Cross-language code search using static and dynamic analyses

George Mathew  
North Carolina State University  
USA  
george2@ncsu.edu

Kathryn T. Stolee  
North Carolina State University  
USA  
ktstolee@ncsu.edu

ABSTRACT
As code search permeates most activities in software development, code-to-code search has emerged to support using code as a query and retrieving similar code in the search results. Applications include duplicate detection for refactoring, patch identification for program repair, and language translation. Existing code-to-code search tools rely on static similarity approaches such as the comparison of tokens and abstract syntax trees (AST) to approximate dynamic behavior, leading to low precision. Most tools do not support cross-language code-to-code search, and those that do, rely on machine learning models that require labeled training data.

We present Code-to-Code Search Across Languages (COSAL), a cross-language technique that uses both static and dynamic analyses to identify similar code and does not require a machine learning model. Code snippets are ranked using non-dominated sorting based on code token similarity, structural similarity, and behavioral similarity. We empirically evaluate COSAL on two datasets of 43,146 Java and Python files and 55,499 Java files and find that 1) code search based on non-dominated ranking of static and dynamic similarity measures is more effective compared to single or weighted measures; and 2) COSAL has better precision and recall compared to state-of-the-art within-language and cross-language code-to-code search tools. We explore the potential for using COSAL on large open-source repositories and discuss scalability to more languages and similarity metrics, providing a gateway for practical, multi-language code-to-code search.

CCS CONCEPTS
• Software and its engineering → Software maintenance tools;  
• Information systems → Similarity measures.

KEYWORDS
code-to-code search, cross-language code search, non-dominated sorting, static analysis, dynamic analysis

1 INTRODUCTION
Code-to-code search describes the task of using a code query to search for similar code in a repository. This task is particularly challenging when the query and results belong to different languages due to syntactic and semantic differences between the languages [20]. Consider the case of code migration, where it is common for applications in a specific language to be re-written to another language [52]. For example, while porting the video game Fez from Microsoft Xbox to Sony PlayStation, the developers faced their biggest challenge in converting the original C# code to C++ as the PlayStation did not support the C# compiler [85]. Code-to-code search is also involved in identifying code clones [63, 66], finding translations of code in a different language [58], program repair [10, 68], and supporting students in learning a new programming language [3]. The growing prominence of large online code repositories and the repetitive nature of source code [48, 70] lead to the presence of large quantities of potentially similar code across languages, providing a viable platform for code-to-code search.

We propose the first cross-language code-to-code search approach with dynamic and static similarity measures. The novelty is in the application of non-dominated sorting [19] to code-to-code search, allowing static and dynamic information (without aggregation) to identify search results. COSAL leverages prior art in clone detection using input-output (IO) behavior [51]. As dynamic clone detection requires executable code, individually it cannot achieve the recall required for practical search applications. This is where the prior art in static analysis shines; we use token-based and AST-based measures to complement the dynamic analysis. COSAL reaps the benefits of dynamic analysis in finding code that behaves similarly, when dynamic information is available, and the benefits of static information when dynamic information is infeasible. It provides results that balance how code looks with how it behaves, in the spirit of returning code that looks more natural to the user.

We evaluate COSAL using 43,146 Java and Python files from AtCoder, a programming contest dataset, and 55,499 Java files from BigCloneBench [79], a Java based clone detection benchmark. We show that combining static and dynamic analyses yields better precision and success rate compared to code search with individual or weighted analyses. COSAL performs better in cross-language and within-language contexts compared to the state-of-the-art code search tool FaCoY and the industrial benchmark, GitHub. COSAL can also detect more cross-language code clones compared to SLACC and CLCDSA, the state-of-the-art code clone detection techniques. The contributions of this work are:

- the first code-to-code search approach using non-dominated sorting over static and dynamic similarity measures,
- an evaluation of COSAL with state-of-the-practice cross-language code search tools, GitHub and ElasticSearch (RQ2),
Effective code-to-code search requires code similarity measures that cover a variety of developer concerns. Code-to-code search should preserve code behavior, and thus IO similarity from dynamic analysis is an important consideration. Prior work has shown that identifiers impact source code comprehension, especially for beginners [14], and as developers must understand the code returned by search, tokens are an important consideration. Prior work in code-to-code search that relies on ASTs have seen high precision and recall [43, 63] suggesting that is an important consideration as well. Individually, each measure has shortcomings. Taken together, however, we show the whole is greater than the sum of its parts.

Consider the code snippets in Figure 1. Three of the functions are behaviorally identical, taking an input integer and returning an array of even integers: 1(a) is a Java function which uses a for loop; 1(c) uses the stream library from Java v8; 1(d) is a Python function which uses a filtered list-comprehension. 1(b) is different: it is a Python function that takes an integer max and returns a list of odd numbers between [0, max). Figure 2 contains AST and token information for the code snippets, discussed extensively in § 3.

To identify behaviorally identical Python code for Figure 1(a), a code-to-code search engine should support both the programming languages Java and Python. In this case, using a purely token-based approach for cross-language search will not be very helpful. First, the syntactic features of each language will skew the search. For example, since Java is static typed, variables are declared with a datatype, while variables in Python lack these tokens due to dynamic typing in Python. Second, even if the language-specific keywords are ignored, there is an over reliance on the names of variables and libraries to infer behavioral context which may not always succeed. For example, 1(a) uses even to denote a Java list, while 1(d) uses nums to represent the same Python array. Still, identifier names can be informative in describing the behavior of code [86, 87] and are thereby useful as a metric.

Using an AST-based approach to identify similar code across languages is useful but not consistently viable due to the language-specific constructs. For example, 1(a) uses a standard for loop to populate the list while 1(d) uses list-comprehension which is a pythonic construct for the same task. The nearest structural match would be 1(b) since it also uses a for loop. However, the functions 1(a) and 1(b) are behaviorally different. In such cases, a dynamic approach based on IO similarity would reveal the differences.

Behavioral approaches also have their limitations. For example, IntStream from 1(c) is specific to Java v8 and above. Similarly range from 1(d) is specific to Python v2.x. Hence, the right version of the
language and libraries is a prerequisite and in many cases a major bottleneck for dynamic similarity.

There is no single best similarity measure for cross-language code-to-code search. It depends on multiple varying criteria which cannot be generalized for all cases. Hence, we need a code-to-code search technique that enables search based on multiple similarity measures. This can be handled by aggregating the multiple measures into a single measure or by using all the measures in tandem.

When aggregating into a single measure, it is easier to compare but there are limitations. Search methods [38, 41] and evolutionary algorithms [25] convert multiple search objectives to a single-objective search problem [53]. Using such methods is not very optimal [92] as ranking the results would be very subjective if the objectives are independent or weakly correlated to each other. An aggregated approach can also lead to bias in comparison [17]. This motivates the use of an approach to ranking that preserves each of the similarity measures, using them in tandem. In this section, we have shown that code that matches on one measure (e.g., AST similarity would match 1(a) with 1(b)) may differ on another measure (e.g., behavioral similarity shows 1(a) and 1(b) are different). When aggregating into a single measure, such nuances are often lost.

Non-dominated ranking orders search results across multiple independent search objectives without aggregating them. We select three similarity measures to represent the context, structure and behavior across programming languages. These measures are weakly and moderately correlated to each other, presenting non-overlapping perspectives when used to compare code. Further, non-dominated ranking could provide information sufficient to explain why one result is ranked above another in a meaningful way, an ability that is lacking in aggregated approaches (e.g., “1(a) and 1(b) have more structural similarity”). While we do not investigate the value of the explanations in this work, we conjecture that such explanations may be useful when developers discern between search results and leave that for future exploration.

3 CODE-TO-CODE SEARCH ACROSS LANGUAGES

Figure 3 depicts the general workflow of COSAL:

1. Offline, a Repository is crawled to extract Code Snippets (e.g., GitHub, a local File System.)
2. Offline, Indices are created for each of the following:
   a) A Token Index for code names and libraries (§ 3.1).
   b) A language-agnostic AST Index for code structures (§ 3.2).
   c) If the code can be executed, the IO Index is recorded (§ 3.3).
3. During search, a Code Query is processed in the same manner as Steps 2(a)-(c), to gather Tokens, AST, and IO information.
4. Non-Dominated Sorting identifies Search Results, which are ranked and returned to the user (§ 3.4).

We illustrate COSAL using the code examples from Figure 1.

3.1 Token-based Search

Fragments of code that are contextually similar often use similar variable names [69], though the naming conventions vary by language. For example, Java primarily uses camelCase conventions while Python uses snake_case. Libraries across languages tend to share similar function names or contexts [4]; for example List class from java.util library and list from the Python __builtins__ library are both commonly used to represent an array. Developers tend to describe the code in comments based on the functionality [33]. We infer context by extracting non-language-specific tokens from source code and comments as follows:

- Remove language-specific keywords based on the documentation [18, 60]. For example, Java tokens public and static, and Python tokens def and assert, are all removed.
- Remove frequently-used words used in a language based on common coding conventions. For example, in Python, the token self is often used to denote the class object.
- Remove common stopwords from the English vocabulary [15], such as does and from.
- Split tokens to address language-specific nomenclature. Variables typically use camelCase in Java and snake_case in Python. These are split into {“camel” and “case”} and {“snake” and “case”}, respectively.
- Remove tokens of length less than MIN_TOK_LEN.
- Convert all the tokens to lowercase.

A repository of code is tokenized using the above approach and stored in an ElasticSearch [29] index. For a user’s code query, the tokens generated from the indexing approach are looked up in the search index and the best matched results are returned using the token similarity distance ($d_{token}$). This distance is the same as the Jaccard Coefficient [59] and is defined as follows:

$$d_{token} = \frac{|tokens_{query} \cap tokens_{result}|}{|tokens_{query} \cup tokens_{result}|}$$

$d_{token}$ will range from $[0, 1.0]$. Larger values of $d_{token}$ indicate higher similarity between the query and result.

For the functions in Figure 1, the generated tokens using this approach are shown in Figure 2 and the token similarity distance for each pair of functions is shown in Table 1 ($d_{token}$). If the Java function 1(a) is the query, the best Python result would be 1(d) ($d_{token} = 0.067$). The common token max is extracted from these functions help in identifying this similarity. Note that none of the functions in Figure 1 have code comments, while in our implementation the comments are analyzed.

In many cases the token-based analysis cannot yield ideal results. It relies on self-describing snippets; the choice of variable names, function names, libraries used, and comments all impact the results.
Table 1: Similarity measures for Java($J$) and Python($P$) functions from Figure 1. High similarity implies high values ($\uparrow$) of $d_{\text{token}}$, low values ($\downarrow$) of $d_{\text{AST}}$, and high values ($\uparrow$) of $d_{\text{IO}}$.

| Snip 1 | Snip 2 | $d_{\text{token}}$ ($\uparrow$) | $d_{\text{AST}}$ ($\downarrow$) | $d_{\text{IO}}$ ($\uparrow$) |
|--------|--------|-------------------------------|-------------------------------|-------------------------------|
| $J$ - 1(a) | $P$ - 1(b) | 0.0 | 1 | 0.0 |
| $J$ - 1(a) | $J$ - 1(c) | 0.333 | 16 | 1.0 |
| $J$ - 1(a) | $P$ - 1(d) | 0.067 | 7 | 1.0 |
| $P$ - 1(b) | $J$ - 1(c) | 0.0 | 17 | 0.5 |
| $P$ - 1(b) | $P$ - 1(d) | 0.0 | 8 | 0.5 |
| $J$ - 1(c) | $P$ - 1(d) | 0.059 | 21 | 1.0 |

Not all code snippets adhere to intuitive naming conventions. For example, in 1(c), the programmer chose very generic names. Still, we show in § 6.1 that our approach for tokenization yields more precise results compared to full text search.

3.2 AST-based Search

A tree-based representation for comparison across languages is challenging since there is no generic AST representation that encompasses syntactic features of different languages. Traditional AST parsers like ANTLR [61], JavaParser [83], python-ast [65] modules use different grammars to denote similar features. For example, a function node in JavaParser is represented as MethodDeclaration while the python-ast parser represents the node as FunctionDef. As a result, to compare ASTs of different languages requires a mapping scheme between each pair of programming languages.

For better scalability to additional programming languages, we built a parser for a generic AST. By mapping the ASTs for Java and for Python onto the generic AST, we can compare across these languages (see § 7.3.2 for a discussion on scalability). The generic AST is based on our intuition and chosen languages, and there may be more effective or efficient representations. It contains a superset of the language features, as follows:

- **Common control structures**: Control structures are simplified and clustered. For example, the loop node is used for the Java constructs: for, forEach, while and do-while; and Python constructs: for, while, and list-comprehension.
- **Normalizing Variable**: Variables are denoted as var nodes.
- **Normalizing Literals**: Literals are denoted as lit nodes.
- **Normalizing Operators**: Operators are denoted as op nodes.
- **Language specific features**: If a feature is implemented in only one language, a custom node is created. For example, switch is specific to Java and not supported in Python. As a result, a custom node switch is created for this statement.

Similarity between ASTs is computed using the Zhang-Sasha algorithm [90] ($d_{\text{AST}}$). The algorithm computes the minimum number of edits required to transform an ordered labeled tree to another ordered labeled tree in quadratic time. $d_{\text{AST}}$ will range from $[0, \infty)$. Lower values of $d_{\text{AST}}$ are associated with higher similarity.

For the functions in Figure 1, the generated ASTs are shown in Figure 2 and the AST edit distance for each pair of functions is shown in Table 1 ($d_{\text{AST}}$). Based on $d_{\text{AST}}$, a query with Python function 1(d) yields 1(a) as the best Java result ($d_{\text{AST}} = 7$). Notably, the syntactic constructs of the two functions are also different. The Python search query 1(d) uses a list-comprehension which is a Python feature and the Java search result 1(a) uses a for loop. Identifying the matching search result is possible, since list-comprehension and the for loop are denoted as loop nodes in the grammar for the generic AST.

There are cases where a generic AST-based approach is non-ideal. For example, if the Java function 1(a) is queried using $d_{\text{AST}}$, the best Python result would be 1(b). This is because both functions use traditional for loops and updates the return array sequentially, and yet, the search result is behaviorally different from the query. Such scenarios can be handled using dynamic similarity.

3.3 Input-Output based Search

Dynamic search in COSAL, is performed by clustering code based on the their IO relationship. To determine the IO relationship between two pieces of code, we use SLACC [51], a publicly-available IO-based cross-language code clone detection tool. SLACC segments code into executable snippets of size greater than MIN_STMTS and executed on ARGS_MAX arguments generated using a grey-box strategy. The executed functions are then clustered using a similarity measure ($\text{sim}$) based on the inputs and outputs of the functions. Consider a query $q$ and a potential search result $s$. Let $Q$ and $S$ be sets of segments identified by SLACC from $q$ and $s$ respectively. We define the IO similarity as:

$$d_{\text{IO}}(q, s) = \frac{1}{|Q|} \sum_{s_i \in S} \text{maximize } \text{sim}(q_i, s_i)$$

The value $d_{\text{IO}}$ range from $[0, 1.0]$. Higher similarity corresponds to higher values of $d_{\text{IO}}$.

The IO similarity between any $q$ and $s$ is not commutative. This is because it is often preferred for a search result to contain extra behavior as compared to the query [76]. Also, there may be a many-to-one mapping where multiple query segments match with a single segment in the result. Consider an example: let $Q = \{q_1, q_2, q_3, q_4, q_5\}$ be set of five segments and $S = \{s_1, s_2, s_3\}$ be set of three segments identified by SLACC. Segments $q_1$, $q_2$, and $q_3$ find segments $s_1$, $s_2$, and $s_1$ to be the most similar, respectively, with similarity scores ($\text{sim}$) of $\text{sim}(q_1, s_1) = 0.8$, $\text{sim}(q_2, s_2) = 0.95$ and $\text{sim}(q_3, s_1) = 0.7$. Notice how $s_1$ is identified as the closest match for both $q_1$ and $q_3$. Segments $q_4$ and $q_5$ did not find segments in $S$ with similarity greater than 0.0. In this case, $d_{\text{IO}}(q, s) = 0.49$.

As a practical example, say a developer is looking for a Java API for QuickSelect\(^2\), which finds the $k^{\text{th}}$ smallest number from an array of integers. It has a method that identifies a random pivot in the array and a method that swaps values. However, these methods do not call each other. Thus, to characterize the behavior of this file, we characterize and aggregate the behavior of segments of the file. Then, when comparing to a custom Python QuickSort\(^3\) API that has a function to recursively find a random pivot and perform a swap operation, a match is identified even though the number of methods and how they accomplish the same task are different.

\(^2\)from org.apache.datasketches
\(^3\)stackabuse.com/quicksort-in-python
The dynamic similarity between all the functions in Figure 1 is shown in Table 1. We noticed in § 3.2 that 1(a) and 1(b) were very similar based on \( d_{\text{AST}} \), but functionally different. Using behavioral analysis, we see that they are indeed functionally different as \( d_{\text{IO}} = 0.5 \) for both measures of \( d_{\text{IO}} \). In contrast, 1(a) and 1(c) are similar based on behavior \( (d_{\text{IO}} = 1.0) \), even though structural similarity \( (d_{\text{AST}}) \) is low.

### 3.4 Non-Dominated Ranking

We use Non-Dominated Sorting, which is a component of NSGA-II [19], and orders results with multiple objectives without aggregation. COSAL uses this algorithm to rank search results based on \( d_{\text{token}}, d_{\text{AST}}, \) and \( d_{\text{IO}} \). We note that the similarity measures considered in this work are weakly correlated, as described in § 7.2, making this an appropriate algorithmic choice. We use the algorithm in a novel context; this is the first work that uses non-dominated sorting for code-to-code search.

In our context, each similarity measure is an objective. COSAL incorporates non-dominated ranking as follows:

1. **Individual Search**: For a query, the top \( \text{TOP}_N \) search results are fetched using each similarity measure \( (d_{\text{token}}, d_{\text{AST}} \) and \( d_{\text{IO}}) \) independently.
2. **Merge**: The individual search results are merged such that duplicate instances of search results are removed.
3. **Sort**: The merged results are sorted by NSGA-II [19] by measuring the dominance of one result over the other.

A search result \( s \) is said to dominate a search result \( t \), if \( s \) is no worse than \( t \) in any objective and is better than \( t \) in at least one objective. Otherwise, there is a tie. In case of a tie, we select the result that has the dominant objective closest to the optimal value. For example, consider the following scenarios of the relationship between \( s \) and \( t \) and three hypothetical similarity measures, \( d_A, d_B, \) and \( d_C \), where higher values mean higher similarity:

| scenario | \( d_A \) | \( d_B \) | \( d_C \) | winner |
|----------|----------|----------|----------|--------|
| 1        | \( s > t \) | \( s > t \) | \( s > t \) | \( s \) |
| 2        | \( s = t \) | \( s = t \) | \( s > t \) | \( s \) |
| 3        | \( s = t \) | \( s < t \) | \( s > t \) | tie    |
| 4        | \( s < t \) | \( s < t \) | \( s > t \) | tie    |

For scenario 1, \( s \) is better than \( t \) on all measures, making \( s \) the winner. In scenario 2, since \( s \) is better than \( t \) on one measure, and is never worse than \( t \), \( s \) is the winner. In the third scenario, \( s \) is better than \( t \) on one measure (\( d_C \)), and worse on another (\( d_B \)). Therefore, there is a tie. Similarly on scenario 4, \( s \) is worse than \( t \) on two measures and better on one, so it is also a tie. Ties are broken by looking at the search results that are better for each similarity measure and then comparing to optimal values (typically 1 or 0, depending on whether high or low values represent better similarity).

Using the examples from § 2, consider the Python functions 1(d) and 1(b) in Figure 1 as queries. The potential cross-language results are 1(a) and 1(c). We show the relationships between the potential results using three similarity measures; see Table 1 for specific values. The winner for each comparison is bolded for clarity.

| query | \( d_{\text{token}} \) | \( d_{\text{AST}} \) | \( d_{\text{IO}} \) | winner |
|-------|-------------------|-----------------|-----------------|--------|
| 1(d)  | \( 1(a) > 1(c) \) | \( 1(a) = 1(c) \) | \( 1(a) = 1(c) \) | \( 1(a) \) |
| 1(b)  | \( 1(a) = 1(c) \) | \( 1(a) < 1(c) \) | \( 1(a) < 1(c) \) | \( \text{tie} \) |

When the query is 1(d), \( 1(a) \) is better than \( 1(c) \) on two of the measures, and equal on the third, thus making \( 1(a) \) the winner. When the query is 1(b), \( 1(a) \) is the winner for \( d_{\text{AST}} \) and \( 1(c) \) is the winner for \( d_{\text{IO}} \), meaning we need to break the tie.

To break ties, we compute distances between each search result and the optimal value for each similarity measure (omitting similarity measures on which the results are tied). The optimal value for \( d_{\text{AST}} \) is 0, as that represents isomorphic ASTs. The optimal value for \( d_{\text{IO}} \) is 1, as that represents a perfect match in code behavior (i.e., the search result and query return the same output for all the provided inputs). The optimal value for \( d_{\text{token}} \) is also 1, as this represents highly similar syntax. We use a normalized distance because the similarity measures have different ranges of values. Thus, normalizing ensures a uniform comparison scale between the different similarity measures and subsequently avoid the precedence of one similarity measure over other similarity measures. The normalized distance of a similarity measure \( X \) on a snippet \( s \) is computed as \( \frac{d_X(s) - \min(d_X)}{\max(d_X) - \min(d_X)} \).

### 4 RESEARCH QUESTIONS

There does not exist a cross-language code-to-code search tool to compare against directly (see § 8). Thus, our evaluation assesses each part of COSAL: the ranking algorithm, within-language code-to-code search compared to state-of-the-practice public code search tools, and cross-language clone detection. The first research question (RQ) examines the similarity measures and ranking:

| RQ 1 | Does non-dominated ranking using tokens, AST and IO yield better results for cross-language code-to-code search as compared to any subset or aggregation of those search similarity measures? |
| RQ 2 | How effective is COSAL in cross-language code-to-code search compared to state-of-the-practice public code search tools? |

After validating the choice of using multiple code similarity measures and non-domination ranking, COSAL is compared to the state-of-the-practice search in GitHub Search and ElasticSearch which are based on full text search. We ask:
FaCoY [40], the state-of-the-art in code-to-code search is within-language, but COSAL is a multi-language tool. We limit our tool to within-language code-to-code search and evaluate it against FaCoY.

**RQ 3**

How effective is COSAL in within-language code-to-code search as compared to the state-of-the-art?

Code-to-code search is often used in clone detection [66, 68]. Using COSAL for clone detection, we compare against ASTLearner [63], CLCDSA [56], and SLACC [51]:

**RQ 4**

Can COSAL effectively detect cross-language code clones?

## 5 STUDY

The setup for each RQ is different. In all evaluations we make a best effort to be fair in the comparison. The RQs are summarized in Table 2, which lists the application (either code-to-code search or clone detection), baseline approaches, language(s), and benchmarks.

### 5.1 Data

The data used in this study are available online [2].

#### 5.1.1 AtCoder (AtC)

We require a labeled set of similar code snippets in multiple programming languages for queries and search results. Hence, like prior studies [56, 63], we use AtCoder [5] to create a dataset of similar code snippets across different programming languages. Competitive programming contests like AtCoder [5] have open problems where users can submit their solutions in most common programming languages. Solutions which are syntactically incorrect or do not pass the extensive test suite are filtered out by AtCoder. All the accepted solutions for a single problem implement the same functionality and are behavioral code clones. If a search query and a result belong to the same problem, we consider the result to be valid and the query-result pair as valid code clones; the problem solutions are the ground truth in our experiments. We limit our study to the most recent 398 problems which had solutions in Java or Python. For these problems, we crawled 43,146 files from all the accepted Java and Python solutions. Table 3 lists an overview of the dataset used for the study; 307 of the 398 problems have both a Java and Python solution.

#### 5.1.2 BigCloneBench (BCB)

BigCloneBench [79] is one of the largest publicly available code clone benchmarks for Java with over 55,000 source code files harvested from approximately 25,000 open-source repositories. Table 3 lists a summary of BigCloneBench. We consider query-result snippets belonging to the same functionality as a valid search result. Fragments of code with less than 6 lines or 50 tokens are not considered which is a standard minimum clone size for benchmarking [12, 40, 79].

### 5.2 Baselines

We compare COSAL to each of the other tools by searching over the same data sets. For RQ3 and RQ4, we used the source code in the GitHub repositories of the tools for experimentation.

#### 5.2.1 RQ2 – Text Search

Google search is commonly used by developers for code search [75]. Textual queries can take the form of keywords, expected code, or exceptions raised. In our study, Google failed to index our code repository after a six week wait. As a result, we turned to a custom full text search using ElasticSearch [29] which takes in a code snippet, tokenizes the code and identifies results based on Lucene’s Practical Scoring Engine [6]. For this study, each Java and Python file is added to an ElasticSearch index and searched using the ElasticSearch programmatic search API.

#### 5.2.2 RQ2 – GitHub Search

GitHub search engine is an IR-based search model over code repositories, including issues, pull request, documentation, and code data [81]. Using the built-in code search on GitHub, code can be searched globally across all of GitHub, or searched within a particular repository or organization. We add the Java and Python files from the dataset to a single GitHub repository and search within the repository using the GitHub Search API [80].

#### 5.2.3 RQ3 – FaCoY

FaCoY [40] is a Java-based code-to-code search tool that uses a query alternation approach using relevant keywords from StackOverflow Q&A posts. FaCoY can be modified to change its search database from Q&A posts to custom datasets. In our experiments, we redirected the search to the repositories of code from the AtCoder and BigCloneBench datasets. Similar to the experiments in the FaCoY evaluation when comparing against research tools, FaCoY does not use StackOverflow in our baseline.

#### 5.2.4 RQ4 – ASTLearner

Perez and Chiba developed a semi-supervised cross-language syntactic clone detection method that we call ASTLearner [63]. It uses a skip-gram model and an LSTM based encoder. The encodings train a feed forward neural network classifier using negative sampling to identify clones. ASTLearner considered code as clones if the classifier score is greater than 0.5.

#### 5.2.5 RQ4 – CLCDSA

Cross Language Code Clone Detection [56] (CLCDSA), uses syntactic features and API documentation to detect cross-language clones in Java, Python and C#. Nine features are extracted from the AST; API call similarity is learned using API documentation and a Word2Vec [54] model. The vectorized features train a reconfigured Siamese architecture [8] using a large amount

---

**Table 2: Summary of RQs with the application (code-to-code search, clone detection), baselines, benchmarks (AtCoder or BigCloneBench) and the language(s). COSAL_{single} represents COSAL using a single similarity measure.**

| RQ | Purpose | Application | Baselines | Language(s) | Benchmarks |
|----|---------|-------------|-----------|-------------|------------|
| 1  | Merit of using multiple similarity measures | Code-to-Code Search | COSAL_{single} | Java→Python | AtCoder |
| 2  | Vs state-of-the-practice cross-language tools | Code-to-Code Search | FaCoY | Java→Python | AtCoder, BigCloneBench |
| 3  | Vs state of the art within-language tools | Code-to-Code Search | CLCDSA, ASTLearner | Java→Python | AtCoder |
| 4  | Using COSAL to identify similar code | Clone Detection |  |  |  |
of labeled data. CLCDSA uses cosine similarity to detect clones; the best F1 scores were when the similarity threshold was 0.5.

5.2.6 RQ4 – SLACC. Simion-based Language-Agnostic Code Clones [51] (SLACC), uses IO behavior to identify clones. It is also used in COSAL for dynamic similarity. Here, we use SLACC as a baseline in its original context, clone detection. We use the same values for the hyper-parameters set by the authors of SLACC: MIN_STMTS is set to 1; ARGS_MAX is set to 256; SIM_T is set to 1.0.

### 5.3 Metrics

#### 5.3.1 Code Search. For code search applications (RQ1, RQ2, RQ3), we use Precision@k, SuccessRate@k, and MRR.

\[
\text{Precision@k} = \frac{\#\text{valid clones identified}}{\#\text{total clones}} \quad \text{and} \quad \text{Recall} = \frac{\#\text{valid clones identified}}{\#\text{total clones}}
\]

\[
\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

\[
\text{MRR} = \frac{1}{\#\text{queries}} \sum_{\text{query}} \frac{1}{\text{TOP}_N} \text{BR}(q) \quad \text{and} \quad \text{SR} = \frac{\#\text{relevant results}}{\#\text{total results}}
\]

\[
\text{SR@k} = \frac{\#\text{relevant results}}{\#\text{total results}}
\]

\[
\text{MRR} = \frac{1}{\#\text{queries}} \sum_{\text{query}} \frac{1}{\text{TOP}_N} \text{BR}(q)
\]

\[
\text{Precision@k} = \frac{\sum_{q \in Q} \text{delta}(k) + \text{delta}(q)}{|Q|} \quad \text{and} \quad \text{SuccessRate@k} = \frac{\sum_{q \in Q} \text{delta}(k) + \text{delta}(q)}{|Q|}
\]

5.3.2 Clone Detection. For clone detection [59, 63] (RQ4), we use Precision, Recall and F1 score. Precision (P) is the ratio of valid clones to the number of retrieved clones. Recall (R) is the ratio of the number of accurately detected clones to the number of total actual clones. F1 or F-Measure, is the harmonic mean of precision and recall. We define |C_v| as the number of valid clones identified, |C_\#| as the number of valid clones not identified, and |NC_\#| as the number of invalid clones identified:

\[
P = \frac{|C_v|}{|C_v| + |NC_\#|} \quad R = \frac{|C_v|}{|C_v| + |C_\#|} \quad F1 = \frac{2 \times P \times R}{P + R}
\]

5.4 Experimental Setup

Our experiments were run on a Ubuntu 18.04 LTS Virtual Machine with 32 CPUs and 64GB memory using a Dell PowerEdge R640 server with Intel Xeon Silver 4210 CPU @ 2.2 GHz and VMware ESXi 6.7.0 hypervisor. The experiments have four hyper-parameters:

5.4.1 Minimum token size. (MIN_TOK_SIZE in § 3.1) This is set to three. IR based techniques [33, 86] on source code find that tokens less than three characters are irrelevant.

5.4.2 Minimum segment size. (MIN_STMTS in § 3.3) A small value of MIN_STMTS results in more granular snippets. We set it to 1 for maximum number of behavioral snippets of code.

5.4.3 Maximum number of arguments. (ARGS_MAX in § 3.3) Prior work finds ARGS_MAX<256 was sufficient for cross-language clones in Google Code Jam [27], so we use the same.

5.4.4 Number of individual search results. (TOP_N in § 3.4) This is set to 100. We experimented on COSAL with 10% of the AtCoder dataset varying TOP_N in [10, 20, 50, 100, 200, 500]. For TOP_N greater than 100, we see a minimal change in the code search metrics. Hence, for each individual search, we report the top 100 search results.

### 6 RESULTS

We present the results of each RQ in turn.

#### 6.1 RQ1: Single vs Multiple Search Similarity Measures

In a cross-language search context, we compare the results of COSAL with multiple search similarity measures to COSAL with subsets of the similarity (e.g., COSAL_AST is COSAL with only the AST similarity). The validation of this study was performed using 'leave-one-out' cross-validation [72] where each code fragment is used as a query against all other fragments in the repository. We use this approach over the traditional k-fold cross validation since ‘leave-one-out’ is approximately unbiased and more thorough [50].

Each of the 43,146 code fragments is used as a query. The results are detailed in Table 4. Overall, COSAL outperforms the other formulations that use subsets of the similarity measures. It also outperforms an alternate ranking approach based on weighted measures (KDTTree [13]).

We observe that token-based search (COSAL_TOKENS) and AST-based search (COSAL_AST) are less precise individually compared to dynamic search (COSA_GLASS), but have higher success rate for k = [5, 10]. When both the static similarity measures are used as part of a bi-similarity search (COSAL_STATIC), we see better metrics compared to each similarity individually, and better metrics than the dynamic approach COSAL_GLASS in P@k and SR@k when k > 1.

The power of the technique comes from using static and dynamic information without converting them into a single search metric. Rather than non-dominated ranking, an alternate avenue would be a weighted approach. For example, KDTREE+AST+TOKEN uses d_TOKEN, d_AST and d_IO to build a KDTREE [13], a common approach used

### Table 3: Summaries of AtCoder and BigCloneBench datasets

| Metric        | AtCoder (ATC) | BigCloneBench (BCB) |
|---------------|---------------|---------------------|
| #Problems     | 364           | 43                  |
| #Files        | 20,828        | 55,499              |
| Avg. Files/Problem | 57          | 1291                |
| #Methods      | 81,896        | 765,331             |
| Avg. Lines/Feature | 10,020      | 22,318              |
| Avg. Lines/File | 51            | 278                 |

| Metric        | Java          | Python            |
|---------------|---------------|-------------------|
| #Problems     | 333           | 43                |
| #Files        | 22,318        | 55,499            |
| Avg. Files/Problem | 67          | 1291               |
| #Methods      | 10,020        | 765,331           |
| Avg. Lines/Feature | 14          | 278                |
| Avg. Lines/File | 14            | 278                |
Table 4: RQ1 & RQ2: Cross-language code search results on AtCoder dataset comparing COSAL against the state-of-the-practice (SotP) GitHub, and ElasticSearch. COSAL\_\text{token}, COSAL\_\text{AST}, COSAL\_\text{SLACC} use single search similarities (Single Sim.) \(d_{\text{token}}\), \(d_{\text{AST}}\) and \(d_{\text{IO}}\) respectively. COSAL\_\text{static} uses \(d_{\text{token}}\) and \(d_{\text{AST}}\) with non-domination. KD\_\text{AST+token} performs code search with KDTree using \(d_{\text{AST}}\) and \(d_{\text{IO}}\). Code search techniques using multiple similarity measures are represented with Multi Sim.

| Search            | MRR      | \text{P@1/3/5/10} | \text{SR@1/3/5/10} |
|-------------------|----------|-------------------|--------------------|
| SotP              | ElasticSearch | 29/37            | 27/34/27/33/34     |
|                   | GitHub   | 32/33/35/34/39   | 32/34/28/34/32     |
| Single            | COSAL\_\text{token} | 31/32/33/34/35 | 32/33/34/35/36     |
|                   | COSAL\_\text{AST}  | 34/34/35/36      | 34/34/35/36/37     |
|                   | COSAL\_\text{SLACC} | 45/46/47/48     | 45/46/47/48/49     |
| Multi             | COSAL\_\text{static} | 43/44/45/46/47  | 43/44/45/46/47     |
| Sim.              | KD\_\text{AST+token} | 39/40/41/42/43  | 39/40/41/42/43     |
| COSAL             | 64       | 58/64/65/66/67   | 58/68/79/81/94     |

Using non-dominated ranking with static and dynamic similarity measures improves the quality of results for code-to-code search compared to subsets or a weighted aggregation of measures.

6.2 RQ2: State-of-the-Practice Cross-language Code-to-Code Search

We compare COSAL against GitHub Search (§5.2.2) and a custom full text search based on ElasticSearch (§5.2.1). We use ‘leave-one-out’ cross-validation with each of the 43,146 code fragments as a query. Results are shown in Table 4.

We observe that between the textual code search tools, GitHub Search has better MRR, \text{Precision}@k and \text{SuccessRate}@k compared to ElasticSearch except for \text{SuccessRate}@10. Yet, GitHub Search and ElasticSearch are worse off compared to COSAL in all metrics.

| Search     | MRR       | \text{P@1/3/5/10} | \text{SR@1/3/5/10} |
|------------|-----------|-------------------|--------------------|
| SotP       | ElasticSearch | 29/37            | 27/34/27/33/34     |
|            | GitHub    | 32/33/35/34/39   | 32/34/28/34/32     |
| Single     | COSAL\_\text{token} | 31/32/33/34/35 | 32/33/34/35/36     |
|            | COSAL\_\text{AST}   | 34/34/35/36      | 34/34/35/36/37     |
|            | COSAL\_\text{SLACC} | 45/46/47/48     | 45/46/47/48/49     |
| Multi      | COSAL\_\text{static} | 43/44/45/46/47  | 43/44/45/46/47     |
| Sim.       | KD\_\text{AST+token} | 39/40/41/42/43  | 39/40/41/42/43     |
| COSAL      | 64        | 58/64/65/66/67   | 58/68/79/81/94     |

6.3 RQ3: Single-language Java code search comparing COSAL to the state-of-the-art (SotA) FaCoY on AtCoder and BigCloneBench.

| Search     | MRR       | \text{P@1/3/5/10} | \text{SR@1/3/5/10} |
|------------|-----------|-------------------|--------------------|
| AtCoder    | SotA      | FaCoY             | 51/52/53/54/55     |
|            | Single    | COSAL\_\text{token} | 46/47/48/49/50     |
|            | Sim.      | COSAL\_\text{AST}    | 40/41/42/43/44     |
|            | COSAL\_\text{SLACC} | 40/41/42/43/44 | 39/40/41/42/43     |
| Multi      | COSAL\_\text{static} | 53/54/55/56/57 | 43/44/45/46/47     |
| Sim.       | KD\_\text{AST+token} | 39/40/41/42/43 | 39/40/41/42/43     |
| COSAL      | 64        | 58/64/65/66/67   | 58/68/79/81/94     |

The results for MRR, \text{Precision}@k and \text{SuccessRate}@k are tabulated in Table 5. COSAL has better scores on all metrics compared to FaCoY. Even if COSAL is used with only static similarity measures (COSAL\_\text{static}), it consistently performs better than FaCoY.

Since, FaCoY supports only Java, we also compare FaCoY to COSAL using BigCloneBench. This experiment moves us toward evaluating the feasibility of COSAL with open-source projects. We again use ‘leave-one-out’ cross-validation where each file from BigCloneBench is used as a query and the other files are used as search results. A search result is considered valid if it has the same functionality group as the search query.

Compared to AtCoder, the BigCloneBench dataset yields better results for all techniques. This is because the 43 functionalities in BigCloneBench have minimal overlap. This can be corroborated by the better scores for token-based search compared to the AST-based search on BigCloneBench dataset. In contrast, on AtCoder, AST-based search out-performs token-based search. Like the AtCoder dataset, search based on a combination of measures (COSAL\_\text{static}) yield better results compared to FaCoY.

Only 4,984 (9%) of the files from BigCloneBench are executable by SLACC; the remaining files depend on external libraries. Thus, dynamic similarity (COSAL\_\text{SLACC}) has much lower scores in Table 5. Subsequently, the inclusion of dynamic similarity hardly contributes to the results of COSAL as highlighted by their similar values for COSAL\_\text{static} and COSAL. We dive deeper into the role of dynamic similarity in §7.1.

6.4 RQ4: Cross-language code clone detection

As there is no existing tool for cross-language code-to-code search, we instead compare to cross-language code clone detection techniques: ASTLearner, CLCDSA and SLACC. While code-to-code search for information retrieval [21, 30]. Although KD\_\text{AST+token} and COSAL use the same similarity measures for code search, the former under-performs on all metrics compared to the latter. This suggests that aggregation of similarity measures into a single measure does not help code search as these measures complement each other.

COSAL obtains better \text{Precision}@k, \text{SuccessRate}@k and MRR compared to GitHub Search and ElasticSearch.
We have evaluated COSAL extensively against prior work in code-AST Learner and CLCDSA, and SLACC on AtCoder. To account for variance, we repeat this validation set in the search database to ensure a fair comparison to the validation set and five for the test set. We used the default hyperparameters for scalability, and threats to the validity.

In §6.3 and Table 5, we observe a low scores for code search using IO-based similarity (COSAL<sub>static</sub>) compared to other techniques due to the small sample of files in BigCloneBench (9%) with executable code. To study the relative contribution of dynamic analysis to COSAL results, we repeat the validation study on BigCloneBench but restricted to the files that can be executed (4,984).

Since the gains are not very high with the BigCloneBench dataset, the correlation between the static similarity measures using an aggregated weighted approach (K<sub>D</sub><sup>token</sup><sub>0.5</sub><sup>AST</sup><sub>0.5</sub><sup>token</sup><sub>0.5</sub><sup>IO</sup><sub>0.5</sub>) is still better than ASTLearner and comparable to CLCDSA. However, this cannot be generalized across datasets. BigCloneBench is built on Java code from open-source projects. For cross-language search (Table 4), using dynamic and static similarity measures vastly improves the results. This is due to the syntactic differences between languages which can be overcome in many cases with dynamic information [35]. Hence, the benefit of including dynamic similarity data must be balanced against the cost and context.

### 7.2 On Non-Dominated Sorting
For cross-language code search, combining the search similarity measures using an aggregated weighted approach (K<sub>D</sub><sup>token</sup><sub>0.5</sub><sup>AST</sup><sub>0.5</sub><sup>token</sup><sub>0.5</sub><sup>IO</sup><sub>0.5</sub><sup>token</sup><sub>0.5</sub><sup>IO</sup><sub>0.5</sub><sup>AST</sup><sub>0.5</sub><sup>IO</sup><sub>0.5</sub>) results in lower MRR, P@k and SR@k compared to the non-dominated sorting approach (Table 4). As one potential explanation, this poorer performance for the aggregation approach could be a result of bias due to the independence or weak correlation between the three similarity measures [17]. In this section, we explore the impact of the correlations between the similarity measures.

---

**Table 6: RQ4: Cross-language performance of COSAL in clone detection compared to ASTLearner, CLCDSA, and SLACC on AtCoder.**

| Clone Detector | Precision | Recall | F1   |
|----------------|-----------|--------|------|
| Single Sim.    |           |        |      |
| ASTLearner     | 25        | 80     | 38   |
| CLCDSA         | 49        | 83     | 62   |
| SLACC          | 66        | 19     | 30   |
| Multi Sim.     |           |        |      |
| COSAL<sub>static</sub> | 48        | 85     | 61   |
| COSAL          | 55        | 89     | 68   |

**Table 7: Performance based on 4,984 executable code snippets from BigCloneBench.**

| Dataset | Search | MRR | P@1/3/5/10 | SR@1/3/5/10 |
|---------|--------|-----|------------|-------------|
| AtCoder | GitHub | 81 | 78/74/76/81 | 81/83/99/94 |
| Java ↔ Python | FaCoY | 79 | 74/70/68/57 | 74/76/81/84 |
| Java ↔ Python | COSAL | 82 | 81/78/74/67 | 81/83/99/94 |
| Java ↔ Python | SLACC | 80 | 78/75/72/66 | 79/83/97/91 |
| Java ↔ Python | COSAL | 83 | 81/79/74/68 | 81/86/91/96 |

**Table 8: Pearson’s correlation (r) between d<sub>token</sub>, d<sub>AST</sub> and d<sub>IO</sub> for cross-language snippets on AtCoder (AtC) and within-language Java snippets on AtCoder and on 4,984 executable BigCloneBench (BCB) datasets.**

| Dataset | Language | Correlations (r) |
|---------|----------|------------------|
| AtC     | Java ↔ Python | token, AST | token, IO | AST, IO |
| AtC     | Java ↔ Java | -0.38 | 0.33 | -0.41 |
| BCB     | Java ↔ Java | -0.46 | 0.53 | -0.71 |

For code clone detection, COSAL obtains better precision, recall, and F1 scores compared to ASTLearner and CLCDSA, without the need to build models. COSAL has lower precision to SLACC but much better recall and F1 score.
We explore three scalability concerns: indexing and searching open-cross-language static and dynamic information.

Consider the code snippets in Figure 4. For this example, COSAL identifies a similar code snippet from the collections library in Python: 4(b) returns the count of an element from a Counter. A Counter is a Python collection, like a bag, that takes elements and maintains a count of their occurrences. For this pair, we can see that they share few common tokens (count, get), do not have similar ASTs, but are behaviorally similar. Hence, the token-based and IO-based similarity in COSAL influence the ranking of search results and returns 4(b) as a valid search result for the query 4(a).

In our experiments, we see low scores for COSAL since only around 9% of the files in BigCloneBench had executable code. In this open-source exploration, around 68% of the Java and all the Python classes had executable code. The presence of dependent code in the libraries compared to the isolated files in BigCloneBench actually facilitated more widely applicable behavioral analysis.

Thus, we conclude that COSAL can be scaled to support open-source projects in the current implementation. The token-based and AST-based similarity measures for COSAL can be used on any project or file(s) in its current version. Since the behavioral similarity measure used by COSAL is heavily dependent on SLACC, scaling to support new projects would require the projects have all its dependencies satisfied and executable.

### 7.3.2 Support for new languages

COSAL currently supports Java and Python. While we have not demonstrated scalability to new languages, we comment on the effort required.

For dynamic behavior, COSAL is dependent on SLACC [51], so adding a new language to COSAL requires support in SLACC. However, COSAL-\textit{static} can be extended to new languages by adapting the token and AST analyses. A language-specific tokenizer like c-tokenizer [32] or a generic tokenizer like ANTLR [61] can be used to parse code and convert it into tokens as detailed in §3.1. For the AST, COSAL uses a generic AST to represent source code across different languages. Using a language-specific AST Parser like clang for C [45] or roslyn for .NET [22], code could be parsed and converted to the generic AST-based on the grammar available in the GitHub code repository for COSAL [2]. If a feature specific to a language is not supported by the grammar, a new node should be created based on the feature’s syntactic structure.

### 7.3.3 Adding New Search Similarity Measures

COSAL uses three search similarity measures for code-to-code search, which provides a start for this line of research. New search similarity measures can be added or existing similarity measures can be replaced in COSAL. First, a similarity measure to compare code snippets has to be defined. The similarity measure has to be a numerical value to support non-dominated ranking of the search results. Next, an index must be created characterizing the similarity measure. Lastly, the similarity measure has to be updated in the configuration file.

### 7.4 Threats to Validity

#### Language Bias

COSAL was implemented for Java and Python and may not generalize to other languages.

#### Baseline Bias

The ElasticSearch baseline for cross-language code-to-code search (in RQ2) is not an exact representation of a code-to-code search tool used by developers [70].
We present work in code similarity, search, and clone detection. 

8 RELATED WORK

8.1 Code Similarity

Source code similarity is used to characterize the relationship between pieces of code in software engineering applications such as program repair [28, 37, 57, 74, 75], code search [40, 49, 66], software security [67, 84, 89] and identifying plagiarized code [7]. Code similarity can be measured through static or dynamic analyses.

Techniques that use static code attributes to compute similarity often parse code into an intermediate representation based on text [7, 36, 47], AST [11, 34] or graph-based [26, 46] and compute a measure for syntactic similarity. For cross-language syntactic similarity, most techniques are text-based [43, 56, 58]. Tree- and graph-based approaches have not been explored for cross-language similarity due to language specific grammar. We tackle this challenge by creating a language-agnostic grammar by abstracting out common features across languages to build a generic AST (§ 3.2).

Techniques that execute code to determine similarity are classified as dynamic. For some techniques, functions are adjudged to be similar if they have similar inputs, outputs, and side-effects [24, 35, 51, 78]. Other techniques use abstract program states after execution to analyze the behaviors of the code fragments [39, 64, 77]. Dynamic measures are particularly successful in detecting code clones across languages since it does not rely on syntactic properties [35, 51]. Limitations to this approach include the need to execute the code which dictates the granularity [20] and runtime.

8.2 Code Search

In code search, the goal is to find code that is similar to a given query. Historically, developers have preferred general search engines such as Google and Bing when searching for code to reuse [73, 75, 76]. Some code search tools [1, 44] use code snippets as the query, a problem called code-to-code search. Solutions to code-to-code search vary in several dimensions, we list three: within [31, 40] vs. across languages [49, 56, 63], static [1, 34, 36] vs. dynamic analysis [51, 68], and index-based [40, 49, 81] vs. model based [31, 56, 63].

In cross-language code-to-code search, the query is a code snippet in one source language and the results are from a different target language(s). AROMA [49], supports cross-language code-to-code search across Java, Hack, JavaScript, and Python using static analysis based on the parse tree. Since AROMA is not publicly available, it is not used as a baseline in this study. InferCode [16] is a self-supervised cross-language (Java, C, C++ and C#) code representation approach using Tree-based Convolutional Neural Networks based on syntax subtrees. Since this work was performed in parallel to our study, we have not benchmarked COSAL against InferCode and leave that for future work. FaCoY [40] is a within-language code-to-code search tool on JAVA that uses query alteration to find semantically similar code snippets using Q&A posts.

8.3 Clone Detection

Clone detection is a special case of code-to-code search; results are identified as clones if they meet a specified similarity threshold. Clones are often categorized into four types: types I-III are based on syntax and type IV is based on behavior.

Most code clone detection tools [11, 26, 34–36, 46, 47, 78] have been proposed for single language clone detection and on static typed languages like Java [34, 42] and C [11, 34, 36, 88]. A small number of tools support cross-language code clone detection [51, 56, 58, 63]. API2Vec [58] detects clones between two syntactically similar languages by embedding source code into a vectors and subsequently comparing the similarity between the vectors. CLCDSA [56] identifies nine features from the source code AST and uses a deep neural network to learn the features and detect cross language clones. Perez and Chiba [63] propose an LSTM-based deep learning architecture using ASTs to detect clones in Java and Python code. These three tools build machine learning models to detect code clones. As a result, these techniques require a large number of annotated training data to build the model and the hyper-parameters need to be carefully optimized to avoid over-fitting.

SLACC [51] is a cross-language code clone detection tool that uses IO profiles. It succeeds in detecting code clones with high precision between programming languages with different typing schemes. However, SLACC requires the code snippets to be executable and as a result has low recall and a large runtime.

In a clone detection context, we use CLCDSA, the Perez and Chiba approach, and SLACC as baselines for comparison (§ 6.4).

9 CONCLUSION

We present COSAL, a cross-language code-to-code search tool that uses static and dynamic analyses. It uses two static similarity measures based on extracted tokens from source code and a tree edit distance based on a generic AST, and one dynamic similarity measure to compute IO similarity. For a given code search query, these three similarity measures find results using non-dominated sorting. Our experimental evaluation on 98,645 Java and Python files from AtCoder and BigCloneBench datasets show that COSAL outperforms state-of-the-art code search tools FaCoY and industrial benchmark of GitHub code search. We also compare COSAL to state-of-the-art clone detection techniques using the AtCoder dataset and find that COSAL has better Recall and F1. Cross-language code-to-code search appears to have a bright future, but more work is needed to evaluate it for more languages and in relevant applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is supported in part by the National Science Foundation under NSF SHF #1645136, #1749936, and #2006947.
