INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

Recognition of Code Patterns from Stackoverflow Answers in Computer Programs

Christof Ferdinand Schramm

Masterarbeit

Aufgabensteller	Prof. Dr. François Bry
Betreuer	Prof. Dr. François Bry, Yingding Wang
Abgabe am	20. Mai 2017

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 20. Mai 2017 Christof Ferdinand Schramm

Abstract

Writing good quality source code is a task that becomes increasingly relevant as the software-industry grows and it's products permeate all aspects of life. The economic cost of the maintenance and faults of badly written code is a major concern for the ever growing software industry.

The information in web resources plays a central role in how programmers solve implementation problems [8]. Frequently these web resources provide code examples in the shape of coding patterns – pieces of relatively short code, that solve a specific task; an example could be using an API to access web service functionality.

This master thesis proposes the CodeKōan search engine for locating regions of source code that are similar to code patterns in a large database. The pattern database, that is used in this work is the set of all code fragments in answers to user submitted questions on the popular Q&A site Stack Overflow.

One of the key questions that this thesis is concerned with, is to find a definition of source code similarity, which is suitable for identifying reuse of short code patterns. The proposed definition attempts to capture similarity on both a syntactic and semantic level. This notion of similarity is founded on four primary characteristics, which can be objectively measured.

To detect similarity according to this definition, a pipeline approach is developed. It starts with pre-screening for regions of the submitted query and syntactically similar regions of indexed source code. The matches from this step are then refined and filtered in further steps that increasingly aim to produce semantically similar results.

While the modules of the developed pipeline approach are based on programming language independent reasoning, a further plugin for every targeted programming language is necessary. Each programming language is tokenized differently, and has a unique way of structuring code. The purpose of language specific plugins is to make these features accessible. This thesis includes plugins for the Java, Python and Haskell programming languages. These are used to demonstrate the platforms capabilities.

Evaluation of the pipeline module's correctness and the method's boundaries is carried out on hand-crafted examples and by property-based testing.

The proposed methodology is used to analyze open-source repositories in order to test the popular hypothesis that a lot of practical, everyday coding is done by copying and pasting from Stack Overflow [1].

Zusammenfassung

Die deutsche Zusammenfassung ist eine eins-zu-eins-Übersetzung der englischen Zusammenfassung.

Acknowledgments

I would like to thank Professor Bry for offering me the opportunity to write my thesis at his lab. He has been tremendously supportive throughout the long process of writing this thesis. Our discussions have helped me to find new inspiration and refine the ideas behind this project.

I would also like to thank my advisor Yingding Wang for his support in overcoming the many obstacles that present themselves when working on a project of this scope.

Furthermore I want to thank both Yingding as well as Sebastian Mader for giving a lot of their time to give me advice on writing a better thesis and helping me in finding a lot of mistakes and unclear passages in this text.

Table of Contents

CHAPTER 1

Introduction

1.1 Why Code Quality Matters

On the face of it, computer programs are just lists of instructions that are executed by pieces of circuitry. This simplification, however, omits the essential role that computer programs play in maintaining modern society. They control virtually every piece of technology that surrounds us. This ranges from trivial devices such as toasters and electric toothbrushes to live saving technology like pace makers and critical infrastructure like power stations and road signs. Without computer programs there could be no modern communication; there would be no satellite communication, no email and even the machines sorting letters run using software, that reads postal codes from the backs of envelopes. Programs are what enables current transportation infrastructure, health care, aviation, industry and a plethora of other industries.

Remarkably, people generally don't realize the degree to which source code has taken control of their lives. That can rapidly change, however, when something goes wrong. The technical failures that lead to unreachable email servers, trains running late due to malfunctioning signal boxes and airplane crashes[1] highlight the degree, to which twenty first century life depends on correctly functioning computer programs.

Faults in software have been attributed to cause large economic damages. In the US alone, a 2002 report finds faulty software to have caused damages of *„just under 1 percent of [...] gross domestic product (GDP)“* [29]. Since then, the US software industry has nearly doubled in size [27] and it stands to reason, that the cost of faulty software grew accordingly. In addition to financial cost, software bugs are known to have resulted in numerous deaths due to faults in medical devices [35] or airplane control systems [34].

There is a very substantial body of computer science research into writing less faulty software. The insights gained in this research have lead to the adaption of software development techniques and abstractions like:

[1] like the 2015 crash of a A400M transport aircraft with 4 casualties in Seville, Spain

- avoiding goto statements [10]
- exception handling [15]
- the paradigms of object oriented, functional and logic programming
- static type systems

and newer abstractions like monads [20]. Classic computer science research is mainly concerned with finding techniques, that eliminate or reduce certain types of errors and problems in programming by imposing limits on what programmers can do. Additionally, this research has led to a great number of abstractions, that allow the expression of program logic in much clearer ways.

Analysing the Human Factor in Programming
However, the aforementioned research is mainly focused on technological aspects of computer programming. It is important to keep in mind that most computer code is written by humans, and therefore the errors in the produced code are caused ultimately by mistakes made by a programmer in the process of writing source code. Because of this it is very useful to analyze, how code is actually written and to generate reliable data on the prevalence of good and bad practices in programming.

To address the aforementioned issues, this master thesis seeks to generate new insight into how one of the fundamental activities in programming – solving technical implementation problems on a small scale – is performed in the real world. When approaching implementation problems, programmers are often faced with unknown factors such as:

- application programming interfaces (APIs) unknown to a programmer
- multiple alternatives for solutions
- uncertainties over the correct usage of components

Usage of Online Documentation
A common way to find a solution in such circumstances is to consult documentation and discussion with peers (off- and online). Both in peer discussions, as well as documentation, one of the most popular ways to communicate ideas about implementation are code patterns. These code patterns are small and commented pieces of code (see section 1.2.1), that provide one or more archetypal solutions to a specific programming problem. Code patterns are one of the most popular forms of documentation, because understanding and appropriately adapting an already implemented solution is often easier than coming up with a good solution from scratch. Additionally these code patterns are often provided together with some natural-language explanation and discussion highlighting a solution's advantages, drawbacks and edge-cases.

It is important to note, that there is a good and a bad way to use code patterns. A diligent programmer would make sure, that she fully understood a code pattern and that the pattern she chose to adapt was actually a suitable way to solve the problem she faced. The outcome of such a strategy would be logically consistent code, that covers all relevant edge cases and follows current best practices. However, programmers, that have collaboratively worked on larger projects will attest, code patterns are often less well used.

Due to time pressure, lack of skill or other constraints, people writing software will often take shortcuts to quickly arrive at a working solution. With code patterns, there is always a temptation to just take a solution, that works in other circumstances, and perform just

the bare minimum of adaptions to overcome an implementation task. Such an implementation method will quickly arrive at a solution, that will appear to work in the short term. However, doing this will have a long term negative impact on a codebase.

The solution provided by a code pattern often doesn't solve the precise problem that a programmer is trying to solve. Even worse, it might do something unexpected. A pattern might use a different API from the rest of the codebase, or it might be written in a style that isn't used in surrounding code. It might throw unexpected exceptions when inserted into an unintended context or it might change the behaviour of other code. When reusing code patterns, it is imperative, that programmers actually adapt a pattern to suit their needs precisely and cover all edge-cases. Therefore it is necessary to actually understand precisely what is going on in the code pattern. Otherwise the resulting source code will become increasingly inconsistent both in it's logic and style.

CodeKōan: A Search Engine for Discovering Usage of Online Documentation
To the author's best knowledge, there is no previously published empirical study of the usage of code patterns from online documentation in real-world source code. The principal contribution of this master thesis is the CodeKōan search engine. This search engine can efficiently detect reuse of code patterns in user-supplied source code queries. The CodeKōan search engine is a versatile tool, that makes conducting such empirical studies of code pattern reuse feasible.

The search algorithm used in the CodeKōan search engine combines methods from code plagiarism detection and code-clone detection, as well as natural language processing and automata theory in a novel way. An additional benefit of the proposed algorithm is that it processes search queries by applying a pipeline of filtering steps to the results of a very sensitive initial search. This makes the algorithm well suited to further use and adaption in future work as outlined in chapter 6.

The Source of Code Patterns for This Thesis: Stack Overflow
There are a great number of resources like forums or manuals online that provide documentation and solutions, including code patterns. In this master thesis, the most widely used database of code patterns – the Q&A site Stack Overflow – is used for developing and evaluating CodeKōan, a search engine for code pattern reuses in user submitted source code documents.

1.2 Relevant Concepts

This section gives a brief outline of some concepts that are central to this thesis. This part of the thesis sets out with a discussion of code patterns and some examples thereof. After that, a very brief overview of Stack Overflow, source code quality and similarity will be given.

1.2.1 Code Patterns

Code patterns are a core concept of this master thesis. A code pattern is a short, frequently used piece of code that solves a specific programming problem. In that regard they are different from the frequently used object oriented design patterns, because these solve prob-

```
try{
    // Do something that can cause an exception
} catch (SomeException e){
    // Exception handling code here
} finally {
    // Code that gets executed despite possible exceptions
    // (cleanup code, etc.)
}
```

Figure 1.1: A very common coding pattern in Java: exception handling

lems of software architecture, while code patterns solve common implementation problems.

Since code patterns address specific small scale problems, they are generally short – usually no longer than fifty lines of code (LOC) in common imperative programming languages like Java. Depending on the programming language, these patterns may vary in length. Haskell for example is known to generally be comparatively terse. Whereas a C programmer would commonly use multiple lines to e.g. apply a function to each element in an array, Haskell commonly uses just one expression like `function <$> array` for the same task.

Code patterns are used to describe various and diverse solutions to problems. Most language idioms can be documented by coding patterns. A frequently used code pattern in Java is the handling of exceptions using the `try/catch/finally` construct. In documentation or online discussion platforms this pattern could be presented in the form it is depicted in figure 1.1.

1.2.2 Structure and Mechanisms of Stack Overflow

The evaluation of the methods proposed in this thesis is performed on code patterns from the popular question and answer site Stack Overflow[2]. This section gives a short overview of how data is organized on Stack Overflow.

Organization of Data – Q&A Threads
Fundamentally Stack Overflow is a platform for programmers to ask questions and gather answers to these questions. Content on the site is organized in several millions of question and answer (Q&A) threads. Each of these threads is started by a user's *question*. At any given point in time, each question can have zero or more *answers* from members of the community. Only one of these answers can be marked as an *accepted answer* by the poster of the question [22]. Both questions and answers are referred to as *posts*. Additionally each post can have zero or more comments. These comments are not relevant for the methods proposed in this thesis.

Furthermore it should be noted that questions on Stack Overflow virtually always have one or more tags that are added by either the submitter of a questions or members of the community. These tags are intended to concisely summarize a question's topic.

An example of a Q&A thread is shown in figure 1.2.

[2] http://stackoverflow.com

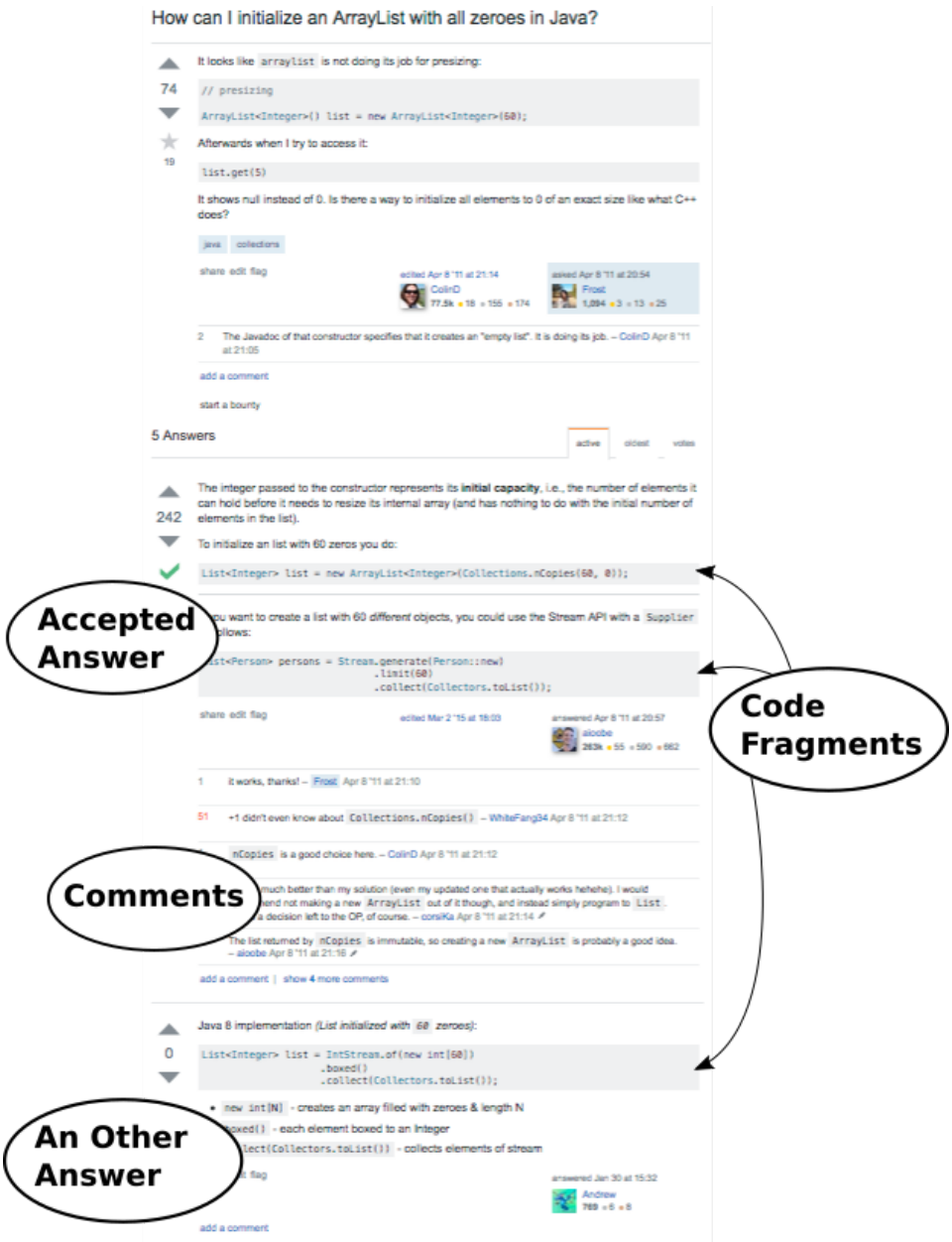


Figure 1.2: A concrete example of Q&A thread on Stack overflow

Content of Stack Overflow Posts
For the purposes of this thesis, the content of Stack Overflow posts is defined to be made up of source code *fragments* (specially delineated, formatted pieces of source code) and other content in the form of natural language text, images, hyperlinks, etc. Only the source code fragments are considered in the rest of this thesis. While the non - source code parts of the content of Stack Overflow posts could be used as well, this is beyond the scope of this work.

Rating & Reputation System
Posts on Stack Overflow can be both up- and downvoted by users; this results in each post having a *rating*. The rating influences a post's visibility and also the sorting order of displayed answers to a question. The sum of all of the ratings of a user's posts and comments makes up that user's *reputation*. Reputation is used to unlock more of the site's functionality to users that have already generated content of high-enough quality to be upvoted by other users. Upvoting, for example, is only available to users that have a minimum of 15 reputation points as of October 2016, downvoting requires 125 reputation points[3].

1.2.3 Source Code Quality

Many of the common terms and concepts that are generally used to communicate ideas about source code are only loosely defined and context dependent. One such term is, e.g., source code quality. What exactly makes good source code? These are a few traits that are desirable in any piece of source code:

Readability The source code should be easily understandable by human readers

Conciseness The source code should not be unnecessarily long

Efficiency The compiled source code should fulfill its task using as little resources as possible

Modifiability Over the time, that source code is part of a computer program it is often subject to change due to e.g. changing requirements. Source code should be structured in a way that makes this process as simple as possible.

Correctness Source code has to perform its function correctly and (as far as possible) without crashing.

These goals are frequently at odds with one another. The most concise code may often be less readable. A need for efficiency might be conflicting with modifiability and readability, etc. In different contexts, some of these goals are more important than others. In high performance applications, modifiability and conciseness might be sacrificed for efficiency. In settings where failures in the software may lead to injury or loss of life, correctness trumps all other concerns.

Often it is not clear, how these factors should be weighted. What constitutes a good quality piece of source code under which circumstances is often a subjective judgment. Different people that are familiar with different coding styles, feel more familiar with some solutions over others. Developers that prefer working with the functional programming paradigm may prefer solutions that use higher order functions and recursion, while developers that mostly work on low-level systems might prefer iterative and imperative approaches.

[3] A detailed list of privileges that are awarded upon reaching a certain level of reputation: http://stackoverflow.com/help/privileges

1.2.4 Source Code Similarity

1.2.4.1 The Importance of a Clear Definition

Just like source code quality, source code similarity is a context dependent, subjective concept. This is a problem in any methodology that attempts to detect similar pieces of source code. It is hard to do a thorough evaluation of results without having a precisely defined notion of what they should be. Since automatic evaluation of any similarity-based method of source code analysis will be hindered by lack of a precise definition of source code similarity, the concept will be discussed in this section. First, the notion of similarity in related work will be discussed, and subsequently the notion that is used in this work is developed.

1.2.4.2 The Concept in Related Work

There is much scientific literature that is concerned with recognizing similarities between pieces of source code. This research is important for a great number of applications such as *„[...] program understanding [...], code quality analysis [...], aspect mining [...], plagiarism detection, copyright infringement investigation, software evolution analysis,code compaction [...], virus detection,and bug detection“* [25]. While a great number of publications [25, 24], are concerned with detecting similar source code, there is a conspicuous absence of a widely agreed-upon definition for what precisely makes source code *similar*. As one publication on source code clone detection puts it: *„this level of definitional vagueness is typical within clone detection publications.“*[32]

Code Similarity is Context Dependent
The absence of a clear definition of source code similarity is due to a number of difficulties. One of the major issues is that *the very concept of similarity* – independent of source code similarity – is difficult to define [31]. In fact, there is an argument to be made that the concept of similarity is dependent on context. What may be considered similar under some circumstances will be considered dissimilar under a different set of circumstances.

The Merriam-Webster dictionary defines the word „similar“ as **„having characteristics in common“**[4]. What is noteworthy, is that it is not specified, *which* characteristics these are. So for example a dog might be considered similar to a whale, as they are both mammals. Genetically speaking they are certainly more similar than a whale and a virus, however they are vastly different animals based on morphology, habitat, diet and a host of other characteristics. The key to whether or not any two arbitrary things can be considered similar is the set of characteristics that are being used as a basis for the assessment.

Categories of Source Code Similarity
There is an excellent discussion [31] of the concept of similarity in respect to computer programs by Walenstein et al. Like a number of other publications [24], Walenstein et al. group the characteristics for measuring source code similarity into two broad categories:

Syntactic Similarity: Source code generally refers to a textual representation of algorithms and data structures. Characteristics in this category can be further divided into the source code as text, the source code's syntax and the source code's structure.

Semantic Similarity: The most important part of any source code is what it does, not how it is written down. Semantic characteristics of source code are largely independent

from its textual representation. Semantic similarity is much harder to precisely define than syntactic similarity.

1.2.4.3 Measuring Source Code Similarity

The concepts of syntactic and semantic similarity are elaborated upon in this section. This thesis only considers similarities of source code that is written in the same programming language, except for cases where the opposite is explicitly stated.

Syntactic Similarity
Of the two broad kinds of similarity that we discerned above, syntactic similarity is comparatively more tractable. There are several syntactic characteristics that can be used for similarity analysis. Some of these are:

Source Code Character-Strings Similarity can be analyzed on the level of the raw source code text. As will be discussed later, this is a rather limited concept of similarity.

Look at the token-strings of programs The next „higher level“ at which source code similarity can be considered, is at the level of token-strings. These token-strings are independent of identifier-naming, layout, etc. and therefore the set of source code that is recognized as similar to any given piece source code is larger than with character strings alone.

Compare abstract syntax trees A further characteristic of pieces of source code in high level languages is their abstract syntax tree. These tree-like structures represent the syntactic structure of source code with less regard to the particular tokens that the source code consists of.

Each of these characteristics has been used for similarity detection in previously published work, parts of which will be surveyed in section 1.3.1.

Semantic Similarity
Characteristics for the semantic similarity of source code are much harder to define. The essence of what these characteristics try to capture is the degree to which two pieces of source code fulfill the same tasks.

In contrast to syntactic equality, semantic equality of two pieces of source code is not generally decidable due to the undecidability of the halting problem for arbitrary turing machines[4].

Special Case: Source Code String Equality and Semantic Equality
The only exception to the general undecidability of semantic equality that is relevant in this thesis is pieces of code that have identical string representations. Identical source code strings that are run by the same compiler or interpreter on a machine in the same state are guaranteed to produce the same output, as computers do not have a source of true randomness.

[4] Theoretically, the equivalence of two programs running on real computers can be decided, since any computer that can be constructed has a finite number of possible states. This is practically irrelevant, however, because that number is intractably large.

This case can be further generalized. Let $sem(C)$ denote the semantic meaning of a piece of code C. Now let C_a and C_b be two pieces of code for which it holds that $sem(C_a) \neq sem(C_b)$. Furthermore let f be a deterministic transformation of source code $f : C \to C$. Then it holds that if $sem(C_a) = sem(f(C_a))$, $sem(C_b) = sem(f(C_b))$ and $f(C_a) = f(C_b)$ it follows, that $sem(C_a) = sem(C_b)$.

This means, that if a deterministic transformation of program source code like appropriately replacing all while loops with for loops or naming all identifiers according to some fixed scheme leads to identical source code strings, then the two pieces of source code are semantically equivalent.

Extending the search algorithm proposed in chapter 2 with such transformations will be discussed in section 6.2.1.3.

General Case: Semantic Similarity Heuristics

Generally, no source code transformations f as outlined previously are available. Therefore, due to the discussed undecidability problems, heuristics for deciding semantic similarity have to be used. Some of the steps of the pipeline algorithm outlined in chapter 2 are such heuristics.

1.2.4.4 Further Discussion of Source Code Similarity

By definition, identical pieces of source code are both syntactically and semantically similar. The question of whether or not two pieces of source code are „similar“ gets more murky – and more interesting – in less obvious cases. Consider, for example, the two Java - code snippets in figure 1.3.

An iterative approach:

```
public int calculateSum(int[] arr){
    int sum = 0;
    for(int i = 0; i < arr.length; i++){
        sum += array[i];
    }
    return sum;
}
```

A recursive approach:

```
public int caluculateSum(int[] arr){
    return calculate(arr, 0);
}

private int calculate(int[] arr, int i){
    if(i == arr.length){
        return(0);
    } else{
        return arr[i] + calculate(arr, i+1);
    }
}
```

Figure 1.3: Syntactically different but semantically similar: calculating the sum of an `int` array in Java

Accessing each key of map.

```
public void method(Map<String,Int> mp){
    String[] keys = mp.keySet().toArray();
    for(int i = 0; i < keys.length; i++){
        mp.get(keys[i]);
        System.out.println(keys[i]);
    }
}
```

Emptying a map.

```
public int method(Map<String,Int> mp){
    String[] keys = mp.keySet().toArray();
    for(int i = 0; i < keys.length; i++){
        mp.remove(keys[i]);
        System.out.println(keys[i]);
    }
}
```

Figure 1.4: Syntactically similar but syntactically different source code fragments

At a glance the recursive and iterative approach to calculating a sum appear syntactically different but semantically identical. They both calculate the sum of an array of `int` values. However there is a subtle difference: if the input array is very large, the recursive variant will lead to a stack overflow error (depending on JVM configuration).

An other example case (again in Java) is presented in figure 1.4. In this case, the similarity of two pieces of code is considered; one of the two gets all values from a map, the other empties it. These two pieces of code are syntactically very similar while they do two semantically different things.

1.2.5 A Working Definition of Code Pattern Similarity

Up to this point, this thesis hasn't given an answer to the question: „when are two pieces of source code similar?". As previously discussed, this question has no generally accepted answer that covers all use cases. The reason for this is, that different notions of similarity are useful for different application scenarios [31], and that the general concept of similarity is highly context dependent.

A More Precise Notion of Similarity

Thus, the question this thesis is concerned with is: „when are parts of a piece of source code similar enough to a code pattern that they can be assumed to solve the same problem in a similar way?". Posing the question for similarity analysis this way captures what this thesis fundamentally tries to accomplish: to find the reuse of common coding patterns in computer program source code.

Given that we want to answer this specific question, it is necessary to outline the similarity measures and characteristics of source code that will be used to give a satisfying answer. It is noteworthy that an approach that focuses on semantic similarity alone is not sufficient to answer the question above. Any purely semantic similarity measure would only show that two pieces of code are „doing similar things". This, however is not what this thesis is

solely interested in. For this thesis it is also important to be able to discern if two pieces of code use a similar approach in implementation style.

Four Characteristics for Similarity

Like any objectively quantifiable similarity measure, the one developed in this thesis is based upon certain shared characteristics of pieces of source code. These characteristics are:

1. The token-strings of pieces of source code. This basic characteristic for similarity analysis is primarily chosen for practical reasons. Operating on token-strings is highly efficient but still more robust to minor changes than using character-strings.
2. Since we are dealing with similarity of arbitrary code to a fixed set of patterns, code can only be similar to a pattern if it is similar to a relevant fraction of the patterns overall code.
3. A hierarchical block structure that is present in many programming languages and given by delimiters like {} in Java or C. This structure is chosen as a characteristic because it carries information about the organization and layout of code.
4. The degree of semantic similarity of the words in identifiers. This characteristic is important because it captures a layer of meaning of source code that is not covered by the other characteristics. This characteristic should ideally lead to considering code similar if it pertains to the same problem domain.

This thesis' notion of similarity seeks to capture both syntactic and semantic characteristics of source code. The presented approach is fundamentally based on token-strings. Token-strings are preferred over more elaborate measures like program dependency graphs mainly for efficiency reasons. On this low level, loose syntactic similarities are detected, which are subsequently filtered in a series of heuristics that seek to capture semantic meaning. These are outlined in chapter2.

1.3 Related Work

Since this thesis is primarily concerned with searching for reuse of code patterns, work related to this thesis is mostly about detecting code similarities and searching in large string-sets. Work that deals with judging whether two pieces of source code ares similar mainly aims to find code clones and plagiarism. While this thesis is not mainly concerned with either of these two things , the methods developed in these two field are very relevant to the methods proposed here. Therefore they will be discussed first.

1.3.1 Code-Clone Detection and Software Plagiarism Detection

As already mentioned previously, a large part of the related work of this thesis is concerned with code-clone and plagiarism detection. This is because both fields develop methods to find similar pieces of source code. However it should be kept in mind that there is a relevant difference between such work and the subject of this thesis. Code-clone detection and plagiarism detection are both attempting to find the results of intentionally copying code. This thesis deals with a wider topic as it tries to find (intentional or unintentional) reuse of any of a large database of already defined code patterns in given source code.

1.3.1.1 Typologies of Code-Clones and Plagiarism

Most [18, 5, 23] of the related work discussed in this thesis is based on the assumption that similar code is the result of deliberate obfuscation. While this work does not share the assumption that the discovered similarities are the result of malicious intent, the terminology of these publications is still useful in our context. Reuse of a code pattern without copying can look similar to code plagiarism / code-cloning without being a case of either.

Disguises Used in Plagiarism
A scheme for classifying plagiarism disguises was described in Liu et al. [18]. This scheme identifies five types of common alterations to source code that plagiarists make to disguise their activities. These are:

1. Format Alteration – changing layout of code so that it appears different in layout
2. Identifier Renaming – replacement of identifier names
3. Statement Reordering – reordering of statements that does not affect the result of a program
4. Control Replacement – the replacement of control structures by similar structures. An example of this is the replacement of a for-loop by an equivalent while loop.
5. Code Insertion – the insertion of unnecessary source code

Any of these five practices could also be found when reusing code patterns without malicious intent. Code patterns are after all just small, common solutions to frequent programming problems. Therefore those solutions are likely to be pairwise similar except for any of the introduced five plagiarism disguises. Code insertion for example can happen without malicious intent simply by interleaving a common code pattern with a solution to an other problem in the same function. Something like this could e.g. result if the java - exception handling code pattern in figure 1.1 were filled with additional code.

Types of Code Clones
There is a common typology [25] in code-clone detection publications that assigns pairs of source code fragments that form a code clone to one of four classes. These are:

Type I clones: The character-strings are the same except for variations in layout, comments and whitespace.

Type II clones: The token-strings of the two source code fragments are identical.

Type III clones: Anything that would fall into categories I or II plus the variations from points three to five of the plagiarism disguises (statement reordering, control replacement and code insertion).

Type IV clones: The two pieces of source code lead to the same computational result, but are denoted differently on a syntactic level.

As with the plagiarism disguises, any of these four code-clone variants can occur as a result of the (maybe unintentional) reuse of common source code patterns. The algorithms that are developed in this thesis need to therefore stand up to scrutiny with respect to this typology.

1.3.1.2 Pipeline Approaches in Code-Clone Detection

The algorithm in this thesis uses a pipeline based approach. Similar approaches are also used in a number of code-clone detection publications, therefore a generic model for such an approach that was described in a 2009 survey [24] by Roy et al. of code clone-detection methods will be introduced here. The relevant differences and intersections of this work's approach and the concept from Roy et al. will be highlighted.

The approach follows a basically linear approach in six steps:

Preprocessing This step removes likely irrelevant code, and determines units for comparison. There is a notable difference here. In this thesis' specialized use case the identified units for comparison are already given in the form of Stack Overflow answer - code fragments

Transformation In this part of the pipeline, an intermediate representation of the code is generated. In the case of this work's algorithm this a token-string representation is used.

Match Detection Code-clones are identified in this step by using the result of the transformation step.

Formatting In this step, the clones that were detected on an intermediate representation in the match detection step are mapped back to regions in the original source code.

Post-Processing/Filtering After the formatting step, the matches are ranked and filtered either manually or automatically using heuristics. This brings the most relevant search results to the user's attention.

Aggregation In the final part of the pipeline, summary statistics are generated and the matches are organized into classes of similar code-clones.

The approach of this thesis that is described in chapter 2 also works as a pipeline, and it is conceptually similar to the stereotypical pipeline outlined above.

1.3.2 Full Text Search and Sequence Alignment

1.3.2.1 Suffix Trees

Suffix trees are an established data structure in computer science that will be discussed in further detail in section 2.4.3. This section will only give a short overview of the general properties and applications of this data structure along with some salient sources on the matter.

Originally developed in 1973 by Weiner [33], suffix trees are a data structure that finds its use in *„linear time solutions to many strings problems more complex than exact matching“* [16]. Formally suffix trees are defined in [16] as :

Definition 1 *A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from root to leaf i exactly spells out the suffix of S that starts at position i.*

An important property of suffix trees is that a suffix tree for a string S of length n can be constructed in $O(n)$ time. There are multiple algorithms for achieving that achieve this linear time bound [14]. The most popular and easy to understand one is Esko Ukkonen's algorithm [30].

Suffix trees can be easily extended to represent not only a single string S, but a set $\mathscr{S} = \{S_1, ..., S_k\}$ of strings [16]. The resulting data structure is called a *generalized suffix tree*. The construction of a generalized suffix tree is still possible in $O(\sum_{i=1}^{k} |S_i|)$.

One of the classic problems that suffix trees are applied to is the *exact matching* problem. The goal is to find the zero or more occurrences of a short pattern-string P in a larger text-string T. Assuming a suffix tree $\mathscr{T}$ of T is already available, this problem can be solved in $O(|P|)$ time [16].

The real advantage of suffix trees comes into play, when using them as an index structure. Once the suffix tree is built, many search problems are solvable in a number of steps that is *proportional to the size of the search-pattern*. The largest effort is put into the preprocessing step of building the suffix tree, the subsequent query steps are then comparatively computationally efficient.

1.3.2.2 Levenshtein Automata and Fast Dictionary Lookups

Levenshtein Distance
The *levenshtein distance* [17] $d_L(W,V)$ of two strings $W,V \in \Sigma^*$ is the minimal number of edit operations which is needed to transform W into V. An edit operation in this context is:

- Deletion of a symbol
- Insertion of a symbol $x' \in \Sigma$
- Substitution of a symbol x for an other symbol $x' \in \Sigma$

Levenshtein Automata
Levenshtein Automata were first proposed by Schulz [26] in 2002. The main idea behind levenshtein automata is, to construct deterministic finite state automata (DFAs) $LEV_k(W)$ that recognize the language of all words V, for which $d_L(W,V) \leq k$.

The seminal paper by Schulz [26] also outlined an algorithm, which could decide whether $V \in \mathscr{L}(LEV_k(W))$ without first constructing the entirety of the (potentially very large) automaton $LEV_k(W)$.

The original use case that levenshtein automata were proposed for is to offer a quick way to do word spelling correction, assuming the availability of a dictionary $\mathscr{D}$ in the form of a trie.

CHAPTER 2

Code Pattern Recognition

2.1 Relevant Notation

For the description of the algorithm in this chapter it is necessary to define some notation. A more comprehensive overview of the terminology and notation in this thesis is given in the glossary on page 71.

Substrings
Let s be a string of letters from an alphabet Σ. Then $s_{a:b}$, $a \geq 0 \wedge b \leq |s| \wedge b \geq a$ is a substring of s starting at position a and ending at position b. The length of $s_{a:b}$ is $b-a$. Furthermore $s_{a:}$ is the suffix of s beginning at position a. Conversely, $s_{:b}$ is the prefix of s starting at position b.

So for example if $s = \texttt{test}$ then $s_{0:1} = \texttt{t}$ and $s_{1:} = \texttt{est}$

Stack Overflow Answers and Answer Fragments
All Stack Overflow answers are uniquely identified by some $i \in \mathbb{N}$. So A^i is the i^{th} Stack Overflow answer. As already described in section 1.2.2 each A^i contains $0 \leq j \leq n \in \mathbb{N}$ code fragments. The j^{th} fragment of answer A^i is identified as $A^{i,j}$.

Combined with the substring-syntax this means that $A^{i,j}_{a,b}$ refers to the substring from the a^{th} to the b^{th} character / token of $A^{i,j}$.

2.2 Outline of the Proposed Approach

Before an in-detail description of the algorithm powering CodeKōan, this section will give a high-level overview of the pipeline-structure of the proposed algorithm. A visual overview of the pipeline is given in figure 2.1.

Queries and Replies
The proposed CodeKōan search engine, accepts source code in the form of a *query document* D. Query documents are arbitrarily long pieces of source code. These query documents are typically files containing source code like Haskell or Python modules, Java-classes or something similar.

The replies of the search engine are individual reuses of code patterns in the query document. These reuses are not just reuses in the sense of exact string matching, but in the sense of the similarity concept developed in section 1.2.4.

The Index
The search engine maintains a separate index of code patterns $\{P^i, i \in \mathbb{N}\}$ for each supported programming language. Every index contains a generalized suffix tree of all the token-strings of code-patterns for one programming language in the Stack Overflow data set.

How Search Works
The search algorithm itself is structured as a pipeline. The first steps of this pipeline analyze source code similarities on a token-string level, and then progress increasingly to analyze further for semantic similarity based on heuristics. This approach is sensible, considering that it is computationally very expensive to analyze a database of millions of code patterns for semantic similarity directly. Rough syntactic similarity, however, can be detected with more economic resource use, which is important for developing a search engine that has to produce results quickly.

Pipeline Step: Alignment
The first step in the pipeline is called *alignment*. It finds all regions $D_{a:b}$ of the query document token-string, that are similar to token-string regions $P^i_{x:y}$ of code patterns in the index. These matched tuples $(D_{a:b}, P^i_{x:y})$ are referred to as *alignment matches*. They are filtered and grouped in further steps. The alignment step combines common algorithmic techniques using suffix trees and levenshtein automata into a novel algorithm for inexact string search that is well suited to this thesis' use case.

Pipeline Step: Aggregation
The next step, *aggregation*, filters out all alignment matches that are shorter than a parameterized length. The remaining alignment matches are grouped by the origin of their pattern-side, P^i. These groups are then referred to as *search results*. The set of search results is then further filtered and eventually returned. After the following semantic processing steps, one search result indicates a single reuse of a code pattern.

Pipeline Step: Code Block Structure Analysis
Source code in most programming languages is structured in hierarchical blocks. Some languages, like LISP, make this very explicit through ubiquitous use of parenthesis. The same is also true for languages with a c-type syntax where code blocks are delimited by curly braces, and languages like haskell and python where code blocks are defined by indentation.

This block structure has deep semantic meaning. It controls the scope of identifiers, and code in a valid search result that is in the same block in a pattern should be in the same block in the query document.

The pipeline's code block structure analysis step defines a binary relation called *block accordance* over alignment matches. The idea is, to split search results into groups of alignment matches that fall into the same block structure on both the query-document and the pattern side.

Pipeline Step: Word Similarity Analysis
This pipeline step tries to capture the problem domain source code is concerned with. This is done by using methods from natural language processing. For each search result, the words in all identifiers from alignment-match pattern-sides $P^i_{x:y}$ and all alignment-match document sides $D_{a:b}$ are collected into lists. For each search result's two lists, a score between 0 and 1 is then calculated using NLP methods. Search results that have a score below a certain – parameterized – threshold are then considered irrelevant and removed from the result set.

2.3 Normalization and Tokenization

The purpose of the algorithm presented in this chapter is to quickly find reuses of code patterns in any given computer program. The description of the algorithm itself is programming language independent, but its implementation defers certain features to language specific plugins. As outlined in figure 2.1, the algorithm works by first constructing a large inverted index from a database of source code fragments. The search then consults this inverted index to find pattern usages.

Since the algorithm is largely token-based, the description of the algorithm begins with a discussion of tokenization and normalization procedures. The next section in this chapter describes, how tokenized/normalized code from the pattern-database is used to build an index to for search. Following that, a step by step description of the pipeline is given.

Tokenization
Like many other algorithms, that deal with matters of source code similarity, the proposed algorithm does not deal with the character-string representation of given program source code directly. Instead, the token-strings of source code are used as the basis of analysis. In tokenization, the character-string of a piece of source code c is transformed into a token-string $tokens(c)$. For each programming language implementation there is a finite set of *token types* T so that $\forall_{0 \leq i < |tokens(c)|} : tokens(c)_i \in T$. The specific token types and tokenizers have to be implemented by a language specific plugin. Such token types might be for example:

Number for numbers, that occur in program source code. There may be different tokens for octal or decimal numbers

Identifier variable names, function names, class-names, etc.

Keyword like `switch`, `case` or `def`

Type for e.g. primitive data types in Java

Indent/Unindent Tokens that mark indendations in white space sensitive languages like python or haskell

Figure 2.1: An overview of the search engine's pipeline.

An important property of token types is, that there exists a distance function:

$$distt(a,b) : (T,T) \rightarrow \{x | x \in \mathbb{R} \wedge x \geq 0\} \tag{2.1}$$

In this thesis, the usual choice for this distance function is: $distt(a,b) := 0$ if $a = b$ and 1 otherwise. Unless explicitly stated otherwise, this distance function is used. While this case of a distance function seems trivial, uses for more complex *distt* will be outlined in chapter 6, especially in section 6.2.1.2. The default distance function can only express equality, but a distance function with output values other than 0 or 1 can be used to express more nuanced degrees of similarity among token types.

Token-Annotation
Simple tokenization alone is not sufficient for implementing a useful search algorithm. The reason for this is, that once search is done, regions in source code that are related need to be identified precisely, so that they can be displayed. Therefore a tokenizer that is useful for this thesis, needs to produce a token-string, where each token is generated along with the range in the original source code, that it covers. That means, a tokenization function has to have the type [1] `String -> [(Token, (Int, Int))]` This means that every token, which is generated is annotated with a start and end-position in the actual character string of the underlying source code.

Normalization
Some regions of program source code are not relevant for analysis, because they don't lead to meaningful analysis results. One example of this are import statements in Java-code. Such code regions can be omitted during the tokenization step. This leads to token-strings, that only contain tokens which are deemed relevant for analysis.

What exactly – if anything – is omitted or transformed in the tokenization step is dependent on a programming language plugin's implementation.

2.4 Index Construction

This section describes the construction of index structures that are relevant for the proposed search engine and its applications. A separate index for each programming language has to be constructed. The reason for this is, that the token-types of each programming language are different. The index-generation itself, however, is polymorphic over the types of tokens. This means that the index construction process is independent of the concrete token types. It is only required that all indexed token stings are made up of tokens from the same set of token types T, which was outlined earlier.

2.4.1 Accessing the Structure of Stackoverflow Data

Recall section 1.2.2, where it was outlined that each question Q^i has a set of answers $\{A^j | A^j answers Q^i\}$. Conversely, each answer A^j has exactly one parent question Q^i, which it answers.

[1] Type signatures for functions are simplified Haskell type signatures. These are not exactly equal to the ones used in actual the program.

For displaying formatted results and other applications, this structure is made accessible through a PostgreSQL database.

2.4.2 Programming Language Index Part 1: NGram Bloom filter

Fast Probabilistic Membership Detection
Bloom filters [6] are a widely used data structure, that allows to determine for any item x if $x \in S$ for some set S. In fact, for a given bloom filter this determination can be made in a constant amount of time irrespective of how many items it contains.

A caveat, however, is that *the membership determination of bloom filters allows false positives*. This is offset by the fact, that the membership determination does not admit false negatives, and the error probability can be kept arbitrarily small by increasing the memory available for the bloom filter.

This means, that *bloom filters allow constant time set membership lookups with a low, controllable probability of false positives*.

A Bloom Filter for Ngrams
A bloom filter is part of any programming language specific index. For each indexed code-fragment c , the set of all its *ngrams*, $ngrams(n,c)$ is determined. $ngrams(n,c)$ is defined as:

$$ngrams(n,c) = \begin{cases} \{c_{i:i+n} \mid \ i \leq |c| - n\} \text{ , if } n \geq |c| \\ \varnothing \text{ otherwise} \end{cases} \tag{2.2}$$

This means, that an ngram is a substring of n-contiguous characters.
It holds that $|ngram(n,c)| = |c| - n$. For example:

$$ngrams(3, \texttt{contiguous}) = \{\texttt{con}, \texttt{ont}, \texttt{nti}, \texttt{igu}, \texttt{guo}, \texttt{uou}, \texttt{ous}\} \tag{2.3}$$

The bloom filter BF_l in the index for the programming language l contains the set:

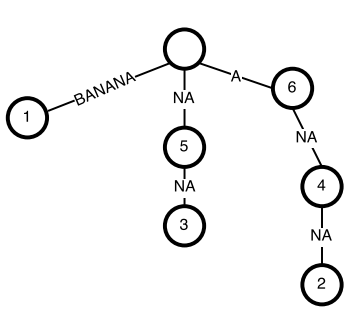
$$\bigcup_{i=1}^{|A|} \bigcup_{j=1}^{|A^i|} ngrams(n, A^{i,j}) \tag{2.4}$$

which is the set of all token-string ngrams of all Stackoverflow answer-fragments for a given language l. The n-gram size n has to be specified as a parameter for index construction.

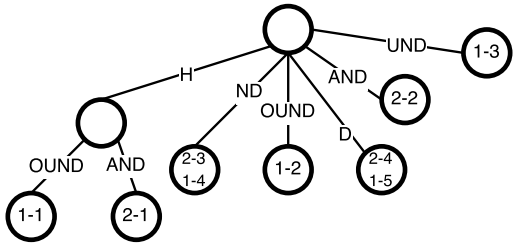
This part of the index will be used later for quickly determining if an ngram of a query document D occurs in any of the indexed answer fragments.

2.4.3 Programming Language Index Part 2: A Generalized Suffix Tree (GST)

As already discussed in section 1.3.2.1, generalized suffix trees are data structures, that allow for efficient exact search in a set of strings $\mathscr{S} = \{s_1, ..., s_k\}$.



(a) A suffix tree for the string BANANA



(b) An example of a generalized suffix tree for the strings 1: „HOUND“ and 2:„HAND“. Note, that nodes can have multiple labels in GSTs and that each label now contains an identifier for the string $\in \mathscr{S}$ it refers to.

Figure 2.2: Examples of a suffix tree / generalized suffix tree

In the proposed algorithm, a generalized suffix tree is used to index the token-strings of all Stackoverflow answer fragments $A^{i,j}$ for a programming language l. This is accomplished by first constructing a suffix tree for every $A^{i,j}$ and then merging these suffix trees into one large suffix tree T_l. Whether an answer should be part of the index for a language l is resolved by inspecting answers' parent question tags. This section will outline how this is performed in detail.

2.4.3.1 GST: Construction

Suffix Trees for Individual Strings
First, suffix trees for individual strings and their construction will be discussed. For this purpose, let s be a string of letters over an alphabet Σ. The aim is to construct a suffix tree for s.

Recall the definition from 1.3.2.1. The suffix tree T of s is a rooted tree, where the edges of the tree are labeled with strings consisting of letters from an alphabet Σ. The suffix tree T for the string s has $|s|$ nodes that are labeled with values from 1 to $|s|$. Each path from the root to a labeled node spells out a suffix of s, and all suffixes of s are contained in such a path. No two edges starting form a node can have labels that begin with the same character.

A suffix tree for the word BANANA is shown in figure 2.2a. In this example, all inner nodes of the tree except for the root node are labeled.

A suffix tree for a string s can be constructed in $O(|s|)$ steps. One algorithm to do this is Ukkonen's algorithm [30].

Application: The Substring Problem
Suffix trees are versatile data structures with a great number [16] of applications. The application, that suffix trees are likely best known for is the *substring problem*. The problem is to find whether a pattern-string p occurs in a longer text-string s. This problem can be solved in $O(|p|)$ time, if a suffix tree T for s is available.

A solution is reached by starting at the root of the suffix tree T and following the path that is spelled out by p. If p is a substring of s, this can obviously be done in $\Theta(p)$ steps. If p is not a substring of s, then there is a character in p, that can not be used to descend further down a the path in T.

This problem is relevant, because a variation of it is used in the alignment step, that is outlined in section 2.5.1.

Generalized Suffix Trees for Sets of Strings
Generalized suffix trees are suffix trees, that encode information for a set of strings $\mathscr{S} = \{s_1, ..., s_k\}$. A GST is again a rooted tree with edges labeled with strings of characters $\in \Sigma$. In a GST, all paths from the root to labeled nodes spell out a suffix of some $s_i \in \mathscr{S}$. Furthermore, analogous to suffix trees for single strings, all suffixes of every $s_i \in \mathscr{S}$ are present as a path from the root to a labeled node.

A simple example for a GST for $\mathscr{S} = \{\text{“}HOUND\text{”}, \text{“}HAND\text{”}\}$ is given in figure 2.2b.

It is remarkable, that the substring problem for single text-strings can be easily extended to solving the substring problem for all $s_i \in \mathscr{S}$ by using a GST. A decision of whether a pattern p is part of any (or multiple) of the elements of $\mathscr{S}$ can be made in $O(p)$ time if the GST for $\mathscr{S}$ is available.

Also, a GST for $\mathscr{S}$ can be constructed in $O(\sum_{i=1}^{|\mathscr{S}|} |s_i|)$ [16, 30].

2.4.3.2 GST: Merging and Splitting

A GST containing all code from Stackoverflow answer fragments for a much discussed programming language – like Java or Python – can become very large. Furthermore, using single indices does not scale very well with larger volumes of requests, and direct index construction can't well be parallelized. The GSTs for popular languages tend to be larger than small machine's RAM, and the construction of large GSTs can take several hours.

These points can be addressed by exploiting a very favorable property of generalized suffix trees, namely that they can be split and merged and still yield the same solutions for for a great number of problems – among others the substring problem. GSTs for sets $\mathscr{S}$ of strings can be constructed by merging GSTs of subsets [16].

Construction by Merging
A simple procedure for merging two GSTs T_a and T_b would be to create a new tree by merging the root nodes of T_a and T_b. However, this will likely not be a valid suffix tree because of edges with the same starting character originating from the root node. This is remedied by merging these edges and inserting a new node, if a full merge is not possible. In the example in figure 2.2b, such edges would be the edges for `HAND` and `HOUND`. Both have a common prefix (`H`). After this common prefix, a new node is inserted to resolve the conflict. This node then has two children for the differing parts `AND` and `OUND`. This conflict resolution method is then repeatedly applied until no conflicts are left, and the result is a valid GST.

Horizontal Scaling by Splitting
It has been shown [16], that running multiple queries on all GSTs for subsets of $\mathscr{S}_i \subset \mathscr{S}, \bigcup \mathscr{S}_i = \mathscr{S}$ will yield the same results as performing a signle query on a large GST for all of $\mathscr{S}$. That fact makes it possible to distribute the index over multiple machines. This has the advantage of requiring proportionally less memory and CPU time per machine.

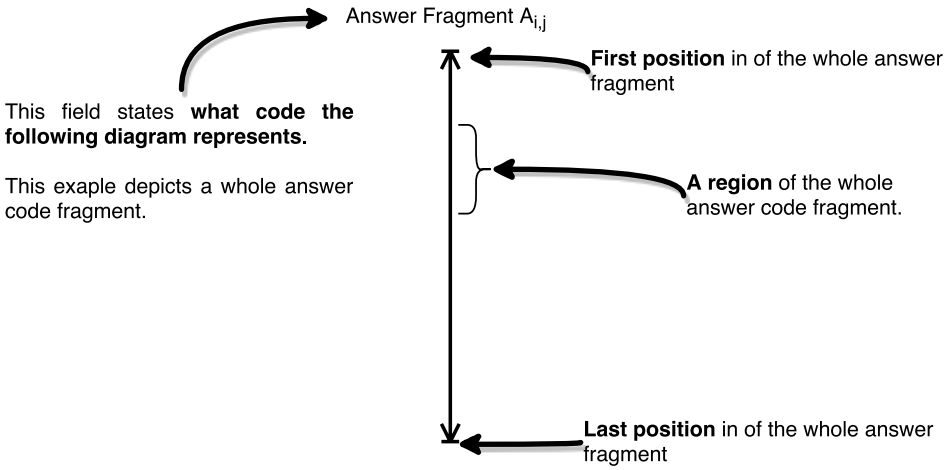


Figure 2.3: A graphical notation for regions in strings.

2.5 Syntactic Filtering

The following is a description of the initial two steps of CodeKōan's algorithm: alignment and aggregation. The idea of these two steps is to use generalized suffix trees to find pieces of a submitted query document, which are (syntactically) similar to parts of answer fragments $A^{i,j}$.

2.5.1 Alignment

2.5.1.1 Outline

The alignment step is the first step of the proposed algorithm's search pipeline. For this description of the algorithm it will be assumed that a single GST of all code-fragments is the basis of analysis. This reasoning is valid because performing alignment on multiple GSTs, that are subsets of all indexed code fragments, and yielding a union of the results is the same as using a single tree.

For clarification, a graphical notation for regions of strings will be used. This notation is introduced in figure 2.3.

Parameters, Input and Output
The alignment step takes three parameters:

- The Ngram-Size used for index construction,
- the minimal match length l_{min}, and
- the accepted levenshtein distance d_{max}

The input for the alignment step is the token-string D of a query document. The output of the alignment step is the set of all *alignment matches*:

$$M_A = \{(D_{a:b}, A^{i,j}_{x:y}) \mid b - a \geq l_{min} \wedge d_L(D_{a:b}, A^{i,j}_{x:y}) \leq d_{max}\} \tag{2.5}$$

This means, that an alignment match is a pair of a token-substring of the query document and a token-substring of a code fragment from the database. The first element of the pair is referred to as the *document-side* and the second element is referred to as the *pattern-side* of the alignment match. A graphical schematic of alignment matches is provided in figure 2.4.

The document-side and the pattern-side of an alignment match are within a parameterized levenshtein distance d_{max} of each other, and the document-side of the alignment match is longer than a given minimal length l_{min}.

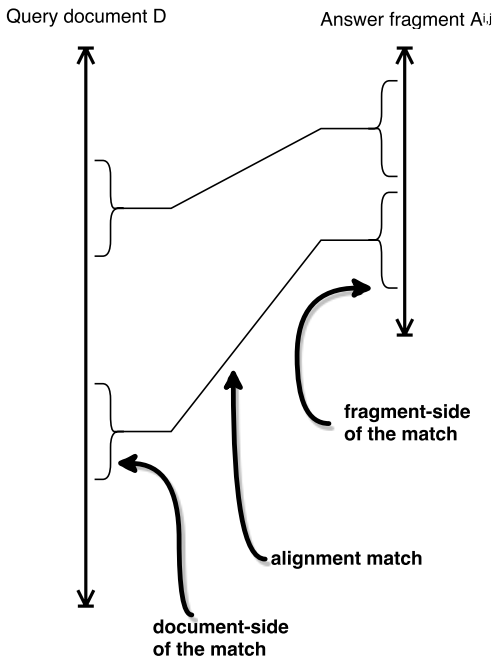


Figure 2.4: Two alignment matches.

2.5.1.2 Searching with GSTs

The description of alignment will begin with a simplified algorithm. This algorithm will then be extended to yield the desired results.

Fixed Length and Starting Point, Levenshtein Distance 0
The base algorithm aims to find the following set of alignment matches:

$$M'_A = \{(D_{0:l_{min}}, A^{i,j}_{x:y}) | D_{0:l_{min}} = A^{i,j}_{x:y}\}$$

This means that M'_A is the set of all exact occurrences of the l_{min}-long prefix of D, in anywhere in any answer code-fragment $A^{i,j}$. Given the GST T for $\bigcup_i \bigcup_j \{A^{i,j}\}$, this is merely an instance of the substring problem.

This is solved by starting at the root of T and entering the edge, that starts with D_0. Then the path spelled out by $D_{0:l_{min}}$ is followed. If there is a D_i, for which we can't extend the path, then $M'_A = \{\}$. Assuming we followed a path for the entirety of $D_{0:l_{min}}$, we know that $D_{0:l_{min}}$ is a substring of at least one answer code fragment $A^{i,j}$.

By doing this, the decision problem of whether $D_{0:l_{min}}$ is a substring of any $A^{i,j}$ is solved. What is still missing, however, is what answer fragments $A^{i,j}$, and what text-ranges of these answer fragments $D_{0:l_{min}}$ is identical to.

Assuming the walk down from the root of T for $D_{0:l_{min}}$ is completed, the algorithm either halts in a node or in the middle of an edge's label. Now the sub-tree of the current position is inspected. All labels of nodes are collected by a depth-first traversal of the sub-tree. The result of this traversal is a set $\{((i,j),x)\}$ of pairs of identifiers for Stackoverflow answer code fragments $A^{i,j}$ and positions x at which the suffix that is indicated by the label starts. What is needed to build alignment matches now, is the endpoint y of the regions $A^{i,j}_{x:y}$, whose starting points are given by the labels $((i,j),x)$. This y can be trivially derived in our case: $y := |D_{0:l_{min}|}$, which is just l_{min}.

The gathering the labels of child-nodes is referred to as *fragment label collection*.

This variant shows one of the major performance issues of the alignment algorithm. After descending down a path for $D_{0:l_min}$, all labels of the sub-tree of the resulting position have to be gathered. This could possibly be a very large number. For the simplified algorithm, that has been described so far, this is not a problem however, because we only have to do this step once.

Improvement: Variable Alignment Match Length
So far, the alignment matches that are generated are not very helpful. For example an alignment match $(D_{0:l_{min}}, A^{i,j}_{x:y})$ could conceivably have n more overlapping letters, extending it to $(D_{0:(l_{min}+n)}, A^{i,j}_{x:(y+n)})$. The current version of the algorithm would miss such extensions.

Upon closer inspection, it is clear, that the ending point l_{min} of the traversal from the root of T is arbitrary. Therefore, the algorithm is modified so that the traversal from the root goes on as long as there are more D_i, that can be used to extend the path. This way, the traversal either ends with a conflict or by reaching the end of D with no more letters to consume.

If the path up to the conflict or end of D is longer than l_{min}, fragment label collection is performed and the resulting alignment matches are returned.

Improvement: Inexact, Fixed Length Matching Using Levenshtein Automata
Levenshtein automata [26] were presented in section 1.3.2.2. These automata can be constructed for a fixed string s and a levenshtein distance d_{max}. Such an automaton $LEV_{d_{max}}(s)$ accepts the language $\mathscr{L} = \{s' | d_L(s, s') \leq d_{max}\}$.

Now assume, that alignment is performed to find all alignment matches within levenshtein distance d_{max} of a *fixed* region $D_{a:b}$ of the query document. We do this by constructing the levenshtein automaton $LEV_{d_{max}}(D_{a:b})$. Then, as before a traversal of the suffix tree T from the root is begun. The letters in the edge-labels of T are used as input for the levenshtein automaton and its state is tracked along the traversal. The traversal continues until the levenshtein automaton enters an accepting state.

With this model, there is more than one path, that the walk through the tree can follow. The higher d_{max} is set, the more of the suffix tree has to be traversed. This higher amount of possible paths is due to the definition of levenshtein automata. Whereas exact search can reject branches immediately because their label's first character does not match, mistakes are allowed when traversing with a levenshtein automaton.

One question remains: when should label collection be performed? The answer to this question is, that fragment labels are collected each time the levenshtein distance increases (i.e. when a mistake is admitted) if more than l_{min} characters were admitted. Furthermore we perform fragment label collection when the levenshtein automaton enters an accepting state.

Improvement: Inexact Matching with Variable Length
The previous improvement sacrificed variable length matching for the possibility of inexact matching in alignment. It turns out, however that this is not a tradeoff that has to be made. In their paper Schulz and Mihov [26] describe a method to use levenshtein automata without completely constructing them.

The alignment algorithm makes use of that fact. At a starting-point i, instead of setting a fixed stopping point, the levenshtein automaton for $D_{i:|D|}$ is used. Previously, traversal

continued until an accepting state of the levenshtein automaton was reached. The last improvement to the algorithm modifies this condition by traversing *until either the end of D or a rejecting state of* $LEV_{d_{max}}(D_{i:|D|})$ *is reached*.

Elimination of Redundant Results
One problematic property of the alignment algorithm is, that it could possibly return redundant alignment matches. Redundant matches are pairs of alignment matches $(D_{a:b}, A^{i,j}_{x:y})$ and $(D_{a':b'}, A^{i,j}_{x':y'})$ where $d_L(D_{a':b'}, A^{i,j}_{x':y'}) \geq (D_{a:b}, A^{i,j}_{x:y})$ and $a' \geq a$ and $b' \leq b$ and $x' \geq x$ and $y' \leq y$. In such cases, the second alignment match $(D_{a':b'}, A^{i,j}_{x':y'})$ carries no additional information that is not given by the first match. The first match therefore *subsumes* the second one. Alignment matches that are subsumed by another alignment match are removed from the set of alignment matches that the algorithm returns.

Performance Enhancement: Variable Starting Points by Bloom Filter Usage
The algorithm currently only searches for matches from a given starting point. A naive strategy to find all valid results would be to simple perform the alignment algorithm from each starting position i where $0 \leq i < |D| - l_{min}$.

However this is not the most efficient strategy, that can be used. Some starting points i will not yield results that are longer than l_{min}. We can find worthwhile starting points by consulting the bloom filter BF, that is part of our index. We identify the set of suitable starting points as

$$startpoints = \{i | i \in \mathbb{N} \cup \{0\} \wedge i < |D| - max(l_{min}, \text{ngram-size}) \wedge member(BF, D_{i:(i+\text{ngram-size})})\} \tag{2.6}$$

This is a viable strategy if the ngram size is smaller than l_{min}. If the ngram-size larger than l_{min}, possible alignment matches could be overlooked. Therefore the ngram-size should be chosen during index construction so that it is smaller than sensible values for l_{min} but so large, that the number of spurious search starts is minimized.

Properties of the Alignment Algorithm
With the previous improvements, the alignment algorithm now generates the desired result set M_A described in equation 2.5. The algorithm has a number of favorable properties:

- starting points for alignment are efficiently found
- the generation of alignment matches is greedy, i.e. alignment will yield the longest possible alignment matches
- inexact matching up to a levenshtein distance d_{max}
- no fixed maximal alignment match length has to be used

Performance of Reduncancy Elimination

In practice, the number of initial alignment matches can get very high especially when using a large indices and query documents with thousands of lines of code. In such cases a naive algorithm for removing redundant matches will make alignment too slow for reasonable application in a search engine. While a $O(n^2)$ comparing each alignment match with

every other alignment match is easy to implement, it has proven to be too slow in practice. This drawback is addressed by exploiting two basic observations:

1. Subsumption is only possible for pairs of alignment matches on the same code pattern $A^{i,j}$. Therefore as a first step, alignment matches are grouped by the origin of their pattern sides. This can be done in $O(n)$ amortized time by using a hash-based map from answer fragment id (i, j) to a linked list of alignment matches.

2. Let m_1, m_2 be two alignment matches. If m_2 is subsumed by m_1 then the document-side range of m_2 must be wholly contained in m_1. This is a necessary, but non-sufficient condition for subsumption. This allows for an optimization to the redundancy-elimination algorithm, that drastically reduces run-time in practice, even though it does not lower the theoretical worst-case bound of $O(n^2)$.

   The idea of this optimized algorithm is, to order the list by the starting point of the document sides of the alignment matches and then perform redundancy checking by traversing this ordered list once. This traversal keeps track of all previous alignment matches that can possibly subsume the current one. The algorithm is given in detail in 1

Algorithm 1 An algorithm for efficiently removing redundant alignment matches

```
1:  l ← list of alignment matches
2:  l_sorted ← l sorted on document side starting pos.
3:
4:  maximal ← []
5:  relevant ← []
6:
7:  for i = 0; i < length(l_sorted); i++ do
8:      (start, end) ← document side start/end position of l_sorted[i]
9:      subsumedByNone ← True
10:     for all r in relevant do
11:         if end(r) < start then
12:             remove r from relevant
13:             add r to maximal
14:         else if end(r) ≥ end then
15:             if l_sorted[i] subsumed by r then
16:                 subsumedByNone := False
17:                 break from this loop
18:     if subsumedByNone then
19:         add l_sorted to relevant
20: add all matches in relevant to maximal
21: return maximal
```

2.5.2 Aggregation

Alignment alone is not enough to capture reuse of code patterns in answer fragments $A^{i,j}$. It merely finds common substrings, but does not carry any information on the significance of that substring. For example an alignment match $(D_{10:20}, A^{i,j}_{30:40})$ will not be very meaningful if the code fragment $A^{i,j}$ is five hundred or more tokens long. The pipeline's aggregation step addresses this concern.

This pipeline step aggregates alignment matches into groups by the origin of their pattern side $A^{i,j}$. Then these groups are filtered by whether the pattern-sides of all alignment matches in a group make up a significant part of this group's pattern.

Parameters, Input, and Output
The aggregation step only takes one parameter: the **coverage - percentage** p_{cov}. The purpose of aggregation is to eliminate all alignment matches from a set of matches, that have pattern-sides from answer code fragment $A^{i,j}$ which are not sufficiently represented in the set of alignment matches.

The Notion of Coverage
To detect reuse of a pattern which occurs in a code fragment $A^{i,j}$ it is assumed that a significant part of $A^{i,j}$ has to occur in D. The reasoning for this is that answer code fragments are assumed to be self-contained sensible units of source code. Thus, if only a small part of this self contained unit is detected, it is judged to not be sufficiently represented.

To detect this, the notion of *coverage* is used. Let s be an arbitrary, nonempty string and § be a set of substrings of s defined as ranges $a : b$ where $a \leq b \wedge a > 0 \wedge b < |s|$. The coverage of S of s is then defined as:

$$\frac{\sum_{i=0}^{|s|-1} cov(S,i)}{|s|}$$

where

$$cov(S,i) = \begin{cases} 1 \text{ if } \exists (a:b) \in S.i \geq a \\ 0 \text{ otherwise} \end{cases}$$

This means, that the coverage of a set of alignment matches for an answer fragment $A^{i,j}$ is the fraction of unique positions of $A^{i,j}$ that are included by at least one pattern-side substring $A^{i,j}_{x:y}$ of $A^{i,j}$. The concept of coverage is graphically illustrated in figure 2.5.

Filtering Alignment Matches for Coverage
Aggregation calculates the coverage of the set of alignment matches for each answer fragment $A^{i,j}$ in the set of alignment matches. If the coverage is below p_{cov}, all alignment matches with a pattern-side from the fragment $A^{i,j}$ are excluded from further processing.

2.6 Semantic Filtering

So far the pipeline has produced groups of alignment matches. Alignment matches are precisely defined in equation 2.5. These alignment matches have been generated for syntactic similarity on a token-string level.

The algorithm's further steps filter the sets of alignment matches and group them into actual, sensible search results. To do this, it is necessary to define what exactly a search result is, which will be done in the discussion of of analyzing code block structure.

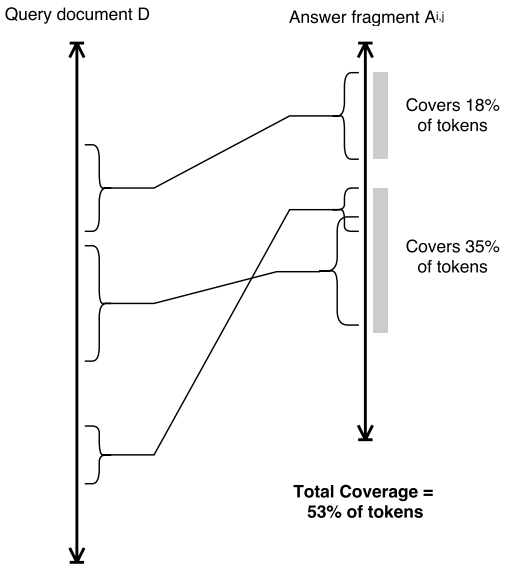


Figure 2.5: An illustration of coverage. The pattern-sides of the three alignment matches cover 53% of the token-string of the answer fragment $A^{i,j}$

The aim of the pipeline steps in this section is to incorporate heuristics for semantic similarity into the pipeline. These heuristics use characteristics of the underlying programming languages that are not immediately apparent from the pure token-string. In this regard, the pipeline steps in this section differ from the first two steps, which are language unspecific (except for a tokenizer).

2.6.1 Code Block Structure Analysis

This step will deal with alignment matches grouped by the origin-fragment of their pattern-side, i.e., the steps process the set of alignment matches M_A by viewing subsets $\mathscr{S}_{A^{i,j}} \subseteq M_A$, where

$$\mathscr{S}_{A^{i,j}} = \{(d, p_{x:y}) | (d, p_{x:y}) \in M_A \wedge p = A^{i,j}\}$$

2.6.1.1 Conflicts

A group of alignment matches $\mathscr{S}_{A^{i,j}}$ is not a sensible search result. The stated goal of this thesis is to find individual reuses of code patterns from Stackoverflow answers. Such an individual use is not present if the alignment matches for a single answer code fragment $A^{i,j}$ in a search result have either overlapping pattern-sides or document-sides.

Two alignment matches $(D_{a:b}, A^{i,j}_{x:y})$ and $(D_{a':b'}, A^{i,j}_{x':y'})$ are *conflicting* if either the intervals $(a:b)$ and $(a':b')$ or $(x:y)$ and $(x':y')$ are overlapping. A very simple example of conflicting alignment matches is shown in figure 2.6.

The first requirement for a group of alignment matches to be called a search result is, that the alignment matches in the result are pairwise non-conflicting.

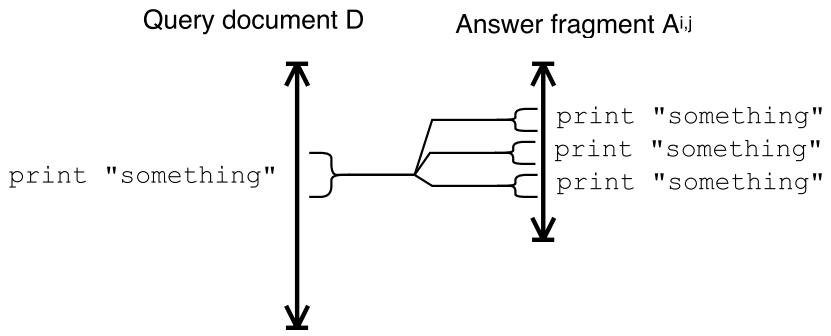


Figure 2.6: A simplified case of three conflicting alignment matches. The pattern sides of all three matches are identical.

2.6.1.2 Hierarchical Position Relationships

Source code in most high-level programming languages is organized into blocks. These code blocks are often denoted by special delimiters. Such delimiters may be parentheses in LISP-dialects or curly braces in languages with c-like syntax. Languages like python denote their blocks by indentation. These delimiters can be inserted into the token-string even if they are not directly present (as is the case in e.g. python which has no explicit block delimiters).

Point-wise Relationships
The code block structure analysis step of the pipeline operates on this code block structure. First we define a function `blockR`: $(\mathbb{N},\mathbb{N}) \rightarrow (\mathbb{N},\mathbb{N})$, that expresses a relationship between two points in a source code token-string. Let a and b be two indices of a source code string. Without loss of generality, let $a \leq b$.

Then `blockR(`a`,`b`)` is a pair $(down, up)$. $down$ is the minimal number of block levels that have to be descended to reach point b. up is the minimal number of block levels that have to be ascended to reach point b. An example of `blockR` is shown in figure 2.7.

Alignment Match Accordance
Using `blockR`, it is now possible to define a binary relation $AC \subseteq M_A \times M_A$. If it holds for a pair of two alignment matches $(D_{a:b}, A^{i,j}_{x:y}), (D_{a':b'}, A^{i,j}_{x':y'})$ that $((D_{a:b}, A^{i,j}_{x:y}), (D_{a':b'}, A^{i,j}_{x':y'})) \in AC$, they are in *accordance*.

Two alignment matches $(D_{a:b}, A^{i,j}_{x:y}), (D_{a':b'}, A^{i,j}_{x':y'})$ are in accordance ,i.e. $\in AC$ if:

1. `blockR(`a`,`a'`)` $= (0,0)$ and
2. the alignment matches are non-conflicting

Sensible Search Results
With the notion of accordance, search results can be defined. A search result $\mathscr{R}_{A^{i,j}} \subseteq \mathscr{S}_{A^{i,j}}$ is a maximal set of alignment matches that are pairwise in accordance with each other.

Such search results $\mathscr{R}_{A^{i,j}}$ fulfill a number of minimum requirements for something, that could justifiably called a search result.

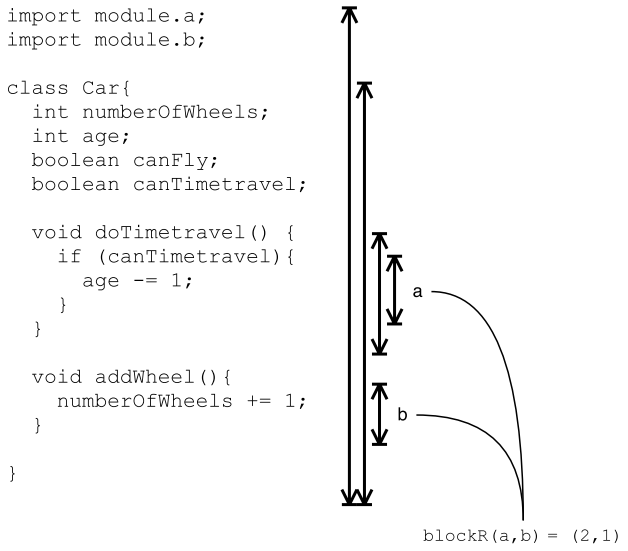


Figure 2.7: `blockR` in a simple example. Two blocks have to be descended when going from a to b and one block has to be ascended. So `blockR` $(a, b) = (2, 1)$.

1. Conflict free sets of alignment matches separate multiple reuses of a pattern $A^{i,j}$ in D into different results.
2. A search result will not be made up by recognition of a pattern that just repeats a small part of D often due to conflict freeness. An example of such an undesirable situation, which is avoided is shown in figure 2.6
3. Pairwise accordance of the alignment matches in a search result guarantees that pieces of a pattern occur in a similar context. For example, if two pattern-sides of a pair of alignment matches occur in different functions, accordance makes sure that they don't occur in the same code block in the document-sides of the matches.

2.6.2 Identifier Word Similarity Filtering

This step of the pipeline operates as a filter on search results $\mathscr{R}_{A^{i,j}} \subseteq \mathscr{S}_{A^{i,j}} \subseteq M_A$. It takes into account one of the „levels of meaning" that have been discarded by tokenization, namely the identifiers. Identifiers carry a lot of meaning. The statement `<identifier>.<identifier>();` tells us nothing, however the statement `databaseConnector.connect();` already tells us something about the intent behind the source code.

This step of the pipeline analyzes similarities of the words that in identifiers of the query document and patterns. For each search result $\mathscr{R}_{A^{i,j}}$, a score $sim \in [0..1]$ of the word-similarity between matched document and pattern regions is calculated. That score is then compared against a threshold value $sim_{threshold}$ and $\mathscr{R}_{A^{i,j}}$ is rejected if $sim < sim_{threshold}$.

Extracting Words From Identifiers

Many identifiers consist of one or more combined words like e.g. `databaseConnection` or `image_cache_state`. The first thing that this pipeline step needs to do is to extract

individual words from identifiers.

Words in identifiers are commonly distinguishable by casing rules. The two most common ones are camel case (e.g. `someObjectCaller`) and snake-case (e.g. `some_object_caller`). In both cases, we can easily split the identifiers into individual words. In the case of snake-case we simply split on underscores. Camel-case case identifiers can be easily split as well.

Some care is taken in the splitting of camel case identifiers to correctly identify upper case abbreviations. So for example the identifier `HTTPConnection` is split into two words „http“ and „connection“.

Bags of Words from Search Results

The pipeline's identifier word similarity filtering step uses the function for splitting identifiers to generate two bags of words b_D and b_P for each search result. The bag of word b_D contains all words in identifiers on the document-sides of the alignment matches, that make up the search result. The second bag of words, b_P, contains all words from identifiers on the pattern sides of the alignment matches.

These two bags of words are used as input for TF-IDF cosine similarity, that yields a similarity value $sim \in [0, 1]$.

In the identifier word similarity filtering step, this measure sim is calculated for all search results, and search results that have a value for sim smaller than a threshold value $sim_{threshold}$ are excluded from the set of search results. A schematic of how this step works is given in figure 2.8.

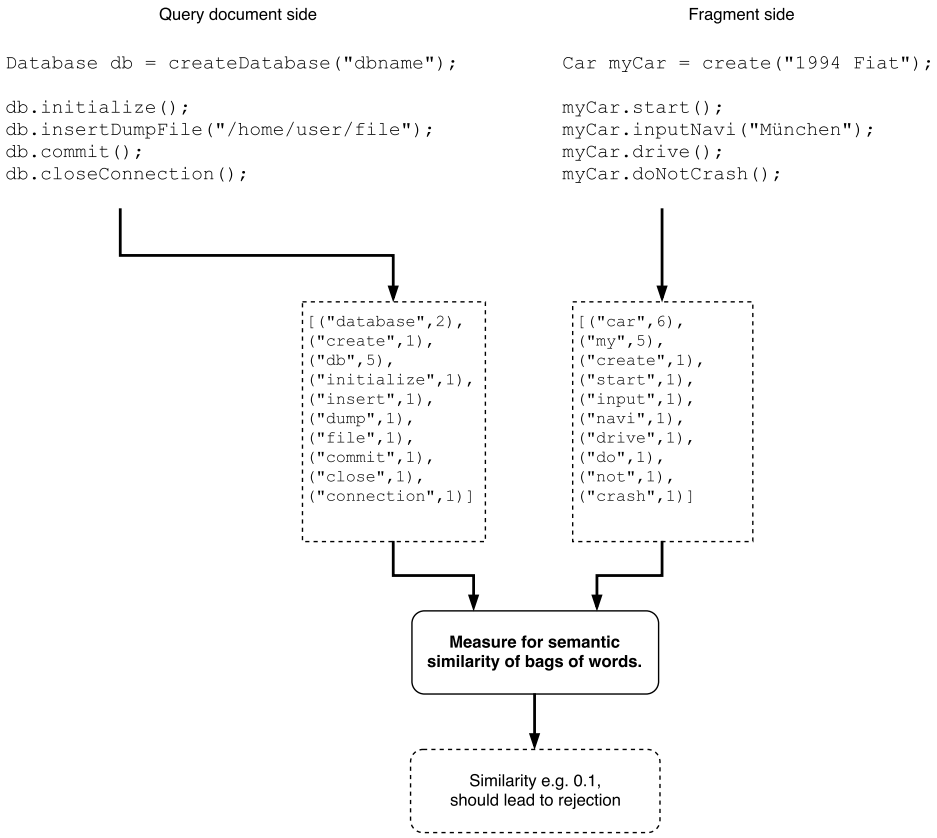


Figure 2.8: A schematic of how the identifier similarity filtering step works. First the identifiers of document-side and pattern-side are processed into bags of words. Then these bags of word are assigned a similarity value $sim \in [0..1]$, which is checked against a threshold value. Search results where *sim* is less than the threshold are removed from the set of all results.

CHAPTER 3

CodeKōan Search Engine Implementation Overview

The CodeKōan search engine is implemented as a distributed web service. The bulk of the search engine is implemented in the purely functional programming language Haskell. The RabbitMQ - messaging system is used for safe and reliable message passing between the search engine's components that may be distributed over a cluster of machines. This chapter will give a broad outline of Codekōan's architecture and implementation.

3.1 Programming Languages, Tools and Frameworks

This section briefly introduces the main tools and frameworks, that were used to implement the CodeKōan search engine.

3.1.1 Programming Languages: Haskell and Javascript

As mentioned previously, the entire backend-code for the CodeKōan search engine was written in Haskell. It is a highly performant, compiled programming language with a strong, static type system.

The web interface to the CodeKōan search engine was implemented using javascript.

3.1.2 The RabbitMQ messaging system

To schedule tasks and distribute work in the backend, the RabbitMQ messaging system is used. RabbitMQ is a highly performant, widely used messaging platform. It allows routing, sending and receiving messages between a large set of participants that may either be located on a single machine or distributed over a network. The software is highly available and has a number of favourable properties in case a worker process dies.

Increased Fault Tolerance
Besides allowing comparatively simple abstraction over the distribution of worker processes in a network, a major advantage of using RabbitMQ is increased fault tolerance. If a worker process unexpectedly dies or is disconnected from the network for some reason, the requests, that it should have processed are not lost. Instead, all messages are stored on the RabbitMQ host and will be available for later processing when the worker process is available again. This means, that temporary failure / disconnection of a worker process does not automatically result in information loss and invalid search results.

The RabbitMQ Message Routing Model
RabbitMQ implements the Advanced Message Queuing Protocol (AMQP). The main idea of AMQP is to send messages from *producers* to *consumers* via a set of *queues*, that temporarily store messages[2]. The advantage of using queues is that a producer can send messages as a fire-and-forget operation, and failure of producer or consumer will automatically not automatically result in message loss.

RabbitMQ extends this model by introducing *exchanges*. These exchanges allow distributing a message originating from a producer to multiple consumers by routing the message to different queues based on message meta data. When using RabbitMQ, messages should always be sent to an exchange and can only be received from a queue. RabbitMQ defines four types of exchanges:

Direct Exchange: This is the simplest type of exchange, it writes any message it receives to a given queue

Fanout Exchange: This type of exchange copies each received message to all queues in a given set

Topic Exchange: All RabbitMQ messages carry a string called a routing key. This exchange uses matching on the routing key to decide, into which queue a message is stored

Headers Exchange: This type of exchange uses other message meta data for routing messages

3.1.3 The PostgreSQL database

PostgreSQL is a mature, performant SQL database. It is used for all data storage needs of the CodeKōan search engine. All Stack Overflow-answers are indexed in this database. Furthermore the database holds user data for the CodeKōan web interface.

For interaction with the database a library, which guards against SQL-Injection vulnerabilities by escaping textual user input, is used.

3.1.4 The Yesod Webframework

Yesod is one of the most widely used Haskell web frameworks [28]. It is based on the performant warp web server and extensively uses Haskell's meta-programming and code generation abilities.

3.1.5 HSpec for Unit Testing

HSpec is a Haskell testing framework, which integrates other Haskell software testing libraries. Two such libraries, that HSpec integrates are HUnit, a library for writing classic unit tests, and QuickCheck[9] which enables property based testing. Especially property based testing with QuickCheck has proven to be very useful in the implementation of this work, since property based testing allows to find edge cases of functions which the developer did not think of.

Property based testing works by randomly generating input values to functions. Properties of functions, that must hold regardless of input values can then be specified. A QuickCheck property will fail if an input which violates a given property could be generated. An example of such a property is that lookups in a GST T must always find each of the strings in the underlying set $\mathscr{S}$ for which it was created.

3.2 Architecture and Components

This section outlines the software architecture of the CodeKōan search engine.

An overview is given in figure 3.1. This figure shows how information flows through CodeKōan's backend. The following sections will outline how query documents are submitted and routed through the backend, resulting in a useful reply.

One of the key ideas for the following services is that each service that is linked to RabbitMQ always return a response, even if an exception occurs. That way, exceptions can be traced for debug purposes.

3.2.1 JSON - Message Passing

All messaging between service components is done via JSON messages that are transmitted via either RabbitMQ or HTTP requests. The information of these messages is wrapped into a JSON object. The wrapper object contains meta information about a message such as the sender, content type, send time, etc. This meta-information is currently mainly used for debugging and validation purposes in the pipeline.

3.2.2 Submitting Messages: The Web Interface

The entry point to the CodeKōan system for most users will be the web interface. At the time of submission of this thesis, the web interface is available to the public on CodeKōan's website `https://codekoan.org/`.

The web interface allows users to enter a query document and automatically detects the language of the submitted code. This code is then submitted to the RmqInjector service outlined in the next paragraph.

Screenshots of the web interface are shown in figure 3.2.

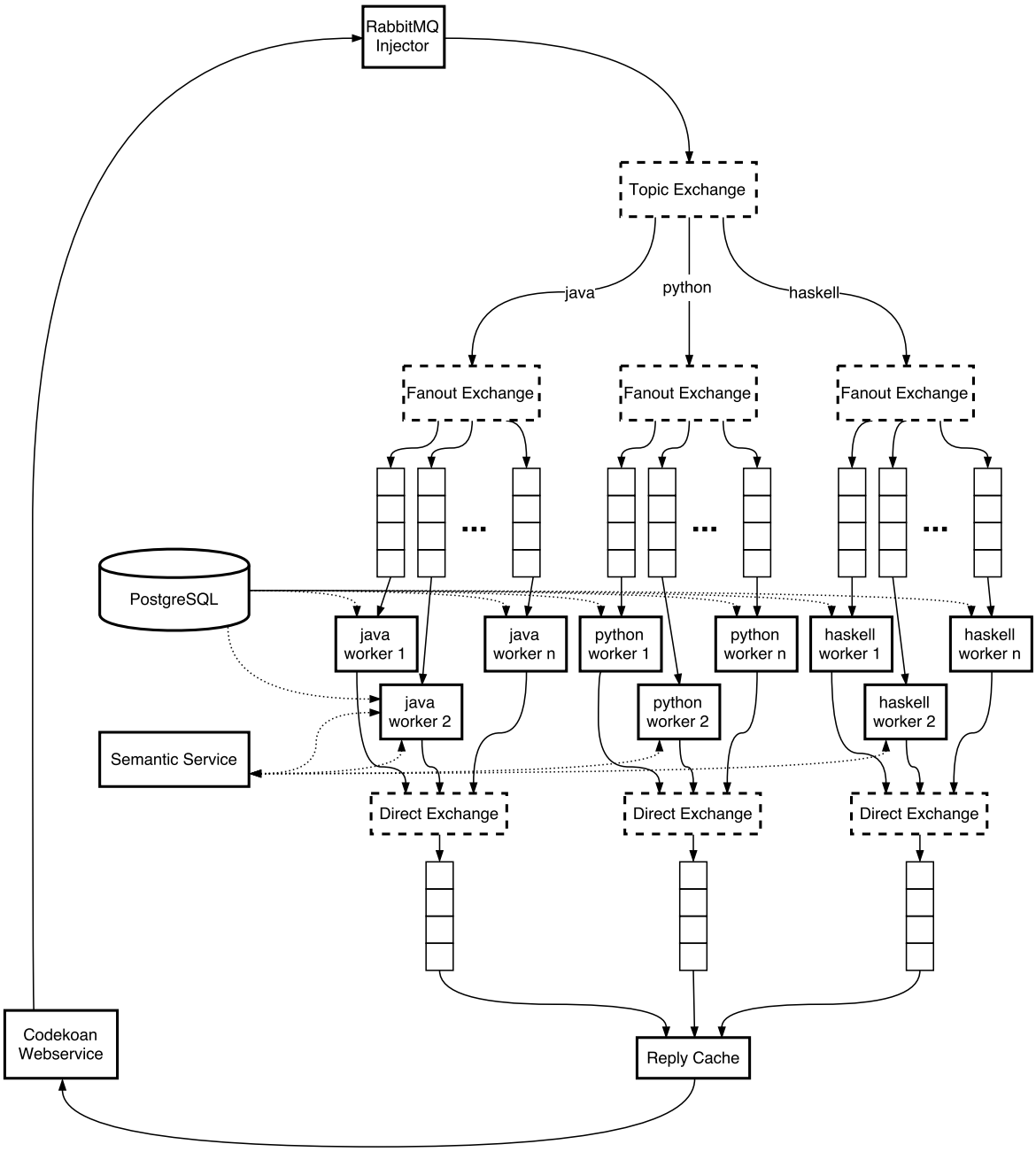


Figure 3.1: Overview of CodeKōans architecture. RabbitMQ exchanges are in dashed boxes, Haskell processes are represented by boxes with solid borders. Some edges in the graph (from PostgreSQL and the Semantic Service) have been omitted for a clearer layout.

3.2.3 Preprocessing, Validity Check and Enqueuing

RmqInjector is a webservice with a REST endpoint, to which all queries to the CodeKōan search engine have to be submitted as a JSON object. This object contains settings, which are to be used in the search pipeline outlined in chapter 2. Furthermore this object contains the text of the query document as such and the programming language that the query document's source code is written in.

The RmqInjector validates this JSON input object and if the settings are valid, the programming language is recognized and the size of the query document is limited to a reasonable size, an ID is assigned to the given query. This ID is returned as a response to the original HTTP request with which the query was submitted. The validated query is then wrapped into a JSON message object as outlined in section 3.2.1. The completed query message is then submitted to a RabbitMQ topic exchange with the query language as the routing key. This exchange in turn routes the query message to one of currently three fanout exchanges (one for each of the three recognized programming languages Java, Python and Haskell).

Recall that a fanout exchange copies each message it receives to each of the output queues and exchanges it is linked to. In case of the CodeKōan backend, these outputs are n queues. Each of these queues is subscribed to by a worker process. Each of these worker processes performs the same analysis task for the query document on a different part of its programming language's index. The function of these worker processes is outlined in the next section.

This routing setup is advantageous because RabbitMQ allows to abstract over the fact that an arbitrary number of worker processes may be distributed over a large number of physical machines without further configuration efforts.

3.2.4 Language Specific Worker Processes

As described in section 2.4.3, the index for a programming language can be split into multiple parts without yielding different search results, provided search is performed on all parts of the index. This is used in the CodeKōan backend by only constructing a part of the index in each worker process. Each of the worker processes operates on a subset of all code fragments in the PostgreSQL database. The workers build a suffix tree and token N-Gram bloom filter only for their individual subset of code fragments. They then perform pipeline algorithm outlined in chapter 2.

The only part of the pipeline that is currently not wholly performed in the worker processes is word similarity analysis as outlined in section 2.6.2. The reason for this is that at the core of the described identifier word similarity filtering is the capability to assign a similarity score between 0 and 1 to two bags of words. This is independent of programming language and is therefore encapsulated into an independent micro service, that is communicated with via a REST interface. This micro service will be discussed in the next section.

The result of processing by a worker process is a partial reply that is sent to a language - specific RabbitMQ direct exchange, which writes it into a queue.

3.2.5 Semantic similarity analyzer

The semantic similarity analyzer preprocesses a large corpus of source code files in all of the three programming languages. This preprocessing consists of extracting all identifiers

in the corpus' source code files and splitting them up into individual words as described in section 2.6.2. These identifiers are then processed into term vectors that can be used for similarity analyses.

This service can be queried via a REST interface with two bags of words in a json-object and will return a similarity score between 0 and 1 based on the cosine-similarity measure.

3.2.6 A cache for search results

The final step in the backend is a reply cache, that waits for the partial results of all worker processes. Users can query this service via a REST-interface for replies to a given query ID as assigned by the RmqInjector service, that was described in section 3.2.3. The JSON response to a request for replies to a given query id will either contain:

- a full search result, or
- a partial search result with the number of outstanding parts, or
- information on possible exceptions that may have occurred.

3.2.7 Displaying Results in the Web Interface

Search results are displayed in a javascript application, that is part of the web interface. This application shows all search results in a human readable way. Furthermore, the display-application gives users the possibility to submit their own feedback to individual search results. This feedback could later be used to construct human computation systems for enhanced data processing capabilities (see section 6.2.2.3).

Figure 3.3 shows a screenshot of a search result in the CodeKōan web interface.

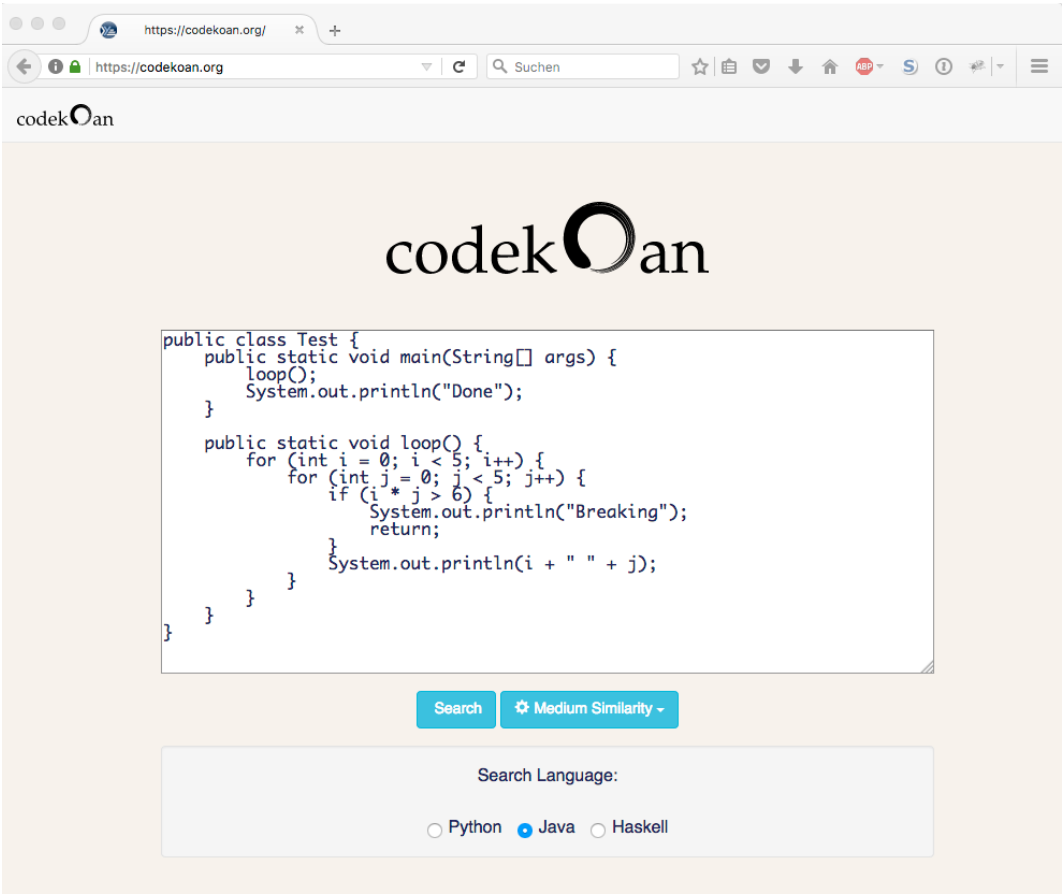


Figure 3.2: A screenshot of the CodeKōan search interface. A query document is entered in the central text-area and users have a selection of three parameter presets for search sensitivity. Sensitivity selection is made in a drop-down menu to the right of the search button.

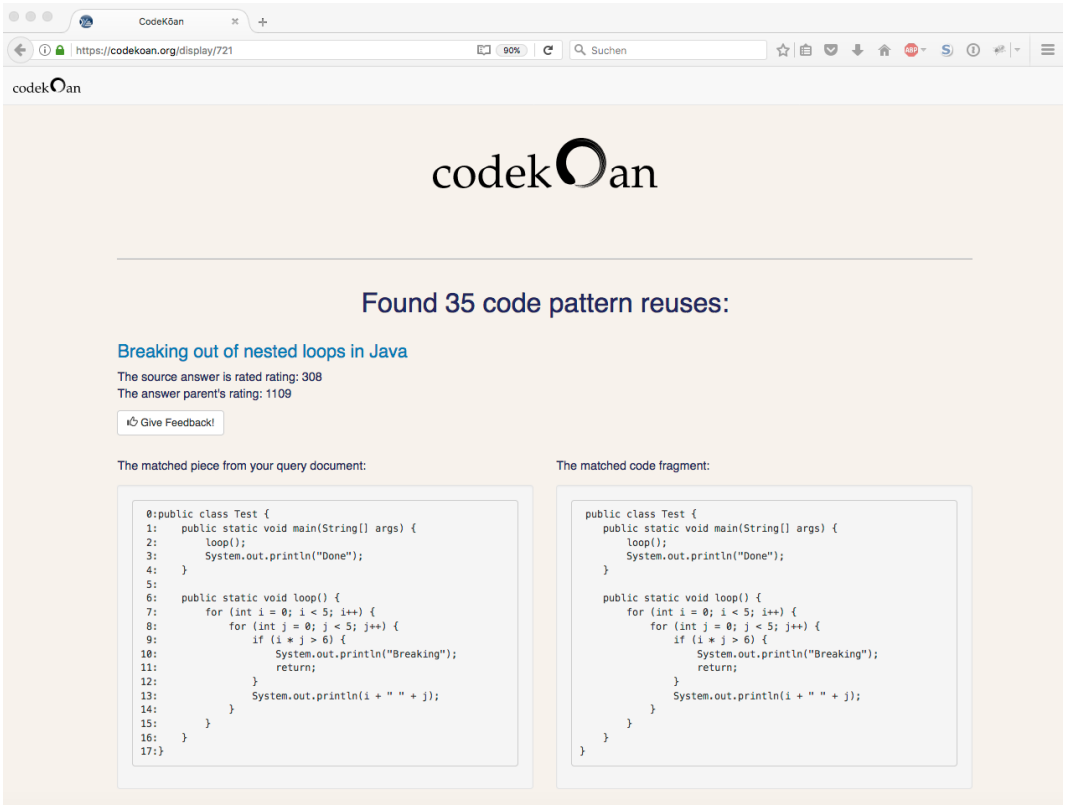


Figure 3.3: A screenshot of the results for a search query. The headline for each search result is the title of the Stack Overflow Q&A thread from which an answer fragment originates. The source code in the box(es) on the right is from the query document D while the code in the right box shows the matched answer fragment $A^{i,j}$.

CHAPTER 4

Validation and Parameter Optimization

This chapter describes the methods by which the correctness of the search pipeline described in chapter 2 was validated. This chapter comprises of two parts. Firstly, basic validation of all of the parts of the CodeKōan pipeline are discussed. This validation was performed for all three supported programming languages (Java, Python and Haskell). Secondly, good parameter settings for different search goals will be discussed.

4.1 Hand Crafted Examples

An important step in the implementation of any algorithm is to show that the algorithm's theoretical goals are fulfilled by the developed software. In the case of the proposed pipeline this is done by running it with a set of hand crafted examples and comparing the expected output with the actual results.

For the sake of brevity most complete listings of handcrafted examples are deferred to the appendix in section 7.2 and only pertinent examples are included here. Complete listings of examples are often redundant because some examples are very similar for all of the three programming languages and some of the used examples are often fairly long.

This section will be subdivided by properties that the pipeline must fulfill for each of the three languages. Unless explicitly stated otherwise, these properties are implemented as unit tests for the CodeKōan base library or its language specific plugins.

4.1.1 Always Find Exact Copies

The first important property is that the CodeKōan pipeline must be able to always find verbatim copies of indexed code fragments. This property was implemented as a unit test. Sets of handcrafted examples were indexed into test-indices that were further extended by

```
public List<User> getUser(int userId) {
    try (Connection con = DriverManager.getConnection(myConnectionURL);
         PreparedStatement ps = createPreparedStatement(con, userId);
         ResultSet rs = ps.executeQuery()) {

         // process the resultset here, all resources will be cleaned up

    } catch (SQLException e) {
        e.printStackTrace();
    }
}

private PreparedStatement createPreparedStatement( Connection con
                                                  , int userId
                                                  ) throws SQLException {
    String sql = "SELECT id, username FROM users WHERE id = ?";
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setInt(1, userId);
    return ps;
}
```

Figure 4.1: A Java-code sample from Stack Overflow showing use of Java's try-with-reuse language feature.

500 randomly generated token strings. The verbatim copied code samples were correctly identified 100% of the time, as expected.

4.1.2 Always Detect Point-wise Changes / Levenshtein Distance

Testing for correct recognition of query documents that are within a levenshtein distance d of an indexed pattern was again performed using a language independent QuickCheck property. The results of that property show that token-strings that are within a certain levenshtein distance of indexed token strings are always correctly identified.

An example of such a correct recognition is the following Java - query document D:

```
public static void main(String[] args){
  System.out.println("testing");
}
```

The above code's token string is within levenshtein distance two of the following Java code (note the concatenated string):

```
public static void main(String[] args){
  System.out.println("testing" + " software");
}
```

4.1.3 Detecting Code Reordering

An important property of the CodeKōan search pipeline is that it recognizes reuse of code patterns after reordering. Consider the Stack Overflow code pattern shown in figure 4.1.

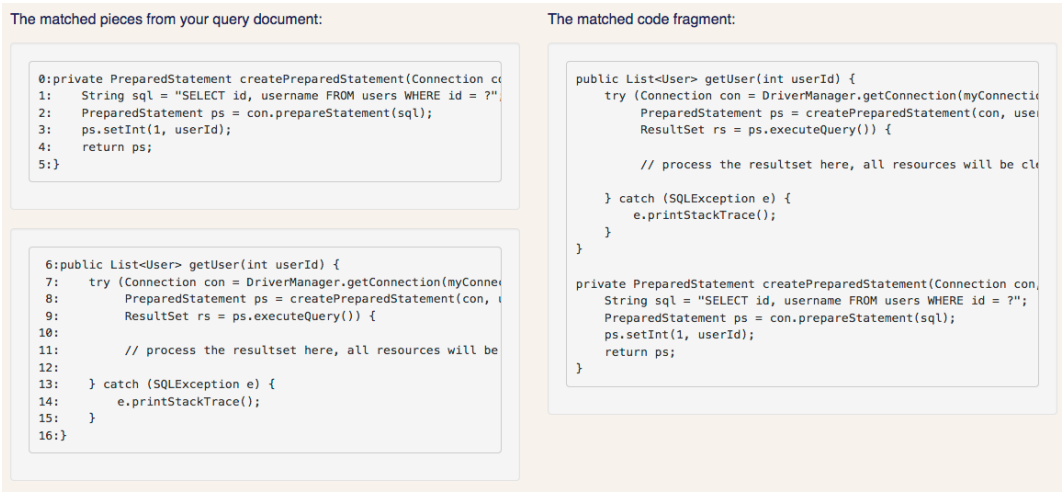


Figure 4.2: A search result with a query that contains the code in figure 4.1 with the order of the two methods switched.

This code pattern illustrates using the try-with-resources feature that has been added to the Java language in its 7th version.[1] It has already been established in section 4.1.1, that searching for this pattern in CodeKōan leads to correct identification of this reuse. What happens when the method `createPreparedStatement` is put before `getUser`? In this case the CodeKōan search engine returns a search result consisting of two alignment matches that contain the two functions as shown in figure 4.2

4.1.4 Detecting Code Insertion

An important part of many code patterns like the one in figure 4.1 are spots at which code is meant to be inserted. Such spots are often indicated by comments, as is the case for the given example.

What happens when if arbitrary code is inserted into such a spot as in the example in figure 4.3.In such cases CodeKōan will yield one search result consisting of two alignment matches: one containing the matched code before the insertion and one containing the rest of the pattern. An example how the document sides of these alignment matches look like in the web UI is given in figure 4.4.

Caveats

This property is, however, subject to some caveats that should be kept in mind. Firstly, the part of a code pattern before and after an insertion must be longer than the minimum match length l_{min} of tokens. The minimum match length is a parameter to the search pipeline's alignment step outlined in section 2.5.1.

Furthermore an insertion of code into a pattern can break the block structure of the surrounding code. Such would for example be the case in Java code if an insertion consisted simply of three curly braces „}}}“. In such a case, the block relationship `blockR` (outlined

[1] official documentation: http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

```
public List<User> getUser(int userId) {
    try (Connection con = DriverManager.getConnection(myConnectionURL);
         PreparedStatement ps = createPreparedStatement(con, userId);
         ResultSet rs = ps.executeQuery()) {

         // Code we inserted for this example:
         while(rs.next()){
             String userName = rs.getString("USER_NAME");
             System.out.println(userName)
         }

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Figure 4.3: The first of the two methods in the code pattern from figure 4.1 with an insertion of code

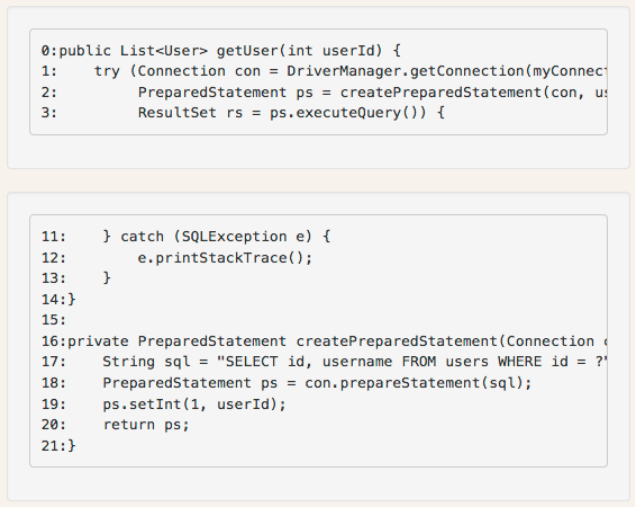


Figure 4.4: Document side of a search result after insertion into a code pattern shown in the CodeKōan Web-UI.

in section 2.6.1) between code before and after the insertion point would differ between query document and pattern. Therefore a search result with an alignment match before and after the insertion would not be possible.

There is a further, third, caveat that is a result of the pipeline's block structure filtering step. This pipeline step necessitates that any part of a pattern is only included in at most one alignment match in a search result. Consider a hypothetical code pattern „*ABC*"; search results with alignment matches that encompass *AB* and *C* or *A* and *BC* would be possible. However, a single search result containing two alignment matches with both pattern-sides *AB* and *BC* would be removed from the result set. The alignment algorithm, however, is greedy. Therefore a query document *ABBC* would produce a search result with alignment matches (AB, AB) and (BC, BC), which would lead to no result being returned. A more concrete python code pattern illustrating this point is presented in the appendix in section 7.2.2

4.1.5 Renamed Identifiers

The examples up to this point have illustrated all parts of CodeKōan's pipeline algorithm, save for identifier word similarity filtering. The impact of this filtering step will be shown with the help of the following piece of Haskell source code from a Stack Overflow answer:

```
trySql :: Connection -> (Connection -> IO a) -> IO a
trySql conn f = handleSql catcher $ do
    r <- f conn
    commit conn
    return r
  where catcher e = rollback conn >> throw e
```

It is apparent that the shown code deals with safely executing SQL statements. The following will illustrate the effects of some similarity threshold settings (as outlined in section 2.6.2.

Similarity Threshold: 0.0

If all identifiers in this piece of code are replaced with „`x`", all but the lowest similarity thresholds $sim_{threshold}$ lead to no search result being found. When replacing every identifier with `x`, results are only found with a similarity threshold of 0.

Similarity Threshold: 1.0

The opposite holds when using a similarity threshold of 1.0. In this case, the bag of words of all words in identifiers in a piece of code has to be identical. In the case of the given Haskell example, this bag of words is: {`a (2x), catcher (2x), commit, connection (2x), conn (4x), e (2x), f (2x), handle, io(2x), r (2x), return, rollback, sql(3x), throw`}. Note, that this only puts a requirement on the bag of words and has no influence on the placement of those words. So for example the following code would still produce a perfectly valid search result since its identifiers still

are from the same bag of words:

```
trySql :: Connection -> (IO -> IO catcher) -> Rollback a
commitSql conn f = throwE a $ do
    r <- f conn
    try conn
    return r
  where catcher sql = connection conn >> handle e
```

Less Extreme Threshold Values

While thresholds of 0 and 1 illustrate the behaviour of CodeKōan's identifier word similarity filtering pipeline-step, they are not very useful in practice. A threshold of 0 is equal to not performing the pipeline step at all; a threshold of 1 is only needed for finding verbatim reuse.

More nuanced $sim_{threshold}$ values are used in practice to limit search results to recognizing usage of patterns that are topically similar to the code in query documents.

4.2 Parameter Choice

This section describes the parameters used for the CodeKōan search engine and outline the reasons for why parameters were chosen this way.

4.2.1 Syntactic Pipeline Parameters

The minimal alignment match length l_{min} and the coverage percentage p_{cov} are important parameters for the CodeKōan pipeline because they determine the set of alignment matches that the semantic pipeline steps can operate on.

Minimal Alignment Match Length and Levenshtein Distance

When choosing an appropriate l_{min}, a tradeoff has to be made between finding as many alignment matches as possible, and increased resource demands. Such an increase happens because every pipeline step after alignment match has to deal with more alignment matches, that also often carry less information. This means that smaller l_{min} lead to a more sensitive, slower search engine.

An other factor to keep in mind is that shorter l_{min} can make searching with higher levenshtein-distances redundant. If a part of a query document is similar to an indexed code fragment except for a deviation of a limited levenshtein-distance, reuse of this code fragment can still be discovered with search settings that have $l_{min} = 0$. This is achieved by choosing a sufficiently small l_{min}, so that a search result consists of multiple alignment matches. These alignment matches capture all regions of the code fragment except for the different tokens that would otherwise detected with levenshtein-search. An example of this is given in the appendix in section 7.2.3.

Generally, higher levenshtein-distances greatly increase the resources used by the algorithm therefore a parameter set with small or zero levenshtein-distance should be preferred when working with large indices.

Coverage Percentage

The coverage percentage p_{cov} serves a very important purpose in the search pipeline: it limits the set of search results to actual potential reuses. If p_{cov} were chosen to be very low, e.g. below ten percent, then reuse a single for-loop or fold would be identified as reuse of a large module or class. On the other hand, excessively high values for p_{cov}, i.e. greater eighty percent, would lead to being able to find only nearly-verbatim reuses (on a token-string basis).

Sum of Match Lengths

This parameter constitutes a minimum for the sum of the lengths of all aligment matches in a search result. The basic idea behind this parameter is the same as behind the minimal alignment match length. The difference is that short lengths for individual alignment matches make sense while whole search results tend to be of higher quality if they are somewhat longer. The reason for this is that very short search results often encapsulate micro-code fragments like e.g. one or two print-statements that don't contain a lot of information.

4.2.2 Semantic Pipeline Parameters

There is not much to say on parameters for the semantic pipeline steps that hasn't already been discussed, therefore this section only gives a concise summary. The only choice with respect to the block structure filtering step is whether to enable it or not. Since this step ensures that parts of a pattern can only occur once per search result, not enabling this step generally leads to nonsense-results.

Practical experimentation has shown, that settings for the identifier word similarity threshold $sim_{threshold}$ are sensibly chosen between 0.2 and 0.9.

4.2.3 Parameter Settings in the CodeKōan Search Engine

The CodeKōan search engine offers users three parameter settings: „High Similarity“, „Medium Similarity“ and „Low Similarity“. This is done because these settings cover most use cases and offer better user-experience. Additionally, allowing users to set their own parameters could lead to search queries that put excessive strain on the search engine, especially if the amount of users were to grow very large.

These settings were chosen based on the discussions in this chapter and large amounts of practical experimentation and optimization. They are largely the same for every language except for Haskell which has a shorter minimal match length on the „High“ and „Medium“ settings.

High Similarity

The „High“ setting aims to produce search results, that very likely constitute reuse of a code pattern or even verbatim copies. Therefore the following parameter-set was chosen:

Minimal Match Length: 15 (10 for Haskell)

Coverage Percentage: 80%

Levenshtein-Distance: 0

Similarity Threshold: 0.6

Block Filtering: enabled

Sum of Match Lengths: 40 (30 for Haskell)

Medium Similarity

This is the default setting of the CodeKōan search engine and represents a more relaxed notion of similarity. The purpose of this setting is to recognize general reuse of code patterns. The „Medium“ setting finds more reuses than the „High“ setting while also finding a limited amount of arguably spurious results. It is the author's opinion that this setting makes a reasonable tradeoff between sensitivity and specificity.

Minimal Match Length: 15 (10 for Haskell)

Coverage Percentage: 70%

Levenshtein-Distance: 0

Similarity Threshold: 0.3

Block Filtering: enabled

Sum of Match Lengths: 30 (20 for Haskell)

Low Similarity

The most tolerant of the three parameter sets is the „Low“-set. This parameter set is the only one, that completely disables the identifier word similarity filter. Searching with these parameters identifies a wider set of code patterns as potential reuses and serves a more exploratory purpose.

Minimal Match Length: 10

Coverage Percentage: 50%

Levenshtein-Distance: 0

Similarity Threshold: 0.0 (disabled)

Block Filtering: enabled

Sum of Match Lengths: 25 (20 for Haskell)

CHAPTER 5

Empirical Analysis of Public Repositories

This chapter presents results of an analysis of open source projects hosted on GitHub. These results were gathered to gain insight into the day-to-day work of programmers and to show the capabilities of the CodeKōan pipeline.

5.1 Selection of Repositories

The analyses in this chapter were performed on a selection of seventeen repositories in which Java was the primary programming language used. Eight of these repositories were either android apps or contained android components. The selected repositories are listed in table 5.1.

5.2 Tagging Repositories

The idea of the first presented analysis is to generate descriptive tags for whole source code repositories from CodeKōan search results and tags of questions on Stack Overflow. The generated tags are always tags of Stack Overflow questions. This analysis aims to show that CodeKōans results manage to capture the topic of analyzed source code. In this way this analysis also serves as further validation of the CodeKōan pipeline.

5.2.1 Generating Tags

Unless stated otherwise the indexed code fragments for the CodeKōan search engine in this section are all code fragments in all answers to questions tagged 'java'. Recall that CodeKōan search results are always sets of alignment matches between a single query document and a particular indexed code fragment.

Name	Brief Description
AlgoDS	Reference implementations of data structures and algorithms
duckduckgo-android	Android app for the search engine duckduckgo
GraphJet	Library of graph algorithms developed by Twitter
HWTDresden	Android developed by HWT Dresden
humanize	Library for formatting dates and timestamps
Jest	Elasticsearch client
maven-shared	Shared components of the apache maven project
open-keychain	PGP-Implementation for android
opennlp	Natural language processing library
stila-android	Android app developed by the university of Munich
Studentenportal	Android app for students of Klagenfurt University
testng	Popular unit testing framework
Tiny	Image compression framework with android components
tomcat	Popular web server developed by Apache
TumCampusApp	Android app developed by students of TU Munich
unisanno-android	Android app developed by University of Sannio, Italy
zxing	Android barcode recognition framework

Table 5.1: Names and short descriptions of analyzed Java repositories

For the purpose of this analysis it is reasoned, that answers to questions on Stack Overflow are related to the same topics as the questions they answer. Therefore it is argued, that a questions tags also apply it's answers. By this reasoning individual search results are tagged with the same tags as the parent question of the code fragments Stack Overflow answer.

Let $res(doc_l, A^{i,j})$ be the number of search results with a document-side in a document doc_i and a pattern-side in a Stack Overflow answer $A^{i,j}$. Furthermore let $tags(Q^k)$ be the set of tags for question Q^k, which is answered by $A^{i,j}$. Then $tags(A^{i,j})$ is defined to be equal to $tags(Q^k)$.

Now let P be a software project made up of a set source code documents $\{doc_l | l \in 1..n\}$. The set T_P of tags for the repository P is defined as

$$T_P = \{t | t \in tags(A^{i,j}) \wedge res(doc_l, A^{i,j}) > 0 \wedge doc_l \in P\} \tag{5.1}$$

This is the set of all Stack Overflow tags that belong to at least one answer that occurs in the CodeKōan search result for any doc_l of the analyzed project P.

5.2.2 Scoring Tags

Above, the set T_P of all tags for a source code repository P has been defined. A problem with this set of tags is that there is no ranking for significance of individual tags t. Such a ranking is necessary in order to determine the tags that actually describe the code of a project. In the following a scoring mechanism for tags is described that aims to score more relevant tags higher, and tags that are less descriptive lower.

Statistical Observations

This discussion is based on determining scores for individual tags t independent from other tags. An important observation is that the probability $p(t)$ of any randomly drawn answer A^l for language l having tag t can be observed from tag frequencies on Stack Overflow. It is simply the number of questions with the language tag (e.g. 'java') and tag t divided by the number of answers tagged with the language tag (e.g. Java). In other words, whether or not an arbitrarily chosen answer for a language l has tag t is a Bernoulli-experiment.

From this it follows that the number k of observed answers with tag t among n randomly drawn answers for language l from Stack Overflow is binomially distributed with parameters n and $p(t)$ as defined above.

Statistical Scoring

The CodeKōan search engine will always find a number of search results from answers that have tags unrelated to the topic of the given source code. This happens because many answers on Stack Overflow contain source code following general patterns like iterating over collections despite its tags being very specific to a topic like the Facebook graph-API. Therefore there is always a certain probabilistic component to the found tags.

The aim is to generate tags for repositories P that are actually relevant and don't arise by a statistical anomaly. This is done by comparing the frequency of a given tag t in search results T_P for a project P to the expected frequencies if the search results origin answers were drawn at random as described above. Under this assumption of completely random search results it is already known that the amount k of results with tag t among all n search results is binomially distributed. A score for tag t is then derived by calculating the value of the cumulative distribution function of the $(n, p(t))$ binomial distribution for the observed value k. The higher this value is, the lower the probability of observing an even higher k with random search results becomes.

The idea behind this score for a tag t is, that a value of 0.5 is exactly what would be expected if results were chosen completely randomly. Increasingly larger values are interpreted as an indication, that the frequency of a tag t is not the result of random chance but rather expresses a genuine deviation from a random search.

Practical Concerns

The density function of the binomial distribution for the parameters n and p is given by $\binom{N}{k} p^k (p-1)^{n-k}$ [21]. Since the the binomial coefficient $\binom{N}{k}$ is computationally expensive to calculate the binomial distribution is approximated using a normal distribution with $\mu = np, \sigma^2 = np(1-p)$. This approximation is known as the de Moivre - Laplace approximation [12], which is invalid for binomial distributions with small (< 10) n. Therefore those tags that occur in less than ten answers in Stack Overflow are excluded.

There is a small number of cases in which multiple tags can have a score of 1.0. Since a strict ranking is desired, a small correction factor $0 < f \leq 0.002$ is added to each tags score which is simply $p(t) * 0.002$. The result of this is that in case of two tags achieving a score of 1.0, the one that occurs more frequently on Stack Overflow is given a slightly higher score. This means that the score for a tag is in the interval $]0; 1.002[$. The language tag for a project

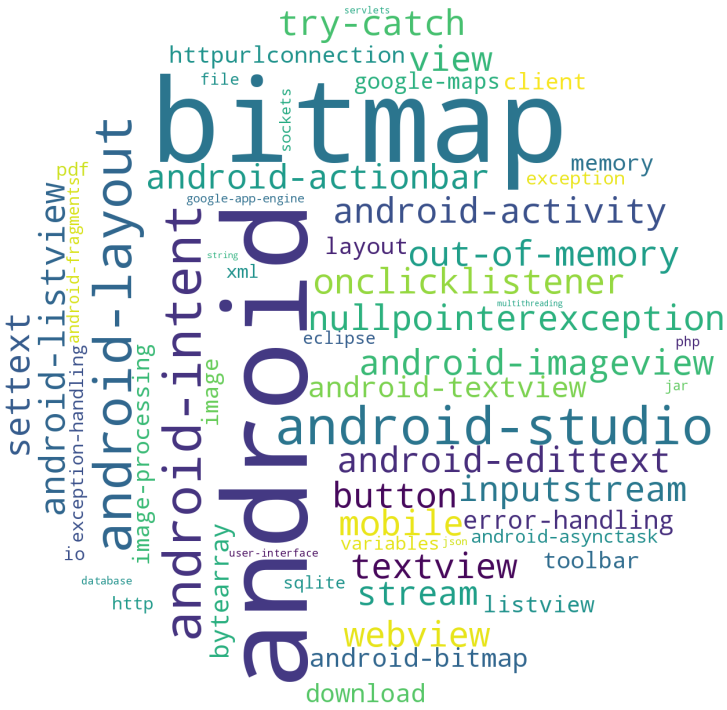


Figure 5.1: A tag cloud for the Tiny repository, an image compression library with android components. Note that 'android', and 'bitmap' are the most prominent tags.

P is always excluded from evaluation because it would always get the maximal score of 1.002.

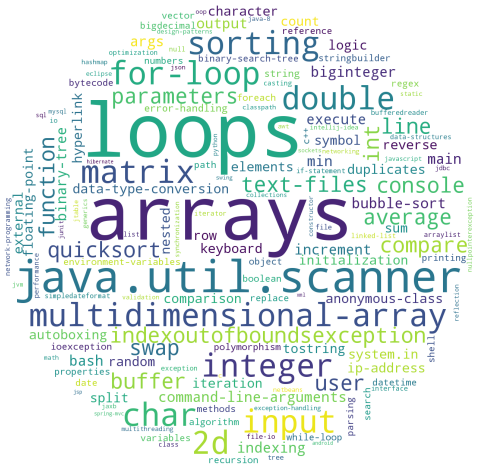
5.2.3 Results

The primary result of this analysis are tag clouds[1], which are a method of displaying weighted tags as shown in figure 5.1. In tag clouds, each word is a single tag occurring in the search results for a project P. The pertinent feature of tag clouds are the size of the words. The larger a word is written, the higher its significance score as defined above. Words are colored in order to make them more discernible, otherwise the color of individual words has no meaning.

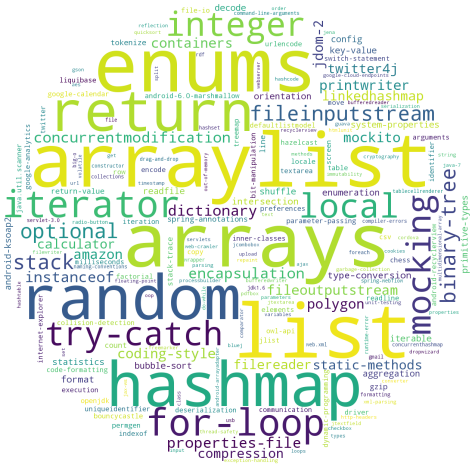
For the analyzed repositories presented in this section the medium search setting as outlined in section 4.2.3 was used unless stated otherwise.

Results for selected repositories are shown in figures 5.2 and 5.3. What is noteworthy is, that usage of android in repositories is always clearly identified.

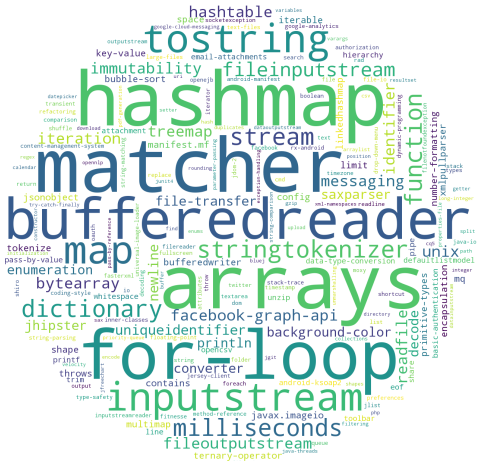
[1]The tag clouds in this thesis were generated using the wordcloud python script by Andreas Mueller



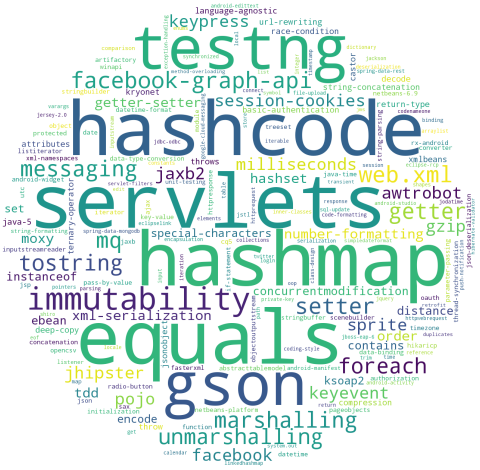
(a) AlgoDS: A package with reference implementations of algorithms. Primarily written by students. Note the prevalence of basic data structures like arrays or techniques for iteration.



(b) GraphJet: A library for graph algorithms developed by twitter. Like AlgoDS, this repository is primarily tagged with fundamental data structures and iteration.



(c) OpenNLP: A library for natural language processing. The most significant tags for this repository are likewise fundamental operations but also string and text specific ones like 'stringtokenizer'.



(d) Tomcat: This webserver repository is most significantly tagged with 'servlet' and also a number of other tags that are relevant for webservers like 'marshalling' or 'unmarshalling'. A prominent tag is also 'testng' which is a testing library.

Figure 5.2: Tag clouds for some projects that don't use large frameworks.

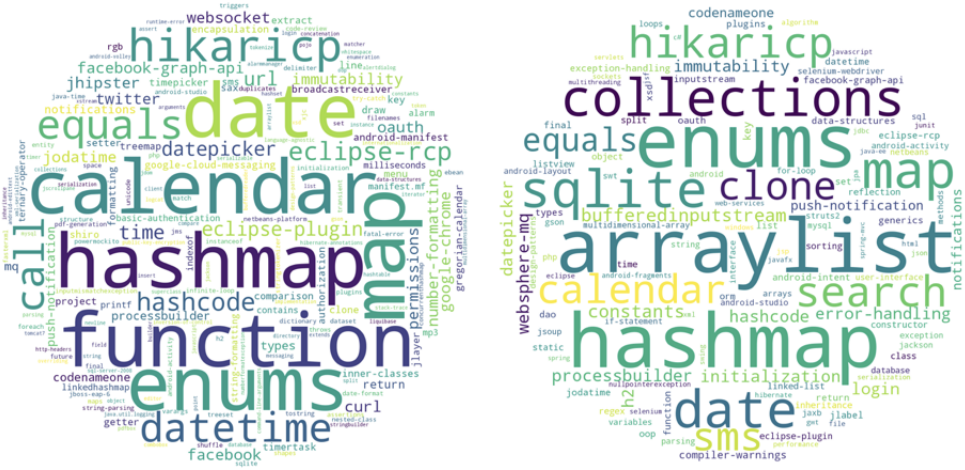
(a) Duckduckgo-Android: The android app for the search engine duckduckgo.

(b) TumCampus: An android app for TUM students with contributions from TUM students.

(c) Uni Sannio: An android app for students of the university of Sannio, Italy.

(d) Open-Keychain: A PGP implementation for android

Figure 5.3: Tag clouds for some android projects. Note that the tags clearly identify android projects as such.



(a) 'Medium' setting (b) 'Low' setting

Figure 5.4: Comparison of tagging-results for the Humanize repository. Humanize is a library for formatting dates and timestamps. The tags for date and calendar are prominent in both tag clouds even though the effect is weaker in the 'low' setting.

Tagging Without Identifier-Similarity

A concern when evaluating these results was, that it was unclear to what degree the identified tags arose as a side effect of the identifier-similarity pipeline step. To test this, search on selected repositories was run with 'low' pipeline-settings to compare with 'medium' results. Results are shown in figures 5.4 and 5.5.

These results indicate, that it is possible to generate descriptive tags for a repository using the CodeKōan pipeline and Stack Overflow data *on token-strings alone*.

5.3 Prevalence of Code Pattern Reuse

The CodeKōan search engine was furthermore used to study the degree to which source code in repositories is covered by code patterns from Stack Overflow. The results are shown in figures 5.6 and 5.7. What is shown in these chart is the degree to which repositories are covered (as defined in section 2.5.2) by code patterns. So for example coverage of 0.05 per project means that five percent of a projects total source code are covered by at least one search result.

Verbatim Reuse

An unexpected result is, that in none of the surveyed repositories verbatim reuse of code fragments could be identified. Verbatim reuse is defined as reusing a code example with the precisely same identifier names and comments and only allowing for formatting differences. This suggests that the proverbial „copying and pasting from Stack Overflow“ occurs less frequently than the popular hypothesis would suggest.

(a) 'Medium' setting (b) 'Low' setting

Figure 5.5: Comparison of tagging-results for the Duckduckgo-Android repository. Note that in both cases android tags are most prominent.

Discussion

The obtained results show strong discrepancies in code pattern reuse among projects. Professionally developed libraries without framework-use like GraphJet have less than five percent of reused code while some android apps consist of almost fifty percent reused source code.

The results suggest two general trends, firstly that student projects tend to show greater code pattern reuse than professionally developed projects and secondly that frameworks greatly increase the degree of code pattern reuse.

Projects that are developed by students are consistently among the ones showing the highest degree of code pattern reuse both in android apps and non-android projects. For example both the AlgoDS and GraphJet packages develop algorithms and have limited dependencies. However GraphJet has a 3.46 percent of code pattern reuse while that number is 21.5 percent for AlgoDS. The same holds for android apps, in which category all the ones with the highest rates of reuse are developed by students.

A possible explanation for this is, that students tend to more frequently adapt patterns from Stack Overflow or other resources rather than writing them out by hand which may increase the rates of detection in students projects. An alternative hypothesis is that student code shows a higher degree of code duplication than code written by more experienced developers. None of these hypotheses is so far backed up by empirical work and these questions should be investigated in the future.

The second trend, that android apps show higher degrees of reuse, can be more readily explained. Android code follows a rather rigid structure. Much of the functionality of android apps is implemented by inheriting from standard classes like `Activity` or `Fragment` and overriding these classes methods. The way this is done is often syntactically very similar (on a token-string basis) to code on Stack Overflow. It is hypothesized, that this is because android strongly encourages developers to implement functionality in a certain way, and that CodeKōan recognizes these patterns.

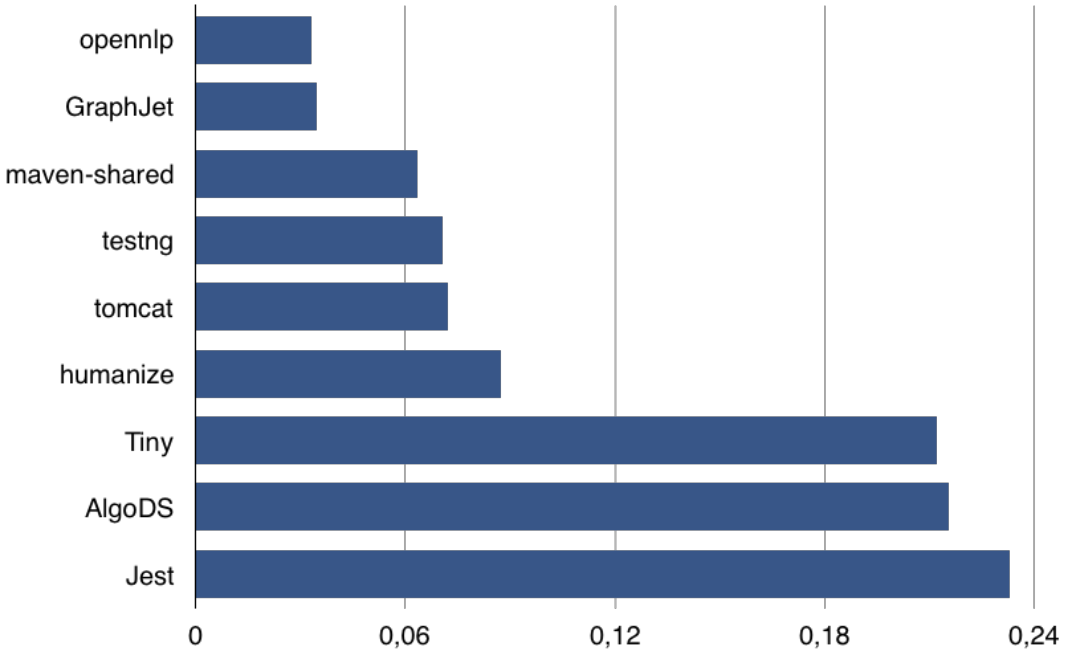


Figure 5.6: Total fraction of code that that is covered by at least one indexed code pattern.

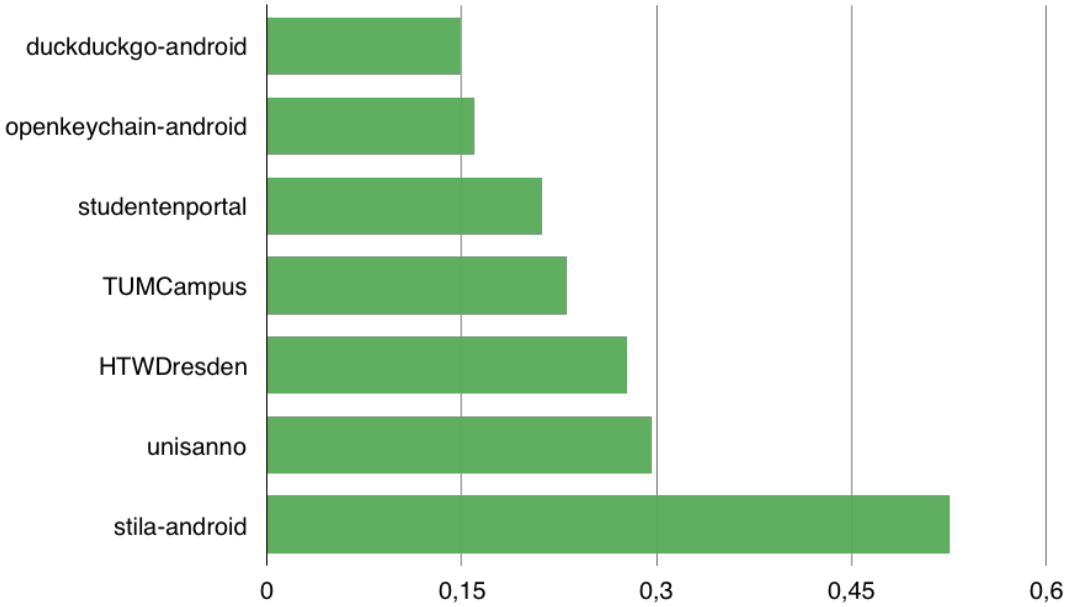


Figure 5.7: Total fraction of code that that is covered by at least one indexed code pattern. The repositories in this chart are all android apps.

CHAPTER 6

Outlook and Future Work

This thesis proposes a search engine for finding reuses of code patterns in computer programs. The proposed algorithm works as it is, but there are a number of worthwhile improvements and applications of the proposed method, that would go beyond the scope of this work. Improvements can be grouped into two categories. The first set of improvements focuses on enhancing the current implementation without modifying the existing feature set. The second category is consists of modifications and extensions of the existing search pipeline as described in chapter 2. These extensions will allow to further reduce false-positive results, and to better rank returned results.

This chapter will be concluded by outlining a number of applications for the developed method that could prove to be beneficial for both researchers and software developers.

6.1 Implementation Enhancements

This section discusses improvements to the implementation, that do not extend the set of features.

6.1.1 Security Considerations

The current webapplication for the CodeKōan search engine was not build with information security as a primary goal. Therefore there are still some vulnerabilities that need to be addressed.

Users Can See Other's Submissions
Currently it is possible for users to view the results of other user's queries. If users were to submit confidential and/or proprietary source code to the search engine this would be a problem. This issue could be addressed by giving each user an account, and logging which

user requested which query. The system could then be altered in a way that prohibited viewing an other user's search results. This could be achieved by using

DDOS Prevention
Furthermore the CodeKōan search engine is currently not secured against distributed denial of service attacks (DDOS). Should the search engine attract a wider audience the probability of such an attack is likely to increase. This should be addressed as CodeKōan gains popularity.

6.1.2 Efficiency Optimization

CodeKōan was primarily designed for a complete feature set and does not employ the fastest possible algorithms in all places. Since the CodeKōan backend may come under a lot of load, it is important, that queries are handled efficiently and index construction is as fast as possible.

More efficient index construction
To achieve quicker horizontal scalability for the CodeKōan search engine, index construction needs to be sped up. For this purpose the suffix tree construction algorithm should be improved. Currently suffix trees are generated by an algorithm that does not run in linear time like Ukkonen's algorithm. While this has no effect on space usage, this increases the time it takes to launch new instances of search - worker processes.

Detailed profiling
Additionally, there may be further performance gains achievable by profiling the CodeKōan backend in detail. The largest rewards for doing this are likely yielded by analyzing the algorithms for alignment and aggregation, which currently take up the majority of processing time.

6.2 Improvements to Presented Methods

In this section, extensions to the feature set of the proposed search pipeline are presented. These new features are grouped by whether they improve syntactic or semantic analysis.

6.2.1 Improvements to Syntactic Analysis

6.2.1.1 More Meaningful Tokens

The tokens, that are described in section 2.3 are very simple. They encompass, for example, numbers or single occurrences of an identifier. These tokens were chosen as "smallest units of meaning", because it would not be beneficial to analyze source code at a finer granularity (i.e. the character-string level) [18]. These token types may however prove to be too simplistic.

Many programming language have sets of syntactic constructs that are made up of several tokens. Such syntactic constructs are, for example, method declarations, if-statements,

assignments, function calls, type signatures etc. While these constructs are made up of different tokens, their structure is always fixed. Consider, for example, Java method declarations like this:

```
public final static int add(int a, int b)
```

These method declarations are always made up of:

- An (optional) access modifier like `public`, `private` or `protected` followed by
- (optional) non-access modifiers like `final`, `static` or `synchronized`,
- a return type,
- the method name and
- a list of parameter type declarations in parentheses, which are comma separated type-identifier pairs.

It is proposed, that such fixed constructs be dealt with as single tokens. So instead of converting the above method declaration into a token string `modifier, modifier, modifier, atomic_type, identifier, left_paren, atomic type, identifier, comma, atomic_type, identifier, right_paren`, a single token should be created. The reason for this is, that these constructs are only meaningful as a whole but not in parts. There can be no half method declarations and no half assignments. Fixed constructs are atomically meaningful and cannot be sensibly split up, therefore they should be treated as single tokens.

This leads to a problem, however, because fixed constructs with slight differences would be deemed do be completely different by our standard distance function *distt*. Recall, that this distance function defined in section 2.3 returns a value of $distt(a,b) = 0$ if $a = b$ and 1 otherwise. This problem will be addressed in the next section.

6.2.1.2 More Nuanced Distance Function

Degrees of Token-Similarity
A problem with the standard distance function *distt*, as defined in section 2.3, is that it only distinguishes between equal and unequal tokens. What it does not take into account is that some tokens are completely dissimilar while some are more similar. A more nuanced notion of similarity can be captured by modifying the distance function *distt*.

Such a modified distance function could appropriately reflect the degrees of similarity between tokens. For example in Java `long` and `int` are more similar than `bool` and `class`. Such a relation could be modeled by assigning a distance function *distt'* so that, e.g, $distt'(\texttt{long},\texttt{int}) := 0.5$ and $distt'(\texttt{int},\texttt{class}) := 0.8$.

Relation to Levenshtein Distance
An obvious question is how to integrate a distance function *distt* that yields value that are different from 0 and 1. Currently, the distance function integrates into the levenshtein framework by assigning a levenshtein distance of

- 1 for an insertion or deletion, and
- $distt(a,b)$ for substitution / equality.

This framework can still be used with modified distance functions. Levenshtein automata don't necessitate, that the distance used equals 0 or 1. It is probably practical, however, to use the value of an insertion/deletion as the maximum value of *distt* because any substitution is likely more "similar" than a deletion/insertion.

Context-Sensitive Distance Function
A further possible enhancement would be to make the used distance function carry some piece of state. This piece of state would have to be a third parameter to a new distance function $distt_s(a,b,s)$ which would yield a pair of $(d \in \mathbb{R}, s')$. This s' could then in turn be used as an input for the next call of $distt_s$.

By doing this, information about context could be used in the alignment step of the pipeline. This would add an ability to assign different token-distances, depending on the context, i.e. tokens that occurred before. A concrete example of this would be to be more lenient in arguments passed into functions compared to other regions of code.

6.2.1.3 Intermediate Language Compilation

As discussed in section 1.2.4.3, if a transformation of arbitrary source code f exists, so that $sem(f(C)) = sem(C)$, we can use that transformation to reason that if $sem(f(C_a)) = sem(f(C_b))$, then $sem(C_a) = sem(C_b)$.

Such a transformation is essentially what intermediate language compilation does. This process is used by many compiler to transform very high-level source code into a language with fewer low-level instructions. Examples of this process are compilation of Haskell into Core by GHC; compilation of Java, Scala and Clojure Code into JVM-Bytecode or compilation of .NET languages to the .NET common intermediate language.

This work could be extended by an intermediate-language compiler as a source-code normalization step. This means that any source code would be compiled into an intermediate language before being used for search or index construction. Search would work as usual, but not on token-strings of Java, Python and Haskell. Instead, the token-string of the intermediate language compilation result would be used.

This would give two great advantages:

- Code of multiple programming languages would become comparable and
- some language constructs could be normalized away in favour of others like substituting every for-loop with an equivalent while-loop.

Especially the second point is noteworthy, because it would make the syntactic steps of the search pipeline a lot more useful.

Of all the proposed extensions of the feature set of this work, this one is the most likely to yield great improvement but also the one that would take the most effort. Some of this work may be saved by using existing work on intermediate language compilation e.g. between .NET languages or JVM-based languges.

6.2.2 Improvements to Semantic Analysis

6.2.2.1 Dependency Graph Analysis

A promising technique for analyzing source code similarity is the comparison of program dependency graphs [18]. These graphs are an intermediate data structure generated from source code, in which relations between e.g. declarations of variables and their uses or loops are expressed in a directed graph.

With these graphs, detecting similarities between pieces of source code can be reduced to solving the subgraph-isomorphism problem. This method yields promising results but it is computationally very expensive because the subgraph-isomorphism problem is *NP*-complete. Therefore the method was not used in this thesis, it might, however, yield useful results as a step in the semantic analysis part of the pipeline, at which point the set of possible search results has already been reduced to a moderate number.

6.2.2.2 Type Mismatch Filtering

The Usefulness of Type Checking
Most programming languages use a notion of types. Type checking allows to eliminate whole classes of runtime errors [19], and types carry information about what any given program can do. As an example, consider the classic example of the `map` function in Haskell:

```
map :: (a -> b) -> [a] -> [b]
```

This function takes a function that converts a value of type `a` to a value of type `b` and a list of elements of type `[a]` and returns a list of `b`. This type signature guarantees, that the function always returns lists and it has to use the parameter-function on elements of the input list to generate the output-list (unless the output-list is empty).

Such information about types can be used to build a further filtering step in the semantic analysis part of the pipeline. How this can be achieved will be outlined in the following.

Type Conflicts on Functions
Let f_{doc} and f_{frag} be two functions that make up an alignment match, with f_{doc} on the document-side and f_{frag} on the fragment-side. A type-information based filtering step would check such alignment matches for compatibility of the types of the functions f_{doc} and f_{frag}. Note, that this proposal is not the only possible one, but rather a possible implementation that would have to be weighed against others.

In the follwing it will be assumed, that f_{doc} and f_{frag} are both written in the same programming language. If this programming language does not admit any polymorphism or if the two functions are monomorphic, the type based filtering step for the alignment match becomes simply a check for equality of the types of f_{doc} and f_{frag}.

A more interesting case occurs if the type of one of the two functions is polymorphic (f_{poly}) and one is monomorphic (f_{mono}). In this case, the two type signatures are compatible if the polymorphic parameters and result of f_{poly} can be instantiated in a way, that yields the type of f_{mono}. Such a case could be the parametrically polymorphic Haskell function

f_{poly} = `(+) :: (Num a) => a -> a -> a`

and a specialized version

$$f_{mono} = \texttt{(+.)} \quad \texttt{=> Double -> Double -> Double}$$

Here, the variables in the signature of f_{poly} can be instantiated to match the type signature of f_{mono}. This would not be possible if the type of f_{mono} were e.g. `String -> String -> String`, since the `a` in f_{poly} is constrained by the typeclass `Num`.

A more complex problem arises, if the two compared function types are both polymorphic. Under such circumstances, it is proposed that the two function type signatures are judged to be compatible if it is possible to instantiate the type variables in the signatures in a such way, that both type signatures become identical.

Type Conflicts on Statements, Expressions, Identifiers
The idea of filtering based on type conflicts of matched functions can be extended to work on more fine grained parts of programs. Such may e.g. be identifiers in series of statements or whole expressions. Depending on the precise kind of language construct, that is the subject of analysis the idea above would have to be adapted.

Overall filtering based on type information seems to be a promising approach because typing greatly limits what piece of source code can do. It is the author's estimate, that implementing this filtering step will further increase the accuracy of the CodeKōan search engine.

6.2.2.3 Human Computation

The CodeKōan search engine's web interface provides a dialog in which users can submit feedback to search results. This user supplied information can be used to enhance the quality of CodeKōan's search results.

Optimizing Parameters

User feedback would could be very useful to further optimize the pipeline algorithm's parameters. If, for example, a majority of users perceived the false positive rate as too high, parameters could be adapted in a way that reflects users desired search results.

Machine Learning Based Pipeline Step

User feedback can be used to generate labels for search results. These labels could in turn be used to train machine learning models to rate the quality of search results. Such a trained model could potentially be used as a further step in CodeKōan's search pipeline.

Increasing User Engagement

For a human computation system to work it is critical to elicit as much user supplied data as possible. While the satisfaction of contributing to an ongoing research project may be enough for some users, some more incentives for participation can be given. A promising

approach might be to only allow anonymous users a certain number of queries per day. If a user wants to make more queries per day, she must register an account with CodeKōan and provide a certain amount of feedback on search results to unlock more queries. By structuring the system this way, new users are still free to try the CodeKōan search engine but the average amount of feedback data per search query would be increased.

6.3 Applications

This section discusses ideas for practical applications of the CodeKōan search engine and methods developed in this thesis. Some of the proposed ideas could be integrated in existing development toolchains with limited effort and directly be used by developers in everyday work.

6.3.1 Further Language Specific Plugins

A useful extension to the CodeKōan search engine in its current form are further language specific plugins. At the time of writing this, this project supports searching Java, Python and Haskell source code. However this omits some of the most popular programming languages. Counting by tag use on Stack Overflow, the most popular language is Javascript [3], which could be derived from the current Java plugin with limited effort because the syntax of both languages is similar.

More language specific plugins would enable CodeKōan to cover more widely used programming languagages and to thereby be useful to a greater number of users.

6.3.2 Plagiarism Detection Systems

A natural application for the proposed search engine would be plagiarism detection. Plagiarism of Stack Overflow code fragments can be uncovered by using the current implementation with suitable parameters. Such parameters would demand a rather large minimal length for alignment matches (fifty or more tokens) and a high coverage percentage as defined in section 2.5.2.

A more general system for plagiarism detection in arbitrary code would be even more useful. Such a system could be implemented on top of the existing CodeKōan platform. The implementation of CodeKōan is independent of the structure of Stack Overflow and could easily be adapted to work on sets of code fragments with a different origin. The only relevant problem then is, how to extract sensible code fragments from arbitrary code.

For most programming languages it would likely be useful to extract methods and code blocks as code patterns. Such an extraction program would be possible to implement with limited effort.

6.3.3 Illucidating the Relationship of Reputation and Reuse

There is published work on reputation on Stack Overflow and its relationship to programmer behavior [7]. An aspect that has not yet been investigated on a large scale is, if code fragments from authors with a higher reputation are actually more frequently reused than code fragments from authors with a lower reputation. Likewise, there are no published

studies (to the best of the author's knowledge), that investigate if highly ranked Stack Overflow answers are more frequently reused in publicly available code.

Such a study could be performed using the CodeKōan platform and the dataset of all answers from Stack Overflow.

6.3.4 Finding Anti-pattern Usage

The term anti-pattern was coined with the frequently used term „design-pattern" in mind. A design pattern is a frequently used solution in software development, often in relation to object oriented program design [13]. An anti-pattern on the other hand is a frequently observed solution in software development that is considered a mistake or otherwise problematic.

Examples of such anti-patterns could be collected per-language in a database, which could in turn be used as a dataset for the CodeKōan search engine.

Search functionality for anti-patterns could be used a wide range of applications. A particularly helpful use of such functionality could be to integrate search for anti-patterns as a plugin in continuous integration (CI) platforms like Travis or TeamCity. If a developer committed code which contained an anti-pattern to a version control - repository, the developer would be immediately notified via email. Such CI plugins could be used by developers and organizations to uphold coding standards by simply specifying a list of examples of undesirable implementation techniques.

6.3.5 Suggesting Improvements from Related Stack Overflow Answers

An exciting use-case for the CodeKōan search engine comes from the structure of the data on Stack Overflow. Each code fragment, that is found by CodeKōan is part of an answer to a question. If a found code fragment makes up most of an answer, it is likely that the fragment functions as a solution to the problem that was presented in the question. This assumption then suggests, that it might be possible to find alternative solutions to the problem in the question by looking at code from other answers to the same question.

Furthermore it is very advantageous, that the answers each have a rating. Therefore, if reused code from a very lowly or negatively rated answer is found in a query-document, that may indicate use of an anti-pattern. In such cases, suggesting use of code from a more highly rated answer to a developer may lead to an actual increase in the quality of the work product of that developer.

This functionality could ideally be implemented as an integrated development environment (IDE) plugin, which could find reuse of code from a badly rated answer and present a developer with solutions from answers with a higher rating.

6.3.6 Suggesting Candidates for Method Extraction

As described in this thesis, the CodeKōan search engine detects reuses of code patterns. If the same code patterns are frequently detected in a project this may suggest, that code is being duplicated. Code duplication has long been recognized as a bad practice, as it reduces system maintainability [11]. The CodeKōan search engine could be used to build a

tool that automatically detects frequently used patterns in a project and suggests extracting these patterns into modules, methods or functions.

Doing this would reduce the overall amount of code in projects and thereby also reduce the amounts of points at which code could fail. This could be a direct contribution of the CodeKōan project to increase software quality.

CHAPTER 7

Appendix

7.1 Glossary of Terms and Notation

A

accepted answer

An answer to a question, that was selected as the best answer by the original poster of a question. A question is denoted by Q_i.. 4

Alignment (pipeline-algorithm step)

The initial step of the search pipeline, that finds alignment matches . 16

Alignment Match

A token-string pair $(D_{a:b}, P^i_{x:y})$ so that $d_L(D_{a:b}, P^i_{x,y}) < k$, where k is a fixed, given parameter. Each alignment match has two sides: a "document-side" and a "pattern-side". The document-side is $D_{a:b}$ and the-pattern side is $P^i_{x:y}$. . 16, 23

answer

An answer to a question on Stack Overflow, formally denoted as A^i. Contains zero or more code Fragments denoted by $A^{i,0}, A^{i,1}$.... 4

B

Block Accordance

A binary relation $A \times A$ of alignment matches. . 17

Bloom filter

A fast probabilistic data structure for determining set mebmership. First prosed by Burton H. Bloom[6].. 20

F

fragment ($A^{i,j}$)

Each post on Stack Overflow can contain zero or more specially denoted code fragments. It is argued in this thesis, that the answer code fragments are self contained units of code that solve sensible parts of problems. The j^{th} fragment of answer i is denoted as $A^{i,j}$. 6

G

generalized suffix tree

A suffix tree for a set of strings $\mathscr{S} = \{S_1, ..., S_k\}$. 14, 16, 20–23

L

levenshtein distance d_L

The levenshtein distance $d_L(W,V)$ of two strings $W,V \in \Sigma^*$ is the minimal number of edit operations, that is needed to transform W into V. . 14

LOC

lines of code. 4

N

NGram

n contiguous characters of a string.. 20

NLP

Natural language processing. 17

P

post

Either a question or an answer on Stack Overflow. Can have zero or more comments. 4

Q

query document (D)

A single code file, that is submitted for analysis by a user. Formally denoted as D. 16, 20

question

A question on Stack Overflow. Each question can have zero or more answers, and the asker can choose one accepted answer, that answers the question.. 4

R

rating

The difference of upvotes and downvotes of a Stack Overflow post. 6

reputation

The sum of the ratings of all of a user's posts including certain bounties that can be awarded by other users. A higher reputation earns a user privileges like upvoting, downvoting, or commenting on other user's posts. . 6

S

Search Result

A $\mathscr{R}_{\mathscr{P}\rangle}$ group of alignment matches, that all have their pattern-side in the same code pattern P^i. . 16

7.2 Listings of Hand Crafted Examples

This section will contain all hand-crafted examples that are mentioned in section 4.1. Most of these examples are implemented as tests using the HSpec testing framework. This appendix not only contains listings of hand crafted and hand-picked examples, it also showcases some of the actual source code from the implementation of the implemented tests. This source code is included to give a deeper understanding of precisely how the CodeKōan search pipeline is validated. Explanations of functions and libraries are sometimes simplified for the sake of brevity. For the full documentation please refer to the CodeKōan source code an library documentations on `http://stackage.org`

7.2.1 Find Exact Copies

One of the first fundamental properties that we expect from the CodeKōan pipeline is, that all code patterns in the index - GST are always found found when searching. This property is verified with a QuickCheck property.

All tests using the HSpec framework are values in the a `SpecWith` monad. Because of this the unit test for this property starts with:

```
-- | Given a suffix tree is constructed for a set of nonempty
-- strings. Then each of these individual strings should be
-- retrievable from the suffix tree by the lookup function.
alwaysFindSuff :: SpecWith ()
alwaysFindSuff = do
  it "Suffixtree lookup should always find each entry" $
```

Since this property is validated by a QuickCheck property, the next thing we do is call QuickCheck's `property` function. This function is applied to another function with the type signature `(Arbitrary a) => a -> Bool`. The typeclass arbitrary includes all things, that can be randomly generated for testing. In our concrete example the type variable `a` is instantiated by `[String]`.

```
    -- This quickcheck property must hold for all lists of
    -- strings. Quickcheck will check this property multiple times for
    -- randomly generated lists of strings.
    property $ \(strs :: [String]) ->
```

Next, these strings, which are just lists of characters are turned into vectors of characters). From these vectors of characters suffix trees are built and then merged into one single GST.

```
      -- This logic builds a suffix tree for each of the strings and
      -- then merges these suffix trees together into a single
      -- generalized suffix tree (GST)
      let vectors = V.fromList <$> (filter (\s -> length s > 1) strs)
          -- Turn the given list of strings into a list of vectors
          -- (i.e. arrays) of characters. Then number these vectors
          -- from zero.
          indexedVectors = zip vectors (S.singleton <$> ([0..] :: [Int]))
          tries = fmap (\(tr,s) -> buildSuffixTrie Nothing tr s)
                       indexedVectors
          mergedTrie = foldl1 mergeTries tries
```

The final part of this test code checks if every single one of the strings from the randomly generated set of strings can be found without mismatches in the merged GST. This makes use of the list monad to make for more concise code.

```
      -- Go over all vectors of characters and their according number,
      -- and lookup each in the merged suffix tree.
      in and $ do
        (v, i :: S.Set Int) <- indexedVectors
        let results = lookupAllSuff
                        -- A simple levenshtein automaton that only
                        -- exactly accepts it's original word.
                        (vectorToLevensteinAutomaton 0 v)
                        -- Run on the whole merged trie.
                        mergedTrie
                        -- Minimal match length is set to zero.
                        0
            resultSets = do
              (_, setsAndPositions, _) <- results
              (set, _) <- setsAndPositions
              return set
        -- Make sure that the original vector is in the results
        return $ i `elem` resultSets
```

7.2.2 Caveat for Code Insertion Handling

This section presents a concrete example for the code-insertion caveat outlined in section 4.1.4. Consider the following Python code pattern from Stack Overflow:

```
for x in xrange(10):
    for y in xrange(10):
        for z in xrange(10):
            print x,y,z
            #Insertion point
            if x*y*z == 30:
                break
        else:
            continue
        break
    else:
        continue
    break
```

Searching for this piece of source code in the CodeKōan search engine will give us a search result containing one alignment match with the above code. Inserting a single line with the code `call.a(function)` at the same indentation level as the comment in the above source code will give us a result with two alignment matches as shown in figure 7.1.

If instead of `call.a(function)`, the code `print x,y,z` is inserted at the location indicated by the comment, no result will be found. Why is this? Because after alignment and

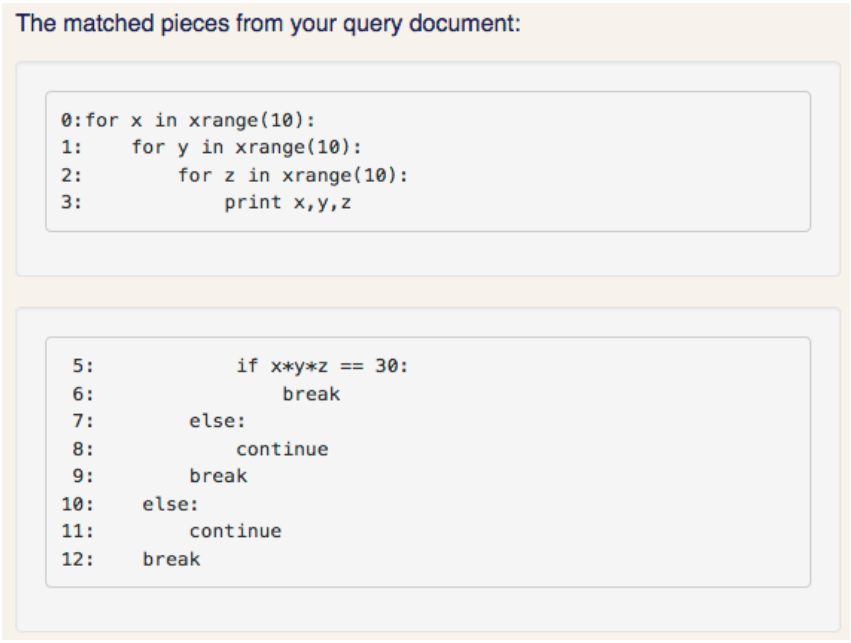


Figure 7.1: Document - sides of the two alignment matches of a search result for valid insertion.

aggregation, there will be two alignment matches. The first of these will be:

```
      DOCUMENT - SIDE            |          PATTERN -SIDE
                                 |
for x in xrange(10):             | for x in xrange(10):
    for y in xrange(10):         |     for y in xrange(10):
        for z in xrange(10):     |         for z in xrange(10):
            print x,y,z          |             print x,y,z
```

and the second will be:

```
      DOCUMENT - SIDE            |          PATTERN-SIDE
                                 |
            print x,y,z          |              print x,y,z
            if x*y*z == 30:      |              if x*y*z == 30:
                break            |                  break
        else:                    |          else:
            continue             |              continue
        break                    |          break
    else:                        |      else:
        continue                 |          continue
    break                        |      break
```

Note, however, that only one print statement occurs in the code pattern above. This single print statement is used in the pattern-side of both alignment matches, which is something that the block-filtering step prohibits. Therefore two search results are generated which each only contain one of these alignment matches. Both of these two search results, however do not meet the necessary coverage criteria and are therefore removed by a second

aggregation step.

7.2.3 Levenshtein-Distance vs. Minimal Alignment Match Length

This section illustrates an example for how choosing a small alignment match length can make levenshtein-search redundant in a lot of cases. For this purpose, consider the following source code fragment:

```
public static void main (String [] args)
{
    Scanner input = new Scanner(System.in);
    System.out.print("Enter Number:");
    int n = input.nextInt();
        while ( n > 0 )
    {
        System.out.println("Hello World");
        n = n - 1;
    }
}
```

With an index containing the above code fragment we use the following query document:

```
public static void main (String [] args)
{
    Scanner input = new Scanner(System.in);
    System.out.print("Enter Number:" + "(below)");
    int n = input.nextInt();
        while ( n > 0 )
    {
        System.out.println("Hello World");
        n = n - 1;
    }
}
```

It is easily observable, that the token strings of these two pieces of source code only differ by a levenshtein distance of two (insertion of `+ "(below)"`). To recognize the code fragment and the query document as similar on a token string levels, CodeKōan offers two choices: either choose a small l_{min} or a levenshtein-distance greater than 1.

With a sufficiently small l_{min} we can obtain two alignment matches (before and after the insertion) and get very high coverage. This can be done with levenshtein-distance zero.

Bibliography

[1] *Computer programming to be officially renamed "googling stackoverflow"*, http://www.theallium.com/engineering/computer-programming-to-be-officially-renamed-googling-stackoverflow/.

[2] *Information technology — advanced message queuing protocol (amqp) v1.0 specification*, ISIO/IEC 19464:2014.

[3] *Stackoverflow developer survey 2016*, http://stackoverflow.com/research/developer-survey-2016.

[4] *The merriam-webster online dictionary*, http://www.merriam-webster.com, 2016.

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

[6] Burton H Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM **13** (1970), no. 7, 422–426.

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

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

[9] Koen Claessen and John Hughes, *Quickcheck: a lightweight tool for random testing of haskell programs*, Acm sigplan notices **46** (2011), no. 4, 53–64.

[10] Edsger W Dijkstra, *Letters to the editor: go to statement considered harmful*, Communications of the ACM **11** (1968), no. 3, 147–148.

[11] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer, *A language independent approach for detecting duplicated code*, Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on, IEEE, 1999, pp. 109–118.

[12] Ludwig Fahrmeir, Christian Heumann, Rita Künstler, Iris Pigeot, and Gerhard Tutz, *Statistik: Der weg zur datenanalyse*, Springer-Verlag, 2016.

[13] Erich Gamma, *Design patterns: elements of reusable object-oriented software*, Pearson Education India, 1995.

[14] Robert Giegerich and Stefan Kurtz, *From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction*, Algorithmica **19** (1997), no. 3, 331–353.

[15] John B Goodenough, *Exception handling: issues and a proposed notation*, Communications of the ACM **18** (1975), no. 12, 683–696.

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

[17] Vladimir I Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Soviet physics doklady, vol. 10, 1966, p. 707.

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

[19] Robin Milner, *A theory of type polymorphism in programming*, Journal of computer and system sciences **17** (1978), no. 3, 348–375.

[20] Eugenio Moggi, *Notions of computation and monads*, Information and computation **93** (1991), no. 1, 55–92.

[21] Kevin P Murphy, *Machine learning: A probabilistic perspective*, The MIT Press, 2012.

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

[23] Lutz Prechelt, Guido Malpohl, and Michael Philippsen, *Finding plagiarisms among a set of programs with jplag*, J. UCS **8** (2002), no. 11, 1016.

[24] Chanchal K Roy, James R Cordy, and Rainer Koschke, *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach*, Science of Computer Programming **74** (2009), no. 7, 470–495.

[25] Chanchal Kumar Roy and James R Cordy, *A survey on software clone detection research*, Queen's School of Computing TR **541** (2007), no. 115, 64–68.

[26] Klaus U Schulz and Stoyan Mihov, *Fast string correction with levenshtein automata*, International Journal on Document Analysis and Recognition **5** (2002), no. 1, 67–85.

[27] Robert J Shapiro, *The us software industry as an engine for economic growth and employment*, Georgetown McDonough School of Business Research Paper (2014), no. 2541673.

[28] Michael Snoyman, *Developing web applications with haskell and yesod*, " O'Reilly Media, Inc.", 2012.

[29] Gregory Tassey, *The economic impacts of inadequate infrastructure for software testing*, National Institute of Standards and Technology, RTI Project **7007** (2002), no. 011.

[30] Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.

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

[32] Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia, *Problems creating task-relevant clone detection reference data.*, WCRE, vol. 3, 2003, p. 285.

[33] Peter Weiner, *Linear pattern matching algorithms*, Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on, IEEE, 1973, pp. 1–11.

[34] Marilyn Wolf, *Embedded software in crisis*, Computer **49** (2016), no. 1, 88–90.

[35] Michael Zhivich and Robert K Cunningham, *The real cost of software errors*, IEEE Security & Privacy Magazine (2009), 87–90.