CodeKōan: A Source Code Pattern Search Engine Extracting Crowd Knowledge

Christof Schramm
Institute for Informatics, LMU
Munich, Germany
c.schramm@csconsultingsoftware.com

Yingding Wang
Institute for Informatics, LMU
Munich, Germany
yingding.wang@ifi.lmu.de

François Bry
Institute for Informatics, LMU
Munich, Germany
bry@lmu.de

ABSTRACT

Source code search is frequently needed and important in software development. Keyword search for source code is a widely used but a limited approach. This paper presents CodeKōan, a scalable engine for searching millions of online code examples written by the worldwide programmers' community which uses data parallel processing to achieve horizontal scalability. The search engine relies on a token-based, programming language independent algorithm and, as a proof-of-concept, indexes all code examples from Stack Overflow for two programming languages: Java and Python. This paper demonstrates the benefits of extracting crowd knowledge from Stack Overflow by analyzing well-known open source repositories such as OpenNLP and Elasticsearch: Up to one third of the source code in the examined repositories reuses code patterns from Stack Overflow. It also shows that the proposed approach recognizes similar source code and is resilient to modifications such as insertion, deletion and swapping of statements. Furthermore, evidence is given that the proposed approach returns very few false positives among the search results.

KEYWORDS

mining crowd knowledge, source code search, code patterns, source code similarity, Stack Overflow, search algorithm

ACM Reference Format:
Christof Schramm, Yingding Wang, and François Bry. 2018. CodeKōan: A Source Code Pattern Search Engine Extracting Crowd Knowledge. In *Proceedings of the 5th International Workshop on Crowd Sourcing in Software Engineering (CSI-SE 2018).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

An overarching theme in programming and software engineering has always been source code reuse [10]. Widely available libraries are major accomplishments for software engineering because they make complex functionalities available to programmers. Reuse of source code in libraries has been an enabling factor for the ongoing growth of the software industry [25].

CSI-SE 2018, May 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A study by Brandt et al. [7] has shown that programmers, when faced with novel implementation tasks, often look up online documentation containing code examples. This work aims to provide a novel approach to empirically, automatically and efficiently detect reuse of short source code fragments (5-50 lines of code) using the crowd knowledge contained in online code examples.

Online code examples commonly are not specific solutions to unique problems, but general solutions to frequently occurring small scale implementation tasks. These sets of small scale solutions are called *source code patterns*.

While library usage can easily be automatically detected by inspecting build files, detecting code pattern reuse is much more complicated. When reused, code patterns are commonly adapted to the relevant context for providing custom solutions. This makes detecting code pattern reuse a challenging task because the fuzziness introduced by pattern adaptation has to be overcome.

Studying code pattern reuse can help in creating new libraries and tools to give programmers feedback. Such feedback could link programmers to relevant online documentation for code examples that are similar to their source code. Furthermore, IDE plugins based on CodeKōan can alert programmers when redundant code is introduced to a project. This article presents CodeKōan, a code pattern search engine exploiting the crowd knowledge of Stack Overflow accessible at https://codekoan.org.

The main contributions of this article are:

- An original programming language independent method for code pattern recognition based on code patterns extracted from Stack Overflow.
- A proof-of-concept deployment of the method for two languages, Java and Python, tapping in the crowd knowledge of Stack Overflow.
- A quantitative analysis of the proof-of-concept application relying on well-known software repositories pointing to the effectiveness of the presented search engine.

2 RELATED WORK

2.1 Source Code Search Engines

The basic approaches for source code search can be differentiated by the type of query submitted by the user. Searching source code for keywords is used by many publicly available source code hosting platforms. This approach is partially helpful due to an often large number of search results that may not be related to what exactly the users have been searching for [1]. Searching source code for specifications is another approach presented in [15, 27]. Searching source code for predefined pattern schemata is a further approach to search for the occurrence of formally predefined query patterns in indexed source code [19]. The approach proposed in this article searches query source code for occurrences of arbitrary patterns. It requires no abstract specification of queries and source code can be searched for directly without any necessity for modifications. Thus, it differs significantly from the aforementioned approaches.

2.2 Code Clone Detection and Plagiarism

Code clone detection is the task of locating shared pieces of similar source code among software systems [24]. While the research in code clone detection focuses on finding copied or redundant code within or among source code projects, this work focuses on searching web resources for source code reuse.

In previous research on code clone detection multiply algorithms have been developed, which can be roughly categorized into the following four methods for comparing source code [22]: (1) Raw string comparison, i.e., analysis of the raw string representation of source code including comments, formatting [2]. (2) Source code comparison on a token string basis. Such algorithms [14, 20, 24] have the advantage of recognizing similarities despite differing layout, identifier names and comments. An advantage of these approaches is that they tend to be very performant. (3) Abstract syntax tree (AST) comparison [4, 12]. It is considered a severe disadvantage of such algorithms, that ASTs are very fixed on source code order and structure. (4) Program dependence graph analysis. Program dependence graphs are a representation of source code that expresses the dependence of some parts of source code on other parts [9]. Such a graph could contain statements and assignments as nodes with edges linking every use of a variable to its declaration. These graphs contain more semantic information than the three previously mentioned source code representations (raw string, token string and AST). Comparing these graphs can ultimately be reduced to solving the subgraph-isomorphism problem which has been shown to be NP-complete [21], meaning that it is very hard to scale this approach well. Several authors [8, 16] have presented approaches for code clone detection based on dependence graphs that attempt to manage the NP-completeness of the underlying problem.

2.3 Previous Work Using Stack Overflow Data

Stack Overflow[1] is a popular online question and answer (Q&A) site for programming problems. User content on Stack Overflow is generated in Q&A threads that are started by a user submitting a question, which other users then answer. An important feature of Stack Overflow is its rating system, by which users can vote on post's relevance. This rating contributes to post's visibility. Stack Overflow has been shown to provide high quality documentation for a great variety of APIs, topics and programming languages [18]. Furthermore, a rich body of previous research uses Stack Overflow data [3, 6, 17, 29]. Moreover, user-generated content on Stack Overflow is licensed under a Creative Commons license, permitting its use in research projects. Stack Overflow publishes a quarterly dump of all posts on its website. Publications about Stack Overflow data include studies of post quality [17], topics that are talked about [3], and user behavior and ratings [6]. Vassallo et al. [29] published the CODES system, an eclipse plugins, that mines Stack Overflow data to generate comments and documentation from Stack Overflow posts for Java classes. The CODES system doesn't generate documentation for arbitrary code, but rather for already existing open source projects that are discussed on Stack Overflow. CODES uses the keyword based search mechanism provided by Stack Overflow and does not perform specialized source code search.

[1]The Stack Overflow URL is: http://stackoverflow.com

3 SOURCE CODE SIMILARITY SEARCH

3.1 Source Code Patterns

The goal of the presented approach is to find reuse of short source code fragments in source code files, that are submitted as user queries. The code fragments that are the primary focus of the presented search engine design are between five and fifty lines of code in length.

An example of a code fragment that would be relevant for the CodeKōan search engine is given in the following example[2], which reads a file and prints it to the standard output stream line by line:

```
Charset charset = Charset.forName("US-ASCII");
// try-with-resources statement
try (BufferedReader reader =
       Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line); }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x); }
```

There are numerous online code examples that explain the same thing with slight variations: some lack the explicit `ASCII` character set definition, others work without the try-with-resources statement (introduced in Java 7). But all of them fulfill a very similar task using similar source code. These code examples are considered to be instances of source code patterns which solve concrete implementation problems like "read files line wise using `BufferedReader`".

3.2 Source Code Reuse

The queries that users submit to the CodeKōan search engine are single source code files. CodeKōan's purpose is to find all reuses of indexed source code fragments in the query file and the regions of the query that make up this reuse. It should be noted that there are two types of source code reuse, that occur for different reasons [28]. The first type of source code reuse occurs as a result of adaption of reference examples from source code documentation. Previous work [7] strongly suggests, that this is common behavior. The second type of source code reuse takes place if functionality is implemented in a way that follows a code pattern without deliberate adaptation of foreign code. In both cases, CodeKōan can automatically link any code fragment reuse in a codebase to relevant documentation, in which the code fragment occurs. This documentation might point out edge cases or weaknesses of the implemented solution and may even contain different, more correct solutions. An advantage that CodeKōan gives programmers is, that it can link source code to

[2]Adapted from the official Java documentation at https://docs.oracle.com/javase/tutorial/essential/io/file.html

existing documentation and give a better overview of the domain of existing solutions.

3.3 Source Code Similarity

This work aims to find regions in source code that are *similar* to indexed code fragments. Walenstein et al. [30] point out, that the concept of similarity is inherently vague, because any notion of similarity is dependent on a set of criteria upon which a comparison is based. In this section the notion of source code similarity, which is captured by the proposed search algorithm is introduced. There are many other definitions of source code similarity, some of which are discussed in section 2.

Broadly speaking, source code similarity can be viewed on a spectrum of two dimensions: syntactic similarity and semantic similarity. A purely syntactic view of source code similarity would only consider the syntax of programs without attempting to capture their meaning, while a semantic view of source code similarity would only consider the meaning of source code without considering syntax at all.

Two programs are syntactically equivalent if their source code is identical and semantically equivalent if they do exactly the same thing, regardless of syntax. From a practical implementation standpoint detection of syntactic and semantic equality would not be very helpful to the proposed search engine. Syntactic equality of source code is trivial to detect but too susceptible to minor changes in aspects such as source code layout to be useful. The ability to recognize semantic equality of source code on the other hand would be enormously useful, but it is (generally) not detectable. The problem of semantic program equality can ultimately be reduced to deciding whether two Turing machines yield identical output for identical input, which is undecidable.

The notion of similarity which is used in the CodeKōan algorithm is a middle ground between these two extremes. The search engine initially generates a set of search results with a high false positive rate, which is then refined by filtering out false positives using heuristics for semantic similarity.

CodeKōan's similarity detection is based on four characteristics. The first characteristic is syntactic similarity on a token string level. Comparing token strings in the first step makes the CodeKōan search algorithm independent of whitespace and comment, which are both not relevant to the function that source code performs. The second similarity characteristic is sufficient coverage of the code fragment by regions in the query document. A code pattern can only be reused if it is mostly present in the query code. Therefore, query regions are only considered similar to patterns if they match with a majority of that pattern. A third characteristic is structural similarity of matched code in the query document and the compared code fragment. The structure to be compared is the block structure of source code, which controls e.g. variable scope. In programming languages with C-like syntax for example, blocks are delimited by curly braces. The fourth characteristic is similarity of words in identifiers. Two pieces of source code are considered similar if they are both dealing with a similar topic. Topical similarity is measured on the words that occur as parts of identifiers among compared source code pieces.

4 AN ALGORITHM FOR THE SEARCH OF CODE PATTERN REUSE

4.1 Indexing Code Fragments

A central part of the proposed algorithm compares the token strings of indexed code fragments with that of query source code. To efficiently search sets of millions of code fragments a generalized suffix trees (GSTs) of the token strings of all code fragments is used. GSTs are data structures that generalize the concept of suffix trees, which expose a single string's structure to sets of strings.

For any given alphabet Σ and set of strings $\mathcal{T}$, a GST is a tree with labeled edges, in which any path from the root to a leaf node spells out a suffix of some $s \in \mathcal{T}$. Furthermore no two edges starting from a node to its children must start with the same letter $c \in \Sigma$. GSTs can be used to determine if a string P is a substring of any of the strings in an arbitrarily large set $\mathcal{T}$ of strings in $O(|P|)$. Constructing a GST is possible in $O(\Sigma_{t \in \mathcal{T}} |t|)$ [11]. The GST that is used in the following is built from the set $\mathcal{A} = \{A^k | k \in \mathbb{N}\}$, which contains the token strings A^k of all indexed code fragments. The used alphabet Σ is a language dependent set of tokens.

The CodeKōan search engine uses one index per programming language. Such an index consists of two parts: the already mentioned GST of token strings A^k from all code fragments for a single programming language and a Bloom-filter of token 10-grams. Bloom-filters are hashing-based data structures which allow queries of set-membership in $O(1)$ [5]. The Bloom-filter is used to identify in a constant amount of time whether a sequence of ten tokens is a member of any indexed code fragment. Bloom-filters have a certain small, well controllable false positive rate but no false negatives.

4.2 A Pipeline Algorithm for Similarity Search

CodeKōan's search algorithm functions as a pipeline, in which an initial step generates a very large set of partial search results. Subsequent parts of the pipeline filter and group these initial search results. The set of recognized code fragments after every pipeline step is a subset of the respective pipeline step's input.

The idea behind the pipeline is to first match token-substrings of code fragments, with identical token-substrings of the query file. These matched token-substrings are then grouped by their origin code fragments, the resulting groups of token-substrings are search results. These search results have very high false positive rates. Pipeline steps 2-4 aim to decrease this false positive rate.

Step 1: Alignment. The initial step of the CodeKōan search algorithm begins with obtaining a token-string D from the query file using a programming language specific lexer.

Definition 4.1. Let S be a string. Then $S_{a:b}, a \leq b$ is a substring of S starting at position a with length $b - a$. $S_{a:}$ is a suffix of S starting at position a.

First all suffixes $D_{i:}$ starting with a 10-gram that is a member of the index Bloom-filter are identified. These suffixes are relevant because it is known from the Bloom-filter, that there is very likely a substring of some indexed code fragment, which occurs in D and is at least 10 tokens long. Then the index GST is used to identify for every relevant suffix $D_{i:}$ all substrings $A^k_{j:j+n}$ of indexed code fragments A^k, which are identical to prefixes $D_{i:i+n}$ of the respective relevant suffix.

Definition 4.2. The resulting set of substring pairs $(D_{i:i+n}, A^k_{j:j+n})$ is referred to as the set of *alignment matches*. The first part of an alignment match is called its "document side", and the second part is called its "pattern side".

Definition 4.3. A *search result* is a set of alignment matches, in which all pattern sides are token-substrings of the same code fragment A^k.

The alignment matches are grouped into search results so that there is exactly one search result for every code fragment A^k, which has a substring occurring in at least one alignment match.

This step introduces a first parameter to the algorithm, which is given by the minimum alignment match length n. Alignment matches that are shorter than n tokens are discarded. This parameter n has to be larger than the n-gram size 10, which is used during the Bloom-filter construction.

Step 2: Coverage Filtering. The first step of the algorithm identified all alignment matches $(D_{i:i+n}, A^k_{j:j+n})$ of the query D and code fragments A^k. The second similarity criterion outlined in section 3.3 is that code pattern reuse only takes place if a substantial part of a pattern is reused. All sets of token-substrings for code fragments A^k that do not fulfill this requirement are filtered out. Consider for example the code fragment given in section 3.1. If some print-statement in a Java query file were matched to the print statement in the fragment without actually reading a file, this would not constitute pattern reuse. Let R^k be a search result for the code fragment A^k. Then $|A^k|$ is the number of tokens in A^k. The *matched*(R^k) is defined as the number of tokens, that are matched by R^k as follows:

$$matched(R^k) = |\{i \mid i \geq 0 \land i < |A^k| \land \exists((D_{a:a+x}, A^k_{b:b+x}) \in R^k) : i \geq b \land i < b + x\}| \quad (1)$$

Definition 4.4. The *coverage* of a search result R^k is the number of tokens in the code fragment A^k which is included in at least one alignment match in R^k. Formally the coverage of R^k is $\frac{matched(R^k)}{|A^k|}$

Algorithm Parameter: Coverage Threshold. All search results that have a coverage below a certain threshold value are removed. This threshold value is a parameter to the search algorithm.

Step 3: Block Structure Filter. Search results generated by algorithm step 2 with more than one alignment match are filtered in this step of the search algorithm. Search results with only one alignment match pass this step unchanged. This algorithm step relies on a notion of pairwise alignment match conflicts, which will be defined in the following. This algorithm step splits search results, which are sets of alignment matches (see definition 4.2) into maximal subsets, so that no pair of alignment matches in the resulting search results are conflicting.

In order to define when two alignment matches are in conflict, function $f_{block} : \Sigma^* \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ between two indices a and b in a token string $T \in \Sigma^*$ is defined. The intuition for f_{block} is to describe how two points in source code relate to each other with respect to the block structure of source code. That block structure may e.g. be defined by indentation in Python or curly brackets in Java.

Independent of programming language `token_indent` refers to a token opening a block and `token_unindent` refers to a token closing a block. A full pseudo code definition of f_{block} is given in algorithm 1. Intuitively, the tuple $(up, down)$ that is returned by f_{block} describes the number of blocks that have to be exited and re-entered to go from index a to index b in the token string T. If a and b are in the same block, for example, $(up, down)$ would be (0,0). If b is in a child block of a's block, the result of f_{block} is $(0, 1)$. Using f_{block}, a definition for alignment match conflicts is developed:

Definition 4.5. Let $(D_{a:a+l}, A^k_{x:x+l})$ and $(D_{b:b+m}, A^k_{y:y+m})$ be two alignment matches in the same search result. These two alignment matches are in conflict iff: **(1.)** if the two alignment matches' query-side or pattern-side token substring ranges overlap, or **(2.)** if $f_{block}(D, a, b) \neq f_{block}(A^k, x, y)$

```
Data: TokenString $T$, indices $a$ and $b$
if $a > b$ then
    return $f_{block}(T, b, a)$
else
    (up, down) = (0,0);
    i = a;
    while $i$ != $b$ do
        i = i + 1;
        t = $T$[i];
        if $t$ == token_indent then
            up = up + 1;
        else if $t$ == token_unindent then
            if $up > 0$ then
                up = up - 1;
            else
                down = down + 1;
            end
        end
    return (up,down)
end
```

Algorithm 1: Block relationship function f_{block} (pseudo code)

Generating Conflict Free Subsets. For every search result R^k returned by algorithm step 2, a graph $G(R^k)$ is built, in which the alignment matches in R^k are vertices. Two alignment matches are connected by an (undirected) edge, if they do not conflict.

A clique in a graph G is a part G' of G, in which there exists an edge from any vertex to every other vertex [13]. All maximal cliques in the constructed graph $G(R^k)$ of alignment matches in search result R^k are identified. These maximal cliques are the new search results that will be further processed in the pipeline. In order to uphold the coverage-invariant of algorithm step 2 the coverage filter is re-applied to the newly generated search results. These results are then passed to step 4. The set of search results after pipeline step 2 contains only one search result for any code fragment A^k. After step 3 this condition no longer holds. There can be multiple search results for any code fragment A^k.

Step 4: Identifier Similarity Matching. One of the key criteria for source code similarity, as outlined in section 3.3, is the information containing in the identifiers in source code. The information in these identifiers conveys the programmer's intent behind a piece of source code. This step aims to be permissive; the intent is not to retain only "correct" search results, but to exclude ones which are very likely false positives.

The first step for processing a search result R^k is to identify the two sets $I_{query}, I_{pattern}$ of all identifiers that are covered by the query sides and pattern sides, respectively of all alignment matches in the search result R^k. Most programming languages use either camel case (e.g. `camelCaseIdentifier`) or snake case (e.g. `snake_case_identifier`). In both of these naming conventions, splitting multi-word identifiers into sets of words is possible.

Out of the identifier sets $I_{query}, I_{pattern}$, two bags of words are generated, which contain the words in the identifiers of I_{query} and $I_{pattern}$. These two bags of words are compared using a TF-IDF cosine similarity measure [26], which yields a value in the range [0,1]. A value of 1 indicates maximal similarity while a value of 0 indicates maximal differences. This step works by excluding all search results, which have a similarity value as described above, that is below a certain given threshold value. This threshold value is a parameter to the algorithm pipeline.

5 IMPLEMENTATION

In this section, some aspects of the search engine's implementation from an engineering standpoint are outlined. The proposed search engine is a concurrent, distributed system, which can be horizontally scaled across a cluster of machines. Data parallel processing is used to meet the scalability demands for the implemented system. This is made possible by the fact, that the index data structure (GST and Bloom-filter) can be split and distributed among a cluster of machines.

It is not necessary to build a single GST for the set of indexed code fragments $\mathcal{A} = \{A^k | 0 \leq k < n\}$, where n is the number of indexed code fragments. $\mathcal{A}$ can be split up into a set of N subsets $\mathcal{A}_0, ..., \mathcal{A}_{N-1}$, for each of which a separate index can be created. The index traversal for a single programming language can thus be distributed over N worker machines. Every query to CodeKōan is independently processed through all four pipeline steps by each of these N worker machines containing a partial index. The final result is the union of these N result sets.

RabbitMQ. The RabbitMQ message broker is used to connect the parts of CodeKōan's distributed cluster. RabbitMQ is highly performant and persists messages in case of worker node failure, thereby minimizing the potential for data loss. RabbitMQ distributes a copy of each query to a task-queue for each of the N worker machines in a cluster. A dedicated process collects all partial results for a query until a full result can be built, which is then returned to the user.

Indexing. The indices for the CodeKōan search engine consist of the code examples in all Stack Overflow answers for the two implemented programming languages Java and Python. Stack Overflow offers a quarterly XML dump of all user submitted content, which is licensed under a Creative Commons license. This dump is used to index all Stack Overflow posts into a PostgreSQL database.

Repository	Search Setting	False Positives / 1000 LOC
Elasticsearch	Low	146.14
	Medium	0.46
	High	0.0
DuckDuckGo (Android App)	Low	76.04
	Medium	0.273
	High	0.0

Table 1: Numbers of search results for various search settings when querying all source code files in the specified repositories using different search settings. All identified search results are false positives by construction, as the used index contains only code patterns that very specifically query PostgreSQL. None of the listed repositories use a SQL database.

The content of Stack Overflow posts is available as HTML, from which source code examples can be parsed by reading `<pre><code>` blocks. Code examples that are shorter than ten tokens are not indexed. It is assumed, that a code example in a Stack Overflow answer is written in a given language if the answer is in reply to a question, which has the corresponding language tag (e.g. "Java").

6 EVALUATION

6.1 Quantifying the Occurrence of False Positives

A search result of the algorithm (as defined in definition 4.3) is a false positive if the code fragment A^k which was reported as reused is not actually reused in the submitted query source code. To analyze this, a specialized index of code fragments that deal with creating and submitting queries for the PostgreSQL database is created. These code examples are a subset of all code examples on Stack Overflow and are manually selected.

Open source repositories are chosen, in which it is certain, that no SQL database access is performed. Selecting such repositories is straightforward. For example a NoSQL database like Elasticsearch doesn't contain source code for accessing a PostgreSQL database. Neither does the DuckDuckGo Android app.

Every source code file of an analyzed repository is submitted to the proposed search engine with a specialized index. This specialized index contains only code fragments, in which PostgreSQL queries are created and submitted. Since the analyzed application doesn't use such code patterns, every single result yield by the search engine is a false positive by construction.

Table 1 shows the numbers of such false positive results for the Elasticsearch and DuckDuckGo Android app repositories, respectively. The "low" setting produces large numbers of false positive results, as expected. The "low" setting was designed to minimize the amount of false negatives, with little regard for the amount of false positives. The "medium" search setting is a tradeoff between minimizing the false positive and false negative rates. The amount of false negatives in the elastic search repository using this search setting is very low, with less than one false positive search result per 1000 lines of code. The "high" setting didn't produce any false positive results. This is expected, as it is a setting that was designed to minimize the false positive rate while accepting an increase in the number of false negative results.

6.2 Quantifying the Occurrence of False Negatives

A conventional approach to identify false negative results would be to use a gold standard, i.e. a set of query documents for which perfect, “true” sets of search results are known. CodeKōan’s indices are too large to have experts decide for every indexed code example, whether it is reused in a query-file or not. Therefore a “synthetic gold standard” is created to measure the false negative rate. This approach is similar to previously published work by Roy et al. [23]. In this “synthetic gold standard” approach, an index with only a single code example E is constructed. Since the search algorithm is reflexive, using E as search query, E is returned in search results. If E' is a slight modification of E, using E' as search query, the proposed search algorithm should still determine, that E' is a reuse of the pattern E and returns E in the search results. In case E is not a search result for the query E', E is considered to be a false negative result.

To conduct this analysis, three basic types of source code modification were used: insertion, deletion and swapping of simple statements. A simple statement is a “statement of which no part constitutes another statement” [31]. This means that headers of for loops, class declarations, and method declarations are not simple statements. In the following a set of one thousand randomly selected code examples for the python programming language on Stack Overflow with exactly 10 simple statements is used.

Each of these randomly chosen code examples is successively modified with one of the listed types of modification. After all code examples are modified once, the fraction of still recognized examples is counted. After this, another modification is applied to each example, etc. Figures 1, 2 and 3 show the results of these experiments for insertion, swapping and deletion.

Code examples with a fixed and exact number of simple statements were selected in order to obtain comparable results. The number of ten simple statements was chosen, because code examples on Stack Overflow are generally around ten simple statements long, therefore these numbers are most representative.

Figures 1, 2, and 3 indicate that the proposed algorithm is very resilient to simple statement insertion, while swapping and deleting code leads to a quicker deterioration in recognition rates. A point to consider is, that in a ten simple statement long code fragment, a deletion of 3 or 4 simple statements is already a substantial alteration so that a significantly reduced frequency of recognitions is to be expected.

Overall, good rates of recognition are achieved for small amounts of alteration. Furthermore insertion of source code is the scenario which is most useful to recognize. Insertion of additional code adds to a pattern’s functionality, and the pattern is still clearly reused. Swapping and deletion of parts of a pattern on the other hand can quickly lead to drastically altered semantics, thus a quicker drop of the recognition rate on such alternation is a desired behavior of the search algorithm, since it is designed to only find meaningful reuse of code patterns.

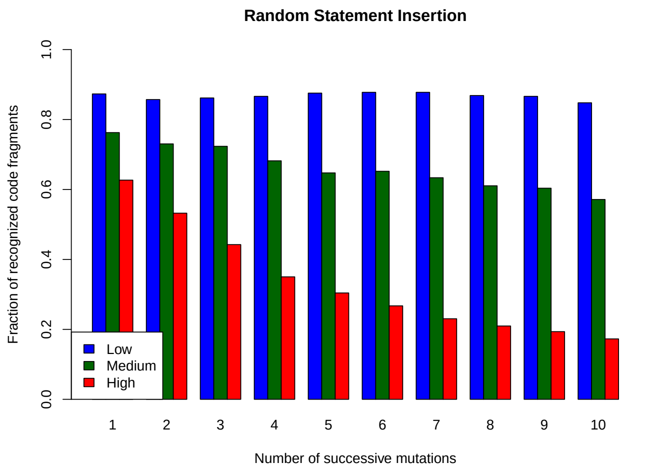

Figure 1: Recognition frequencies when inserting simple statements into randomly selected 10 simple statement long Python code examples. A value of 1.0 means that all mutated examples were deemed similar to the original, a value of 0.0 means that none were deemed similar.

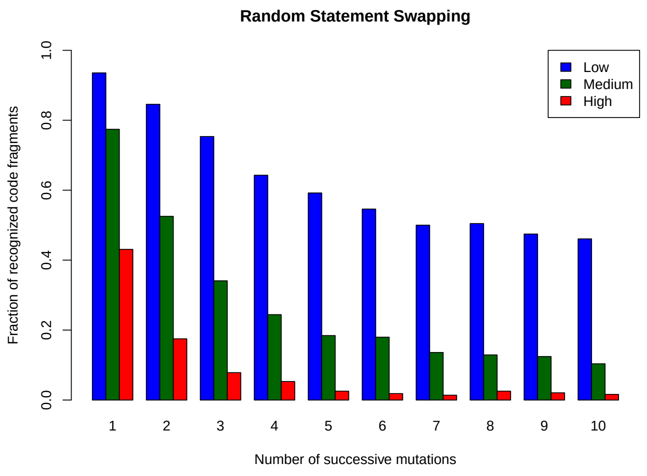

Figure 2: Recognition frequencies when swapping simple statements in randomly selected 10 simple statement long Python code examples. A value of 1.0 means that all mutated examples were deemed similar to the original, a value of 0.0 means that none were deemed similar.

6.3 Empirical Analyses

This section shows results of analyzing source code in publicly available source code repositories. These analyses are carried out on a per-repository basis. CodeKōan is queried with every source code file in the analyzed repository and the result sets are collected. With these result set collections two things are done: firstly the fraction of all tokens in all source code files that are part of at

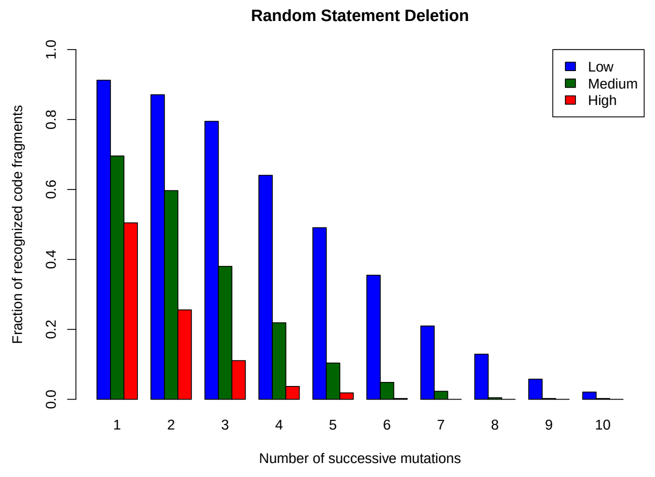

Figure 3: Recognition frequencies when deleting random simple statements from randomly selected 10 simple statement long Python code examples. A value of 1.0 means that all mutated examples were deemed similar to the original, a value of 0.0 means that none were deemed similar.

least one search result is determined. Since only the source code from Stack Overflow answers is indexed, this fraction indicates the fraction of the total source code of a project that is similar to at least one answer on Stack Overflow. Secondly it is shown, that descriptive tags for repositories can be generated based on the tags of matched Stack Overflow answers. A detailed outline of how these results are achieved is given in the following.

Frequency of Stack Overflow Reuse. In this analysis the proposed search engine is used to quantify the amount of source code in several public repositories that is reused from Stack Overflow. As outlined above, the procedure for this is straightforward to implement. For a given source repository every source code file F_i in the project is queried with the CodeKōan search engine. Let $|F_i|$ be the number of tokens in the token string of F_i. For each file the number $covered(F_i) <= |F_i|$ of covered tokens is derived, that are included in at least one search result.

These coverage numbers are used to calculate

$$\frac{\Sigma_i covered(F_i)}{\Sigma_i |F_i|} \tag{2}$$

This fraction is the relative amount of source code in a project, that is part of a source code pattern that is a component of a Stack Overflow answer. Using this methodology, empirical data on the amount of source reuse in real world projects is gathered. The results are split into table 2 with some general projects, and Android apps in table 3.

The gathered results of this analysis indicate a great variance in the amount of reuse of source code examples from Stack Overflow across analyzed projects. It is apparent that Android projects show much higher rates of source code reuse. The differences in the detected rates of source code reuse can largely be explained by the use of popular libraries and frameworks in projects. For example,

Project Name	Reuse %	Description
Open NLP	3.3	NLP library
GraphJet	3.4	Graph algorithms
maven-shared	6.3	Maven components
testng	7.0	Testing framework
tomcat	7.2	Webserver
humanize	8.7	Data formatting
Tiny	21.2	Image processing
AlgoDS	21.5	Algorithms, educational
Jest	23.2	Elasticsearch client

Table 2: Reuse rates of source code examples from Stack Overflow in general purposed open source repositories written in Java.

Project Name	Reuse %	Description
duckduckgo-android	14.9	Web Search Engine
openkeychain-android	15.9	Encryption
studentenportal	21.2	Klagenfurt university
TUM Campus	23.0	TU munich
HWT Dresden	27.6	Dresden university
unisannio	29.6	University of Sannio

Table 3: Reuse rates of source code examples from Stack Overflow in Android applications. Note that the observed four Android applications with most reuse rates are all projects of universities.

there is a great number of Stack Overflow questions about implementing functionality in Android apps. Furthermore libraries with few dependencies, that implement basic algorithms like GraphJet or Open NLP tend to show lower rates of reuse.

Tagging Repositories with CodeKōan. The result sets from the analyses above were used to generate descriptive tags for repositories using tags from Stack Overflow questions. Every located reuse references source code in a Stack Overflow answer which in turn is a reply to a single, tagged question. An answer can be thought of as having same tags as its parent question. By extension every code example A^k in an answer has the same tags as the whole answer. As a result, every search result with pattern sides from A^k has the same set of tags as A^k.

A simple statistical model is used to assign each tag t for a repository a significance between 0 and 1. Let H_0 be the null hypothesis that the pattern sides of generated search results are drawn randomly from all indexed Stack Overflow code examples. Under H_0, the amount k of occurrences of tag t in a result set $\mathcal{R}$ would follow a binomial null distribution. The number of trials n is $|\mathcal{R}|$ and the success probability $p(t)$ of the null distribution would be the fraction of all indexed Stack Overflow examples with tag t. Therefore the null distribution has parameters $(n, p(t))$. This null distribution is approximated by a normal distribution with parameters $\mu = np(t)$ and $\sigma^2 = np(t)(1 - p(t))$. The cumulative distribution function of that normal distribution is used to score the observed tags, giving higher scores to tags that are observed more frequently than would be expected under the null distribution.

With this technique, the content of an unknown software repository can be automatically detected using the tags of Stack Overflow questions provided by the community. After examining the GraphJet repository, the tags such as "arrays, array list, enums, list, hashmap, random, for-loop, iterator, fileinputstream, ... " received high scores and these tages describe reasonably the content of the observed GraphJet repository implemented graph algorithms.

7 CONCLUSION AND FUTURE WORK

The presented search engine is useful for locating reuse of short source code fragments. Currently, the indexed data encompasses all code examples in Stack Overflow answers for the programming languages Java and Python. Since a language independent algorithm is used which only requires programming language dependent tokenizers, this set of indexed data can be extended to include more programming languages in the future. Likewise, future work could crawl the web to index more content in the search index.

The CodeKōan search engine can be used in future development tools and integrated into existing solutions. Likely beneficial applications would be plugins for integrated development environments (IDEs). Such a plugin could be used to make programmers aware of existing solutions as they edit source code, thereby reducing the amount of redundant code and mental effort to solve tasks with already existing solutions. Furthermore, such an IDE plugin could provide programmers with documentation for source code that reuses code examples which are already well documented.

CodeKōan could also be used to improve existing continuous integration (CI) systems. CI systems continuously build and test source code repositories under version control. These build processes may contain style checkers which let builds fail if new source code doesn't conform to e.g. a preferred indentation or identifier naming style. Using the presented search engine, a CI system could be extended with a check for antipatterns in source code. For example if the examined repository were a library, in which no files should be accessed, CodeKōan could be used to specify a list of antipattern-examples for file access, the occurrence of which would cause a build failure.

REFERENCES

[1] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.* ACM, 681–682.

[2] Brenda S Baker. 1995. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on.* IEEE, 86–95.

[3] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. 2014. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering* 19, 3 (2014), 619–654.

[4] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on.* IEEE, 368–377.

[5] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[6] Amiangshu Bosu, Christopher S Corley, Dustin Heaton, Debarshi Chatterji, Jeffrey C Carver, and Nicholas A Kraft. 2013. Building reputation in stackoverflow: an empirical investigation. In *Proceedings of the 10th Working Conference on Mining Software Repositories.* IEEE Press, 89–92.

[7] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 1589–1598.

[8] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, 175–186.

[9] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[10] William B Frakes and Kyo Kang. 2005. Software reuse research: Status and future. *IEEE transactions on Software Engineering* 31, 7 (2005), 529–536.

[11] Dan Gusfield. 1997. *Algorithms on strings, trees and sequences: computer science and computational biology* (12 ed.). Cambridge university press.

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 96–105.

[13] Dieter Jungnickel and D Jungnickel. 2008. *Graphs, networks and algorithms.* Springer.

[14] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[15] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. 2007. CodeGenie: using test-cases to search and reuse source code. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* ACM, 525–526.

[16] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 872–881.

[17] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* IEEE, 25–34.

[18] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. 2012. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. *Georgia Institute of Technology, Tech. Rep* (2012).

[19] Santanu Paul and Atul Prakash. 1994. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering* 20, 6 (1994), 463–475.

[20] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding plagiarisms among a set of programs with JPlag. *J. UCS* 8, 11 (2002), 1016.

[21] Ronald C Read and Derek G Corneil. 1977. The graph isomorphism disease. *Journal of Graph Theory* 1, 4 (1977), 339–363.

[22] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[23] Chanchal K Roy and James R Cordy. 2009. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on.* IEEE, 157–166.

[24] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on.* IEEE, 1157–1168.

[25] Robert J Shapiro. 2014. The US Software Industry As an Engine for Economic Growth and Employment. *Georgetown McDonough School of Business Research Paper* 2541673 (2014).

[26] Amit Singhal, Chris Buckley, and Mandar Mitra. 1996. Pivoted Document Length Normalization. ACM Press, 21–29.

[27] Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 3 (2014), 26.

[28] Medha Umarji, Susan Sim, and Crista Lopes. 2008. Archetypal internet-scale source code searching. *Open source development, communities and quality* (2008), 257–263.

[29] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. 2014. Codes: Mining source code descriptions from developers discussions. In *Proceedings of the 22nd International Conference on Program Comprehension.* ACM, 106–109.

[30] Andrew Walenstein, Mohammad El-Ramly, James R Cordy, William S Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. 2007. Similarity in programs. In *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[31] Niklaus Wirth. 1971. The programming language Pascal. *Acta informatica* 1, 1 (1971), 35–63.