail. Additionally, for *function I/O cardinality*, the best option has a finer level of detail. For our future work, we aim to perform a deeper analysis on this and hopefully learn such configurations, to reduce (or eliminate) the overhead necessary of trying to manually discover such configurations.