Query by Example in Large-Scale Code Repositories

Vipin Balachandran
VMware
Bangalore, India
vbala@vmware.com

***Abstract*—Searching code samples in a code repository is an important part of program comprehension. Most of the existing tools for code search support syntactic element search and regular expression pattern search. However, they are text-based and hence cannot handle queries which are syntactic patterns. The proposed solutions for querying syntactic patterns using specialized query languages present a steep learning curve for users. The querying would be more user-friendly if the syntactic pattern can be formulated in the underlying programming language (as a sample code snippet) instead of a specialized query language. In this paper, we propose a solution for the *query by example* problem using Abstract Syntax Tree (AST) structural similarity match. The query snippet is converted to an AST, then its subtrees are compared against AST subtrees of source files in the repository and the similarity values of matching subtrees are aggregated to arrive at a relevance score for each of the source files. To scale this approach to large code repositories, we use locality-sensitive hash functions and numerical vector approximation of trees. Our experimental evaluation involves running control queries against a real project. The results show that our algorithm can achieve high precision (0.73) and recall (0.81) and scale to large code repositories without compromising quality.**

I. Introduction

Searching code samples in a code repository is an important part of program comprehension. Developers often search code samples during refactoring efforts, fixing bugs, adding new features, or learning the usage of a particular API. The code search is typically performed using an IDE such as Eclipse [1] (for small local repositories) or using the search feature in online source code browsers (for large repositories). OpenGrok [2], Krugle [3], Open Hub [4], and Grepcode [5] are some of the popular code search engines. OpenGrok can be downloaded and customized for custom code repositories. The remaining ones are hosted on the Internet and search only in a fixed set of open source projects.

The search tools mentioned above support searching for language specific syntactic elements. For example, using OpenGrok, developers can search for class definitions using its symbol search. It is also possible to find all references to a symbol using these tools. Tools such as Eclipse also support pattern matching using regular expressions. These tools are basically text-based search engines with options for searching certain syntactic elements in a language. For example, in OpenGrok, a given source file is processed by language specific analyzers to create a Lucene document, which is finally indexed [6].

Most of the common search tasks can be performed using text-based code search engines. However, they are not suitable for queries which are syntactic patterns. Panchenko et al. observed that 36% of all queries in a code search engine are syntactic patterns [7]. As an example, consider the case where a developer is interested in understanding how `IOException` thrown by the `File.createTempFile` [8] API is handled in existing code in a Java code repository. This cannot be searched precisely using "`File.createTempFile`" in a text-based search engine because the enclosing method containing this method invocation may be declared as throwing `IOException`. In such a case, there will not be any `catch` block handling the exception. It is also not effective to search for the symbol "`IOException`" since it could be thrown by many APIs. The situation becomes worse if the method involved is an instance method such as `delete` in the `File` class, in which case the user query might include instance variables (e.g., `File f; f.delete();`). The method name might be a common identifier and the existing code may not use the same identifier for the method invocation as given in the query. Cases like these can be handled in a user-friendly manner if the query can be expressed as a code snippet in the surface programming language and the search engine returns source files containing syntactic structures similar to that of the query. For example, the search intent in the first case may be expressed using the following code snippet:

```
try {
   File file = File.createTempFile("foo", "bar");
} catch (IOException e) { }
```

Intuitively, a file can be considered relevant to a given query snippet if it contains syntactic patterns or structures similar to that of the query. A file can also be considered relevant if it contains code that is semantically related to the query snippet. In this paper, we do not consider semantic similarity and define the relevance score based on syntactic structural similarity between the files and the query snippet. Since ASTs capture the syntactic structure, we use AST similarity (tree edit distance; Section II-B) to compute the relevance score. There are two challenges:

1) It is often the case that only a small subtree of the AST of a source file (source AST) matches the AST of the query snippet (query AST). This implies that the similarity value obtained by directly applying a tree similarity algorithm may not yield desirable results. For example,

978-1-4673-7532-0/15 © 2015 IEEE ICSME 2015, Bremen, Germany

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

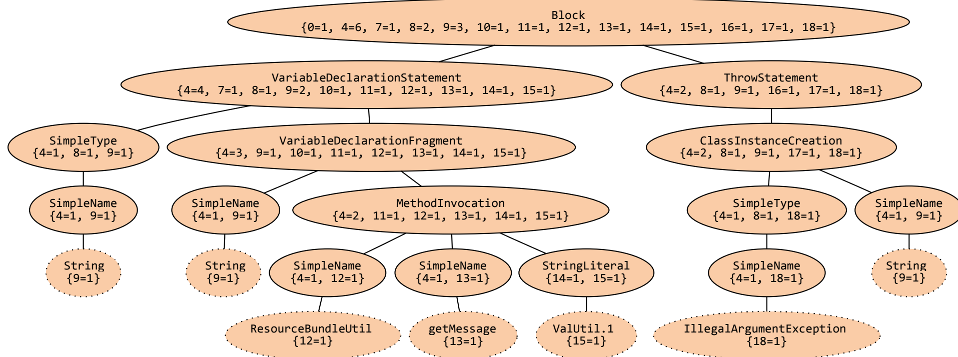


Fig. 1. AST subtree with *1-level* characteristic vectors

the tree edit distance might indicate a low similarity (large edit distance value) even though the query AST matches a small subtree of a very large source AST.

2) There may not exist a single subtree of the source AST that matches the entire query AST; multiple subtrees that may not share a common parent might match different subtrees of the query AST.

To address these challenges, we consider all the subtrees of the query AST with size above a certain threshold and compute similarity between these subtrees and all the subtrees of a source AST. Since the number of subtrees of a source AST can be prohibitively large, we consider only those subtrees of a source AST which has the same root node type as that of the query AST subtree against which it is compared. Finally, a relevance score is computed for each of the source files having one or more matching AST subtrees based on AST subtree similarity values. Since tree edit distance computation is expensive and the number of ASTs can be prohibitively large for large-scale code repositories, we adopt numerical vector approximation of ASTs and use Euclidean distance to approximate tree similarity. To scale the similarity computation to millions of vectors, we use two locality-sensitive hash functions to implement a space and time efficient k-nearest neighbor search to find vectors similar to a query vector.

The rest of this paper is organized as follows. In Section II, we discuss some of the background necessary to understand the problem and our solution. The *query by example* problem is formally defined along with an overview of our solution in Section III-A. The details of the algorithm are given in Section III-B followed by the discussion on scalability in Section III-C. In Section IV, we evaluate our algorithm. The related work is discussed in Section V and we conclude in Section VI.

II. Background

A. Abstract Syntax Tree

Abstract Syntax Tree (AST) is a hierarchical representation of the syntactic structure of source code. In comparison with parse tree, AST does not contain syntactic elements which are not required for program analysis in later stages of compilation. For example, an AST generated for a program in Java will not contain nodes for array subscripts or parentheses because they can be inferred from the hierarchical relationship between other nodes.

Figure 1 shows the AST generated for the `if-block` (lines 24-26) of the following Java code snippet using the Eclipse JDT Core [9] parser.

```
23 if (valRefs == null) {
24    String message = ResourceBundleUtil
25                     .getMessage("ValUtil.1");
26    throw new IllegalArgumentException(message);
27 }
```

The leaf nodes represented by dotted ellipses in Figure 1 do not correspond to leaf nodes in the original AST produced by the JDT core parser. In the original AST, these nodes represent node attribute values. For example, the child node of `StringLiteral` in Figure 1 represents the value of its attribute named `ESCAPED_VALUE` in the original AST. In this work, the original AST is modified to add such attribute values as leaf nodes because of the reasons outlined in Section III-A.

B. Tree Edit Distance and Approximation

The tree edit distance [10] is a commonly used tree similarity measure based on the tree edit operations: insert, delete, and relabel. In insert operation, a new node is inserted and in delete operation, an existing node is removed. The relabel or substitute operation replaces a node label with another label. Given a cost c_i associated with each of the edit operations, we can define the cost of an edit operation sequence $E = \langle e_1, e_2, \ldots, e_n \rangle$ as the sum of the cost of all edit operations in the sequence. If the set of all edit operation sequences which can transform a tree T_1 into another tree T_2 is S_E, then the tree edit distance between T_1 and T_2 ($\mathrm{ted(T_1, T_2)}$) is defined as the minimum of all edit operation sequence cost for each sequence in S_E, i.e.,

$$\mathrm{ted(T_1, T_2)} = \min\{Cost(E) : E \in S_E\} \qquad (1)$$

The tree edit distance algorithms are computationally expensive and therefore cannot be used for comparing larger trees such as ASTs. For example, the algorithm in [11] has a worst case time complexity of $O(|T_1|^2|T_2|^2)$, where $|T|$ denotes the tree size.

Yang et al.[12] approximated tree structural data with n-dimensional vectors (characteristic vectors) and proved that L^1-norm [13] between two vectors is a lower bound of the tree edit distance between the corresponding trees. Based on the following result, Jiang et al. [14] approximated tree edit distance with Euclidean distance of the corresponding characteristic vectors.

Corollary 3.8 in [14] "For any trees T_1 and T_2 with the edit distance $\mathrm{ted}(T_1, T_2) = k$, the Euclidean distance between the corresponding *q-level* characteristic vectors, $\mathrm{eud}(\tilde{v_q}(T_1), \tilde{v_q}(T_2))$, is no more than $(4q-3)k$ and no less than the square root of the L^1-norm", i.e.,

$$\sqrt{L^1\text{-norm}(\tilde{v_q}(T_1), \tilde{v_q}(T_2))} \leq \mathrm{eud}(\tilde{v_q}(T_1), \tilde{v_q}(T_2)) \leq (4q-3)k. \quad (2)$$

where a *q-level* characteristic vector is defined as

Definition 3.5, 3.6 in [14] "A *q-level* atomic pattern is a complete binary tree of height q. Given a label set L, including the empty label ϵ, there are at most $|L|^{2q-1}$ distinct *q-level* atomic patterns. Given a tree T, its *q-level* characteristic vector (denoted by $\vec{v_q}(T)$) is $\langle b_1, b_2, \ldots, b_{|L|^{2q-1}} \rangle$, where b_i is the number of occurrences of the i-th *q-level* atomic pattern in T."

If q=1, the Euclidean distance between *1-level* characteristic vectors corresponding to two trees can be used as a lower bound of the tree edit distance between them. Since a *1-level* atomic pattern is just a single node, the set of distinct *1-level* atomic patterns corresponding to a label set L is a set of $|L|$ nodes– one node for each label. Therefore, a *1-level* characteristic vector of a tree with node labels from L is a L-dimensional vector, where each dimension represents a label and its weight equal to the frequency of occurrence of that label in the tree. If we assume that two trees T_1 and T_2 are similar if their tree edit distance is less than a threshold η, then they are dissimilar if the Euclidean distance between their *1-level* characteristic vectors is greater than η. If the Euclidean distance is less than η, then it is probable that their tree distance is also less than η and hence can be considered similar.

Based on the above results, we can represent an AST or a subtree of an AST with a *1-level* characteristic vector and use Euclidean distance to approximate AST similarity. To generate the *1-level* characteristic vector of an AST subtree rooted at node n, we recursively generate the characteristic vectors of subtrees rooted at n's children, add them together and add 1 to the dimension represented by n's label. Figure 1 shows an AST subtree with *1-level* characteristic vectors, where each dimension with a non-zero weight is represented in the format $Dim_{id} = frequency$. For example, `String` is represented by dimension 9, and therefore, the characteristic vector of the root node (`Block`) has a value 3 for dimension 9.

III. QUERY BY EXAMPLE (QBE)

We formally define the QBE problem as follows:

Given a source code repository `R` *with* `n` *files and a query snippet* `q`*, return the top-k files in* `R` *matching the query* `q`*, ordered by relevance.*

In the next subsection, we give an overview of our solution using an example. The details of the solution are discussed in the subsection following that.

A. Overview

As an example of our query evaluation outlined in Section I, consider the source AST corresponding to the following Java code snippet in Figure 2(a).

```
29 try {
30     File f = File.createTempFile("bar", ".txt");
31     System.out.println(f.getAbsolutePath());
32     f.delete();
33 } catch (IOException e) {
34     logger.error(e.getMessage());
35     e.printStackTrace();
36 }
```

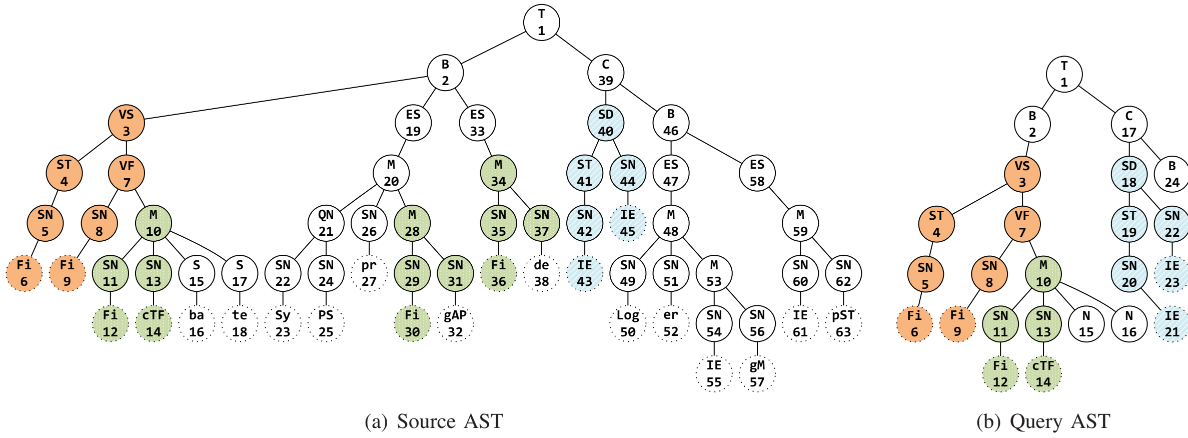
Let the query snippet be

```
1 try {
2     File file = File.createTempFile(null, null);
3 } catch (IOException e) {
4 }
```

with the corresponding query AST in Figure 2(b). Due to space constraints, the AST node labels are replaced by 1-3 letter abbreviations (Table I). The nodes are numbered starting from `1` in depth-first order.

The `try..catch` statement in lines 29-36 is represented by node `1` in Figure 2(a) with the left child, node `2`, representing the `try` block in lines 30-32 and the right child, node `3`, representing the `catch` clause in lines 33-35. The child nodes of node `2`: node `3`, node `19`, and node `33` represent the statements in lines 30, 31, and 32 respectively. The exception declaration in line 33 and the catch block in lines 34-35 are represented by nodes `40` and `46` respectively. In Figure 2(b), node `3` represents the statement in line 2 and node `18` represents the exception declaration in line 3.

These ASTs are modified versions of ASTs generated using the Eclipse JDT core parser (`ASTParser`). The advantage of the Eclipse JDT core parser is that it can generate valid ASTs even when there are unresolved dependencies. As mentioned in Section II-A, the leaf nodes drawn in dotted circles represent attribute values in the original AST. For example, the type names, identifier names, operators, and string literals are attributes in the original AST and are represented as leaf nodes in Figures 2(a) and 2(b) due to the following reasons. It is hard to imagine query snippets without type (class, interface, primitive type) names and method (function) invocations. If we ignore the above *attributes* during AST subtree similarity computation, then the query result will not be precise. If we retain these as attributes in the AST, then our structural matching needs to handle two types of nodes (those with and without attributes). By representing these information as leaf nodes, we can simplify the tree structural similarity



(a) Source AST

(b) Query AST

Fig. 2. Query evaluation

TABLE I
AST NODE LABEL ABBREVIATIONS USED IN FIGURE 2

T:	TryStatement	SN:	SimpleName	C:	CatchClause	VS:	VariableDeclarationStatement
B:	Block	ST:	SimpleType	M:	MethodInvocation	SD:	SingleVariableDeclaration
ES:	ExpressionStatement	QN:	QualifiedName	S:	StringLiteral	VF:	VariableDeclarationFragment
Fi:	File	cTF:	createTempFile	ba:	bar	te:	.txt
Sy:	System	PS:	PrintStream	pr:	println	gAP:	getAbsolutePath
de:	delete	IE:	IOException	Log:	Logger	er:	error
gM:	getMessage	pST:	printStackTrace	N:	NullLiteral		

computation without trading off query precision. We also replaced variable identifier labels with the corresponding type names. For example, we renamed the labels of node `9` and node `30` in Figure 2(a) from "`file`" (identifier) to "`File`" (type). This change will ensure that our query matching is insensitive to identifier names. It also means that it is not possible to search for specific identifiers as in text-based code search engines. For example, the query `File f; f.delete();` might return results containing `File g; g.delete();`.

As discussed in Section I, our query evaluation considers all the subtrees of the query AST with a minimum size (say 3). Let us assume that we have a tree similarity function which marks a subtree ST_s of a source AST as highly similar to a subtree ST_q of the query AST, if the exact matching tree patterns in both the trees contribute to 70% or more to their respective tree sizes. If the percentage of nodes covered by exact matching tree patterns in ST_s is p_s and ST_q is p_q, then ST_s is marked as highly similar to ST_q only if $p_s, p_q \geq 70\%$ and the similarity value is defined as $min(p_s, p_q)$. It should be noted that we use this similarity function instead of tree edit distance (similarity function used in our algorithm) only to simplify the discussion.

The similarity evaluation based on this approach is illustrated in Figure 2. The matching tree patterns in Figures 2(a) and 2(b) are drawn with similar color and fill-pattern. As we can see, the subtree rooted at node `10` in the query AST (Figure 2(b)) does not have a similar subtree in the source AST (Figure 2(a)). There is a tree pattern in the subtree rooted at node `10` in the source AST which matches 70% or more of the query AST subtree, but the matching tree pattern in the source AST subtree does not contribute 70% or more to its size. In the case of a matching subtree subsuming another matching subtree of the source AST, we do not discard the smaller matching subtree since it can have a high similarity value as compared to the larger subtree which could increase the relevance of the source file. For example, the subtree rooted at node `4` in the query AST is similar to the subtree rooted at node `4` in the source AST which is subsumed by another matching subtree in the source AST (node `3`). Finally, we aggregate the similarity values of all matching subtrees to compute the relevance score. There are total 5 matching source AST subtrees in this example: subtrees rooted at nodes `3`, `4`, `28`, `34`, and `40`.

Since a tree similarity function such as the one discussed above or tree edit distance is computationally expensive for large trees, we use characteristic vector approximation of ASTs and use Euclidean distance to approximate their tree edit distance based similarity (see Section II-B). If we consider two AST subtrees to be similar if their tree edit distance is no greater than a configurable threshold (η), then the problem of finding all source AST subtrees similar to a given query AST subtree reduces to finding all source vectors (corresponding to source AST subtrees) which are at a Euclidean distance of at most η from the query vector (corresponding to the query AST subtree). In the context of querying, we are interested only in k most similar characteristic vectors. Therefore, we can further reduce the problem to k-nearest neighbor search in a dataset of characteristic vectors (corresponding to source AST subtrees) based on Euclidean distance.

B. Details

1) Algorithm: The search database creation involves iterating over each of the source files in the repository, generating the AST, generating the characteristic vector of each of the AST subtrees with size no less than a configurable threshold (λ) and adding the vectors to k-nearest neighbor (k-NN) search module. The k-NN search module, when given a characteristic vector and an integer k as input, returns the top-k most similar (as per Euclidean distance) vectors. If exact nearest neighbors are required and if the repository is small, then the k-NN search module can be implemented using a hash table with the AST node type as the key and the value set to the list of characteristic vectors of AST subtrees whose root node type matches the key. The k-NN search in this case is a linear scan over a list of vectors. We call this method of k-NN search as *linear* k-NN search. In our implementation, we included a parameter η which is the max Euclidean distance allowed, above which a vector is not considered similar to the input vector and hence not included in the k-NN result.

The QUERY-BY-EXAMPLE algorithm is shown in Figure 3. In lines 1-2, the AST parser type is determined based on the source language of the query snippet and the query AST is generated. The GENERATE-VECTORS (line 3) generates characteristic vector of each of the subtrees and associates meta-data such as source file path, size of the subtree, and start and end line numbers in the source code covered by the corresponding subtree. The **for** loop in line 4 finds the k-nearest neighbors (with k set to configurable value K; line 6) of each of the query AST vectors with the corresponding subtree size greater than or equal to a configurable threshold (λ) and adds the vectors and the similarity values in the k-NN result to a hash table keyed by the source file path. We used the following equation to compute similarity from Euclidean distance:

$$sim(\vec{v_1}, \vec{v_2}) = \frac{1}{1 + eud(\vec{v_1}, \vec{v_2})} \tag{3}$$

The matching vectors and the similarity values are stored in the query match record M, which is retrieved in line 8. Since a vector (subtree in a source AST) matching a larger query subtree is more useful than a vector matching a smaller subtree, we store a weighted similarity value (line 9), where the weight is the size of the query subtree for which the vector is a match. The ADD-VECTOR method in line 10 adds the vector and its similarity value to the query match record. There could be multiple similarity values associated with the same vector, since the vector could be a match for many subtrees in the query AST. At the end of this step, we have a list of vectors (subtrees) of various source ASTs which are similar to various vectors of the query AST. Next, we will compute a relevance score for each of the source files using the similarity values of matching vectors.

The **for** loop in line 11 iterates over each of the source files which has vectors matching the query vectors and computes its relevance score. The **for** loop in line 13 iterates over each of the matching vectors and uses its similarity values to update the relevance score (line 15). The ADD-LINE-INTERVAL method in line 14 adds the line number interval (needed for reporting) covered by the matching vector to the query match record. Since there could be overlapping line number intervals, we remove redundant line intervals in line 17. Finally, the query match records are sorted based on relevance score and returned in line 18.

```
QUERY-BY-EXAMPLE(querySnippet)
 1  parser = RESOLVE-AST-PARSER(querySnippet)
 2  ast = parser.PARSE(querySnippet)
 3  V = GENERATE-VECTORS(ast)
 4  for each vector v ∈ V
 5      if v.treeSize ≥ λ
 6          knn = KNN-SEARCH(v, K, η)
 7          for each (vector, sim) ∈ knn
 8              M = HASH-GET(vector.sourceFile)
 9              weightedSim = sim × v.treeSize
10              ADD-VECTOR(M, vector, weightedSim)
11  for each entry (sourceFile, M) ∈ HASH-ENTRIES()
12      M.score = 0
13      for each vector v ∈ M.vectors
14          ADD-LINE-INTERVAL(M, v)
15          for each sim ∈ v.similarityValues
16              M.score += sim
17      REMOVE-REDUNDANT-LINE-INTERVALS(M)
18  return SORT-SCORES(HASH-ENTRIES())
```

Fig. 3. Algorithm - Query by Example

2) Time and Space Complexity: In this subsection, we analyze the time and space complexity of the QUERY-BY-EXAMPLE algorithm in Figure 3. For a repository with L lines of code, there are approximately $n = 10 \times L$ AST nodes [15]. In a k-ary tree with n nodes, the number of non-leaf nodes is $\mathcal{O}(n)$; therefore the number of *1-level* characteristic vectors in the search database is $\mathcal{O}(n)$. It is reasonable to assume that the query snippet is often small and the number of lines is a constant. This implies that the number of nodes in the query AST is a constant Q.

Lines 1-3 are constant time operations. Assuming that KNN-SEARCH is linear k-NN search, the time complexity of line 6 is $\mathcal{O}(n \times d)$, where $\mathcal{O}(d)$ is the cost of computing eud for d-dimensional vectors. The maximum number of matching vectors after k-NN search for all query vectors is $Q \times \mathrm{K}$. Therefore, the cost of the **for** loop in line 4 is $\mathcal{O}(Q \times (\mathrm{K} + (n \times d))$. In the worst case, each of these vectors corresponds to a unique source file and hence the **for** loop in line 11 takes $\mathcal{O}(Q \times \mathrm{K})$ time. Finally, SORT-SCORES in line 18 takes $O(Q \times \mathrm{K} \times \log(Q \times \mathrm{K}))$. Therefore, the total time complexity is

$$\begin{aligned}\mathcal{T} = \mathcal{O}(Q \times (\mathrm{K} + (n \times d)) + \mathcal{O}(Q \times \mathrm{K}) + \\ O(Q \times \mathrm{K} \times \log(Q \times \mathrm{K})) = \\ \mathcal{O}(n \times d) = \mathcal{O}(L \times d)\end{aligned} \tag{4}$$

For space complexity, we need to consider only the linear k-NN search method. Assuming a hash table implementation where the elements in a bucket are stored in a list, the space complexity is

$$\mathcal{S} = \mathcal{O}(n \times (\kappa + \rho)) \approx \mathcal{O}(n) \quad (5)$$

where κ is the size of a characteristic vector and ρ is the size of a memory address. Here we assume that the hash table stores only references to characteristic vectors.

C. Scaling to Large Code Repositories

The time complexity discussed in Section III-B2 implies that the QUERY-BY-EXAMPLE algorithm is useful only for small code repositories. Today's code repositories contain projects with *lines of code* (L in Equation 4) in millions. The main bottleneck in our algorithm is the expensive linear k-NN search. Locality-sensitive hashing (LSH) is a widely used technique to solve *approximate* k-NN search in high dimensions efficiently [16]. In this section, we discuss a replacement for linear k-NN search called *fingerprint* k-NN search based on LSH to scale our algorithm to large code repositories. First, we give an overview of LSH followed by the discussion on fingerprint k-NN search and its space and time complexity.

1) Locality-sensitive Hashing: LSH refers to a family of hash functions which map similar objects to the same bucket in a hash table with high probability. For approximate k-NN search, the source objects are mapped to various buckets using the hash functions. Given a query object, it is first mapped to a bucket using the same hash functions and then a linear search is carried out on all objects in that bucket using some similarity measure based on object type and application. Since there are different types of object similarity, not one LSH family will suffice for all cases. Therefore, we have LSH families for Euclidean distance based similarity and cosine similarity.

Given a distance metric d defined on a set of vectors V and distance thresholds d_1 and d_2, an LSH family F is a family of hash functions such that for each $f \in F$, the following conditions should hold:

1) if $d(\vec{x}, \vec{y}) \leq d_1$, then $probability(f(\vec{x}) = f(\vec{y})) \geq p_1$
2) if $d(\vec{x}, \vec{y}) \geq d_2$, then $probability(f(\vec{x}) = f(\vec{y})) \leq p_2$

where $d_1 < d_2$ and $\vec{x}, \vec{y} \in V$. Such a family is called (d_1, d_2, p_1, p_2)-*sensitive*.

As an example, let us define an LSH family for hamming distance between n-dimensional bit vectors. The hamming distance between two bit vectors x, y, denoted by $d(x, y)$ is the number of bit positions in which they differ. If the function $f_i(x)$ denotes the ith bit of vector x, then the family of functions $F = \{f_1, f_2, \ldots, f_n\}$ is an LSH family. Since the probability of $f_i(x) = f_i(y)$ for a random bit position i is $1 - d(x, y)/n$, the family F is $(d_1, d_2, 1 - d_1/n, 1 - d_2/n)$-*sensitive*, where d_1 and d_2 are hamming distances.

2) LSH Amplification: An LSH family can be amplified using *AND* or *OR* operations to obtain an ideal (p_1=1, p_2=0) LSH family. Here we discuss the *AND*-construction. Given a (d_1, d_2, p_1, p_2)-*sensitive* LSH family F with functions $\{f_1, f_2, \ldots, f_n\}$, the *AND*-construction on F results in a new family G with l functions, where each function $g \in G$ is constructed from k' random functions in F. If g is constructed from $\{f_1, f_2, \ldots, f_{k'}\}$, then

$$g(x) = g(y) \Leftrightarrow f_i(x) = f_i(y) \; \forall i = 1, 2, \ldots, k' \quad (6)$$

Since the k' functions to construct each of the functions in G are chosen randomly from F, the LSH family G is $(d_1, d_2, p_1^{k'}, p_2^{k'})$-*sensitive*. By choosing F and k' judiciously, we can reduce the collision probability of non-similar vectors to 0 while keeping the collision probability of similar vectors significantly away from 0 [17].

3) Fingerprint k-NN Search: Since the k-NN search is based on Euclidean distance, an LSH family for Euclidean distance appears to be a good replacement for linear k-NN search. However, it is not suitable for large datasets because of high memory consumption [18]. The cosine similarity between two vectors in the Euclidean space is the cosine of the angle between them. Since nearest neighbor query results based on cosine similarity are similar to those based on Euclidean distance for high-dimensional vectors in Euclidean space [19], we can use an LSH family for cosine similarity instead of an LSH family for Euclidean distance. Simhash [20] is one such technique with very less memory footprint and therefore an ideal candidate for our requirements.

Simhash maps a high-dimensional vector to a compact f-bit vector called fingerprint such that the fingerprints of two similar (as per cosine similarity) vectors are similar. Therefore, estimating vector similarity using cosine similarity reduces to finding the similarity between their fingerprints (bit vectors), for example, using hamming distance. Since there are a large number of fingerprints, we can use an LSH family for hamming distance to find the k-nearest fingerprints of a given query fingerprint. In the rest of this subsection, we discuss how to generate fingerprints of characteristic vectors, followed by the discussion on fingerprint k-NN search method.

To generate a fingerprint of f-bits using Simhash, we initialize an array F of size f with zeros. For each of the dimensions i in the given characteristic vector $\vec{v}$, the dimension is first hashed to an f-bit value h. Then, for each of the bit positions j in h, if h_j is 1, we add v_i to $F[j]$, else subtract v_i from $F[j]$, where v_i is the weight of dimension i in $\vec{v}$. Finally, the fingerprint of $\vec{v}$ is determined based on the sign of values in F– the jth bit in the fingerprint is set only if $F[j]$ is positive.

To find out the k-nearest fingerprints of a given query fingerprint, we use *AND*-construction on the LSH family for hamming distance (see Section III-C1). Therefore, we define l hash functions $G = \{g_1, g_2, \ldots, g_l\}$, where each function g_i returns a k'-bit hash value. For example, if $g_1(x) = (f_1(x), f_2(x), \ldots, f_{k'}(x)$, then the hash value of x given by g_1 is the concatenation of the bits of x at positions $1, 2, \ldots, k'$.

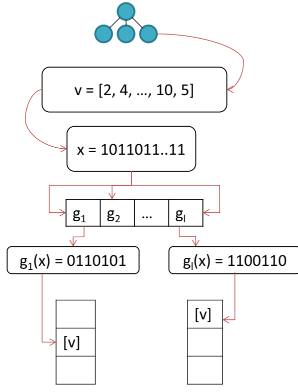


Fig. 4. Initialization of fingerprint k-NN search

During initialization of fingerprint k-NN search method, each of the vectors is inserted into l hash tables. The bucket in a hash table where a vector is inserted is determined by applying the corresponding hash function in G on the vector's fingerprint. This initialization is illustrated in Figure 4. In the figure, v denotes a characteristic vector and x its fingerprint. Given a query fingerprint q corresponding to a query vector, we retrieve the list of vectors in the bucket given by $g(q)$ for each $g \in G$. Finally, a linear scan is performed on the union of the lists of vectors to determine the top-k similar vectors which are at a Euclidean distance of at most η from the query vector.

4) Time and Space Complexity: The time complexity of Simhash is $\mathcal{O}(d \times f)$, where d is the dimension of the characteristic vector and f is the fingerprint size. In the hash table indexed by $g \in G$, there can be at most $\frac{n}{2^{k'}}$ vectors at the index $g(q)$ for a query fingerprint q, assuming an even distribution of vectors. Therefore, the total number of vectors for linear scan is $\frac{n}{2^{k'}} \times l$. Assuming a total cost of $\mathcal{O}(d)$ for each vector comparison and a sufficiently large k' such that n is a constant multiple of $2^{k'}$, the total time complexity is

$$\mathcal{T} = \mathcal{O}(d \times f) + \mathcal{O}(\frac{n}{2^{k'}} \times l \times d) \approx \mathcal{O}(d \times (f + l)) \quad (7)$$

The space complexity of $\mathcal{O}(n)$ characteristic vectors is:

$$\mathcal{S} = \mathcal{O}(n \times \kappa + l \times 2^{k'} \times \rho + l \times n \times \rho) \approx \mathcal{O}(n \times l) \quad (8)$$

where κ is the size of a characteristic vector and ρ is the size of a memory address. The first term is the space consumed by characteristic vectors. The second and third terms denote the space requirement of l hash tables– the space needed for the buckets and references to the vectors stored in it. In comparison with linear k-NN search, the memory requirement increased by l times. For a code repository with 100 million lines of code, the number of 1-level characteristic vectors is approximately 1 billion. Assuming κ=100 bytes, ρ=8 bytes, k'=24, and l=5, the total memory requirement is approximately 131 GB, which is quite reasonable considering the size of the repository and the memory capacity of high-end servers. A distributed implementation of our algorithm could further reduce the memory requirement of a single server.

TABLE II
CONTROL QUERIES

Label	Query
Q1	`LoopTimer timer; while (timer.hasNotExpired()) {}`
Q2	`VirtualDevice device; if (device instanceof VirtualController) {}`
Q3	`try {} catch(CryptoException e) { throw e; }`
Q4	`State state; if (state == State.error) {}`
Q5	`TaskSpec spec = new TaskSpecBuilder().build(); TaskManager mgr; mgr.createLocalTask(spec);`

TABLE III
BEST HITS AND GOOD HITS

Query	Best Hits	Good Hits
Q1	6	11
Q2	3	14
Q3	4	7
Q4	2	8
Q5	1	3

IV. EXPERIMENTAL EVALUATION

Code search tools such as web search engines can be evaluated using information retrieval metrics such as precision, recall, and f-measure. Precision is the ratio of number of valid results to total number of results retrieved. Analogously, recall is the ratio of number of valid results retrieved to total number of valid results. The f-measure combines precision and recall and is computed as the harmonic mean of precision and recall:

$$\text{f-measure} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \quad (9)$$

Since there is no benchmark dataset to compare code search engines, we do not attempt such comparisons. We run control queries on a code base and compute the precision, recall and f-measure of top 5, top 10, and top 20 results. Control queries, as mentioned in [21] have a reasonable number of results which can be analyzed manually and its usefulness can be judged easily. Our evaluation approach is comparable to that followed in [21] and [22]. We discuss the evaluation methodology in Section IV-A and the results in Section IV-B.

A. Methodology

Our experiments were carried out using 5 control queries. For evaluation, we selected an ongoing Java project in VMware cloud suite with 1448 source files and 174.76 KLoC (Kilo Lines of Code). To formulate the control queries, we conducted a survey involving 9 developers working in the

TABLE IV
RECALL (%) OF BEST AND GOOD HITS FOR LINEAR (L) AND FINGERPRINT (F) k-NN METHODS

Query	k-NN	Top 5		Top 10		Top 20	
		Best	Good	Best	Good	Best	Good
Q1	L	50.0	18.18	66.67	54.55	100.0	72.73
	F	50.0	18.18	50.0	36.36	100.0	63.64
Q2	L	66.67	14.29	66.67	42.86	100.0	64.29
	F	66.67	14.29	66.67	35.71	100.0	57.14
Q3	L	25.0	42.86	50.0	71.43	75.0	100.0
	F	25.0	28.57	50.0	57.14	50.0	85.71
Q4	L	50.0	25.0	100.0	62.5	100.0	75.0
	F	50.0	25.0	100.0	37.5	100.0	62.5
Q5	L	100.0	66.67	100.0	100.0	100.0	100.0
	F	100.0	66.67	100.0	100.0	100.0	100.0

chosen project who use OpenGrok for their development activities. In the survey, the participants were asked to provide sample code snippet queries they typically wish to run on their code base. If some of the queries had the same intent (for e.g., how to handle return value of a method?), but used different data types or methods, we arbitrarily selected one and treated all of them as the same query. Based on frequency, we selected top 5 responses as our control queries, which are listed in Table II.

After selecting the control queries, we asked one of the senior developers working in the project under study to identify the result set for each of the queries using Eclipse IDE's search feature. For example, to identify the result set for Q3, one could search for all references to the exception and then filter out irrelevant results. We also asked the developer to categorize the result set into two groups: best hits and good hits. The number of results in each of these groups are summarized in Table III.

To evaluate the precision and recall trade-off due to fingerprint k-NN search, we ran the control queries using two versions of our algorithm– one using linear k-NN search and the other using fingerprint k-NN search. For each k-NN search method and for each control query, we computed the recall of best hits and good hits in top 5, top 10, and top 20 results. We also computed the precision, recall, and f-measure of top 5, top 10, and top 20 results, assuming a result to be relevant if it is a member of either best hits or good hits.

B. Results

The results reported in this section are based on an implementation of the QUERY-BY-EXAMPLE algorithm with parameter values: K=50, η=1.25, and λ=3. For fingerprint k-NN search, we set f=64, l=20, and k'=24. In the case of Simhash, the hash values of dimensions were generated using MurmurHash [23]. Table IV lists the recall of best and good hits in top 5, top 10, and top 20 results. The precision, recall, and f-measure of the results are summarized in Table V.

For 4 out of 5 queries, at least 50% of best hits is retrieved in top 5 results for both linear and fingerprint methods. This shows the effectiveness of our ranking. For both linear and fingerprint methods, all best hits are retrieved in top 20 results for all queries except one. In the case of good hits, for linear method, at least 42% of hits is retrieved in top 10 and 64% in top 20 results. The corresponding figures for fingerprint method are 35% and 57%. We are not reporting the average of recall percentage values due to potential bias caused by smaller result sets. However, our control queries are diverse enough to validate the search effectiveness of our algorithm in a real setting.

Except for query Q5, the relevant result sets of all queries have 10 or more hits. Therefore, it is not meaningful to discuss the recall of top 5 results of queries Q1-Q4. In the case of Q5, 75% of hits is recalled in top 5 results. The average precision of top 5 results of all queries is 0.76 for linear method and 0.72 for fingerprint method, which shows the accuracy of our algorithm. It is natural for the precision to drop for queries with smaller number of hits as the result set becomes larger; for e.g., Q5. The recall of top 10 and top 20 results of Q5 is 1.0 for both linear and fingerprint methods. For top 10 results of Q1-Q4, the average precision is 0.8 and 0.63 respectively for linear and fingerprint methods. The average recall of top 10 results of Q1-Q4 is 0.6 and 0.47 respectively for linear and fingerprint methods. The corresponding average f-measure values are 0.68 and 0.53. The average precision, recall, and f-measure of top 20 results of Q1-Q4 for linear method are 0.73, 0.81, and 0.76. The corresponding values for fingerprint method are 0.62, 0.71, and 0.66. These results show that our algorithm has high precision and recall, which makes it suitable for practical use. Also, using fingerprint method, the decrease in precision and recall is only 15% and 12.3% respectively, which shows that we can achieve high scalability without compromising much on quality.

V. RELATED WORK

There are many tools that use specialized query languages to query ASTs. Crew proposed a Prolog-like query language (ASTLOG) [24] to search syntactical patterns in an AST. In [25], the authors used OCL (Object Constraint Language) to write queries which are evaluated against an object model representing an AST. PMD [26] is a static code analysis tool which allows users to write new defect patterns in XPath expressions which are evaluated against ASTs. .QL [27] is an object-oriented query language used for navigating ASTs and detecting bugs and coding standard violations. In [28], the authors used a variant of AWK to express AST patterns. The main disadvantage of above approaches is the steep learning curve involved in learning the query language and understanding ASTs to write meaningful queries.

There are many query languages for source code meant for different purposes. PQL [29] allows programmers to express queries that would detect design rule violations. The SCRUPLE [30] search engine uses a pattern language based on source programming language to query code exhibiting

TABLE V
PRECISION ($\mathcal{P}$), RECALL ($\mathcal{R}$), AND F-MEASURE ($\mathcal{F}$) OF TOP 5, TOP 10, AND TOP 20 RESULTS

Query	k-NN Method	Top 5			Top 10			Top 20		
		$\mathcal{P}$	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{F}$
Q1	**Linear**	1.0	0.29	0.45	1.0	0.59	0.74	0.74	0.82	0.78
	Fingerprint	1.0	0.29	0.45	0.7	0.41	0.52	0.68	0.76	0.72
Q2	**Linear**	0.8	0.24	0.36	0.8	0.47	0.6	0.8	0.71	0.75
	Fingerprint	0.8	0.24	0.36	0.7	0.41	0.52	0.73	0.65	0.69
Q3	**Linear**	0.8	0.36	0.5	0.7	0.64	0.67	0.63	0.91	0.74
	Fingerprint	0.6	0.27	0.38	0.6	0.55	0.57	0.47	0.73	0.57
Q4	**Linear**	0.6	0.3	0.4	0.7	0.7	0.7	0.73	0.8	0.76
	Fingerprint	0.6	0.3	0.4	0.5	0.5	0.5	0.58	0.7	0.64
Q5	**Linear**	0.6	0.75	0.67	0.4	1.0	0.57	0.21	1.0	0.35
	Fingerprint	0.6	0.75	0.67	0.4	1.0	0.57	0.2	1.0	0.33

certain features. CCEL [31] is a query language based on C++ to express constraints on C++ source code. JQuery [32] is another tool which uses a custom query language to browse source code. In [33], the authors extended Dependence Query Language (DQL) to support queries which contain topics describing functionality of target code and dependence relations between source code entities. Strathcona [34] is a tool which extracts structural information (class, parent class, interfaces it implements, type of fields, and method calls) of the code being edited by the programmer and recommends code with similar structure from an example repository. Code Conjurer [35] is an Eclipse plug-in which allows users to specify their search intent as an interface with function declarations in UML notation. In all these cases, either the query language is highly specialized or not suitable for general purpose code querying.

Like web search engines, there are many code search engines which support keyword-based search. Assieme [36], Codifier [37], and Sourcerer [21] extract syntactic information to enable various keyword-based code search options. JSearch [38] is a tool that indexes syntactic elements in the source code and allows users to retrieve code samples through keyword search. OpenGrok [2], Open Hub [4], Krugle [3], and Grepcode [5] are search engines for open source code which allow searching syntactic elements using keywords. There are keyword-based search engines which target specific software artifacts. For example, Exemplar [39] retrieves software projects relevant to a user query expressed in natural language. Portfolio [40] is a search tool that retrieves functions relevant to a user query. SnipMatch [41] is an Eclipse plug-in which leverages markups specified by snippet authors in code snippets to improve keyword-based snippet search.

There are code search techniques which use queries in surface programming language. In [7], the authors address the *query by example* problem by querying ASTs using XPath [42] expressions. The input code snippet is converted to an AST, which is then represented using an XPath expression. The XPath expression is evaluated against a database of ASTs to generate the query result. The main disadvantage is that this approach constraints the type of syntactic patterns which the user can input (almost 8 types in total). Also, their approach is highly sensitive to XPath query formulation, which in turn depends on the syntactic correctness of the input code snippet. When they tried to generate XPath expression for two or more statements, the expression became complex and the evaluation was unsuccessful. CodeGenie [43] is a search engine which allows developers to input test cases. The code repository is searched using Sourcerer (keyword search) based on the information in the tests. It also verifies the suitability of results by integrating it into a developer's current project and running the input tests.

The *query by example* problem can be considered as a special case of code clone detection. However, code clone detection methods cannot be applied directly due to the challenges mentioned in Section I. In the rest of this section, we discuss code clone detection methods which share techniques similar to ours. In [44], the authors proposed a method for detecting similar Java classes by comparing FAMIX (a programming language independent model for representing object oriented source code) tree representation of classes using tree similarity algorithms. Deckard [14] is a clone detection tool which uses characteristic vector approximation of ASTs and Euclidean LSH to create clusters of similar vectors. In [45], an AST subtree is approximated using a fingerprint which is a tuple of tree size and a hash reflecting its structure. The clones are then detected by comparing fingerprints.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an algorithm based on AST structural similarity to solve the *query by example* problem in source code repositories. We also proposed an approach based on numerical vector approximation of trees and locality-sensitive hash functions to scale the algorithm to code repositories with several million lines of code. Using a set of control queries, we demonstrated the effectiveness of our algorithm in real settings. As part of future work, we plan to conduct experiments with a large code repository to estimate the query response time and CPU and memory usage of our algorithm with varying query rates.

REFERENCES

[1] "Eclipse," http://www.eclipse.org/.
[2] "OpenGrok," https://opengrok.github.com/OpenGrok/.
[3] "Krugle," http://opensearch.krugle.org/.
[4] "Open Hub," http://code.openhub.net/.
[5] "Grepcode," http://grepcode.com/.
[6] "OpenGrok Internals," https://goo.gl/jRpcgj.
[7] O. Panchenko, J. Karstens, H. Plattner, and A. Zeier, "Precise and scalable querying of syntactical source code patterns using sample code snippets and a database," in *Proceedings of the 19th international conference on Program Comprehension*, ser. ICPC '11, June 2011, pp. 41 –50.
[8] "Java Platform, Standard Edition 7 API Specification," http://docs.oracle.com/javase/7/docs/api/java/io/File.html.
[9] "JDT Core Component," http://www.eclipse.org/jdt/core/index.php.
[10] P. Bille, "A survey on tree edit distance and related problems," *Theoretical Computer Science*, vol. 337, pp. 217–239, 2005.
[11] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989.
[12] R. Yang, P. Kalnis, and A. K. H. Tung, "Similarity evaluation on tree-structured data," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 754–765.
[13] "Manhattan Distance," http://goo.gl/CId9do.
[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105.
[15] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998.
[16] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, ser. STOC '98. New York, NY, USA: ACM, 1998, pp. 604–613.
[17] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2014.
[18] W. Zhang, K. Gao, Y.-d. Zhang, and J.-t. Li, "Data-oriented locality sensitive hashing," in *Proceedings of the International Conference on Multimedia*, ser. MM '10. New York, NY, USA: ACM, 2010, pp. 1131–1134.
[19] G. Qian, S. Sural, Y. Gu, and S. Pramanik, "Similarity between euclidean and cosine angle distance for nearest neighbor queries," in *Proceedings of 2004 ACM Symposium on Applied Computing*. ACM Press, 2004, pp. 1232–1237.
[20] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, ser. STOC '02. New York, NY, USA: ACM, 2002, pp. 380–388.
[21] S. Bajracharya, T. Ngo, E. Linstead, P. Rigor, Y. Dou, P. Baldi, and C. Lopes, "Sourcerer: A search engine for open source code supporting structure-based search," in *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '06, 2006, pp. 25–26.
[22] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An information retrieval approach to concept location in source code," in *Proceedings of the 11th Working Conference on Reverse Engineering*, ser. WCRE '04, Nov 2004, pp. 214–223.
[23] "MurmurHash," https://sites.google.com/site/murmurhash/.
[24] R. F. Crew, "ASTLOG: A language for examining abstract syntax trees," in *Proceedings of the Conference on Domain-Specific Languages*, ser. DSL '97. Berkeley, CA, USA: USENIX Association, 1997.
[25] G. Antoniol, M. Di Penta, and E. Merlo, "YAAB (Yet Another AST Browser): Using OCL to navigate ASTs," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003.
[26] "PMD," http://pmd.sourceforge.net/.
[27] O. de Moor, M. Verbaere, and E. Hajiyev, "Keynote address: .ql for source code analysis," in *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, Sept 2007, pp. 3–16.
[28] W. G. Griswold, D. Atkinson, and C. McCurdy, "Fast, flexible syntactic pattern matching and processing," in *Proceedings of the Fourth Workshop on Program Comprehension*, Mar 1996, pp. 144–153.
[29] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: A program query language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 365–383.
[30] S. Paul and A. Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, Jun. 1994.
[31] C. K. Duby, S. Meyers, and S. P. Reiss, *CCEL: A Metalanguage for C++*. Department of Computer Science, Univ., 1992.
[32] E. McCormick and K. De Volder, "JQuery: Finding your way through tangled code," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 9–10.
[33] S. Wang, D. Lo, and L. Jiang, "Code search via topic-enriched dependence graph matching," in *Proceedings of the 18th Working Conference on Reverse Engineering*, ser. WCRE '11. IEEE, 2011, pp. 119–123.
[34] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 117–125.
[35] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008.
[36] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: Finding and leveraging implicit references in a web search interface for programmers," in *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 2007, pp. 13–22.
[37] A. Begel, "Codifier: A programmer-centric search user interface," in *Proceedings of the workshop on human-computer interaction and information retrieval*, 2007, pp. 23–24.
[38] R. Sindhgatta, "Using an information retrieval system to retrieve source code samples," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 905–908.
[39] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd International Conference on Software Engineering*, ser. ICSE '10. IEEE, 2010, pp. 475–484.
[40] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. IEEE, 2011, pp. 111–120.
[41] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal, "Snipmatch: Using source code context to enhance snippet retrieval and parameterization," in *Proceedings of the 25th annual ACM symposium on user interface software and technology*. ACM, 2012, pp. 219–228.
[42] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, "XQuery 1.0 and XPath 2.0 formal semantics," *W3C recommendation*, vol. 23, 2007.
[43] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes, "Codegenie: Using test-cases to search and reuse source code," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 525–526.
[44] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting similar Java classes using tree algorithms," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 65–71.
[45] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, May 2009, pp. 243–247.