Source Code Similarity and Clone Search

Iman Keivanloo

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy (Computer Science)

Concordia University

Montreal, Quebec, Canada

June, 2013

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Iman Keivanloo

Entitled: Source Code Similarity and Clone Search

and submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____Chair
Dr. R. Glitho

____ External Examiner
Dr. G. Antoniol

____ External to Program
Dr. F. Khendek

____ Examiner
Dr. R. Witte

____ Examiner
Dr. N. Tsantalis

____ Thesis Supervisor
Dr. J. Rilling

Approved by ____
Dr. H. Harutyunyan , Graduate Program Director

June 20, 2013 __________________________________
Dr. Robin A.L. Drew, Dean
Faculty of Engineering & Computer Science

Abstract

Source Code Similarity and Clone Search

Iman Keivanloo, Ph.D.

Concordia University, 2013

Historically, clone detection as a research discipline has focused on devising source code similarity measurement and search solutions to cancel out effects of code reuse in software maintenance. However, it has also been observed that identifying duplications and similar programming patterns can be exploited for pragmatic reuse. Identifying such patterns requires a source code similarity model for detection of Type-1, 2, and 3 clones. Due to the lack of such a model, ad-hoc pattern detection models have been devised as part of state of the art solutions that support pragmatic reuse via code search.

In this dissertation, we propose a clone search model which is based on the clone detection principles and satisfies the fundamental requirements for supporting pragmatic reuse. Our research presents a clone search model that not only supports scalability, short response times, and Type-1, 2 and 3 detection, but also emphasizes the need for supporting ranking as a key functionality. Our model takes advantage of a multi-level (non-positional) indexing approach to achieve a scalable and fast retrieval with high recall. Result sets are ranked using two ranking approaches: Jaccard similarity coefficient and the cosine similarity (vector space model) which exploits the code patterns' local and global frequencies. We also extend the model by adapting a form of semantic search to cover bytecode code. Finally, we demonstrate how the proposed clone search model can be applied for spotting working code examples in the context of pragmatic reuse. Further evidence of the applicability of the clone search model is provided through performance evaluation.

Acknowledgements

I would like to thank my supervisor Dr. Juergen Rilling for his continued support and encouragement. Without his openness and encouragement, this journey could not have happened. His unique personality as a supervisor and friend is the main reason behind the success of this research. I also extend my thanks to Drs. Giuliano Antoniol (École Polytechnique de Montréal), Ferhat Khendek, René Witte, and Nikolaos Tsantalis, who served on my committee and provided me invaluable comments at different stages of my research.

I would like to thank Drs. Chancal K. Roy, Volker Haarslev, and Rainer Koschke. I would like to thank Dr. Roy for sharing his knowledge in clone detection with me. His generous presence and collaboration provided me with the spirit to explore diverse problems and solutions. Similarly, I extend my thanks to Drs. Haarslev and Koschke whose early comments on my work opened new directions to my research. I also like to thank my earlier teachers and supervisors, especially Jamshid Rabiei.

I would like to thank all members of the Computer Science & Software Engineering department. In particular, I would like to thank Halina Monkiewicz and Dr. Nematollaah Shiri. They supported me from my first day at Concordia with their patience and kindness in every aspect of student life. I would also extend my thanks to all of my friends and co-authors at Concordia University and University of Saskatchewan: Philipp Schugerl, Aseel Hmood, Laleh Roosta Pour, David Walsh, Ninus Khamis, Mostafa Erfani, Sultan Al Qahtani, Chris Forbes, Chris Neil, George Peristerakis, Farouq Al-Omari, Jeff Svajlenko, and Masud Rahman.

Most of all, I want to thank my immediate family and family-in-law. The grandparents who shaped this journey for us: Aziz, Shahrokh, Mohtaram, Hossein, Shahbanoo, Ali, Sedighe, and

Ebrahim; My parents who gave us the chance to enjoy the journey: Jina, Gholam Reza, Mahshid, and Sirous; My brother, Omid, who has been my friend through the journey; Arash, Kiarash, Nastaran, Maryam, and finally my wife, Mahsa: without her the journey would be meaningless.

To Mahsa

Table of Contents

List of Figures

List of Tables

1. Introduction

The term clone (Greek word *klōn*) was first used by Herbert J. Webber [WEH03] in 1903, referring to the outcome of a derivation activity in the context of living species. In computer science, such autonomous reproduction is limited, but derivation is unavoidable and it is known as cloning. Derivation during software development usually occurs as the result of reuse [PER88][DEE05]. Based on the problem and its granularity, various forms of reuse are introduced [KRU92] [ROT03], including pragmatic reuse [HOI08] (i.e., copy and change). While the concept of reuse is often promoted as a solution for cost reduction and quality improvement [BBO99] [FRA05], there are some forms of reuse that are usually related to discouraged ethical issues (e.g., copy without permission, plagiarism). Common to all of these forms is deriving and introducing new instances (i.e., clones), which share similar concepts and origins with slight modifications (e.g., tailoring).

The ease of reuse and the potential harms caused by cloning in software development became a major motivation for computer scientists to explore the possibility of identifying code duplications (i.e., source code clones). Consequently, a new research discipline - clone detection - [ROS09][BAK92][BEL07] has emerged in computer science, which focuses on devising novel algorithms and heuristics for finding, tracing, and managing [KOS08] clones.

Although the input data for this type of similarity search is source code, which is structured and well organized, the clone detection problem remains non-trivial [ROS09], due to the different types of similarities that can be distinguished. At source code level, clones share two types of similarity: (1) pattern and (2) content similarity. The challenge lies often in determining if two code fragments are actually cloned, as two cloned code fragments, e.g., “int temp=0;” and “float f=2;”, can hold negligible content similarity (i.e., similarity in token names).

In 1976, Ottenstein [OTT76] explored the idea of using code metrics for plagiarism detection in students’ programming assignments that is later extended to software maintenance applications [MAY96] [BAX98] [BAK92], since duplicated code has been widely accepted as a threat to bug fixing and software quality [KAP06]. Consequently, other research directions have emerged, involving algorithms and heuristics for other types of input data such as compiled code (e.g., [BAK98]).

1.1. Motivation

Although pragmatic code reuse through cloning has often been criticized as being harmful, recent studies show that cloning is actually both useful [KAP06] and often unavoidable [HOI08]. Pragmatic reuse occurs when developers are implementing their programming tasks by locating, reusing, and often customizing code examples derived from available local or global code repositories. In general, such source code examples play a major role in programming as both intrinsic resources for learning [NYK02] and reuse [ROS96][JON92]. The lack of good source code examples is one of the major complications in learning [ROB11] and eventually coding during software development. Availability of code examples for reuse and learning can accelerate the development process and improve programmer productivity [MAN05], as well as contribute towards an improvement of product quality [MAR09]. Since it is not a common practice to collect and document code examples [HOL05][SIN98][WAN13], previously written projects [WAN13] and code repositories (e.g., sourceforge.net) have become invaluable resources for code examples.

Due to the sheer size of the data in these repositories, locating code examples without adequate search functionality is a major challenge. Therefore, the community established source code search as common practice to locate code examples for software development [BRA10] [BUS12] when attempting to find relevant code fragments. Specifically, when the search space is extended to cover other resources hosted on the Internet. Such Internet-scale code search is defined as the process of searching over source code available on the Internet to find pieces of working code fragments [GAL09].

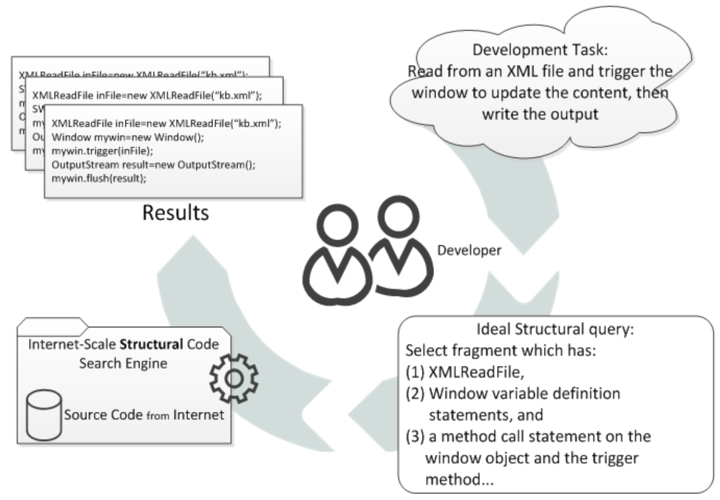
Support for developers in finding code examples for reuse has been widely explored with diverse approaches such as structural code search (e.g., [HOL05]), synthesizing and mining (e.g., [MAN05]), or even Questioning & Answering [NAS12]. Unfortunately, the available search approaches suffer from different challenges. Three major issues are discussed in the literature that are hindering the success of code search for reuse. (1) Mishne et al. [MIS12] argue that one of the challenges in the Internet-scale code search domain is the large number of matches (results returned) for each query [MIS12]. Wang et al. [WAN13] also noted that redundancy in the result set can become a major problem when searching for reusable code fragments. (2) As Holmes et al. [HOL09] point out, relevancy is often not the sufficient condition for such source code search. (3) Buse and Wiemer in [BUS12] discuss that spotted code fragments are usually too long and complicated to be considered as working code examples even after applying program slicing, a program reduction technique.

In summary, it is commonly agreed upon that the usability of the ranked result set provided by current code search engines is limited for finding code examples to support pragmatic code reuse. These result sets are often of poor quality due to the high number of matches returned that contain repeated (exact or similar) hits or missing information.

1.2. Potential solution – exploiting clone detection for reuse

Recently, several similarity search approaches have been proposed to address these ongoing challenges related to code search result sets. Wang et al. [WAN13] proposed an approach for sequence mining to detect reoccurring sequences of code fingerprints (i.e., method call tokens). Buse and Wiemer [BUS12] apply data mining to graph models that were created from data flow and method call sequences to detect duplicates sub-graph occurrences. Mishne et al. [MIS12] suggested using a search rather than a mining approach to detect similarities. Their approach is based on a similarity search model to find repeated code patterns and exploit them to improve the result set (e.g., popularity-based ranking). However, none of the existing code searches have used a clone detection model for their search approach. This is in contrast with the fact that clone detection research covers a vast body of similarity search models for formal source code similarity types (i.e., Type-1, 2 and 3). Current source code search models have relied on ad-hoc similarity search models since the available clone detection counterparts still lack support for some of the emerging requirements of search engines such as ranking, scalability, and short response times. In summary, although clone detection had originally been devised to cancel out the negative effect of reuse via cloning, its core principals can also be applied for clone search and further exploited to support reuse, e.g., online development session support [LEM11].

An example for immediate applications of clone search in this context is the result set expanding [KLX12]. If a clone search model that addresses the core requirements can be derived, it can be further exploited to improve the result of existing structural code searches (e.g., Sourcerer [BAJ12]). Figure 1 illustrates a traditional structural search-based approach to support pragmatic reuse. However, given the size of the search space, the complexity of the queries, and the challenges in presenting relevant result sets, existing structural code search engines are limited in their applicability to support pragmatic reuse. For example, to formulate the query, the user should know the participant elements (e.g., types and methods) in advance. To address this deficiency and improve the usability of the search engine and its result set, it is possible to extend the search strategy (Figure 2) by exploiting a similarity function close to what is available in clone detection [KLX12]. Using such a clone detection based approach, the preliminary result of the structural search, users can now expand the result set based on pattern and content similarities (Figure 2 step 2). This approach allows a user to retrieve the expected answer faster by providing them with an option to browse through the clones.

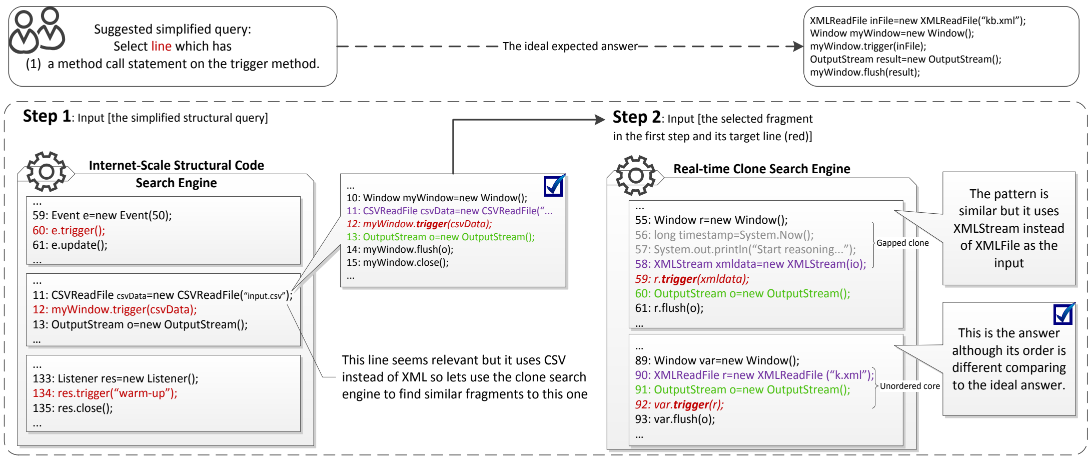

Results
Developer
Internet-Scale Structural Code Search Engine
Source Code from Internet

```
XMLReadFile inFile=new XMLReadFile("kb.xml");
Window mywin=new Window();
mywin.trigger(inFile);
OutputStream result=new OutputStream();
mywin.flush(result);
```

Development Task:
Read from an XML file and trigger the window to update the content, then write the output

Ideal Structural query:
Select fragment which has:
(1) XMLReadFile,
(2) Window variable definition statements, and
(3) a method call statement on the window object and the trigger method...

Figure 1. Traditional structural source code query approach


The ideal expected answer
Internet-Scale Structural Code Search Engine
Real-time Clone Search Engine
Gapped clone
Unordered core

Suggested simplified query:
Select line which has
(1) a method call statement on the trigger method.

```
XMLReadFile inFile=new XMLReadFile("kb.xml");
Window myWindow=new Window();
myWindow.trigger(inFile);
OutputStream result=new OutputStream();
myWindow.flush(result);
```

Step 1: Input [the simplified structural query]

```
...
Event e=new Event(50);
e.trigger();
e.update();
...
```

```
...
CSVReadFile csvData=new CSVReadFile("input.csv");
myWindow.trigger(csvData);
OutputStream o=new OutputStream();
...
```

```
...
Listener res=new Listener();
res.trigger("warm-up");
res.close();
...
```

```
...
Window myWindow=new Window();
CSVReadFile csvData=new CSVReadFile("...
myWindow.trigger(csvData);
OutputStream o=new OutputStream();
myWindow.flush(o);
myWindow.close();
...
```

This line seems relevant but it uses CSV instead of XML so lets use the clone search engine to find similar fragments to this one

Step 2: Input [the selected fragment in the first step and its target line (red)]

```
...
Window r=new Window();
long timestamp=System.Now();
System.out.println("Start reasoning...");
XMLStream xmldata=new XMLStream(io);
r.trigger(xmldata);
OutputStream o=new OutputStream();
r.flush(o);
...
```

The pattern is similar but it uses XMLStream instead of XMLFile as the input

```
...
Window var=new Window();
XMLReadFile r=new XMLReadFile ("k.xml");
OutputStream o=new OutputStream();
var.trigger(r);
var.flush(o);
...
```

This is the answer although its order is different comparing to the ideal answer.

Figure 2. Extending source code search using clone search [KLX12]

1.3. Summary of contributions

Recently, *clone search* (e.g., [LER10]) has emerged as a new research direction that exploits the fundamentals of clone detection research to provide search functionality for similar code fragments (a.k.a. clones). In contrast with the traditional *clone detection*, clone search is only concerned with detecting similar code fragments for a given input code fragment at run-time. A code fragment constitutes the query input making *clone search* also different from regular *source code search* where the input is defined by a set of keywords or concepts. Therefore, *clone search* can be considered as a *function* that accepts a code fragment as its input parameter. The output of a clone search includes all code fragments in the search space that are similar to the input parameter. As a result, clone search forms the core of code similarity search. Furthermore, output items can be sorted based on their similarity degree to the given input query.

In the literature, several terms have been used to highlight the importance of response time in clone search such as just-in-time [BAR10], real-time [KAW09], and instant [LER10]. Several similarity and search models (exploiting clone detection basics) have been proposed to address the core requirements of clone search: scalability, short response time, and search for Type-1, 2 and 3 clones. However, these requirements ignore the importance of ranking and the quality of ranked result set, which we consider both to be core requirements for clone search models to support source code search and pragmatic reuse.

1.3.1. Contribution 1 - the clone search model

In this thesis we propose a clone search model that includes a similarity function for applications such as pragmatic reuse (e.g., [LEM11]), where ranking, scalability, fast response time, and Type-1, 2, and 3 detection are essential requirements. The model is based on our early research attempts [KLX11] [KLZ11] proposing a clone search approach for emerging applications such as pattern-based code search (e.g., [KLZ12]) and source code search result improvement (e.g., [KLX12]). Our studies in [KLX11] demonstrate how a multi-level indexing approach can achieve scalability, short response time, and search capabilities for Type-1, 2 and 3 clones. We have extended this multi-level indexing approach by adapting the Jaccard similarity coefficient [JAC01] and cosine similarity [MAN08] to support another core requirement: the ranking of result sets. Our clone search models' ranking exploits code patterns' (not token) local and global frequencies for assigning different weights depending on the pattern popularity. For example, a domain specific pattern (e.g.,"EclipseEditor foo=new EclipseEditor()") can be assigned higher weights compared to some general code patterns (e.g., "catch (Exception ex) {"). We have studied the applicability of the proposed similarity search model using a representative dataset of 25,000 open source Java projects for line-level granularity. The study focuses on the performance of our search model addressing the core requirements for a clone search approach: scalability, fast response time, Type-1, 2, and 3 detection, and the ranking of the result sets.

1.3.2. Contribution 2 - adaptation of the proposed clone search model for bytecode content

We also conducted studies to provide evidence that our search model is applicable for other types of source code. For these studies we applied our search model on Java bytecode. We consider being able to search bytecode content to be an essential part of Internet-scale code search approach, since bytecode content constitutes a major part of the data (e.g., [BAJ12]). In order to achieve high recall during the Java bytecode clone search, we introduce two detection heuristics for Java bytecode. First, we use relaxation on code fingerprints*,* which only considers certain types of tokens for clone detection. Second, we introduce a multi-dimensional matching heuristic. This multi-dimension heuristic applies the clone detection algorithm independently for each type of token (a.k.a., dimension). These heuristics follow and replicate our multi-level indexing idea for bytecode content. Furthermore, we also extended our original clone search model to support some form of the semantic search [GUH10]. This extension is motivated by the nature of bytecode content, where each token (e.g., a summation token) includes additional embedded information such as data types. Our evaluation with a dataset of 500,000 compiled Java classes showed that our search model is not only scalable but also capable of providing a reliable ranking of the result sets for Java bytecode content.

1.3.3. Contribution 3 - adaptation of the proposed clone search mode for spotting code examples for reuse

As the third major contribution, we illustrate how a clone search model can actually support pragmatic reuse. For pragmatic reuse in a software development context, a key challenge is that any code fragment that meets the query criteria should not be considered as a correct code example. In a pragmatic code reuse context, the answer must be concise, self contained, easy to understand, and integrate [HOL09][MIS12][WAN13][BUS12]. A code fragment meeting these requirements is considered a working code example. In our research we focus on the spotting problem of concrete working code examples using our proposed clone search model. That is, we study the possibility of applying clone search models instead of ad-hoc similarity search models for spotting working code examples. Spotting these code examples is challenging since there exists a tradeoff between various aspects such as popularity, conciseness, and completeness of the results, which have to be considered when selecting the result. We show the applications of clone search for different types of similarity search in state of the art approaches for spotting working code examples. We show that clone search is able to successfully handle the tradeoff between conciseness, completeness, and popularity. Our approach supports free form querying (i.e., bag of words with no ordering constraint). A $query = \{term_1, term_2, \dots, term_n\}$ is composed of different terms, where each term can be a data type, method name, or concept (e.g., download or bubblesort). This is different from most of the earlier work, where search engines require either a partial written code, or API names and the data flow information (e.g., $term_{n-1} \rightarrow term_n$).

1.4. Organization

Chapter 2 outlines related work for clone detection, search, bytecode similarity search, and code search for pragmatic reuse. Chapter 3 overviews our clone search model which is called SeClone. Retrieval and indexing steps of our search model is discussed in Chapter 4. The details of ranking schemas of our search model are covered in Chapter 5. Prior knowledge about the characteristics of the input data is necessary for successful deployment of our search model. Chapter 6 summarizes our observation about the chosen data characteristics in the domain of discourse. For proper performance evaluation we require some measures for the ranking aspect. Chapter 7 introduces the adapted measures from other domains (e.g., information retrieval) for proper clone search evaluation. Chapters 8, 9, and 10 discuss the adaptation and performance evaluation of our similarity search model for the source code clone search, bytecode similarity search, and working code example search problems. Finally, Chapters 11 and 12 provide the ending discussion and conclude the dissertation.

2. Related work

Source code repositories present invaluable sources of information for source code search [WAN13] and pragmatic reuse [HOI08]. For example, reuse patterns can be exploited to infer popular programming solutions for code recommendation [BUS12]. In the past decade, various ad-hoc similarity search approaches (e.g., [MAN05]) have been introduced and applied to define reuse patterns. Alternatively, existing and often well-defined and supported clone detection models can be adapted in place of these ad-hoc similarity detection approaches. This chapter provides an overview of related work covering both the application and solution domains.

2.1. The application domain – code search for reuse

In [GLP13], Gulwani introduces program synthesis (PS) as "the task of automatically discovering an executable piece of code given user intent expressed using various forms of constraints such as input-output examples, demonstrations, natural language …". PS supports a variety of users such as (1) general users of information systems with or without prior programming experience to automate their repetitive daily tasks and (2) professional programmers to accelerate the development process by avoiding coding from scratch. Other topics, such as source code search, recommendation, and completion for pragmatic reuse, are related parts of the program synthesis problem. In particular, their underlying approaches, techniques, algorithms, and heuristics all developed with the common objective to accelerate development processes by helping programmers through working code examples. The (similarity) search functionality is the shared component among these approaches to satisfy the user expectations.

Source code search is not a new research topic in software engineering (e.g., [LIN84]) and has been widely investigated, producing a vast body of research. The diversity in their search models differentiates these solutions. In what follows, we review these search models by highlighting the proposed similarity functions, which form the core of their models. Paul and Prakash [PAU94] propose SCRUPLE that provides code search functionality via queries similar to code templates. The authors focus on applications of code search related to software maintenance, such as locating all matches of a specific buggy fragments. Their pattern language approach addresses deficiencies of grep-like search functions for the code search problem.

Another early approach to finding reuse patterns and association through rule mining suggestions, called CodeWeb, has been introduced by Michail [MIC00]. It applies association rule mining by using generalized association rules for mining [MIC99] reuse patterns. In his research, Michail investigates high-level reuse patterns for C++ covering only fingerprints of the inheritance links, instantiation tokens, method calls, overrides, and receiving invocation messages. This approach is different from traditional fine-grained pattern mining approaches such as sequence of source code tokens. The interesting point about these generalized association rules versus regular association rules is that they are able to employ taxonomies, such as the inheritance trees. Following a similar approach for mining coarse-grained facts and goals, Bruch et al. [BRU06] developed an Eclipse plug-in called FrUiT. However, the focus of their approach was not only on mining of reuse patterns but also on providing a context-dependent presentation within the Eclipse IDE. FrUiT provides reuse support for novice users by recommending the next potential actions.

Hill and Rideout [HIL04] focus in their work on method body completion by using machine learning and exploiting frequently occurred near-duplicate code (small sized clones). The approach focuses on (1) completion of popular methods in Java, such event listeners etc., and (2) extending (i.e., a type of completion) the current method body written from beginning until the cursor. Method bodies are represented as vectors and compared them using Euclidean distance and K nearest neighbor - kNN. The vectors are based on 154 metrics that are calculated for each method body, of which 150 metrics are related to frequency occurrence of Java token types, with the remaining 4 metrics being LOC, cyclomatic complexity of a method, return value, and its number of arguments.

Li and Zhou investigate in [LIZ05] the application of mining code patterns for the detection of buggy code fragments and introduce as part of their work the PR-Miner tool. In their approach, they first use frequent itemset mining to find reputable patterns. Second, they locate code fragments that are not adhering to the mined rules as potential violations/bugs. Specifically, PR-Miner mines closed sub-itemsets using a FPclose algorithm and then creates the association rules to detection violations. Based on their studies for Linux, Apache HTTP server, and PostgreSQL written in C, their approach is capable of successfully detecting actual bugs. PR-Miner considers in its analysis fine-grained fingerprints to generate transactions such as language keywords, method calls, and variables. In order to avoid name collisions, they resolve token names by attaching data types and other metadata since source code tokens constitute the items. Similarly, Wahler et al. [WAH04] use frequent item mining for clone detection. The major differences between Wahler's approach and the other similar works mentioned are that (1) they forced the mining algorithm to detect the consecutive items and (2) they used maximal sets.

Mandelin et al. [MAN05] introduced their PROSPECTOR and Jungloid mining approach to help programmers in acquiring an object (instance) of a specific class (type). The approach produces (synthesizes) Jungloid, a code snippet that performs type transformations, and combines these transformed code snippets to answer queries using the source and destination types. Possible types (templates) of Jungloids are provided for Java. For their approach, the API signatures and examples constitute the input data. The links between both data sources are presented as a single DAG which is used by the synthesize algorithm. The generated solutions are ranked based on their size, with shorter answers being preferred due to their simplicity.

In [BRU09], Bruch et al. focus on re-using code examples for intelligent code completion. Their goal is to improve the auto-completion of search results by removing items irrelevant to a programmer's current work context. Their approach is designed to recommend method names that should be called for a selected variable. The input data are the current programming context and previously mined examples. In their work they evaluated the performance of three approaches to recommend the next method name being called. For their evaluation they used precision, recall, and F-measure for (1) method call frequencies, (2) association rule mining for method call patterns, and (3) code completion using a customized approach based on K nearest neighbors (called best matching neighbors BMN). Their observation shows that for their specific application context, the kNN solution achieves the best F-measure. In order to automate their evaluation, they took advantage of an evaluation approach for API recommender systems presented in [BUT00].

Robbes and Lanza [ROB08] focus on the code completion problem for MS Visual Studio via IntelliSense by using change history. Their objective is specifically geared towards situations with APIs with large number of methods and members, making the use of the completion result list very challenging. Moreover, in their work they also define a benchmark for accuracy measurement of such systems and introduce a new graphical interface. Their approach is mainly based on recorded fine-grained actions and collected data during programming sessions, which are modeled as sequences of changes. The authors argue that ranking can be improved by combining change history information with other types of information such as code and query context.

Hou and Pletcher [HOU10] address the ranking problem for the "auto-complete" box available in IDEs. Their goal is to improve Eclipse's current approach, which supports only alphabetical ranking, by giving priority to answers sharing the same type as the context. They studied the usability of usage frequency for sorting and customized pre-defined rules for the filtering or grouping of the auto complete result sets.

Menon et al. [MEN13] explore the possibility of machine learning (ML) and its benefits for ranking and searching. The objective of their work is to find and efficiently rank some combinations of smaller pre-defined programs as the answer set. Specifically, ML is exploited to learn the weights (i.e., importance) of the possible answers for the given examples. Perelman et al. [PER12] proposed an API discovery approach based on the idea of programming by example that suggests and ranks the APIs matching to the query. They conducted a study on the .NET framework. Their approach supports a variety of code completions, which improve Visual Studio Intellisense for auto completion in some cases. In particular, their approach shows improvements when a method call statement completion is exploited for the completion of an argument list, expression completion, or method name search via candidate types. Their research also proposes a querying approach known as partial expressions, which uses library class/interface definition information and the context data (e.g., local variables) to match candidates to a given query. This solution can be considered as an automatic generation approach that relies heavily on search and matching.

Recently, other mining and search approaches have been proposed towards working code example recommendation for API usage scenarios. In [WAN13], Wang et al. present their UP-Miner implementation as the successor of MAPO [ZHO09]. UP-Miner combines clustering and sequence mining to find reoccurring sequences of API fingerprints (i.e., method call tokens). UP-Miner's probabilistic approach is able to recommend the next most probable step/s for the given API name. Buse and Wiemer [BUS12] apply mining on graph models created from the data flow and method call sequences. Using the mined sequences, their approach synthesizes code fragments as the potential solutions for a given query. Mishne et al. [MIS12] applied another approach, which exploits search instead of mining and synthesizing. Their approach, PRIME

[MIS12], extracts partial temporal specification from method call sequences to find possible solutions, and returns the corresponding code fragments located in the available corpus.

Common to all of these proposed solutions is the fact that they are based on ad-hoc reuse pattern detection techniques, which are used to mine either the library definitions or the given examples to determine how a particular programming task can be implemented. Alternatively, we explore in this thesis how these ad-hoc similarity mining approaches can be replaced by clone detection and clone search models for program synthesis in the context of pragmatic reuse, which not only support the detection of defined reuse patterns (i.e., clone types) but also result ranking.

2.2. The solution domain – code similarity detection

Given the need for finding code duplications in programming content [ROS09], clone detection has emerged as a research discipline in computer science. The underlying algorithms and heuristics target detection of four similarity types [BEL07][ROS09] found in source code. Table 1 provides an overview with examples of the three basic similarity types related to syntactical clones. The types are defined based on their observable similarity in the source code. At source code level, clones share two types of similarity: (1) pattern and (2) content. Clone detection is challenging, as two cloned code fragments, e.g., “int temp=0;” and “float f=2;” can contain negligible content similarity (i.e., token names). Type-1 clones are exact copies of each other, except for possible differences in whitespaces and comments. Type-2 clones are parameterized copies, where variable names and function calls have been renamed and/or types have been changed. Changes (e.g., addition and deletion of statements) in a clone pair result in Type-3 clones. In cases where two fragments share similar functionality with different syntactical presentations, they constitute a Type-4 clone pair.

Table 1. Examples for source code similarity types

The input code sample	
`HashMap var=new HashMap (10);`	
Similarity Type	***Example***
Type-1	Additional Whitespace `HashMap var    =    new HashMap (10);`
Type-2	Different variable name `HashMap` **`list1`**`=new HashMap ();`
Type-3	Additional Code `HashMap list1=new HashMap (`**`list2.size()`**`);`

2.2.1. Clone detection

Source code clone detection has been a major focus of software research and has resulted in a number of clone detection techniques. Common to all of these traditional detection applications is the fact that they have a complete off-line search step to find all possible clone pairs within a static source code repository. In this section, we present a review of early work on (1) source code clone, (2) code clone detection, and (3) code similarity to discuss the origins of these concepts and terms. Our review covers the period between 1930 and present, focusing mainly on the initial use of the terms “cloning” and “clone detection” in the context of “source code” in the literature.

2.2.1.1. Similarity detection in software

One of the first similarity detection approaches dates back to the work by Ottenstein [OTT76] in 1976. Ottenstein introduced a metric-based approach for the detection of plagiarism in student programming assignments. His work also included a discussion on potential dissimilarity types that were supported by a plagiarism detection algorithm, such as re-formatting, re-naming and re-ordering of statements. Later on, Grier [GRI81] in 1981 extended Ottenstein’s work to Pascal code.

2.2.1.2. Source code clone detection

The first actual reference to the clone concept in the source code and programming domain dates back to the work by Abrams and Myrna [ABR79] in 1979. They used the term clone in a Programming Language (APL) context describing it as "… creates an output file and starts a "clone" of itself". In later attempts, the concept of a "clone" in source code was used by Jacobsen [JAC84] to describe a pre-defined command, and by Caudill and Wirfs-Brock [CAU86] as a reproduction of executable files in Smalltalk. Tanenbaum [TAN87] used clone to describe the variations of a software system. During the 1980s, the term clone was further popularized mostly through its use as a reference to computer hardware, such as compatible computer (hardware), an IBM compatible (or short IBMclone) computer [KEL83] or, in [LOM83], as "…can't tell what is on my disk without a clone of my computer". Among the first researchers who actually used the clone detection phrase at the source code level were Carter et al. in 1993 [CAR93]. They described clone detection in their work as the process of finding similar telecommunications systems using neural networks.

While the early work in clone and similarity research had focused mainly on detecting plagiarism in source code, this focus started to shift in the 1990s with software maintenance emerging as a new application for clone detection. In 1992, Baker [BAK92] proposed Dup, a tool to support software maintenance and bug fixing by detecting duplicate code. The Dup tool also implemented a clone detection solution, which exploited hash values and inverted-indexes to facilitate the search process during clone detection. Later approaches, such as metric-based by Merlo et al. in 1996 [MAY96] and AST-based Baxter et al. in 1998 [BAX98], allowed them to use additional facts extracted from source to further improve scalability, performance, and efficiency in their clone detection approaches.

Alternatively, information retrieval has been explored for the purpose of clone detection and clustering, due to its well defined search models. Marcus and Maletic [MAR01] used Latent

Semantic Indexing (LSI) to extract hidden semantics from source code facts (e.g., identifier names) in order to guide the process by detecting code fragments implementing similar features. In [POS07], Poshyvanyk et al. propose an approach that combines Formal Concept Analysis and LSI for the concept location problem. McMillan et al. [MCM12] use LSI to search for similar software applications in terms of their functionality. LSI also has been exploited for clone result set improvement (not the detection itself) and evaluation by Tairas Gray [TAI09]. Additionally, some research exists on using the other IR techniques, such as that by Kontogiannis [KON97] who uses basic retrieval infrastructure, or the work by Mishne et al. [MIS04], who introduced an approach that exploits Conceptual Graphs and structural information (in addition to the other code facts) to find similar code.

In general, the state of the art clone detection tools (e.g., NiCad [ROS08] and CCFinder [KAM02]) are based on *sequence comparison* functions. Recently, novel search and retrieval models have been explored for clone detection focusing on the scalability issue such as the DECKARD [JIA07] model, or suffix trees by Koschke [KOS12]. Uddin et al. [UDD11][UDD13] explored the application of simhash for near-miss clone detection. Lavoie and Merlo [LAV11][LAV12] considered Levenshtein metric and Manhattan Distance in their approach to detect near-miss clones. There is also some work on similarity measures and ranking for clone detection by Smith and Horwitz [SMI09]. While all of these approaches were proposed for clone detection, they simultaneously established the foundations of code similarity detection.

In summary, our research approach is similar to Carter et al. [CAR93], which also uses a cosine similarity function. While we also use vectors similar to DECKARD [JIA07] and Carter et al. [CAR93], we create our vectors using code patterns instead of metrics and predefined fingerprints [JIA07][CAR93]. Furthermore, our approach emphasizes on non-positional similarity search instead of sequence matching and comparison (e.g., as NiCad [ROS08] and CCFinder [KAM02]). While, similar to earlier attempts such as Smith and Horwitz [SMI09], Baker et al.

[BAK98], and Uddin et al. [UDD11][UDD13], our multi-level indexing approach not only detects the major clone types but also is capable of discriminating between Type-2 and 3 clones at the same time.

2.2.1.3. Binary and bytecode clones

In contrast to the traditional source code clone detection, bytecode code clone detection has not been a major research focus in the clone detection community. However, in some domains such as code search [BAJ12] and security [BAK98], the ability to support clone detection at the bytecode level as well becomes a key requirement. A major factor for the use and analysis of binary and bytecode content is often the limited availability of source code. Baker and Manber [BAK98] used a combination of three comparison-based approaches such as Diff for bytecode. The JCD project, [DAV10] introduced by Davis and Godfrey, uses a combination of hill climbing and greedy algorithms to detect the maximum coverage. In [SAN11] the use of process algebra on bytecode was proposed. Selim et al. [SEL10] converted bytecode to the Jimple format [SOO12] and used third-party tools for clone detection.

Recently, license violation and malware detection has become an emerging application area that can greatly benefit from clone detection on binary or bytecode content [CHA11][SAB09][HEM11]. In [HEM11], Hemel et al. explored some generic similarity heuristics for license violation detection using their Binary Analysis Tool (BAT). In their approach they use string literals extracted from the target binary in the central database of literals as part of their first search heuristic. Note that the central literal database can be built using literals extracted from both source code and binary. However, the assumption in their research is that the source code of the target entity is not available. Compression ratio as a similarity metric is their second heuristic, which has been investigated previously in other similarity search domains such as malware detection. Computation of the delta between target and suspect binary extracted from the central repository constitute their last heuristics. For binary content such as native machine code, Sæbjørnsen et al. [SAB09] proposed a more restrictive solution compared to the one by Hemel et al [HEM11]. Sæbjørnsen et al.'s approach is based on common source code clone detection techniques, where the content is indexed based on pattern similarity. They apply some form of normalization, similar to the one used by Baker et al. [BAK98], as part of their token categorization. For bytecode content, they replace possible values of operands (e.g., register name, memory address, and constants) with their category name (i.e., memory, register, value). Finally, to retrieve the similar fragments, they model the normalized data using feature vectors. Chaki et al. [CHA11] explored the applicability of classification techniques on binaries to detect similar binaries that are originating from (1) similar source code and (2) the same compiler. Provenance-similarity is defined for two fragments when both conditions hold. Chaki et al. have argued that holding these two conditions seems reasonable in the malware and virus detection application context. A concrete problem in some environments such as .NET is detecting clones across multi-languages. To avoid dealing with several high-level languages, the intermediate language (i.e., form of compiled content) has been adapted as the sole source of information in recent studies [KRA08][JUR11][ALO12]. Kraft et al. [KRA08] used graph presentations from binaries to detect cloning between languages. In our earlier studies on .NET [ALO12], we addressed the same problem by creating a set of filters for noise reduction to improve the feasibility of such cross platform compiled code clone detection approaches.

2.2.2. Clone search

Although detecting code similarities and patterns is a well-established research area in computer science (e.g., [OTT76][SAN94]), a new research area has recently emerged that is referred to as "source code clone search", but is also known as just-in-time [BAR10], real-time [KAW09], or instant [LER10]) clone search. While clone search still shares its fundamentals with traditional clone detection, both its objective and requirements differ significantly. Common to all traditional detection applications (e.g., plagiarism detection) has been that they have a complete off-line search step to find all possible clone pairs within a static source code repository. In contrast, code clone search models can be considered to be specialized search engines that are designed to find clones of a single fragment within the corpora. Clone search approaches index source code repositories as part of their off-line processing. At run-time the input, in the form of a code fragment (i.e., query criteria), is then used to trigger the search process.

Hummel et al. [HUM10] use an inverted index which groups similar lines of code using a hash table with 128-bit hash values. Their approach locates similar fragments via the inverted index to detect and search Type-1 and Type-2 clones. In [KLX11] and [KLZ11], as part of our earlier work on clone search, we also introduced a hash-based inverted indexing approach. However, our approach combined multi-level indexing in order to support also Type-3 clone search.

SHINOBI [KAW09] builds a suffix array as their index based on transformed tokens using CCFinder's [KAM02] transformation rules. A multidimensional token-level indexing approach has been introduced by Lee et al. [LER10][LEM11] using an $R*\text{tree}$ on DECKARD's [JIA07] approximate vector matching. The language elements (e.g., assignment) constitute the dimensions of the search space. Barbour et al. [BAR10] introduce a result sampling approach that uses results obtained from other clone detection tools to find candidate clones to be indexed by their approach and then apply the Knuth-Morris-Pratt string searching algorithm [KNU77] to find the closest matches amongst indexed clones. Schwarz et al. propose in [SCH12] an approach to detect and store code similarity links to facilitate code search at run-time. Similarly, De Wit et al. [DEW09] developed a tool that monitors copy and paste commands during development for the management of code clones at run-time. Zibran and Roy [ZIB12] introduced an IDE-support for Type-3 clone search based on Rabin's fingerprinting algorithm and suffix trees. Bazrafshan and Koschke [BAZ11] exploit Chang and Lawler's search algorithm, which was originally proposed for the bioinformatics domain to find approximate code patterns.

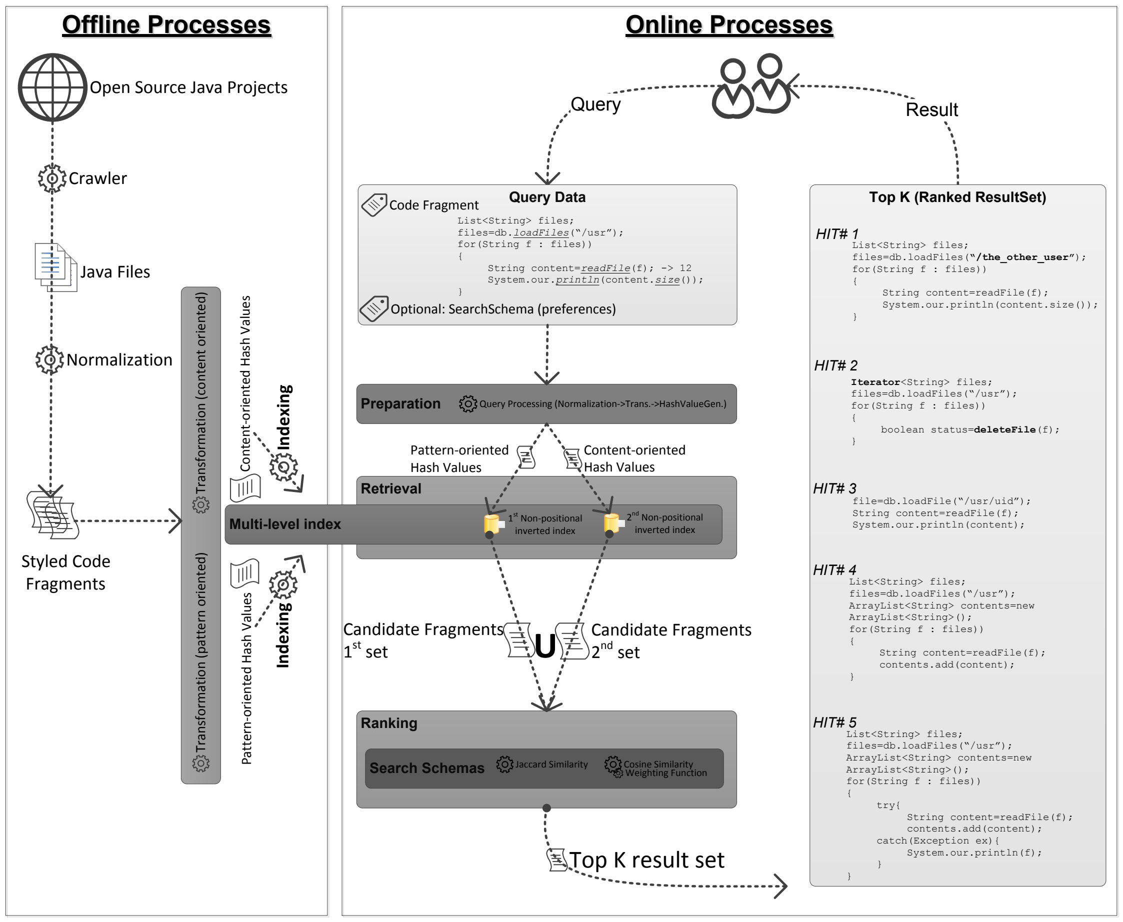
2.3. Summary

In this chapter, we illustrated the need for source code similarity search models in the programming synthesis problems, specifically, code search for pragmatic reuse. However, the proposed solutions ignore the clone detection solutions, while clone community has established the baselines for code similarity detection and measurement. This approach can be attributed to the lack of a proper clone search that supports ranking, scalability, fast response time, and Type-1, 2 and 3 detection. Such clone search model can be used as a standalone code similarity function (including search) for the program synthesis and source code search research.

3. Clone search model

In this chapter, we provide an overview of our solution for the clone search problem when Type-3 detection, scalability, fast response time, and ranking are required. The clone search model is based on the vector space model (VSM), cosine similarity, and Jaccard similarity coefficient (JSC). The VSM and JCS are two of the major models that have been used for similarity search specifically in information retrieval (IR) [MAN08]. Common to both of these models is their low computational complexity and non-positional matching. It is the non-positional aspect in particular that differentiates these algorithms from other algorithms, such as the longest common subsequent model (LCS) [HUN77], which is commonly used in the clone search and detection community. In this research we are interested in exploiting VSM and JSC, as both have been widely used in other domains such as Web retrieval [BRI98][MAN08] due to their features such as scalability.

Figure 3 illustrates our clone search solution, which is based on multi-level indexing and information retrieval ranking models. This approach is able to find the closest matches to a given query (e.g., Figure 3 query data), while returning hits from the search as a ranked result set based on their similarity degree to the search query.



Offline Processes
Open Source Java Projects
Crawler
Java Files
Normalization
Styled Code Fragments
Transformation (content oriented)
Transformation (pattern oriented)
Content-oriented Hash Values
Pattern-oriented Hash Values
Indexing
Indexing
Multi-level index
Online Processes
Query
Result
Query Data
Code Fragment
Optional: SearchSchema (preferences)
Preparation
Query Processing (Normalization->Trans.->HashValueGen.)
Pattern-oriented Hash Values
Content-oriented Hash Values
Retrieval
1st Non-positional inverted index
2nd Non-positional inverted index
Candidate Fragments 1st set
U
Candidate Fragments 2nd set
Ranking
Search Schemas
Jaccard Similarity
Cosine Similarity Weighting Function
Top K result set
Top K (Ranked ResultSet)
HIT# 1
HIT# 2
HIT# 3
HIT# 4
HIT# 5

```
List<String> files;
files=db.loadFiles("/usr");
for(String f : files))
{
     String content=readFile(f); -> 12
     System.our.println(content.size());
}
```

HIT# 1
```
List<String> files;
files=db.loadFiles("/the_other_user");
for(String f : files))
{
     String content=readFile(f);
     System.our.println(content.size());
}
```

HIT# 2
```
Iterator<String> files;
files=db.loadFiles("/usr");
for(String f : files))
{
     boolean status=deleteFile(f);
}
```

HIT# 3
```
file=db.loadFile("/usr/uid");
String content=readFile(f);
System.our.println(content);
```

HIT# 4
```
List<String> files;
files=db.loadFiles("/usr");
ArrayList<String> contents=new
ArrayList<String>();
for(String f : files))
{
     String content=readFile(f);
     contents.add(content);
}
```

HIT# 5
```
List<String> files;
files=db.loadFiles("/usr");
ArrayList<String> contents=new
ArrayList<String>();
for(String f : files))
{
     try{
          String content=readFile(f);
          contents.add(content);
     catch(Exception ex){
          System.our.println(f);
     }
}
```

Figure 3. SeClone – the proposed clone search approach

3.1. Overview

This section provides an overview of our SeClone clone search approach and its major processing steps, which include: (1) *preprocessing,* (2) *indexing,* (3) *retrieval* and (4) *ranking*. The performance of our search approach is configurable via its search schema, which consists of nine parameters (Figure 4) that can be used to customize the off-line and online processing. These configurations are not only used for performance evaluation and comparison studies, but also allow for the configuration of our approach to match the requirements of a specific search and application.

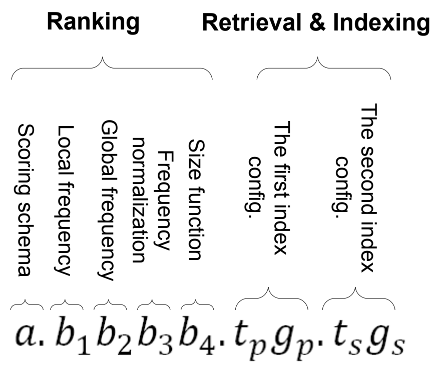


Figure 4. The SeClone search (configuration) schema parameters

Preprocessing. SeClone is a line based clone detection approach that uses Java Abstract Syntax Trees (AST) as its input for the offline preprocessing step. SeClone parses the ASTs of individual files to create a uniform representation, annotated by token types. The preprocessing step also transforms AST tokens using transformation rules, which are specified through the search schema parameters t_p and t_s. These transformation rules generate the corresponding encoded code patterns (ep) for each input code fragment. Encoded code patterns are defined to be able to identify all code fragments with certain degree of similarity.

Indexing. For this processing step, the ep dataset generated by the transformation rules t_p and t_s is used to create two[1] hash table-based indices to represent all code fragments in a single repository. The hash values can be generated for different granularities: g_p and g_s which are specified as part of the search schema.

Retrieval. During the retrieval step, all indexed code fragments are compared at run-time with the input code fragment (i.e., query). We generate two vectors ($v_{primary}$ and $v_{secondary}$) for each query q, based on the hash values of the encoded code patterns (i.e., t_p, t_s, g_p and g_s). These vectors do not hold the ordering of the elements.

[1] As discussed later, our multi-level indexing idea proposes that the actual number of indices should be at least two when both pattern and content similarity are important (e.g., Type-3 clone search).

$$Input\ (ordedred\ bag): \qquad q\ \ (l_1, \dots, l_y)$$

$$v_{primary} < ep_1^p, ep_2^p, ep_3^p, \dots, ep_n^p >$$

$$v_{secondary} < ep_1^s, ep_2^s, ep_3^s, \dots, ep_n^s >$$

A vector represents a code fragment which is used for the retrieval process from the corresponding search space in our multi-level indexing and search approach. For each vector, a look up action is performed to retrieve all code fragments indexed in the corpus, which share at least one hash value ep_x^y with the query. The union of the two clone candidate sets derived from the primary and secondary indices constitute the complete set of hits (clone candidates).

Ranking. Our ranking models are based on VSM and JSC, which can be configured as part of the search schema ($\boldsymbol{a}.\boldsymbol{b_1}\boldsymbol{b_2}\boldsymbol{b_3}\boldsymbol{b_4}.t_p g_p.t_s g_s$), with the ranking parameters being highlighted in bold. The *relevance score* is calculated for each hit returned by the *retrieval* step and these hits can be sorted by their relevance score. Figure 5 summarizes the SeClone search algorithm for both retrieval and ranking steps.

```
Algorithm $Retrieval\_and\_Ranking(q, ix^p, ix^s, a.b_1b_2b_3b_4.t_pg_p.t_sg_s)$

Input    $q$ : query's code fragment, $ix^y$: primary and secondary indices

Output   $hits$: ordered set of all candidate clone fragments based on their relevance to the query

1.  $v_{primary}[]$        $\leftarrow HashValue(q, t_p, g_p)$    //$v_{primary}$:   the un-ordered set of hash values
2.  $v_{secondary}[]$      $\leftarrow HashValue(q, t_s, g_s)$
3.  for $h$ in $v_{primary}$
4.          $hits_{primary}[]$ $\leftarrow ix^p.lookup(h)$            //find and add all fragments with at least one occurrence of $h$
5.  for $h$ in $v_{secondary}$
6.          $hits_{secondary}[]$ $\leftarrow ix^s.lookup(h)$
7.  $hits[] \leftarrow hits_{primary} \cup hits_{secondary}$              //this is an un-ordered set of all candidate clones
8.  for $hit$ in $hits$
9.          $hits[]$         $\leftarrow relevance\_score(q, hit, a, b_1, b_2, b_3, b_4)$
10. sort($hits\ on\ relevance\_score$)
11. return hits
```

Figure 5.   Retrieval and ranking (i.e., search) steps

3.2. Computational complexity

Table 2 summarizes the computational complexity of our approach for both run-time complexity and memory consumption. For the analysis, we excluded style unification, transformations, and AST build times, since they are negligible and mostly linear to the size of the input data set. We separate our analysis in three major processing steps: (1) off-line indexing for creating the hash table indices, (2) the actual search, which includes retrieval and ranking, and (3) the corpus update. T represents the inverted index size, which is $O(n)$ with n being the size of the corpus in terms of lines of code (LOC). The size of the result set is represented by c, and the total number of updated lines of code by l, with the expected lookup complexity for the inverted index as $O(1)$, since the index is hash table-based.

Table 2. SeClone computational complexity

Processing step	**Time complexity**	**Memory complexity**
Repository preparation (Indexing)	$O(n)$	$O(n)$
Clone search	$O(c * \log c)$	$O(c)$
Repository update (content addition/deletion)	$O(l)$	$O(l + T_l)$

The clone search time is $O(c * \log c)$, since in order to create the ranked result set all hits must be sorted based on their relevance scores. As a result, our model features a low time complexity for both clone search (including Type-3 clones) and repository preparation using non-positional indexing. Memory consumption for indices is also almost linear, since T is $O(n)$ as well. This cannot be further optimized without the use of compression and other abstraction mechanisms. In theory, the time and memory complexity of our clone search approach supports the core requirements (i.e., scalability and real-time response time) that we defined earlier in this thesis.

3.3. Summary

In summary, our information retrieval-based approach towards clone search provides significant improvements over our earlier SeClone solution [KLX11], as it includes the adaptation of non-positional retrieval and ranking for clone search problem. The support for ranking is an important step towards providing full-fledged similarity search for further value-added services (e.g., code search for pragmatic reuse). Nevertheless, a potential threat for our approach is the use of non-positional search models on source code content. Using non-positional models might lead to a high number of false positives since the order of source code statements determines the soundness and semantics of the program.

4. SeClone indexing model

Our clone search model uses encoded code patterns (ep) to construct its search space. An encoded code pattern is a template that defines a certain degree of similarity to match concrete code fragments. Our solution is based on the idea of encoding of code patterns to support Type-2 and 3 clone detection. However, instead of using these encoded code patterns directly, they are transformed to hash values. Hash values are useful in providing an efficient numeric representation of textual content in terms of space consumption. Furthermore, hash value based indexing and retrieval also provides fast lookup time, with a lookup complexity of $O(1)$. Both of these properties are important for our model to ensure that it is both scalable and efficient.

4.1. Encoded code pattern generation

In our approach, encoded code patterns represent a single line of code. Encoding the original code content as it is constitutes the most restrictive ep, which will only be applicable for detecting/matching exact (Type-1) clone search. Less restrictive encoded patterns will increase the recall and support both Type-2 and 3 clone search while obtaining lower precision. In our research, we defined a number of models for encoding code patterns in order to address the tradeoff between recall and precision in different contexts.

In our approach, the model of encoded pattern is defined through a transformation function and its encoding granularity. The granularity determines the number of neighboring lines of code that will be considered for the encoding. The transformation function, on the other hand, determines the template and parameterization rules. Table 3 reviews the transformations (t) supported by our approach, including their semantics (type of transformation being performed). Table 4 illustrates a concrete example for the supported transformations. A key difference is their emphasis on either content or pattern resemblance. Content resemblance focuses on token name similarities whereas the pattern resemblance enforces the order of tokens regardless of the token names. For example, the transformation function w will ignore the token ordering completely, while m attempts to keep the balance between patterns and content resemblance.

The hash function H is responsible for generating hash values that represent the encoded code pattern. The hash function uses four input parameters: the code fragment c, its offset o, clone granularity g, and the transformation function t. Since our solution is based on a line-based clone search problem, the hash function operates at line level granularity. Consequently, the input code fragment has to be at least one syntactically complete line of code. The offset refers to the line of code that is used as a target line for the hash value generation process. In order to generate all corresponding hash values of a code fragment, the function must be called several times, iterating over the target line parameter (i.e., the offset parameter).

$$H(c, o, g, t) = v$$

Granularity. The H function is able to generate hash values not only based on the target line content, but also on its neighboring content. While having a single line granularity can increase recall, such fine-granularity level often also results in a decrease in precision, as the overall similarity depends not only on the resemblance of the participating lines, but also on their order. Therefore, in order to improve our search precision, code patterns could be encoded for higher granularity levels as well. As such, we generate hash values of the target line at two granularity levels in our search approach (Table 5).

Table 3. SeClone source code transformation functions – the t parameter

Transformation function	**Full name**	**Description**	**Style unification**	**Preserve code ordering within line**	**Preserve method call names e.g., toString()**	**Preserve class names e.g., Stream**	**Preserve symbols e.g., []**	**Preserve primitive types e.g., int**	**Preserve language keywords**	**Preserve constants and literals**	**Preserve variable names**
x	exact	Same as input except for changes in style	x	x	x	x	x	x	x	x	x
l	loose Type-1	Same content for all code fragments which can be considered as Type-1 clone	x	x	x	x	x	x	x	-	x
w	word set	An unordered set of the selected tokens (i.e., only method and type tokens)	x	-	x	x	-	x	-	-	-
m	transformed tokenized method fingerprints	Preserves only method names in method call tokens and the overall pattern, while the content (i.e., names) of the other tokens are ignored via replacing them by a single place holder (e.g., #).	x	x	x	-	x	x	x	-	-
c	transformed tokenized method and type fingerprints	Similar behavior as *m* except it preserves the content of both method and type tokens.	x	x	x	x	x	x	x	-	-

Table 4. Sample outputs of SeClone source code transformation functions

The input code sample for SeClone transformation functions		
… 5: String msg=”exit 0"; 6: for(**AttributeEntity** var : t.**getAttributes**()) 7: { …		
Transformation function	***Output***	***Major changes***

x	…
	String msg=”exit 0";
	for(**AttributeEntity** var : t.**getAttributes**()){
	…
		Style unification

l	… String msg=#; for(**AttributeEntity** var : t.**getAttributes**()){ …	Unifying the literals and constants
w	{String, AttributeEntity, getAttributes}	An unordered set of selected keywords (Table 3)
m	… # #=#; for(# # : #.**getAttributes**()){ …	Unifying almost all token types except langue keywords and method names
c	… # #=#; for(**AttributeEntity** # : #.**getAttributes**()){ …	Unifying almost all token types except langue keywords, class, and method names

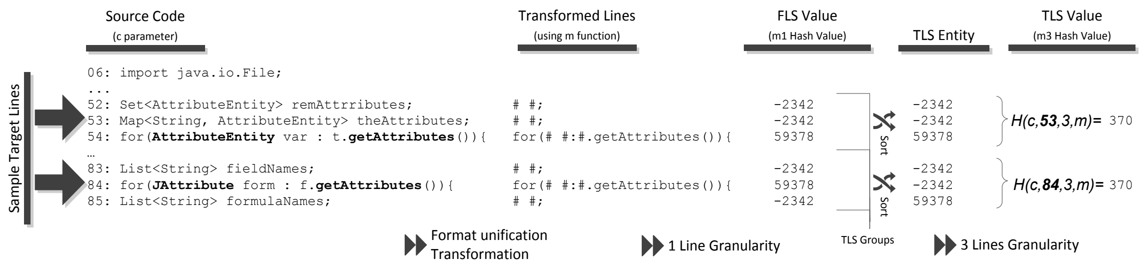
Table 5. Pre-defined granularities for the hash function – g parameter

	Granularity	**Description**
FLS	1	Only the target line that is specified by the offset parameter must be considered
TLS	3	The target line specified by the offset parameter o including $o - 1$ and $o + 1$ lines must be considered - Three lines in total

4.1.1. Hash function implementation

For line-based detection approaches, code layout unification through formatting and normalization is an essential processing step to increase recall of the retrieval algorithm [KAM02]. The layout unification requires normalization for all source code extracted from the code repository and also that of the search queries. During the layout normalization, information from Abstract Syntax Trees for each source code file in the repository is used to extract both tokens and their types. The extracted information is then used by the different transformation functions (Table 3) to perform the selected normalizations.

A combination of transformation function and granularity parameters can be used to specify the encoded pattern model. For example, $m3$ refers to the TLS granularity using the transformed lines of code with only method name preservation (m function described in Table 3). Figure 6 illustrates the complete process of how our hash function assigns an identical value to two different code fragments by exploiting the $m3$ encoded pattern model. In this case, the code fragments identified by the target lines 53 (i.e., lines 52-54) and 84 (i.e., lines 83-85) share the same pattern but their content resemblance is low due to dissimilarity in class and variable names. Unlike, syntactic token matching that will fail to identify these fragments, our approach will identify them as Type-2 clones. In this section we present, how the fundamental idea behind code transformation (e.g., normalization) and hash value based matching originally proposed for traditional clone detection approaches (e.g., [BAK92]) can be exploited for our clone search.



```
06: import java.io.File;
...
52: Set<AttributeEntity> remAttrributes;               # #;                            -2342      -2342
53: Map<String, AttributeEntity> theAttributes;        # #;                            -2342      -2342
54: for(AttributeEntity var : t.getAttributes()){      for(# #:#.getAttributes()){     59378      59378
…
83: List<String> fieldNames;                           # #;                            -2342      -2342
84: for(JAttribute form : f.getAttributes()){          for(# #:#.getAttributes()){     59378      -2342
85: List<String> formulaNames;                         # #;                            -2342      59378
```

Figure 6. Examples of SeClone hash function outputs for a specific granularity and transformation function

4.2. Non-positional and multi-level indexing and retrieval

The encoded code patterns represented by hash values are able to enforce two similarity forms (i.e., pattern and content). Figure 6 provided an example of two cloned fragments which are identified using the $m3$ model. Therefore, any hash value-based indexing and retrieval approach using the Figure 6 hash values (i.e., 370) is able to detect the two participant code fragments as clones. However, if a third fragment identical to the first pair (line 52-54) exists in the corpus, a single indexing model using a single encoded code pattern will not be capable of distinguishing differences in the degree of similarity among all three fragments. In order to support the ability to distinguish and rank the result set, we had to extend our encoded code pattern search approach. We introduced a multi-level indexing and retrieval approach for the clone search problem that deploys two (or more) indexes at one time, where each index is responsible for a specific type of similarity (i.e., content or pattern).

Additionally, the multi-level approach addresses some deficiencies related to our non-positional retrieval. The state of the art in clone detection is to consider the positional information (i.e., line number and offset) as the key information source (e.g., [KOS12]). In our solution we relax this requirement by using non-positional indices to (1) decrease the computational complexity of the retrieval and ranking algorithms, (2) reduce the memory consumption of the indexing and (3) improve the recall for the detection of Type-3 clones. However, omitting positional information in the index can lead to low precision, since the order of statements captures implementation logics and syntax. We address this concern in our multi-level indexing model by using indices at different granularity levels and thus reducing the dependency on a single information source.

Finally, to maximize both recall and precision, our indexing and retrieval solution is based on indices, which are representing different granularities and transformation functions, e.g., $l1$ and $m3$. In this example, the first index $l1$ would be used for fine-grained precise content-based similarity search. The second index ($m3$) improves the recall by adapting a relaxed pattern-based transformation function (i.e., m).

4.3. Summary

This chapter introduced our core approach of creating a search space that is based on encoded code patterns rather than source code itself. The encoded code patterns support Type-1, 2, and 3 clone search. Since our encoded code patterns can be presented as hash values, it is possible to satisfy the retrieval by a hash table-based indexing approach, which provides scalability and fast response times. Given the tradeoff between recall and precision for any encoded pattern-based retrieval model, we use a multi-level indexing approach. In this approach, each index is based on a different encoded code pattern model. Furthermore, to decrease the computational complexity of both retrieval and ranking algorithms, we adapted non-positional indexing for our clone search. In the following sections, we show how, in connection with a good ranking model, our approach can achieve reasonable precision even without access to the ordering information.

5. Ranking model

A main focus of our research is to addresses the need for *ranking* of the clone search result sets. Support for ranking is a key requirement for clone search, which determines the position of results (hits) within a result set. The position in the ranked result set depends on the degree of similarity of the $< query, hit_i >$ pair.

In our research context we are not interested in fine-grained textual similarity models (e.g., LCS [HUN77]) for relevance-based ranking. Although these are common models in the traditional clone detection context, we need a different approach since (1) the required information, such as the ordering of the fragments, is not supported by our retrieval model and (2) there are other factors to be considered such as code fragment popularity. In this chapter we describe in detail our clone search ranking model, which combines our multi-level indexing approach with different information retrieval (IR) ranking models.

5.1. Ranking approaches

As discussed earlier, the generated hash values of the encoded code patterns constitute the basic entities within our search space. Any code fragment (minimum one line of code) that shares at least one hash value with the query will be considered for the ranking. The ranking model is based on two models that have been used in IR [MAN08]: (1) Jaccard similarity coefficient and (2) the vector space model with cosine similarity.

5.1.1. Jaccard Coefficient

Jaccard similarity coefficient is a widely used set theory function, which we adapt for content matching to measure the semantic similarities. We calculate the semantic resemblance of two blocks based on their shared content (e.g., lines), regardless of their order. Our ranking model measures the content similarity of two code fragments using the numerical output of the Jaccard coefficient. We denote $s_1$and s_2 as the sets which contain entities (hash values) that belong to the search query fragment (s_1) and the matched fragment (s_2). Both sets neither contain duplicate occurrences nor do they preserve the ordering among entities, since our indexing approach is non-positional.

$$J(s_1, s_2) = \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|}$$

5.1.2. Vector space model

In addition to the Jaccard coefficient, we also take advantage of the vector space model (VSM) for the ranking of the result sets. VSM has been widely used in the information retrieval domain (e.g., [BRI98]) and a key advantage of VSM is that it provides additional flexibility during ranking compared to the Jaccard coefficient. It can exploit the entity frequency to discriminate among entities by considering their local and global popularity (occurrences). Using VSM, code fragments are represented as vectors of frequency values. In contrast to other vector based approaches, in our case a vector captures encoded code patterns of code fragments rather than terms. The similarity degree between two code fragments is calculated using the cosine similarity function that measures the angle between participating vectors.

$$cosine\_similarity(\vec{s}_1, \vec{s}_2) = \frac{\vec{s}_1 . \vec{s}_2}{|\vec{s}_1||\vec{s}_2|}$$

5.1.3. Weighting factors

In our approach, the $|x| - dimensional$ space consists of code fragments presented as vectors, e.g., $\vec{s_i} = < h_1, h_2, h_3, h_4, \ldots, h_x >$, with h_x being the weight (frequency) of an encoded code pattern x. Similar to traditional information retrieval, we also determine the local and global popularity of an entity using the occurrences from both the complete corpus and the target code fragment. While the local frequency captures the number of occurrences of an encoded code pattern within a particular code fragment, the global frequency represents the total number of code fragments with at least one occurrence of the pattern. Several models exist to calculate these local and global frequencies and weights of the entity x within a code fragment i.

The different types of weighting functions supported in our model are summarized in Table 6 and 7. For example, a combination of l local frequency (Table 6) and t global frequency (Table 7) leads to the well-known IR tf-idf model [MAN08]. Having several ranking options available provides us with the flexibility to configure the weights at run-time. In this research, we also use these functions to study the effect of different weighting approaches on the clone search performance.

Table 6. Weighting support for local frequency

Function Name	b_1 parameter value	Formula
Boolean	b	$\begin{cases} 1 & if \quad lf_{x,i} > 0 \\ 0 & otherwise \end{cases}$
Natural	n	$lf_{x,i} = \lvert local\ frequency_{x,i} \rvert$
Logarithmic	l	$1 + log(lf_{x,i})$

Table 7. Weighting support for global frequency

Function Name	b_2 parameter value	Formula
No	n	1
Simple	s	$gf_x = \lvert global\ frequency_x \rvert$
IR idf	t	$log\left(\frac{N}{gf_x}\right)$

5.2. SeClone's search schema

The search schema (Figure 7) in SeClone is used to configure different properties of the search model, including: (1) the preprocessing of the data and the creation of indices for the retrieval phase, (2) the scoring schema, (3) local frequency function, (4) global frequency function, and (5) additional information such as normalization and size comparison functions.

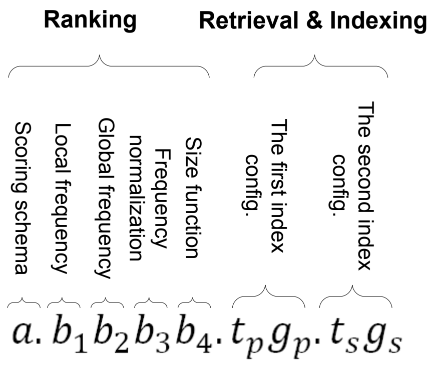


Figure 7. SeClone search schema

The first parameter of our schema template determines the high-level scoring model (Table 8), which can be a variation of cosine similarity, Jaccard similarity, or a combination of both. Furthermore, b_1 and b_2 refer to the local and global frequency functions being used (see Tables 6 and 7). If the Jaccard coefficient is used, only the boolean local frequency is applicable for the b_1 parameter; in this case b_2, b_3 and b_4 will not affect the final result and must be set to n (none) to ensure conformance with our schema template. Additionally, we consider the size resemblance between the query and the matched code fragment, which is denoted by b_4. This option is only applicable for the VSM scoring model. The size functions which are supported in SeClone are summarized in Table 9. Our search schema also supports relevance score normalization, which is denoted by b_3. Available normalization functions are n (none) and c (cosine).

$$\textit{cosine normalization} \qquad \frac{1}{\sqrt{\sum_{y=1}^{x} {h_y}^2}}$$

Table 8. SeClone scoring schemas (a parameter)

Function Name	a parameter value	Formula
Jaccard coefficient	j	$J(s_1, s_2)$
Cosine similarity	w	$cosine\_similarity(\vec{s}_1, \vec{s}_2)$
Cosine Similarity augmented with Size similarity	c	$cosine\_similarity(\vec{s}_1, \vec{s}_2) + b_4(\vec{s}_1, \vec{s}_2)$

Table 9. SeClone size functions (b_4 parameter)

Function Name	b_4 parameter values	Formula
Jaccard coefficient	j	$J(s_1, s_2)$
Naïve	n	$\begin{cases} \frac{1}{\lvert s_{hit} \rvert} & if \quad \lvert s_{hit} \rvert > 0 \\ 1 & \lvert s_{hit} \rvert = 0 \end{cases}$

5.3. Summary

In this chapter, we introduced the search schema of our clone search model. The search schema configures both the retrieval and ranking parameters used to optimize the search for a specific application context. We described in detail our ranking model, which takes advantage of IR models applicable to our non-positional indexing and retrieval approach. Given the ability to configure our search schema, an end-user can alter the search behavior at run-time based on the search requirements. For example, the $c.ltcj.l1.m3$ schema denotes that the search will use the cosine similarity scoring schema which is augmented with the Jaccard-based size function to create an IR like $tf - idf$ weighting by using a cosine normalization function. The indexing is based on single line hash values of Type-1 clones and 3-line hash values of encoded code patterns where only method names have been preserved.

6. Data characteristics study

Several issues related to our indexing heuristics can threaten the success of our research, including: (1) the ability to perform clone search with near real-time (e.g., [KAW09][LER10]) response time (latency time $\approx$100 milliseconds that is expected for interactive querying e.g., [BAS13]) affected by the characteristics of the outliers, retrieval granularity, and index growth rate, and (2) the ability to maintain the precision of the search result due to the potential collisions in our hash function.

In order to evaluate how these threats might affect our approach, we first conducted a study to observe the required characteristics of the data. A representative dataset was required as the necessary condition for such data analysis task. For this reason, we adapted the UCI dataset [UCI10], which covers over 18,000 Java open source projects from online repositories on the Internet.

6.1. Granularity effect on the clone search latency time

In the first part of our studies, we analyzed the effect of different search granularity levels on response time to (1) determine if fine-grained granularities (e.g., single line) are actually practical for real-time clone search over large amounts of data, and (2) estimate the increase in the response time by reducing the granularity. In order to answer these questions we first analyzed the number of retrieved entities for each element of a query. Identifying the number of returned matches for each query provides us with some insight about the boundaries of the response times. For this part of our study, we observe and compare the worst-case scenarios with respect to the number of matches at our two predefined levels of granularity (single and tree-line granularity).

In our empirical analysis, we first grouped source code fragments within the dataset in chunks of three lines, with each Third Level Similarity (TLS) group denoting a set of potentially similar three-line code fragments (i.e., code clone) where all fragments are satisfying an identical encoded code pattern. We then repeated the same study for a single-line granularity level, for which we used a First Level Similarity (FLS) based on pattern similarity at single-line granularity.

The total number of non-distinct source code lines extracted from the dataset is ~300 MLOC, which provides us with a sufficiently large dataset to reduce the potential bias in the data. From this dataset, we generated 30 million unique TLS groups, covering 71 million distinct lines of source code within method blocks. In our index, each TLS group refers to all occurrences of the same three-line code fragment in the whole repository. The objective is to study the number of occurrences (including average, min and max) for each encoded code pattern captured in a TLS group, since fewer occurrences result in a lower response time.

The first observation we made was that almost all TLS groups contain less than 2,000 occurrences (instances) and only a few outlier patterns, 1,220 out of the 30M (0.004%) patterns, exist that actually have more than 2,000 occurrences. Figure 8 illustrates the distribution (excluding the outliers) of TLS groups with fewer than 2,000 members across our complete dataset. Based on these observations, it is apparent that the three-line granularity tends to produce large numbers of small groups and very small numbers of large groups. On average, each TLS group (code pattern) has 2.37 occurrences. However, if we exclude patterns with only one occurrence (match) and outliers (with more than 2000 matches), the average would go up to 5.25 (Table 10).

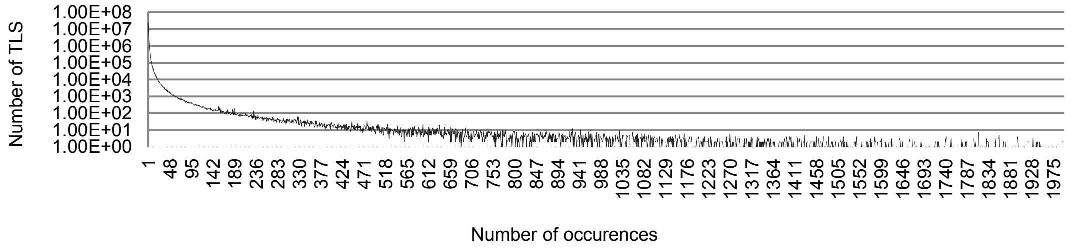


Figure 8. Occurrence frequency distribution for the 3-line (TLS) encoded code patterns

Table 10. TLS and FLS characteristics

Property	**Value**	
	TLS	*FLS*
Number of encoded code patterns	30,232,018	7,606,433
Total number of distinct lines	71,911,376	71,911,376
Number of single-member encoded code patterns (one occurrence)	22,824,697	4,770,010
Largest group size (the pattern with most occurrences/members)	1,048,575	2,937,700
Average occurrence frequency	2.37	9.45
Standard Deviation occurrence frequency	293.23	1898.75

From our analysis, we were able to conclude that three-line granularity is practical for real-time clone search, as long as outlier patterns are handled separately, since it is only for these few outliers that the response time degrades considerably. Our analysis also shows that using TLS, patterns typically occur in small-size groups (on average around 5 members). This is an important observation for real-time search context since given the small group sizes and the hash-based indexing approach, the query has to be compared against a small number of candidates at run-time.

In addition, we studied the distribution of patterns using a single-line level granularity (FLS) index, similar to our TLS study. While one would expect the performance of both granularities to be quite similar, our experiment (Figure 9) actually showed some differences between the two indices. For example, the distribution of the FLS (plus indicators) based patterns shows that the number of FLS outliers (patterns with more 2,000 occurrences (matches)) is considerably larger than the TLS's.

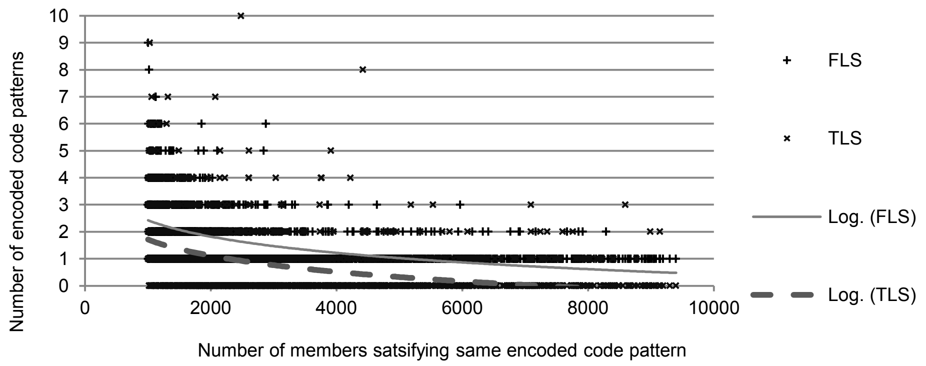


Figure 9. TLS and FLS outlier groups' distribution comparison

This observation is further supported by data in Table 10, which shows that TLS distributes the candidates into 3.9 times more groups, while its group is ~5 times smaller than the FLS's group size. Moreover, the outliers in the FLS index tend to be much larger when compared to the TLS index.

The group size directly affects the response time, since the ranking at the group level has a computation complexity of $O(c * log c)$, where c corresponds to the group size (Table 2). Our study shows that while both TLS and FLS are applicable for real-time search since c remains in a certain boundary when outliers are excluded, TLS outperforms on average the FLS granularity by a factor of ~5 (Table 10).

6.2. The outlier patterns

Outliers often introduce threats to the quality and non-functional performance of search approaches. For example, in text retrieval research, outliers known as stop words are typically eliminated as part of a pre-processing step. As our previous study showed, while we only have to deal with a very small number of outlier patterns (patterns with more than 2000 occurrences) in our dataset, these outliers might have a significant effect on the overall performance of our clone search approach. In order to be able to mitigate this potential threat, it is necessary to identify and study these outlier code clones in more detail. For example, our study showed that there exists a three-line pattern with more than one million occurrences (Table 10). If such an outlier pattern occurs in the search result set, the ranking algorithm will have to evaluate and rank all occurrences, potentially slowing down the search by a factor of 1000 compared to non-outlier searches. For this reason, we further analyzed the source code matching these outlier patterns to observe what kind of programming tasks are associated to the outliers. When analyzing the TLS patterns, we observed that only 1,220 of 30 million TLS groups (three-line code patterns) contain more than 2,000 pattern occurrences. Source code examples for the top 10 outlier patterns are summarized in Table 11.

Some of the detailed observations are: (1) members of outlier pattern #3 belong to one of the largest open source projects in the dataset (gov.nih.ncgc), which is related to genomics and contains very large files containing these pattern instances. (2) Code fragments in the outlier #6 pattern belong to classes related to the initialization of Graphical User Interfaces. (3) Outlier pattern #8 occurrences can typically be found within extraordinarily large java classes (larger than 10K LOC). In summary, the provided examples in Table 11 support the fact that, similar to the other search domains, outliers in clone search can be discarded because they are not associated with vital programming problems. It should be noted that while the (partial) exclusion of these outlier patterns has no or very little effect on the recall of our search engine, we did not exclude them in our further performance evaluation studies to ensure unbiased and repeatable results.

Table 11. The outlier code patterns

Rank	Number of Occurrence	Pattern Title	Sample Code

1	1304840	Local getter	method() {
			return variable;}

2	636846	General Setter	method(type arg) {
			this.variable = arg;}

3	445552	Unknown	s.addToWellOneBased(… new WellComponent(… l.getCompound(…), …));

4	246082	General getter	method() {
			return variabale.property;}

5	239604	Local setter	method(type arg) {
			variable = arg;}

6	124836	Consecutive new	jEdtTest = new JEditorPane();
			lblToken = new JLabel();
			jCmbLangs = new JComboBox();

7	124693	Variable&null	type var1 = null;
			type var2 = null;
			type var3 = null;

8	115230	Consecutive case	case 'value': case 'value': case 'value':

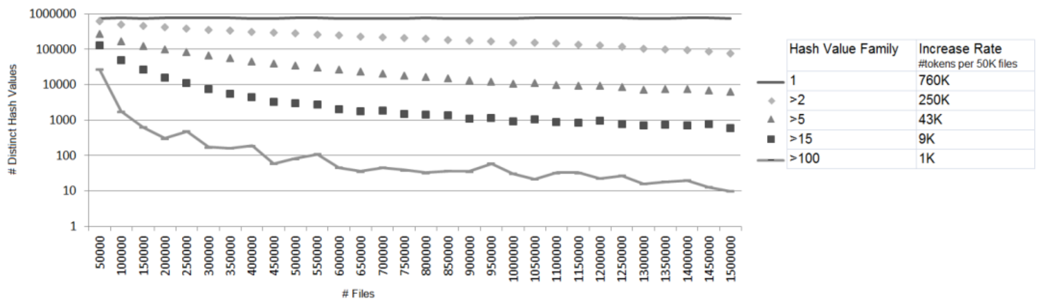
9	100900	Case&return	return "Mountain";
			case TYPE_GAS:
			return "Gas";

10	72842	Throw&new	method(…) {
			throw (new type());
			}

6.3. Index growth rate

Retrieval systems such as [BRI98] keep their indexes accessible/stored in the main memory, rather than swapped to the disk, to reduce latency times when accessing their lookup indices. In most text retrieval systems [BRI98], the approximate index size is known in advance, as it is directly related to the data characteristics in the domain of discourse (e.g., natural languages). However, data characteristics for code patterns used for the clone search problem have not yet been well studied, and as a result there exists no insight on the index size growth rate as new patterns and occurrences are being indexed. This issue can cause a threat to our approach scalability, since we do not have any prior knowledge about growth rates of indices and, consequently, the required memory resources.

For a hash table-based indexing system, total memory consumption can be estimated based on: (1) the number of distinct hash values being indexed and (2) the total number of objects. Given the fact that no prior information is available on potential growth rates, we studied the effect of repository size on the index growth rate in our research context. To be more specific, we observed how different pattern categories (and their indices) evolve as the repository size increases. For this analysis, we incrementally increased our dataset by adding chunks of 50,000 source code files to the repository. We evaluated the index increase rate for each pattern group, which is summarized in Figure 10. The analysis shows that for popular code patterns (with at least 2 occurrences), the growth decreases over time. This was expected, since as more code content is being indexed, the likelihood that newly added code fragments have already been indexed increases. However, the observation also shows that the growth rate for uncommon code patterns remains stable. That is, each chunk of 50K files will introduce an equal number of code patterns that are not going to be cloned in the future as the index grows. Finally, using the increase rate table in Figure 10, we can now estimate the index growth via the number of distinct hash values and possible pointers (duplicated patterns), to optimize memory resources and improve scalability of our search approach.



Hash Value Family	Increase Rate #tokens per 50K files
1	760K
>2	250K
>5	43K
>15	9K
>100	1K

Figure 10. Analysis of the increase rate of new hash values (TLS hashes) per file. Patterns are categorized based on their total # of occurrences per hash value.

6.4. Hash value strength

Hash table based indexing relies on its ability to maintain indices in the main memory to ensure consistent and fast access times. One approach to reduce the memory footprint is by reducing the length of hash codes, as this will directly affect the memory consumption. However, reducing the length of hash codes can potentially introduce a new threat to the strength (uniqueness) of these indices. In our approach, we opted to use only a 32-bit hash code, which is in contrast to other existing work such as Hummel et al. [HUM10], who used a 128-bit code for their clone search approach. The use of a smaller hash code (32 versus 128 bits) will not only provide (1) a 75% lower memory requirements for the indices, but can also (2) reduce the latency times.

We conducted an experiment to evaluate whether the use of a 32-bit hash value might potentially introduce a threat to the index quality in terms of collisions. For our evaluation we created 32-bit hash keys for all single transformed source code lines, using our default transformation function and the Java library hash function for strings. We extracted more than 4 million distinct transformed lines of code and analyzed the possibility of having an ambiguous key that might be used for more than two distinct lines. The result of our analysis showed that for our 32-bit hash function, the error (collision) rate is very small with 0.002%. Note this is the minimum error rate. Using different transformation functions and granularities the error rate might increases. Given this low error rate and the resulting tradeoff between precision and memory consumption, we can conclude that the 32-bit hash keys can be considered strong enough for indexing source code in our research context. This conclusion particularly holds for our research context, since for clone search we are mainly concerned with scalability and response times as key factors.

6.5. Summary

Gaining insights about data characteristics such as the index growth rate and outliers is an essential requirement and step towards creating a scalable search engine. Contrary to the other research domains [BRI98], these aspects had not yet been studied or investigated for the clone search problem. This chapter presented the result from our analysis of various data characteristics based on the code adapted from the UCI dataset [UCI10]. The insights from these studies are essential to be able to predict the latency time, index sizes, and overall quality of clone search approaches. Finally, the observations made in this chapter support the feasibility of our proposed approach based on multi-level indexing and retrieval approach for real-time scalable clone search.

7. Performance evaluation measures

As discussed earlier, our research problem shares many features with information retrieval, including ranking. Due to the fact that traditional clone detection evaluation is not yet concerned with result ranking, current performance measures used by the clone detection community do not include the evaluation of ranking feature. Therefore, to be able to evaluate the quality of our clone search ranking approach, we use existing quality and performance criteria for ranked result sets commonly used by the IR search community. The detailed definitions of the measures in this chapter are adapted from Manning et al. [MAN08].

7.1. Requirements

A key quality criterion used in the information retrieval domain for evaluating the quality of search engines is the relevancy to user expectation. That is, a search is considered to be successful if it locates documents that are not only related to the query, but also meet the end-user expectations [MAN08]. Therefore, a hit that only satisfies the relevance condition from an end-user perspective is considered to be a true positive. For example, a result returned by the query "Java", can only be considered relevant when one considers the user's expectation [MAN08], which might be referring either to the coffee concept or the programming language concept. The relevancy concept can be measured on a binary scale (relevant vs. non-relevant) or by using a more refined scale, which might consider different degrees of relevancy (e.g., highly relevant, relevant, marginal, and non-relevant).

Benchmarks are required to measure the quality of result sets reflecting the feedback of either users or experts. They constitute the "gold standard" or "ground truth". A benchmark or test suite includes three major parts: (1) the input data, (2) some queries, and (3) the pre-tagged dataset of relevant items. The dataset also typically includes relevance scores for each query and the input data, with these scores being subjective to the human experts creating the benchmark. In cases when no benchmarks are available, user studies might be performed.

7.2. The measure suite

For evaluation of ranked result sets in source code search applications (e.g., [LEM11], [KLZ12], and [KLX12]), no single measure can be considered sufficient. For our study, we identified the following categories of measures that we consider to be essential for evaluating the clone search models. The detailed definitions of the measures in this chapter are adapted from Manning et al. [MAN08].

- *Traditional measures.* Traditional measures, such as recall or precision are typically used by the clone detection and search community to evaluate the quality of any unranked result (sets). These basic measures are widely accepted since they are easy to calculate and interpret. They are also frequently applied to search engines, even if they are not able to deal with ranked result sets.
- *IR measures for ranked results*. Since most IR systems return result sets that contain some true positives (TP) and false positives (FP) within an ordered list, these measures evaluate the true positives and their rank (position) in the result set. Furthermore, similarity degree is exploited by a subset of measures in this category when all true positives are not equal in quality.
- *Measures for highly positive ranked results.* In some cases there are only a few FP in the hit list, or even none at all. While all (most) hits are TP, some of the TP should be ranked higher than others based on their relevance degree to the user expectation. In order to evaluate the ranking among TPs, additional measures are required.

7.2.1. Traditional measures (unranked result)

Precision and recall, introduced by Kent et al. (1955), are some of the most well established measures for evaluating unranked result sets. In IR they are typically based on the total number of (1) relevant items in the result set r, (2) total number of relevant items a, and (3) total number of items in the result set s. However, their application is limited, since in most cases the total number of relevant items is not known.

$$Precision = \frac{r}{s}$$

$$Recall = \frac{r}{a}$$

Accuracy is widely used to measure the quality of classifications created by machine learning algorithms. However, it has been less commonly used for IR systems [MAN08], since the datasets being search/analyzed in this domain typically contain significantly more non-relevant (99%) items for a given query compared to relevant items (1% of all data). This problem, also known as skewed data problem, will lead to situations where the size of true negatives (tn) is large enough to cancel out the effect of other relevant values such as true positives (tp). Since we are dealing in our approach with an IR system and a large dataset that will lead to skewed data, accuracy as a measure will not provide a meaningful quality measure for our search approach, and therefore has been omitted from our evaluation.

$$Accuracy = \frac{(tp + tn)}{(tp + fp + fn + tn)}$$

F-measure (introduced by Rijsbergen 1979) is another well-known candidate measure in this context. Typically, a tradeoff between recall and precision can be observed, and the importance of each measure as quality attribute might differ between users and application contexts. In some cases, e.g., regular Web search, higher precision is preferred, whereas in cases such as plagiarism detection, high recall is expected. F-measure attempts to balance both by considering recall and precision. It is also possible to discriminate between importance of the precision and recall via the β value. F-measure is calculated using precision (P) and recall (R) using a weighted harmonic mean.

$$F - measure = (1 + \beta^2) \times \frac{P \times R}{(\beta^2 \times P) + R} \qquad where \;\; \beta \geq 0$$

The default F-measure (Balanced F-measure or F_1) assigns equivalent weight to recall and precision ($\beta = 1$). Due to the significant differences between recall and precision values, F-measure uses a harmonic mean (which is always closer to the minimum value) instead of geometric or arithmetic mean.

$$F_1 = \frac{2 \times P \times R}{P + R}$$

7.2.2. Measures for ranked result sets

While many traditional measures like precision or recall are designed to evaluate unranked lists, such as an unordered set, the IR community has emphasized special measures for assessing the quality of ranked sets. In this section, we identify and introduce measures that are mostly adapted from IR [MAN08] to evaluate the ranked result set return by clone search models.

7.2.2.1. First False Positive measure

The commonly used evaluation criteria for search engines in the IR domain are the top displayed items (hits) in the result set. Studies in IR have shown that end-users tend to browse only the top items in a displayed result set [MAN08]. Furthermore, since search engines typically do not produce 100% precise results (some non-relevant hits might be displayed), search engines are expected to place as many true positives as possible in the highest ranked position of their result set (e.g., top-10). Therefore, the place of the first false positive in the displayed result list can be used as a measure for evaluating the performance of search engines. For example, given two order result sets R1 and R2, with both result sets containing 10 hits (R1 = $\langle h_1, \boldsymbol{fp}, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9 \rangle$ and R2 = $\langle h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, \boldsymbol{fp}, h_9 \rangle$), of which nine results are correct hits and one is a false positive (fp). While the precision for both results sets is 90% (9 out of 10 hits are correct), the user satisfaction for R2 would be considered higher, since the first false positive (FFP) occurs later in the ranked result set R2 (position 9 versus 2 in result set R1).

Discussion. In clone search, one typically deals with a corpus that contains a significant amount of noise (irrelevant code fragments). Therefore, from a code/clone search perspective, our search approach has to deal with two major challenges: (1) being able to detect the few relevant fragments, and (2) assigning these true positive results a higher priority than the false positives in the result sets. In such cases, First False Positive (FFP) provides a result that is easy to understand and interpret.

Weakness. Given the fact that the measure is highly dependent on the data and query characteristics, the applicability of the First False Positive measure to evaluate system performance is often limited. For example, if a corpus contains a skewed dataset with only X true positives for a given query, the best achievable result using this measure is $X + 1$. This becomes an issue particularly in cases where the number X (true positives) varies considerably for different queries. Specifically, the First False Positive measure cannot be generalized since results cannot be averaged across different queries.

7.2.2.2. "Precision at k" measure

Precision at K (P@K) is a measure that reports the number of true positives within the hit list (top K), where K can be any positive number to reflect the window size for the assessment. However, window sizes of 10, 20, and 30 are typically used for Ks. The value of K is derived by the general rule of thumb from the search engines Graphical User Interface design, where the first page usually shows only the top 10 hits. The measure itself is closely related to end-users quality perception, since users tend to consider only results on the first result page to be important and consequently are less likely to browse subsequent result pages.

$$Precision_k = \frac{tp_k}{tp_k + fp_k} \qquad where\ tp\ and\ fp\ are\ limited\ to\ the\ top\ K\ hits$$

This measure is in particular applicable when (1) the total number of relevant results is unknown and therefore no standard recall can be calculated, and (2) the number of returned items is too large to be fully validated, making the calculation of standard precision measures impossible.

Weakness. While this measure is a good candidate for evaluating search engines, especially when no very detailed and strict evaluations (e.g., “first false negative” measure) are required, its major drawback is its dependency on the query. For example, in order to provide a fair evaluation using “Precision at 10” measure, at least 10 actual relevant items must exist in the corpus for all executed queries. Furthermore, similar to the first false positive measure, the results from this measure cannot be generalized (averaged) across queries.

7.2.2.3. Normalized Discounted Cumulative Gain measure

The Normalized Discounted Cumulative Gain (NDCG) measure assesses the quality of search engines and their ranking algorithms in terms of their ability of assigning higher ranks to high quality true positive answers. This measure takes into consideration not only the relevance of hits with respect to a query but also the order of the results. Therefore, it is possible to compare the search result set for each query with an oracle. These oracles are typically manually created result sets (for each query) in the form of a list of all possible answers. Moreover, each answer in the oracle must be assigned a relevance score that presents its similarity degree (to the query). This oracle represents the best achievable result set and order, regardless of local search configurations, search algorithm, and search schema. The measure result is a number that can be used to compare different search and ranking schemata/configurations.

Details. DCG calculates the discounted cumulative gain achieved using a given search schema for query q when compared to the oracle with its manually assigned relevance scores for the top n hits. The output of DCG depends on the query and available data within the corpus ($DCG \in [0, \infty]$), and therefore it is not possible to compare the DCG of different queries with each other since the number of positive hits will depend on the data characteristics. To overcome this issue and to be able to summarize our study result we use NDCG, which is a normalized value of DCG. For the calculation of NDCG, we need to calculate the Ideal DCG (IDCG) first. $IDCG$ returns the ideal (highest achievable) DCG using the given relevance score set (from the oracle). Finally, using DCG and IDCG, we can calculate the final NDCG value.

Since the output of the NDCG function is normalized, it can be used for both (1) query comparison and (2) as an averaged measure for the overall performance of a search engine. The ability to average the measure results can also provide a concrete single output value for performance comparison purposes. For example, in our studies we use this single output value to compare the performance of different search configurations (schemata). The maximum value for the NDCG function is 1.0 for a result set that exactly matches the one from the oracle, and the minimum value is 0.0 for result sets with no true positive. The function $r(q, i)$ returns the relevancy score for the given query and the corresponding hit from the oracle.

$$DCG\ (q, n) = r(q, 1) + \sum_{i=2}^{n} \frac{r(q, i)}{log_2(i)}$$

$$NDCG(q, n) = \frac{DCG(q, n)}{IDCG(q, n)}$$

Weakness. The measure provides a fine-grained evaluation of the quality and ordering of result sets, providing a single value assessment that can, for example, simplify the comparison among different options or configurations of a system. However, the measure is only applicable when fine-grained ordering is important, otherwise measures such as Precision at K are preferred. Applying NDCG is expensive, for not only must all possible answers for each query be manually evaluated, it also requires a *similarity score* (e.g., identical, highly similar, similar, and irrelevant) for each answer. Nevertheless, NDCG is still considered as one of the state of the art search engine measures in the IR domain.

7.2.2.4. Mean Average Precision measure

Mean Average Precision (MAP), a single value measure, has been commonly applied to compare different ranking systems. For a single query experiment, the measure will simply compute the average of all precision at Ks, where Ks refers to the position of all relevant retrieved items in the result set. For experiments involving more than one query, the output is the average of all queries. MAP has been used to identify systems that assign a higher rank to relevant items.

$$AP = Average\ Precision = \frac{1}{|R|} \sum_{k \in R} Precision\ at\ k$$

$$where\ \boldsymbol{R}\ is\ the\ \boldsymbol{set}\ of\ all\ relevant\ retrieved\ items$$

$$MAP = \frac{1}{|Q|} \sum_{q \in Q} AP_q$$

$$where\ \boldsymbol{Q}\ is\ the\ \boldsymbol{set}\ of\ all\ queries$$

Weakness. MAP is an essential and low cost measure that does not require the creation of relevance scores (unlike NDCG). Only the positions of the true positives are necessary. However, since MAP does not include relevance scores, it lacks the ability to compare the relevancy-based ranking of true positives. Moreover, it is generally only suitable for queries where a reasonable number of relevant items are available; otherwise its output can be biased.

7.2.2.5. Mean Reciprocal Rank measure

Mean Reciprocal Rank (MRR) is applicable in cases where FPs (non-relevant hits) are returned at the top of the result set, specifically before the first relevant hit. This measure takes into account the fact that there is huge difference between 5 and 10 but little to no difference between 500 and 600, where the numbers are the rank of the first TP in the hit list.

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank\ of\ the\ first\ \boldsymbol{tp}\ for\ \boldsymbol{q}}$$

$$where\ \boldsymbol{Q}\ is\ the\ \boldsymbol{set}\ of\ all\ queries$$

Weakness. MRR focuses on the position of the first true positive in the result set, and works best for cases with (1) very few tps, and (2) some fps occurring at the top of the result set. Therefore, it can be considered a complementary measure to MAP that is best applied in cases when only a few relevant hits are reported and MAP is not suited.

7.2.2.6. R-precision measure

R-precision is equal to the output of Precision at R measure within the result set, with R being equal to $|AllRelevants|$. $AllRelevants$ is the set containing all relevant results for a given query (which even could be incomplete sometimes). From a different point of view, it is equal to the recall at R.

$$R - precision\ = \frac{r}{|AllRelevants|}$$

$$\boldsymbol{r}\ is\ the\ number\ relevant\ retrieved\ hits\ within\ the\ most\ top\ \boldsymbol{R}\ hits\ (R = |AllRelevants|)$$

Weakness. While the measure is useful to average its results (which is in contrast to the Precision at K measures), interpreting this measure is more difficult.

7.2.3. Measures for highly positive ranked results

In some cases, there is no (or only a few) fp in the hit list (e.g., top 10). While all hits might be true positives, some true positives are typically ranked higher by end-users than others. Assessing this type of ranking requires measures to take into consideration the order of tps in the ranked result set based on user preferences. Several measures have been introduced to assess the ranking performance of positive result sets [KZH10].

7.2.3.1. Normalized Kendall's τ distance

Kendall's τ measures the dissimilarity of the items' order against the ideal order [LAP06]. Suppose π and σ denote two orderings of same item set with size of N. $S(\pi,\sigma)$ is the minimum number of switches between adjacent items to make the first ordered list identical to the second ordered list.

$$\tau = 1 - \frac{2 \times S(\pi,\sigma)}{N(N-1)/2}$$

7.2.3.2. Spearman's rank correlation coefficient

This measure compares the rank of each shared retrieved item in the two subject ranked lists denoted by π and σ where number of items is equal to N.

$$Spearman = 1 - \frac{6\sum_{i=1}^{N}(\pi(i) - \sigma(i))^2}{N(N^2-1)}$$

$$\boldsymbol{\pi}(\boldsymbol{i}) \text{ and } \boldsymbol{\sigma}(\boldsymbol{i}) \text{ are referring to the rank of item } \boldsymbol{i} \text{ in the corresponding hit list}$$

Discussion. As Lapata [LAP06] pointed out, the main difference between Spearman's and Kendall's measures is that Spearman's measure is more popular and focuses on the pure rank values, whereas Kendall's measure has more emphasis on the relative order of items.

7.2.4. Non-functional performance measures

In our research context, non-functional measures can also have an effect on user satisfaction, mainly related to the ability to provide near real-time services for other applications. Among the measures that evaluate non-functional performance of a clone search engine are: (1) indexing time, (2) querying latency time, and (3) corpus size. These performance measures can be calculated automated and are simple to derive.

7.3. Summary

Assessing the quality of clone search (models) differs from traditional clone detection. While traditional clone detection approaches deal with unranked result sets where measures like recall and precision matter, they do not consider the order of the results being displayed. This is in contrast to clone search, where, as in other search approaches, the ranking of results (ranked hits) becomes a key quality criterion. While evaluation measures designed for unranked result sets are useful (e.g., precision and recall), other evaluation measures which are developed for ranked result sets must be adapted to provide a more comprehensive evaluation of a clone search model. As part of our research, we selected and summarized several ranked result set quality measures, originally used by the IR community [MAN08], for our clone search context.

8. Performance evaluation

The preliminary insight regarding the feasibility of our solution and run-time behavior is provided by the data characteristic study in Chapter 6. In order to conduct a detailed performance evaluation study, we have used the gained insight to deploy a concrete instance of our clone search approach with our source code corpus, which contains source code facts from over 25,000 open source Java projects [KLF12] that are crawled from the Internet. The key objectives of our evaluation is (1) to confirm that our proposed model can meet the core requirements of a clone search, such as scalability and fast response time and (2) to compare the different search schemas (search configurations) supported by our model.

Benchmarks are a commonly used approach for evaluating the quality of search engines. In order to be able to evaluate the different features of our model (SeClone), including both retrieval and ranking, we require a benchmark that meets a set of minimum requirements: the corpus (1) should be large to reduce the effect of individual outliers, (2) contains a set of representative queries (code fragments) to be used as search criteria, (3) includes a sufficient number of relevant Type-1, 2, and 3 clones, and (4) covers the clones' fine-grained relevance scores. To the best of our knowledge, there exists no clone search benchmark that satisfies all these requirements. Therefore, prior to our evaluation, we had to create such a clone search benchmark based on the mutation generation framework [RJC08][ROY09][SVJ13]. An overview of our benchmark creation process and the evaluation process is shown in Figure 11. As part of the benchmark creation, we take advantage of an existing mutation generation framework [RJC08][ROY09][SVJ13], which we used to automatically generate Type-1, 2, and 3 clones from 50 randomly selected code fragments (query inputs). For these 50 code fragments, we generated a total of 650 related Type-1, 2, and 3 clones. Note that 50 is the acceptable number of queries that a benchmark must cover [MAN08].

For the benchmark preparation, we injected not only these 650 clones (code fragments) generated by the mutation framework into our repository (which contains 356M LOC), but also performed an extensive manual inspection of ~80K code fragments for relevance score assignment. We then used this benchmark to assess SeClone's search performance using the six measures introduced in the previous chapter, while analyzing over 32 different SeClone configurations (search schemata). This evaluation involved 1600 querying actions for which a clone search was performed, resulting in 117,000 search results (hits)[2]. The following sections describe in more detail our evaluation approach, its outcome, and the summary of our findings.

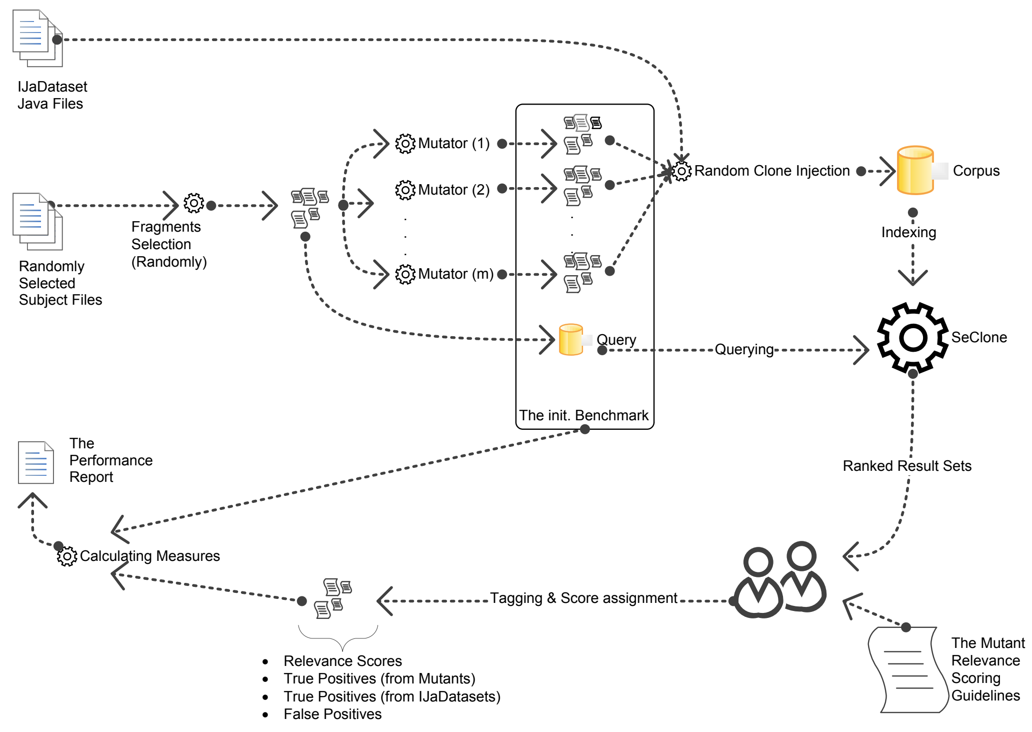


Figure 11. The performance evaluation approach

[2] Note, 117,000 hits belong to the complete benchmark that includes 2,000 querying actions, with 400 of these querying actions being used for our preliminary studies.

8.1. The candidate search schemas

SeClone supports different configurations through its search schemata, which allow different search models, indexing granularities, and content transformation functions. From an end-user perspective, the selection of a search schema (configuration) is often the key to meeting the specific application needs. We conducted a detailed analysis based on 32 candidate configurations to determine their effect on the quality of the result sets and to be able to provide end-users with some guidance during the search schema selection.

In chapters 3, 4, and 5 we introduced in detail the SeClone search schema and the two categories of options: (1) parameters related to the ranking approach (parameters: $a.b_1b_2b_3b_4$) and (2) parameters on how the data is processed for indexing and clone analysis (parameters: $t_pg_p.t_sg_s$). We selected four ranking configurations and eight indexing (analysis) configurations, which provided us with 32 combinations (details are shown in Table 12).

$$SeClone\ Search\ Schema \qquad\qquad a.b_1b_2b_3b_4.t_pg_p.t_sg_s$$

Table 12. Selected SeClone search schemas for the evaluation phase

The first parameter group (ranking) $\boldsymbol{a.b_1b_2b_3b_4}$		**The second parameter group (indexing)** $\boldsymbol{t_pg_p.t_sg_s}$	
j.bnn (Jaccard coefficient similarity approach)	×	*x1.m1* *x1.m3* *l1.m1* *l1.m3* *c1.m1* *c1.m3* *w1.m1* *w1.m3*	=32 search schemas
w.nscn (Cosine similarity using natural frequency)			
w.ltcn (Cosine similarity using tf-idf like freq.)			
c.ltcj (Cosine similarity augmented with Jacacrd size similarity using tf-idf like frequency)			
Total 4		Total 8	

8.2. The corpus and environment configurations

For the deployment of SeClone, we used a Linux-based system with a 3.07 GHz CPU (Intel I7) and 24 GB of RAM. During our run-time evaluation, a single process/thread schema was used, except for the Java virtual machine processes such as garbage collection.

In order to evaluate the scalability, response time, and ranking, and to observe the handling of extreme noise, we require a reasonably large corpus. For the SeClone evaluation we originally created IJaDataset, a large multipurpose source code data set. The dataset contains Java source code data crawled and downloaded from major open source code repositories (e.g. Sourceforge) [UCI10]. The compressed raw data size is approximately 390 GB and contains 3,431,111 Java files from over 18,000 open source projects. After downloading the source code files [UCI10], we performed several data cleaning steps, such as: (1) we removed all non-Java source code and duplicate Java files, (2) using a Java parser, we detected and removed all unparsable files (a total of 14,386 files), and (3) we identified and excluded 197,056 Java interfaces, as interface files do not contain any significant amount of code. After these cleaning steps, our IJaDataset contains 1,500,000 unique Java classes, with a total of 266,635,570 raw lines of code.

The most recent version of the IJaDataset (Version 2.0) has been updated with data crawled in 2012 as part of our SeCold project [KLF12]. This dataset covers approximately 25,000 projects and includes Java classes without package specification (default package). The dataset is based on source code files that were downloaded from SVN, Git, and CVS repositories from SourceForge and Google Code. To remove high-level duplications in the dataset, only one Java File is selected for each available class name identified by its fully qualified name (FQN). During the filtering of such duplications, we were biased toward files that appeared in the "trunk" directory. The crawled data (with duplicated files) initially included 12 million files, but were reduced (through filter) to 3 million files (2.7M regular Java class source code files and 140K files with default package).

We then successfully indexed all 356M LOC in the IJaDataset (Version 2.0) with SeClone to create a single, searchable corpus.

8.3. The benchmark

A high-quality benchmark for clone search should not only include queries and their correct answers, but should also contain a variety of clone types (specifically Type-3 clones) for these queries. Having such a rich benchmark provides not only the basis for evaluating our core SeClone search engine, but also for evaluating its capacity for ranking and Type-3 detection. Using the mutation framework introduced in [ROY09], we created our initial benchmark using 50 code fragments (queries) and their mutants in the form of Type-1, 2, and 3 clones. We selected a mutation framework configuration that automatically generates 13 clones (4 Type-1s, 3 Type-2s, and 6 Type-3s) for each query. In case of code insertion when generating Type-3 clones, the mutation framework uses random code snippets available in its corpus. An overview of the 13 automatically generated clone variations using the mutation framework is given in Table 13. The generated clones were then included and indexed as part of our SeClone corpus. Using this mutation approach provides us with known true positives in advance. Therefore, we are able to (partially) measure the recall in addition to the other precision-like measures. It should be pointed out that since the corpus contains millions of indexed lines of code, SeClone will not only detect and retrieve the seeded clones, but will also most likely include other (correct) clones in the search results.

Table 13. Available clones for each query in the benchmark and their details

ID	**Description (changes comparing to the query)**	**Clone type**	**Our relevance score**
1	no change	Typ-1	5
2	changes in whitespace	Typ-1	5
3	changes in comments	Typ-1	5
4	changes in formatting	Typ-1	5
5	semantic renaming of identifiers	Typ-2	4
6	arbitrary renaming of identifiers	Typ-2	4
7	arbitrary change of an literal	Typ-2	4
8	replacement of identifiers	Typ-3	3
9	small insertion within a line	Typ-3	3
10	small deletion within a line	Typ-3	3
11	insertion of one or more line	Typ-3	2
12	deletion of one or more line	Typ-3	2
13	modification of entire line	Typ-3	3

8.4. Assignment of relevance scores

As discussed earlier, when evaluating the performance of search engines, solely measuring true positives is not sufficient, since one also should consider the relevance (score) of the return search results (hits) with regard to a given search query. Therefore, for our evaluation, we assign scores in the range between 0 to 5 to indicate the relevancy of a hit to the given input query, with a score of 0 indicating that a particular result shows no relevancy (false positive in our research context), and scores between 1 and 5 denoting that a hit has some degree of similarity (true positive $\langle query, hit \rangle$ clone pair). Increasing scores indicate higher levels of similarity/relevance, with a score of 5 being an exact (Type-1) match. As part of creating our benchmark we have initially assigned relevance scores to the 650 cloned fragments that were generated by the mutation framework, indicating their relevancy to the corresponding (clone fragment) query. Table 14 summarizes the basic guidelines we applied for assigning the relevance scores to clone fragments.

Table 14. Relevance scores guideline

The assigned score	Scoring guideline
0	Non-relevant
1	Relevant (partial similar under Type-3)
2	Relevant (Type-3 with modification of few lines)
3	Relevant (Type-3 with one line different)
4	Highly Relevant (Type-2)
5	Highly Relevant (Type-1 / exact)

Given the size of our corpus (25,000 projects and 356 MLOC), there is a good chance that other true positives might be reported during the evaluation process. The relevancy of detected and reported clone pairs depends not only on the returned injected clones but also on the non-seeded and reported clones, which must also be considered as part of an overall evaluation. We therefore manually (1) evaluated all reported hits to determine if they are actual true or false positives and (2) assigned the proper relevance scores.

Since it is both impossible and unnecessary to consider all potential hits retrieved for each query in the benchmark (a query might return thousands of hits), we decided to consider only the top K hits. While it is common best practice in the IR and search community to consider the top 10 hits, we decided to increase the evaluation scope by including the top 60 hits. This extended evaluation is motivated by the characteristic of our corpus, considering the fact that we have generated and included at least 13 controlled, true positives (clones generated by the mutation framework) for each query.

As part of our evaluation, SeClone reported for the 2,000 executed queries[3] a total of 117K hits (clone results) using the top 60 criterion. We used some basic heuristics (based on hit size and keywords) to automatically identify some of the false positives and eliminate them from the manual analysis process. Using these heuristics, we were able to automatically eliminate 37K false positives that no longer needed a manual inspection/scoring. We then manually assigned

[3] 400 querying actions out of the total 2,000 executed queries belong to our preliminary studies and testing

relevance scores to the remaining 80K results (32K distinct $\langle query, hit \rangle$ pairs) following the guidelines (Table 14). Table 15 summarizes the details of the manual assignment of relevance scores. As part of the scoring process, we not only considered syntactical but also semantic similarities. That is, hits that can be considered as Type-3 and relevant (True Positive) in other application domains (e.g., clone detection) might be non-relevant in our context (due to the semantic and syntactical differences), and therefore receive a relevancy score of "0".

Table 15. The evaluation steps and hits manual investigation details

Property	**Value**	
Total search schemas	32	
Total benchmark queries	50	
Total querying experiments	2000	
Result set limit	Top 60	
Total retrieved hits	117K	
Total number of hits which are automatically ignored using heuristics	7.7K (size heuristic) 28K (keyword heuristic)	
Total number of hits which are tagged manually	81K (32K distinct $\langle query, hit \rangle$ pairs)	
	Breakdown	
	Relevance Score	***#hits***
	0	34K
	1	14.9K
	2	3.6K
	3	15K
	4	4.9K
	5	8.8K

8.5. Evaluation result

After our initial review of the reported hits and their characteristics (Table 15), we selected six measures from our measure suite introduced in chapter 7. The evaluation showed that our clone search model is not only scalable and provides fast response times (~100 ms), but is also capable of successfully detecting Type-1, 2 and 3 clone types. Assessing the quality of our ranking approach shows that the model is capable of placing the true positives at the top of the result set. Certain search schemata were capable of achieving even a 100% recall and precision for top K (e.g., top 15) result sets. Since SeClone search schemas rank result sets based on their content similarity, in most cases, Type-1 and Type-2 clones (similarities) are consistently placed in the correct relative order and position within the result sets. For Type-3 clones, the position in the result set depends on the dissimilarity between the clone and the query fragment.

8.5.1. First False Positive

Figure 12 provides a summary of the result for the *First False Positive* (FFP) measure based on the average values (for all queries) across all 32 search configurations (schemata). The results show that the first false positive appears on average at the 25th position for most schemas. Among the 32 schemata, four of them considerably outperform the others by achieving the first false positive at position 30. Furthermore, using the results provided in Figure 13, it can be observed how $c.ltcj.l1.m3$ and $j.bnnn.l1.m3$ schemata outperform the other schemata, specifically $w.ltcn.l1.m3$ and $w.nscn.l1.m3$.

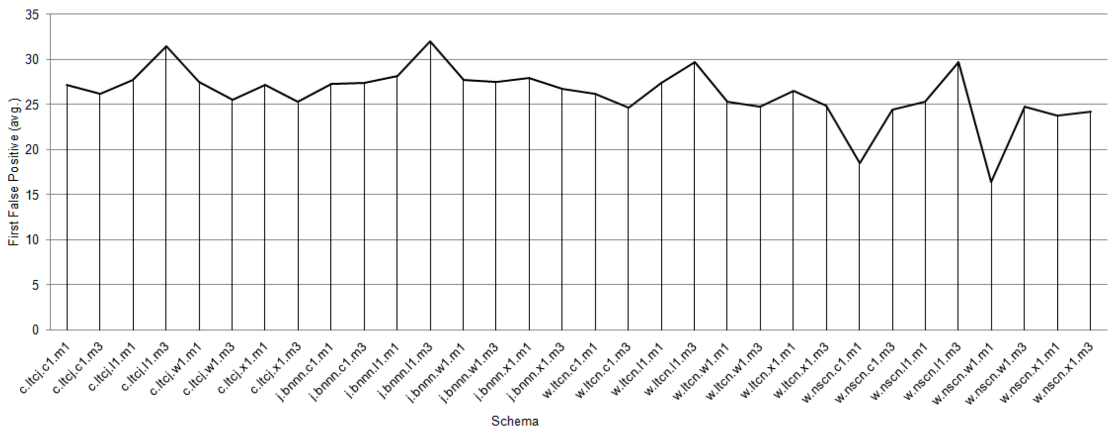


Figure 12. Summary of First False Positive measure result (average values)

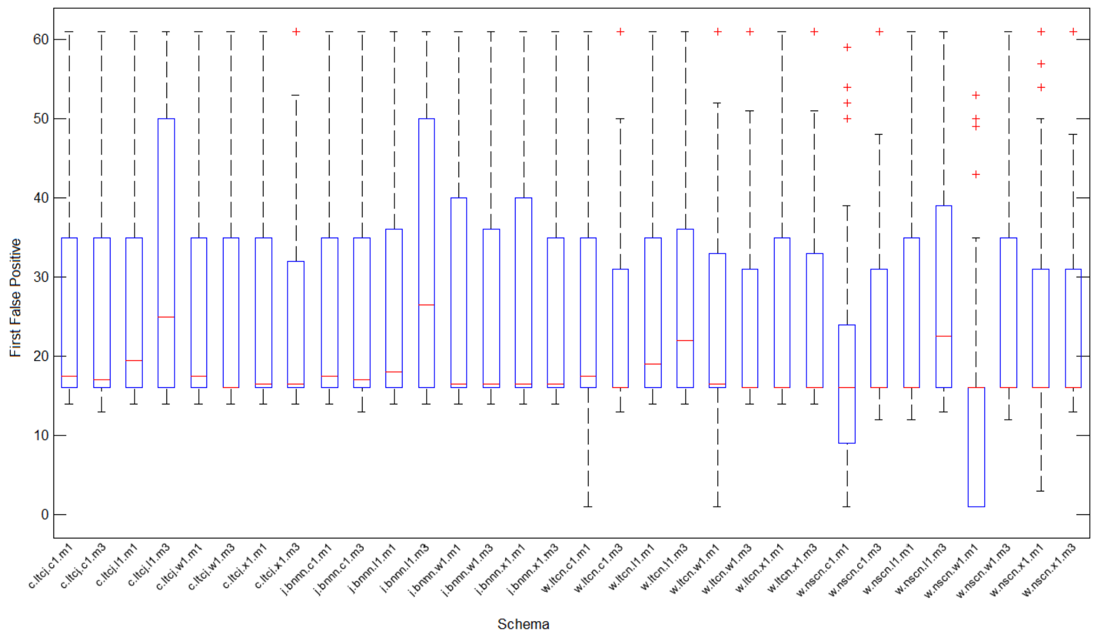


Figure 13. Details of First False Positive measure result

As discussed previously, we use two heuristics to reduce the number of hits for the manual relevance score assignment process. In order to evaluate the impact of these heuristics, we applied them on only half of the queries in the benchmark (queries 26-50), while we manually evaluated all hits of the other queries (#1 to 25). The results in Figures 14 and 15 show that our heuristics have not affected the *overall* outcome of the study considerably.

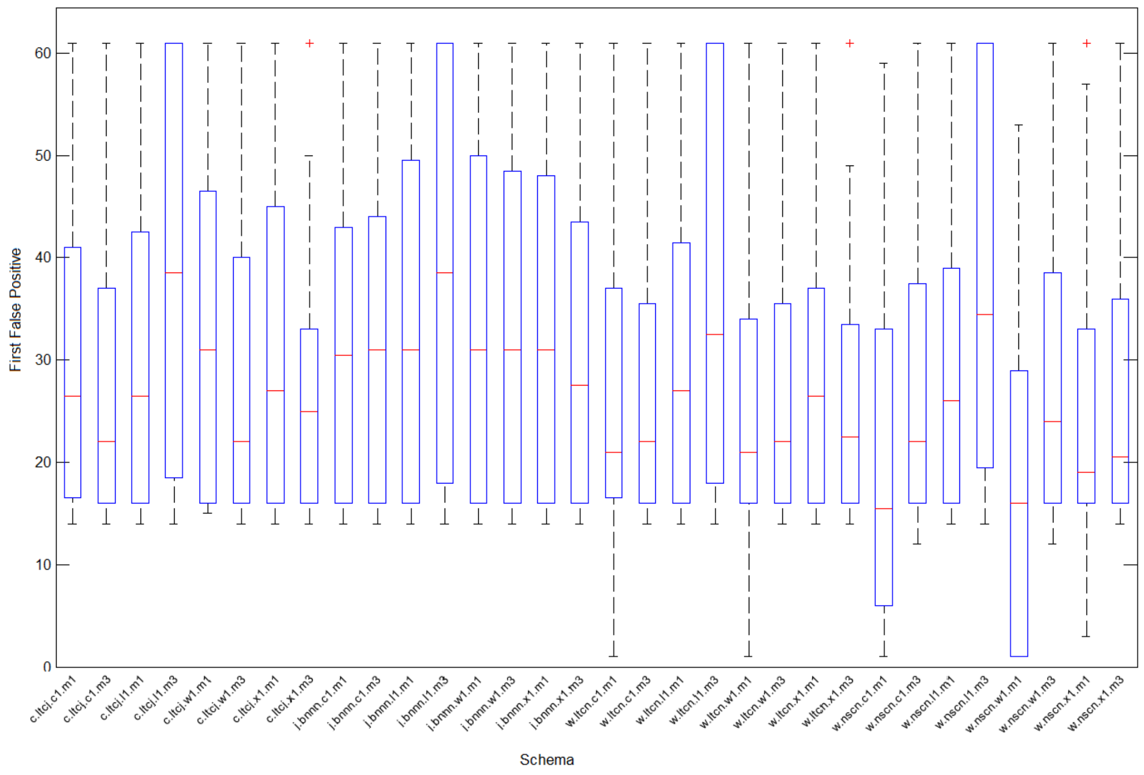


Figure 14. First False Positive measure result (queries 1 to 25, without heuristics scoring)

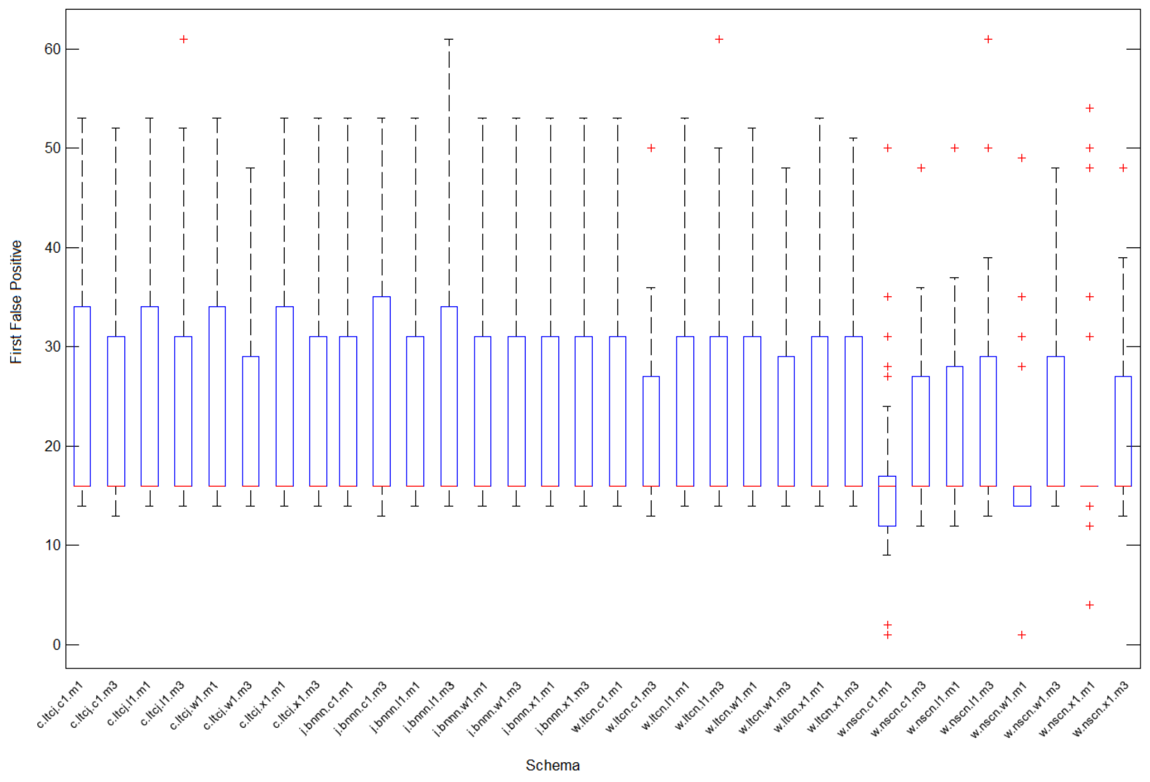


Figure 15. First False Positive measure result - only queries 26 to 50

8.5.2. Precision at K

For our evaluation using Precision at K measure (P@K), we considered 7 different scenarios: K = 10, 15, 20, 30, 40, 50, and 60. The motivation for evaluating these different K values was to provide us with a more comprehensive picture of SeClone performance as K increases. We limited the K value to a maximum of 60, since we only tagged the top 60 hits during our relevance score assignment step. Figures 16 and 17 show the precision at 10 and 15 results, with SeClone achieving 100% precision for both ranges. As expected, the precision values drop as the K values increase from 20 to 60 (Figures 18, 19, 20, 21, and 22). The major reason for this drop in precision is mainly related to data scarcity, since as part of our benchmark we generated (through the mutation framework) and injected only 13 confirmed clones for each query. That is, precision at values higher than 13 depends on data available in the corpus, which is nondeterministic given the size of the corpus and the differences among queries. An interesting observation can be made for Precisions at K=20, 30 and 40 for schemata such as $c.ltcj.l1.m1$, when the second index uses the m transformation function at the single line granularity level. In these cases, the search schemas actually achieved the highest median value. This observation can be explained by the fact that for such a fine-grained (line-level) index, the search engine was able to detect a large enough number of true positives in the corpus to achieve higher recall.

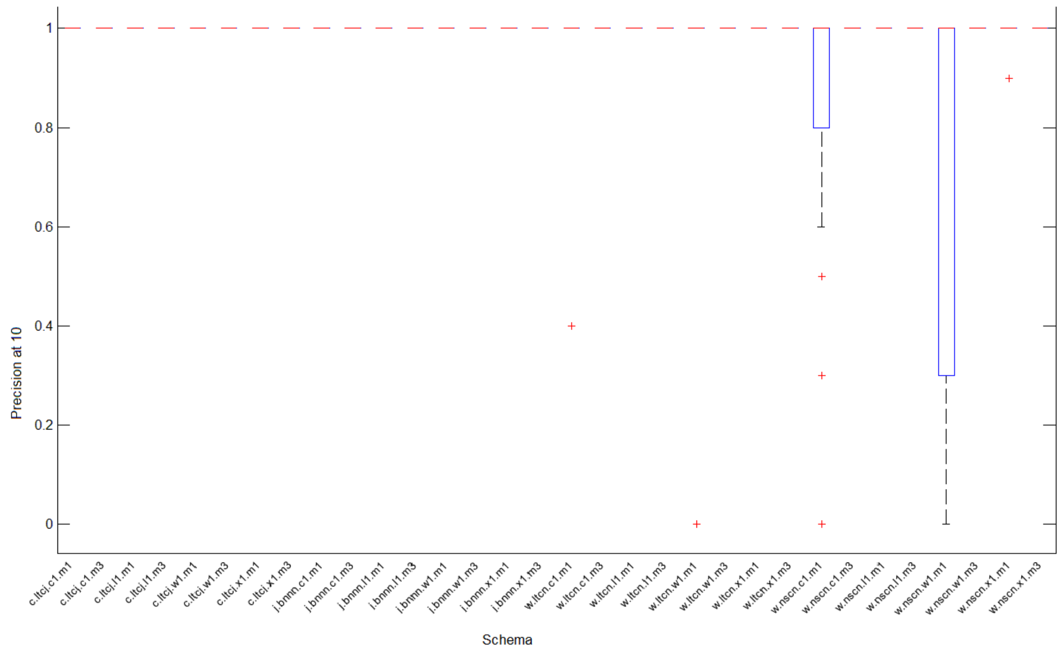


Figure 16. Summary of Precision at 10 measure

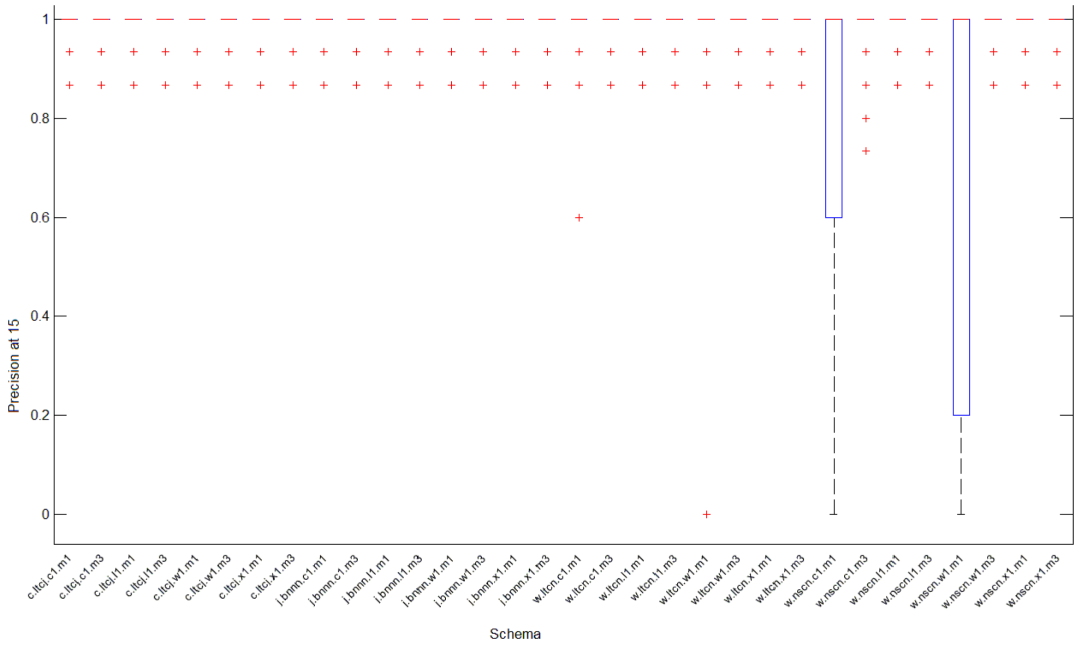


Figure 17. Summary of Precision at 15 measure

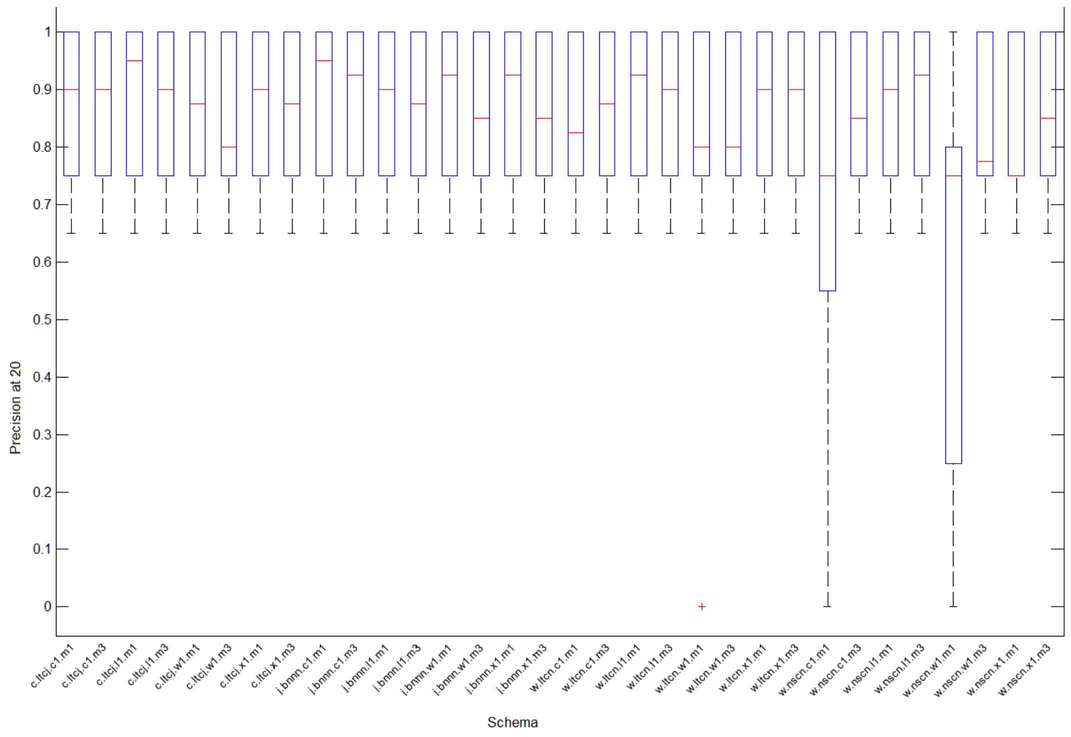


Figure 18. Summary of Precision at 20 measure

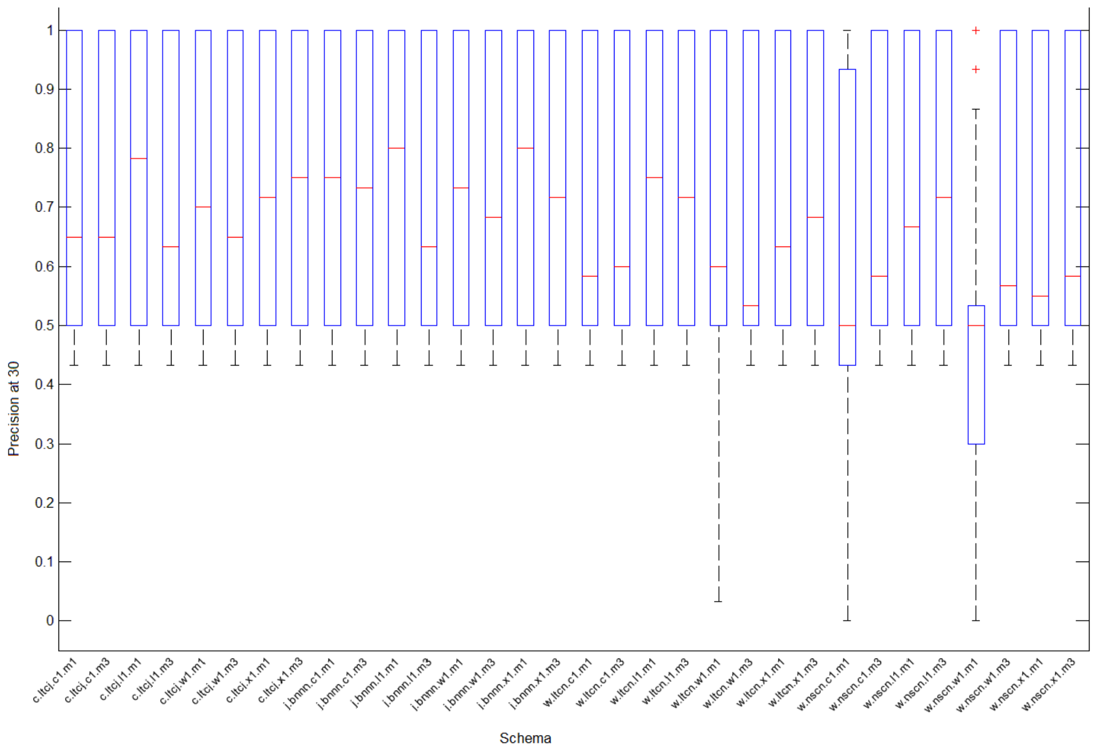


Figure 19. Summary of Precision at 30 measure

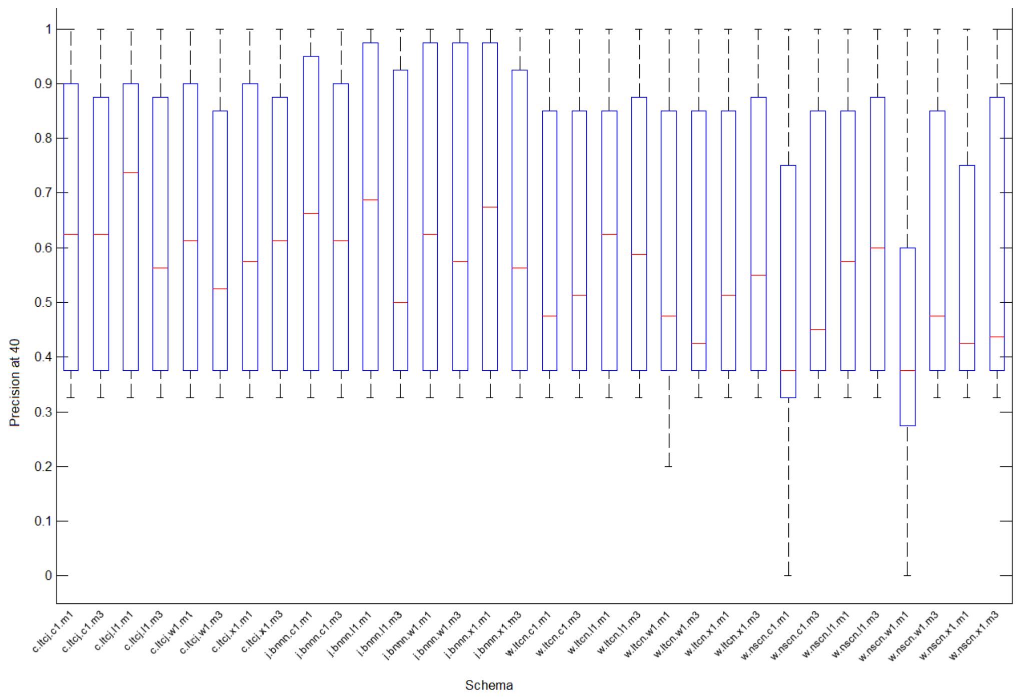


Figure 20. Summary of Precision at 40 measure

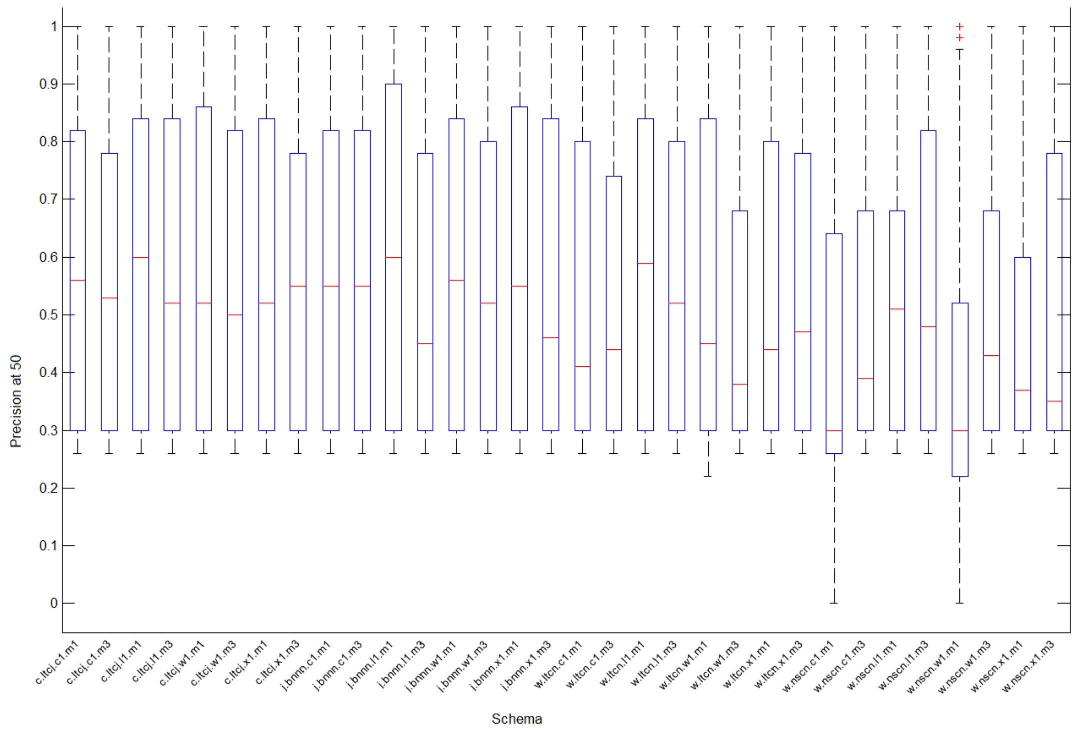


Figure 21. Summary of Precision at 50 measure

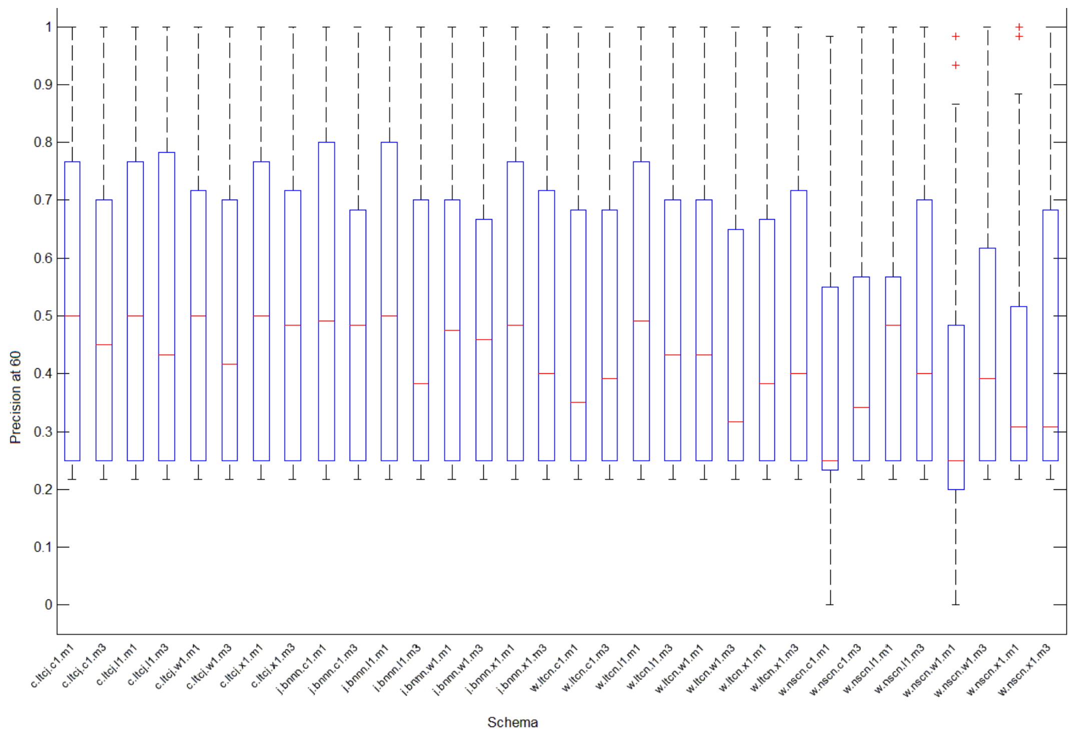


Figure 22. Summary of Precision at 60 measure

8.5.3. MAP

As part of our evaluation, we further assessed the SeClone ranking feature using the Mean Average Precision (MAP), a single value measure typically used in the IR community to compare different ranking systems. For a single query experiment, the measure will simply compute the average of all Precision at Ks where Ks refers to the position of all retrieved relevant items in the result set. MAP is useful when the degree of *similarity* (relevance score) of true positives is not of importance. Figure 23 compares the 32 different schemata with respect to the MAPs. While most of the schemata achieved a MAP of close to 1 (best), we could also observe that, similar to the First False Positive study, $c.ltcj.l1.m3$ and $j.bnnn.l1.m3$ outperform the other schemata.

We also studied the effect of our automated heuristics for benchmark tagging on the MAP. In Figure 24, one can observe that the results for queries 26-50 (after applying the heuristics) decreased slightly, providing more evidence that the heuristics have no lasting effect on the evaluation overall outcome.

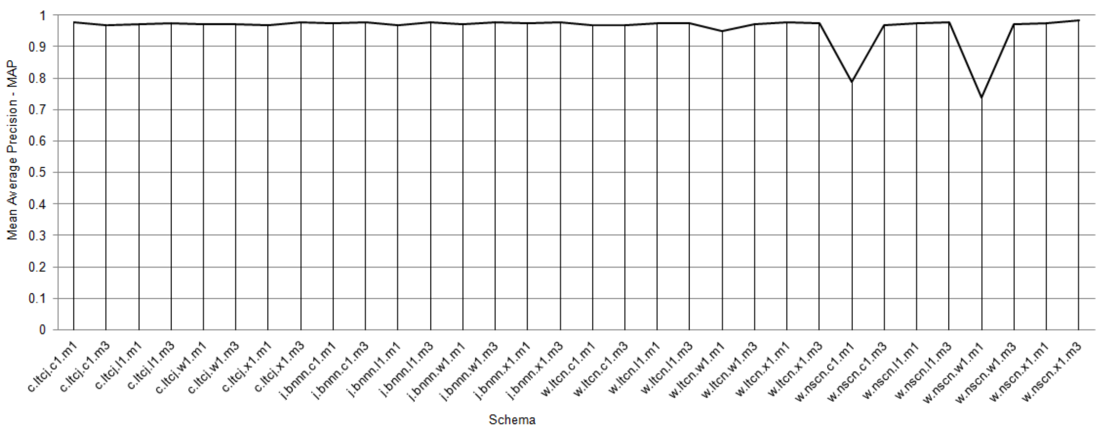


Figure 23. Summary of MAP measure results

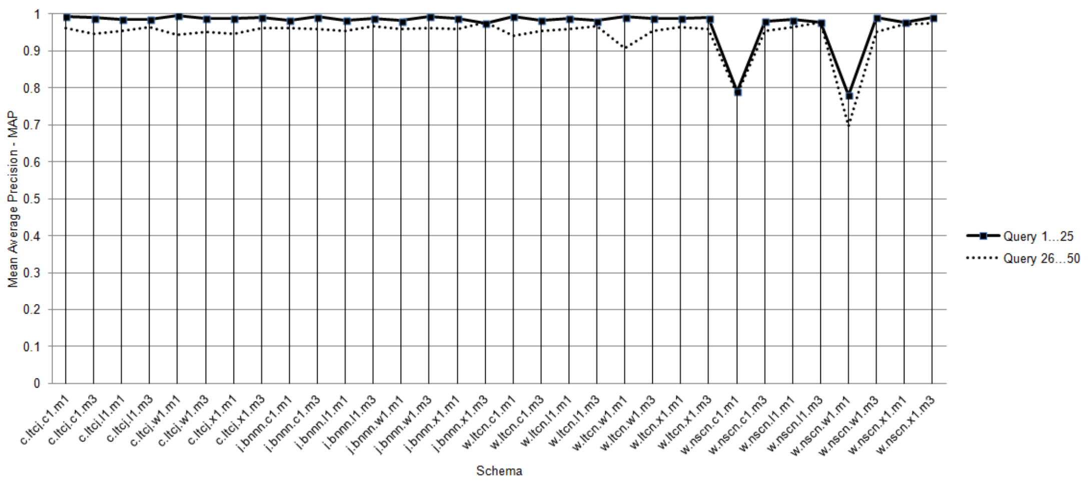


Figure 24. MAP measure results for queries tagged with (26-50) and without (1-25) heuristics

8.5.4. Normalized Discounted Cumulative Gain

To evaluate SeClone *ranking* for applications where the relevance score of true positives are emphasized, we used the Normalized Discounted Cumulative Gain (NDCG). The average values, as well as details of our NDCG experiments, are shown in Figure 25 and 26. In general, the result supports and confirms our earlier observations. Additionally, Figure 26 highlights that from NDCG perspective, the *x1.m3* index configuration outperforms the other configurations.

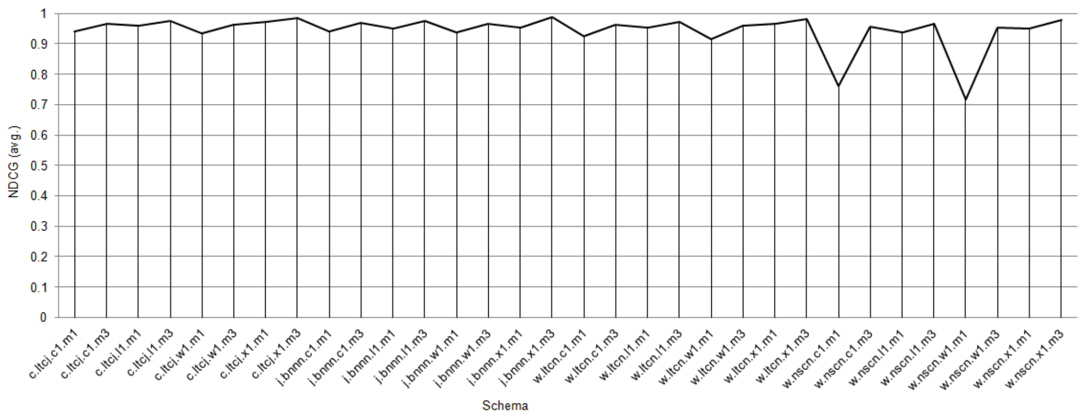


Figure 25. Summary of NDCG values

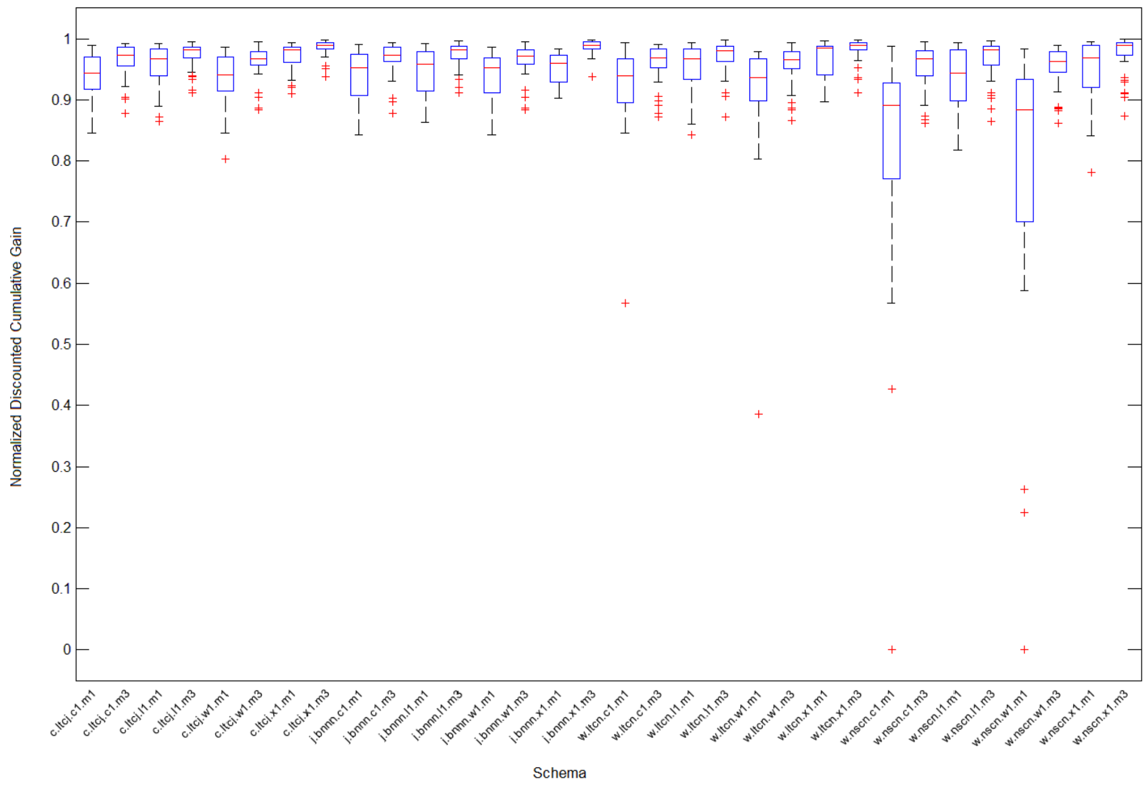


Figure 26. Details of the NDCG studies

8.5.5. Kendall tau

While the focus of the previous studies was mainly on evaluating the performance of the different schemata, in this study we focus the ability of two candidate schemas to achieve a perfect detailed ranking, where the ranking would report Type-1, Type-2, and Type-3 clones based on the scoring guideline introduced in Table 13. Kendall tau is exploited as a measure for this study, as it is capable of providing a fine-grained comparison of highly positive result sets.

The two candidate schemata are selected from amongst the schemas with promising results for FFP, P@K, MAP, and NDCG, with each candidate using a different ranking model (VSM vs. Jaccard). Figure 27 presents the Kendall tau results. All Kendall tau related calculations are made using Wessa online services [WES12]. The result shows some difference among the two $c.ltcj.l1.m3$ and $j.bnnn.l1.m3$ schemata. Although the median values for both schemata are close, the Jaccard coefficient search schema ($j.bnnn.l1.m3$) outperformed the VSM-based schema by providing consistent (better) ranking results.

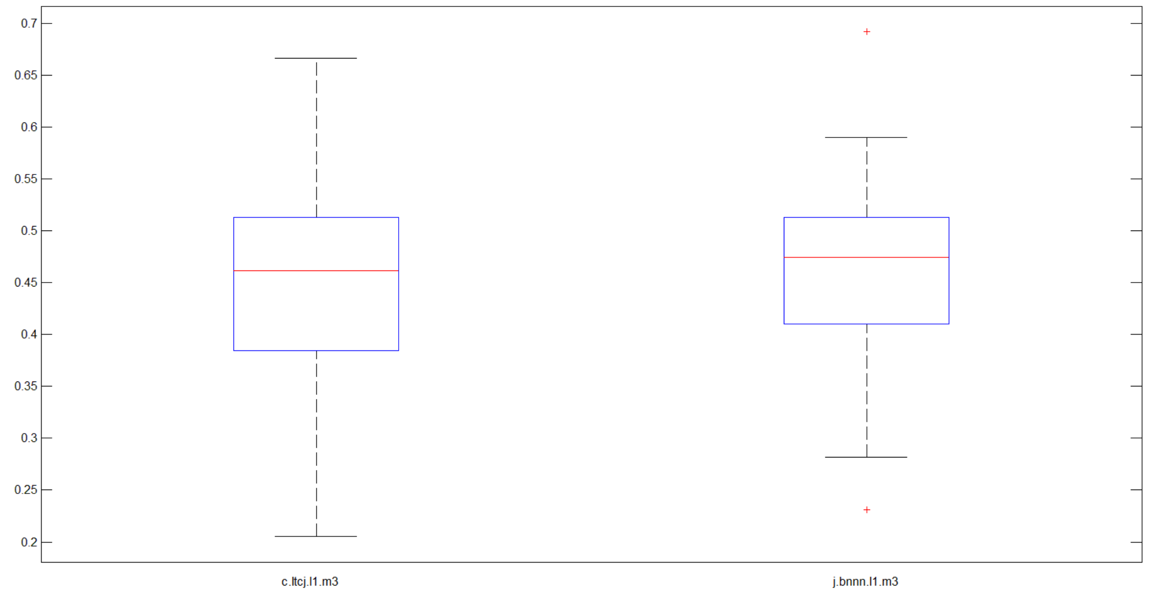


Figure 27. Kendall tau based comparison of $c.ltcj.l1.m3$ and $j.bnnn.l1.m3$ schemas

8.5.6. Response time

A key requirement for SeClone, seeing as it is a specialized search engine, is that it can provide search results in near real-time. In what follows, we discuss SeClone's run-time performance based on the execution of our benchmark queries. For the analysis, we consider clone lookup times, ranking, and sorting as the total response time, which is reported in milliseconds. It should be noted that to deploy the SeClone server application and its indices, SeClone requires ~10 minutes for the incremental indexing of the encoded code patterns for the 356M LOC (3M Java files).

Figure 28 summarizes the observed response times for the 50 queries executed for each of the 32 schemata. The results show that some of the schemata (e.g., $c.ltcj.l1.m3$ and$j.bnnn.l1.m3$) are not only capable of returning high quality search results, but also provide these results in near real-time, with response times around 100 ms. The analysis also shows that both *index* *granularity* and *transformation function* can affect the response times considerably (e.g., all *l1.m3* configurations vs. the remaining configurations). Moreover, our detailed analysis also indicates that the search approach (e.g., Jaccard coefficient) does not affect response time. The response times of each query across all schemata are summarized in Figure 29, highlighting that SeClone performance (i.e., response time) is close to constant for most of the queries.

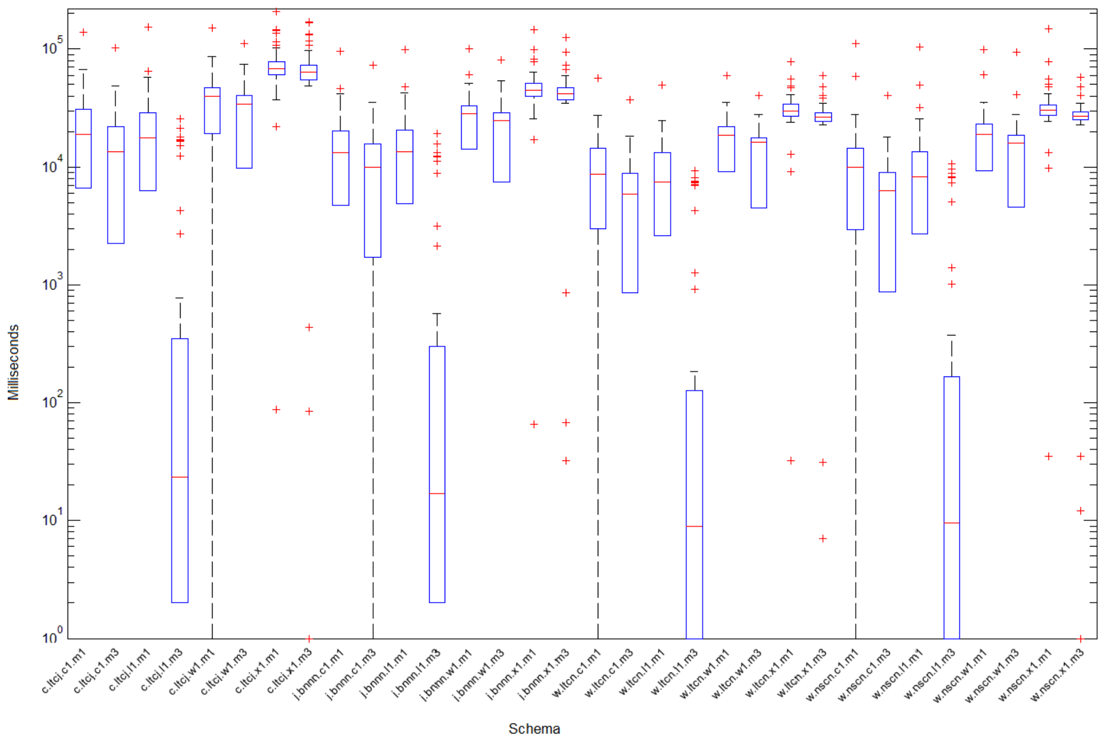


Figure 28. SeClone response time using a 356M LOC corpus

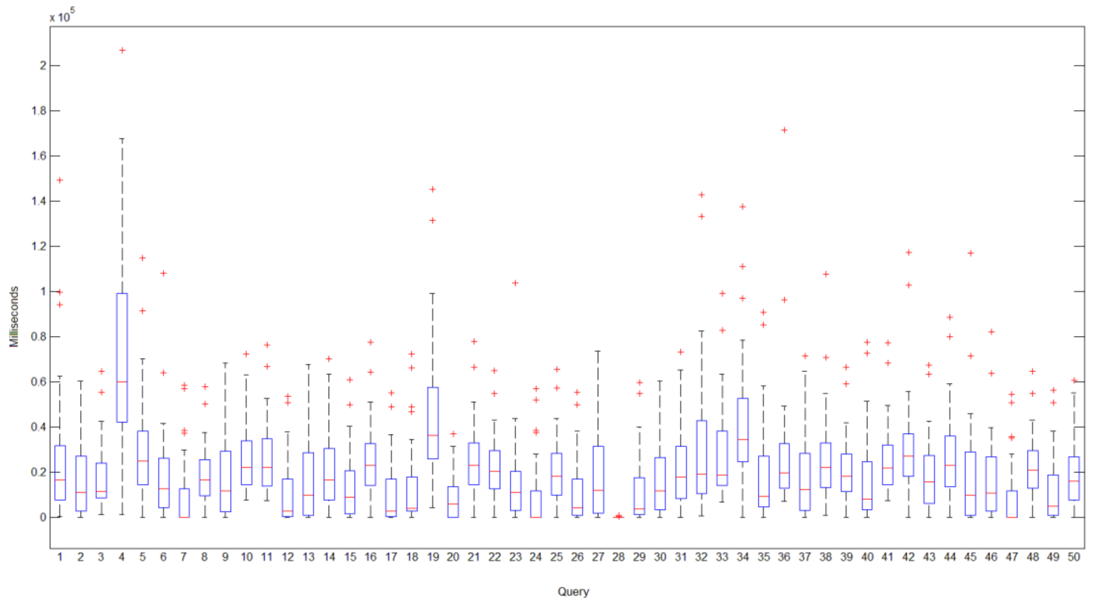


Figure 29. SeClone response time using a 356M LOC corpus grouped by query number

8.6. Summary

Our performance assessment of SeClone shows that the non-positional multi-level indexing approach for clone search can, depending on the search configuration, achieve approximately complete precision and recall for top K, with K being equal to the number of known positive answers/mutants. Moreover, our studies also showed that SeClone detects and ranks Type-1, 2, and 3 clone types as true positives correctly in most cases by exploiting the defined ranking models which we adapted from the IR community.

As part of our studies, we also observed that the *l1.m3* indexing configuration will outperform the other configurations when both response time and quality are important. If there is less an emphasis on response time, the best recall (based on the Precision at K observations) and overall quality (NDCG observations) can be achieved using the configuration *l1.m1* and *x1.m3* schemas respectively. Amongst the ranking schemas, the cosine similarity, augmented with logarithmic local and global frequency ($c.ltcj$) and Jaccard similarity ($j.bnnn$), achieves the best performance. Considering both indexing and ranking, we can recommend the $c.ltcj.l1.m3$ and $j.bnnn.l1.m3$ configurations as a default schema.

9. Bytecode clone search

While source code clone detection is a well-established research area, limited work exists in finding similar bytecode and other intermediate code representations. We are particularly interested in exploiting our clone search model for finding similarities in bytecode content, since bytecode constitutes an essential part of the search space when one implements an Internet-scale code search engine (e.g., [BAJ12]).

This chapter introduces SeByte, which is based on our clone search model (SeClone) and supports Java bytecode clone search. For the bytecode clone search problem, we adapted the two core ideas of our SeClone: multi-level indexing and information retrieval-based similarity search. In order to achieve high recall, we include two heuristics for Java bytecode clone detection, which can be considered as extensions of the SeClone's multi-level indexing for bytecode content. (1) We include *relaxation on code fingerprint,* which only considers certain types of tokens for clone detection. (2) We include what we refer to as a *multi-dimensional matching*, which applies the clone detection algorithm *separately* and therefore *independently* for each type of token (*dimension*). Furthermore, the similarity search task for each dimension is delegated to the SeClone search model. Finally, we extend our original clone search approach to support semantic search [GUH10], which is motivated by the nature of bytecode content where each instruction includes additional embedded information such as data type. As a result, SeByte provides a scalable bytecode clone search model that also supports the ranking of result sets. For our evaluation of SeByte, we conducted a performance evaluation study on a dataset of 500,000 compiled Java classes, which we extracted from the six most recent versions of the Eclipse IDE. The objective of this study was to illustrate that the SeByte search model is not only scalable, but is also capable of providing a reliable ranking of the result sets for bytecode content.

9.1. Java bytecode overview

9.1.1. Instruction families

Java bytecode is considered a stack-oriented language, with the stack being the major computation entity in the Java runtime environment. The compiler translates source code statements to their corresponding Java bytecode instructions, with source code usually being mapped to several bytecode instructions. Bytecode provides instructions to manipulate the stack, such as simple push and pops. A total of 256 instructions[4] are defined in the Java bytecode reference model. These instructions can be classified in 10 major families (summarized in Table 16) based on the Java 7 specification.

Table 16 further highlights an interesting aspect of Java bytecode, namely the fact that many bytecode instructions include additional embedded information such as the data type for which a specific instruction is applicable. For example, several variations of the symbolic load instruction are available in Java bytecode (e.g., iload, iload_0, dload, lload, fload, and aaload), with the prefix specifying the data type that is being manipulated. Table 17 highlights how some implicit semantics are captured in these bytecode instructions and can be further interpreted for fact extraction. There are other pre/postfixes that are less popular, such as postfixes belonging to the "comparison instruction family" (e.g., "fcmpg" where "g" is referring to the presence of greater condition in the comparison function).

As an example, Figure 30 shows a Java bytecode fragment as plain text, where the instruction in line 127 pushes an Integer with value 0. Line 122 shows a method call statement, which calls println from the java.io.PrintStream class. In this example, class and method names are automatically resolved from pointers to the string table.

[4] http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

Table 16. The Java bytecode instruction overview

Instruction Family	Description	Example
Data manipulation	This meta-family covers several areas such as: (1) load and store data onto/from the stack from/to local variables etc. (2) primitive arithmetic functions such as add, multiply etc. (3) data type conversion	“dload” loads a Double local variable onto the stack. “dadd” sums up Double values. i2d converts Integer-typed value to Double format.
Load and store	The two instructions types are related to stack operations involving loading onto and storing from the stack.	“dstore” stores a Double value from top of the stack to a local variable
Arithmetic	This family provides primitive instructions required for arithmetic and logical computation. The required data will be retrieved from the stack and the result will be saved onto the stack. The major families of functions are Add, Subtract, Multiply, Divide, Remainder, Negate, Shift, Bitwise OR, Bitwise AND, Bitwise exclusive OR, Increment, and Comparison	“fadd”, “ishr” (Shift right Integer value) “ior”, “iinc” (such as var++), fcmpg (compare – the greater operand)
Type conversion	The dedicated family for type conversion	“i2d” and “i2f”
Object creation and manipulation	Create, load, and store object or array instances. Note that Java provides dedicated instructions for array creation and manipulation.	“new”, “newarray”, “getfield” (access Java classes’ fields), “iaload” (load an array of Integer type to the stack), “arraylength”, “instanceof”
Stack management	Primitive operations required for stack manipulation. These operations changes the state of the stack directly	”pop”, “dup”, “swap”
Control transfer	Program control flow instructions. Several types of “if” are provided for simulation of all possible conditional branches.	”ifeq”, “ifnull”, “goto”
Method invocation and return	The major instructions for handling method call statements are presented under this family. Although there are two major types which are invocation and return, specialized instructions for Object-Oriented semantics are available	“invokevirtual” (the regular method call in Object Oriented where the receiver of the message is known in advance), “invokeinterface”, “ireturn”
Throwing exception		“athrow”
synchronization	The primitive instructions for synchronization in case of concurrency. Note that the specified synchronization semantics at the source code will be handled using monitor enter and monitor exit	“monitorenter” specifies entering the secured code block in terms of concurrency.

```
…
122: invokevirtual   java/io/PrintStream.println:(I)V
123: astore_1
124: aload_1
125: arraylength
126: istore_2
127: iconst_0
128: istore_3
129: iload_3
130: iload_2
…
```

Figure 30. Java bytecode example (presented as plain text)

Table 17. The symbol table assigned to known data types by Java bytecode

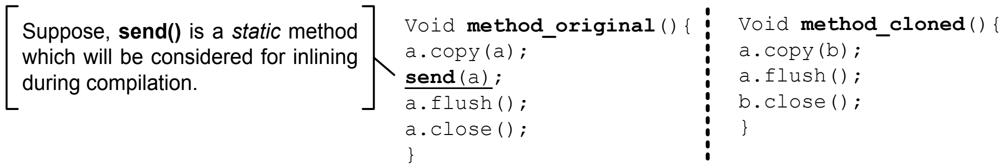
Symbol → The corresponding type			
a → reference	i → integer	s → short	l → long
c → character	b → byte	f → float	d → double

9.1.2. Motivation and challenges

Similar to the other low level languages, Java bytecode uses machine instructions to represent basic functionalities such as conditions and loops. Different types of tokens, such as Java virtual machine instructions, strings, method names and Java type names, are available in the bytecode representation. These tokens form the *code fingerprint,* which we use as input data for our research. Throughout the chapter we use Java bytecode and bytecode keywords interchangeably to refer to any content similar to the textual representation created after our first extraction step.

Motivation. Clone detection at bytecode level can detect clone pairs that might not be syntactically similar at source code level but are in fact semantically similar. The compilation of source code to a bytecode format generates a unified representation of source code, which is based on the transformation of syntactic dissimilarities of various loops and conditional blocks in the source code to the unified format. As a result, the bytecode representation can facilitate "semantic" clone detection even if syntactical matching is considered.

Challenge. While compilation techniques such as method inlining are useful for run-time performance optimization, they also introduce new challenges. For example, the two methods in Figure 31 could be detected as clone pair with high confidence using the source code representation. However, detecting them as clones at the bytecode level is inherently more difficult since its success depends on the original size of the *send()* function in the first method block. Due to the method inlining effect, these two method blocks might end up with completely different sizes.



Suppose, **send()** is a *static* method which will be considered for inlining during compilation.

```
Void method_original(){
a.copy(a);
send(a);
a.flush();
a.close();
}
```

```
Void method_cloned(){
a.copy(b);
a.flush();
b.close();
}
```

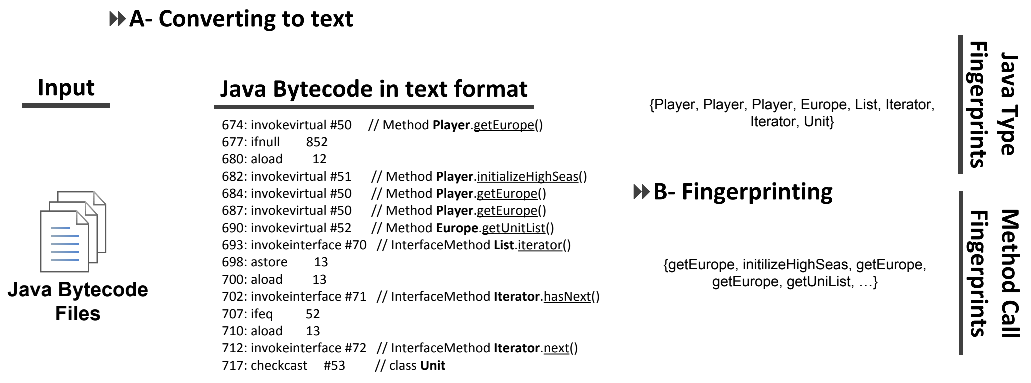
Figure 31. An example with one line dissimilarity at source code level, at the bytecode level due to method inlining effect, the actual bytecode dissimilarity depends on the size of method *send()* implementation.

9.2. SeByte data presentation and manipulation approach

A major part of clone detection revolves around matching code content. The state of the art is to consider a sequence of source code statements as a single fused information source to be compared. In contrast to the current approaches, we include a heuristic called *relaxation on code fingerprint,* which leads to a *multi-dimensional comparison* approach that is described in detail in this section. Instead of comparing code content as lone fused fact sequences, we extract different pieces of information based on their token types, each of which corresponds to a dimension in our approach. This approach is motivated by the fact that each Java bytecode statement (Figure 32) can contain several predefined types of information in a single line of bytecode, such as instruction, class and method name.

Each of these dimensions presents a specific perspective of a method block and its characteristics. In our multi-dimensional approach, we then compare these dimensions *independently* using a clone detection algorithm to detect candidate clone-pairs. We then merge the different result sets created from the analysis of the individual dimensions to create our final clone pair set.

Figure 32 Step B shows an illustrative example of using two different dimensions as part of the *relaxation on code fingerprinting*. In the bytecode column, Java type fingerprints are marked as bold and method names are underlined. The first dimension contains the names of accessed Java types. The second dimension only contains the names of the called methods. Based on their actual appearances in the bytecode, all dimensions will be represented using *ordered sequences*. Due to our relaxation heuristic, it is possible to ignore the other information resources.


A- Converting to text
Input
Java Bytecode Files
Java Bytecode in text format
{Player, Player, Player, Europe, List, Iterator, Iterator, Unit}
Java Type Fingerprints
B- Fingerprinting
{getEurope, initilizeHighSeas, getEurope, getEurope, getUniList, …}
Method Call Fingerprints

```
674: invokevirtual #50    // Method Player.getEurope()
677: ifnull       852
680: aload         12
682: invokevirtual #51     // Method Player.initializeHighSeas()
684: invokevirtual #50     // Method Player.getEurope()
687: invokevirtual #50     // Method Player.getEurope()
690: invokevirtual #52     // Method Europe.getUnitList()
693: invokeinterface #70   // InterfaceMethod List.iterator()
698: astore        13
700: aload         13
702: invokeinterface #71   // InterfaceMethod Iterator.hasNext()
707: ifeq        52
710: aload       13
712: invokeinterface #72   // InterfaceMethod Iterator.next()
717: checkcast    #53       // class Unit
```

Figure 32. Examples for Java bytecode fingerprinting

Motivation #1. The underlying rationale for the relaxation on code fingerprint is to develop a robust clone detection approach that can survive extreme dissimilarities when they are limited to a specific dimension. Using our multi-dimensional matching, we can increase the recall by comparing each data family independently. Therefore, dissimilarity in each dimension is limited only to its corresponding result set.

Motivation #2. Our multi-dimensional approach also reduces input data size (search space) for the clone detection process, since each dimension only contains a subset of available data that will be considered for comparison. In our example (using two dimensions), we use either Java types or the names of called methods. Figure 33 illustrates this reduction in terms of number of tokens to be analyzed. Using this fingerprinting approach for the bytecode datasets (Table 18), we were able to achieve a reduction in data size of 50-80% approximately, where the number of tokens in each dimension (e.g., method or type columns) is compared to the total number of lines in the raw data (i.e., the regular bytecode column). Therefore, the multi-dimensional approach not only supports the detection of clone-pairs with extreme pattern dissimilarity, but also improves its scalability by several folds.

Table 18. Prelimaniry bytecode datasets

Dataset	**Size (#files)**		**Application Context**
	Bytecode	***Source code***	
EIRC	83	64	Network-based comm. client
Freecol (server)	220	79	Server application
Freecol (full)	1120	570	A strategy-based game
HBase	1093	448	Database system

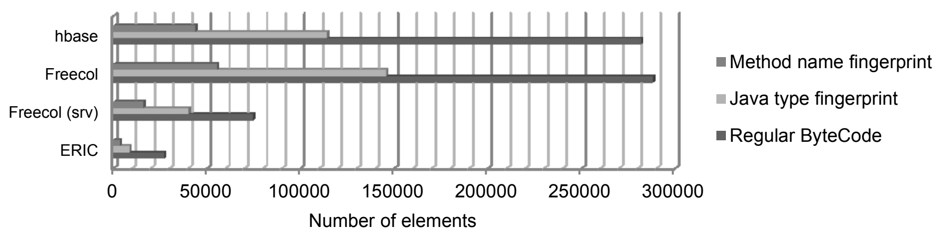
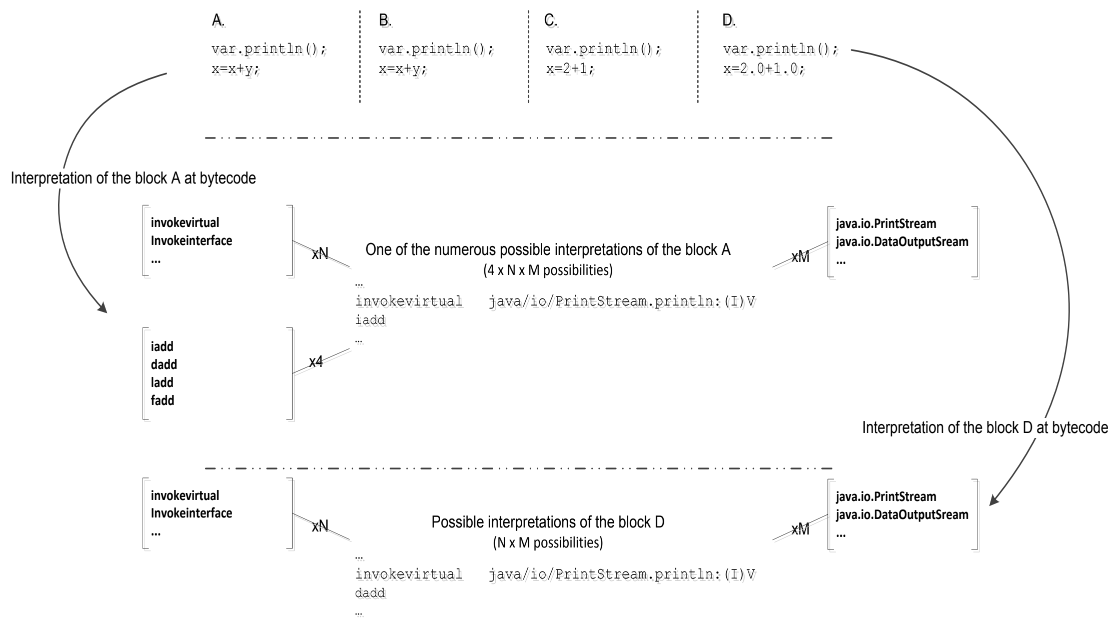


Figure 33. Effects of the relaxation on code fingerprint on data size reduction (with respect to the raw data / number of lines)

9.3. SeByte search approach

As discussed, Java bytecode contains less ambiguity compared to the higher-level languages, due to the availability of additional explicitly embedded information. For example, bytecode level summation instructions explicitly include the data type they are capable of manipulating as part of the instruction. As a result, for each primitive data type, there is a dedicated "add" instruction (e.g., iadd and fadd). Similarly, object creation/access, method call, and field access instructions embed the data types (or other metadata). For example, in line 122 Figure 30, the type of message receiver (i.e., println) is already resolved not only for the receiver class name (PrintStream), but also for the actual implementation captured by its fully qualified name (java.io.PrintStream) and the file address. Although, from a clone detection/search perspective, input data with less ambiguity is typically preferred (to improve precision), it reduces the recall of Type-2 clone detection.

Figure 34, illustrates the challenges of detecting clones at Java bytecode level versus source code level. While only one token (i.e., +) is used to present the add functionality at source code level, the bytecode representation actually depends on the source code's implicit semantics. As a result, the `x=x+y` source code can have four possible corresponding bytecode level representations (depending on the actual data types of variables `x` and `y`). This issue becomes even more challenging with the inclusion of other statements (e.g., `var.println()`). There exist $4 \times N \times M$ different bytecode interpretations for the original source code fragment, where N is the number possible instructions (available for method calls) and M is the number of possible types. As a result, while blocks A and B might be considered identical clones (Type-1) at the source code level, their bytecode representation could be different, and therefore their clone type could be different as well.



A.
var.println();
x=x+y;
B.
var.println();
x=x+y;
C.
var.println();
x=2+1;
D.
var.println();
x=2.0+1.0;
Interpretation of the block A at bytecode
invokevirtual
Invokeinterface
...
xN
One of the numerous possible interpretations of the block A
(4 x N x M possibilities)
xM
java.io.PrintStream
java.io.DataOutputSream
...
invokevirtual java/io/PrintStream.println:(I)V
iadd
iadd
dadd
ladd
fadd
x4
Interpretation of the block D at bytecode
Possible interpretations of the block D
(N x M possibilities)
invokevirtual java/io/PrintStream.println:(I)V
dadd

Figure 34. A few examples showing the differences between source code and bytecode clone detection

9.3.1. Existing solutions

In cases where the input data contains more information than the clone detection algorithm requires or can process, filtering and normalization are applied. For source code content, normalization is commonly used to remove unnecessary differences so that pattern-matching algorithms can achieve higher recall. For example, many approaches [HUM10][KAM02] replace token names (e.g., class names) with predefined symbols (e.g., $ or enumerated $ where the order information must be preserved, such as $1, $2). By using such normalization approaches, detection of Type-2 clones at source code level becomes feasible. Similarly, normalization for intermediate language has been proposed in the literature, e.g., Baker et al.'s for Java bytecode [BAK98] and our work on .NET intermediate language [ALO12].

9.3.2. Our solution - semantic search

Existing solutions for intermediate languages have focused in the past on the use of data filtering and normalization, which often involves some form of data loss, to prepare the input data for clone detection algorithms. While this approach works well for clone detection, the information loss caused by the filtering will restrict its applicability for *clone search* specifically in the bytecode context. A key aspect of any search approach is its ability to differentiate and rank hits based on the closeness of hits to the query. However, the data loss (including semantics) through the data filtering used by traditional clone detection approaches will affect their ability to provide an accurate ranking.

For example, a user is looking for code fragments that implement the summation of two numbers, in particular the summation of float type. In this example, search results containing a float summation corresponding to a Type-1 clone, such as fragment D in Figure 34, should be ranked higher than research results containing summation of other data types, e.g., summation of integer numbers such as fragment C in Figure 34 - i.e., Type-2 clones. Likewise, semantic information associated with other bytecode level instructions can be used to enhance the search and ranking processes. This issue can be solved by adapting the semantic search concept [GUH10]. In order to support semantic search in our approach, we require access to two types of information: existence and degree of similarity (between two tokens). In what follows, we define both the existence and degree of similarity in our research context, which will be used to semantically rank the search results.

Existence of Similarity: Given the classification of bytecode level instructions, it is possible to identify similar instruction types based on their relationship with each other. These similar instructions can be identified by analyzing the associated tokens in the domain of discourse. For example, in Figures 35 and 36, *iadd* and *java.io.PrintStream* can be associated with other tokens either in the instructions or inheritance tree. The key idea is that these association links can be used to help the interpretation of similarities between tokens, and therefore allow us to infer that, for example, an *iadd* (add for integers) token is similar to *dadd* (add for doubles) and other siblings in the same graph (e.g., the semantic network).

Degree of Similarity: While the existence of similarity only identifies whether two token are related (e.g., *iadd* is related to *dadd* and *XOR*), their actual degree of similarity might differ. In addition to the presence of links, the distance between tokens can be used to interpret their degree of similarity. In our example (Figure 35), both *iadd* and *dadd* are closer to each other than *XOR*, since they both belong to the Summation family (Figure 35). Including these additional semantics in our search process allows us to assign different ranking to the *dadd* and *XOR* occurrences for the given token *iadd* (part of the query), which we capture by our degree of similarity measure.

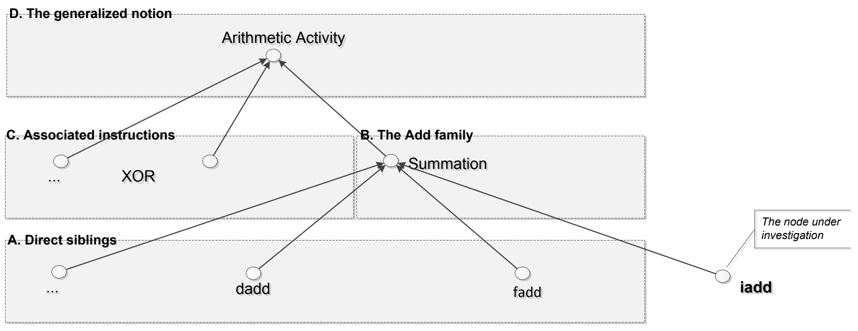


Figure 35. A slice of domain of discourse (i.e., Java bytecode specification) related to iadd instruction

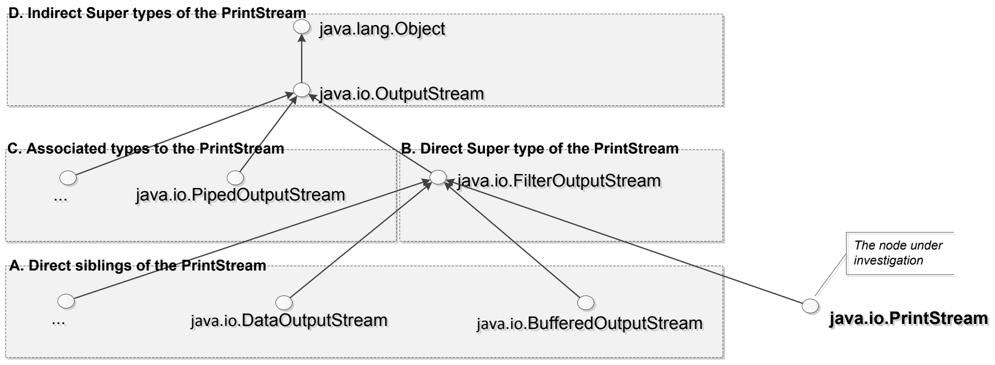

D. Indirect Super types of the PrintStream
java.lang.Object
java.io.OutputStream
C. Associated types to the PrintStream
...
java.io.PipedOutputStream
B. Direct Super type of the PrintStream
java.io.FilterOutputStream
A. Direct siblings of the PrintStream
...
java.io.DataOutputStream
java.io.BufferedOutputStream
The node under investigation
java.io.PrintStream

Figure 36. A slice of domain of discourse (i.e., the program inheritance tree) related to java.io.PrintStream token

9.4. Bytecode ontology

For the successful implementation of our bytecode level semantic search approach, we require a type of semantic network (e.g., [QUI67]) that formalizes the concepts and their connections. We created this semantic network as an ontology based on the Java specification (e.g., Figure 35 and Table 16). The ontology called Bytecode Ontology (byteon) represents a hierarchical conceptualization of bytecode instructions, and includes all 256 bytecode instructions. All instructions are classified into families of related instructions. As discussed earlier, at bytecode level, ten major families can be distinguished (see Table 16). We extend this initial classification by including (1) additional classifications (horizontal extension), and (2) hierarchies between families (vertical extension). For example, intermediate concepts, such as "IntegerAccess", are added to associate all functions defined over integer data types.

We manually created this ontology by reviewing the Java bytecode instruction specification covering all 256 instructions. The resulting bytecode ontology and its documentation are available online at http://secold.org/projects/sebyte. The ontology contains 296 concepts (40 family entities and 256 instructions). Figure 37 provides an overview of the high-level concepts. A complete overview of the ontology is shown in Figure 38, with its major families being labeled by circles. The complexity of the graph is high due to the large number of links (~650 links), since most instruction types belong to several families.

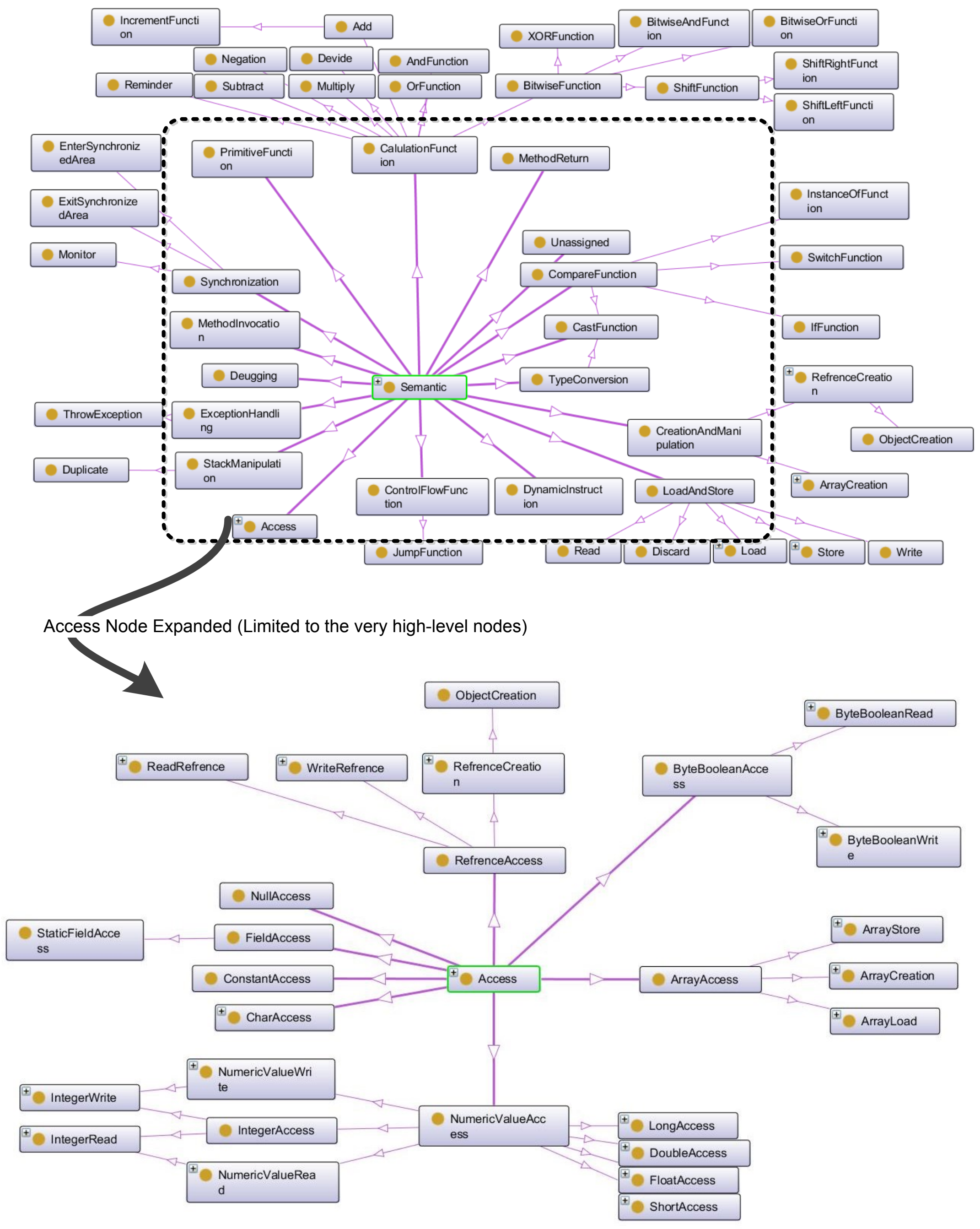


Figure 37. Partial preview of Bytecode Ontology[5]

[5] Created by http://protege.stanford.edu/

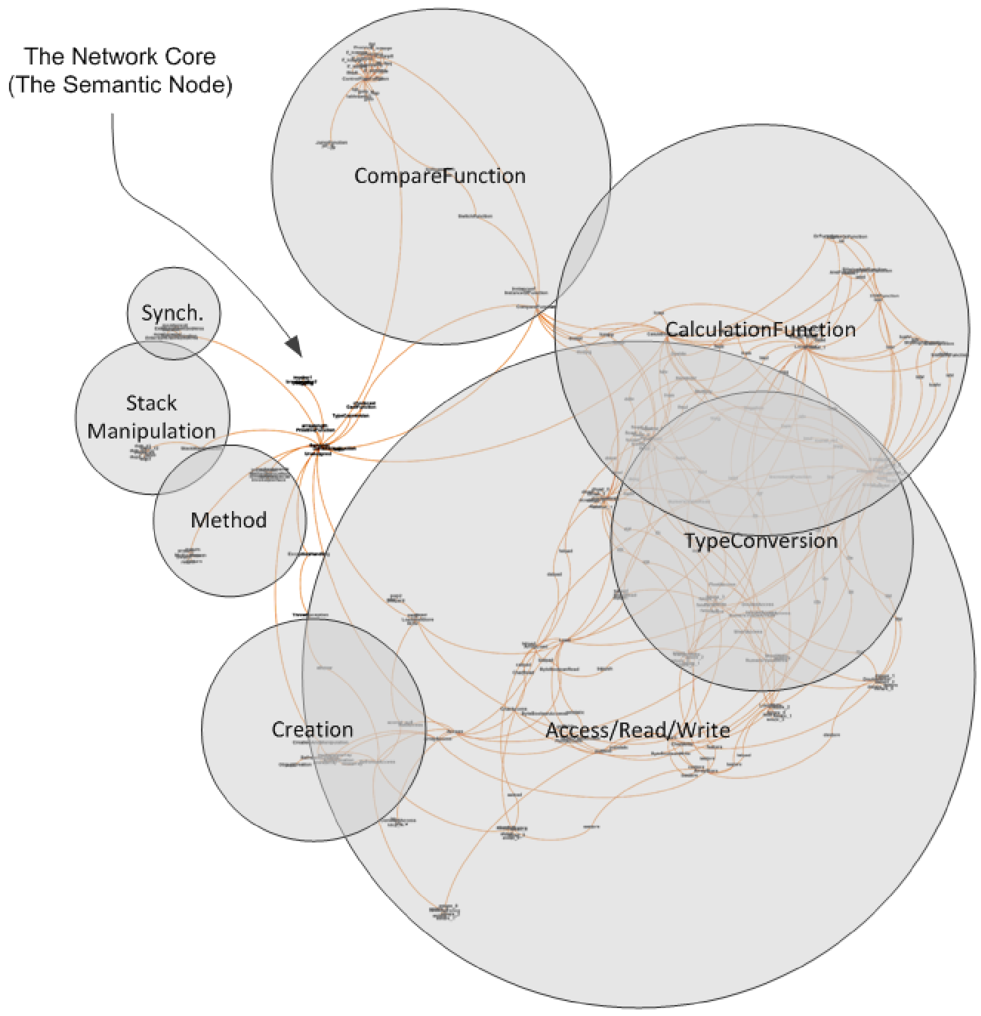
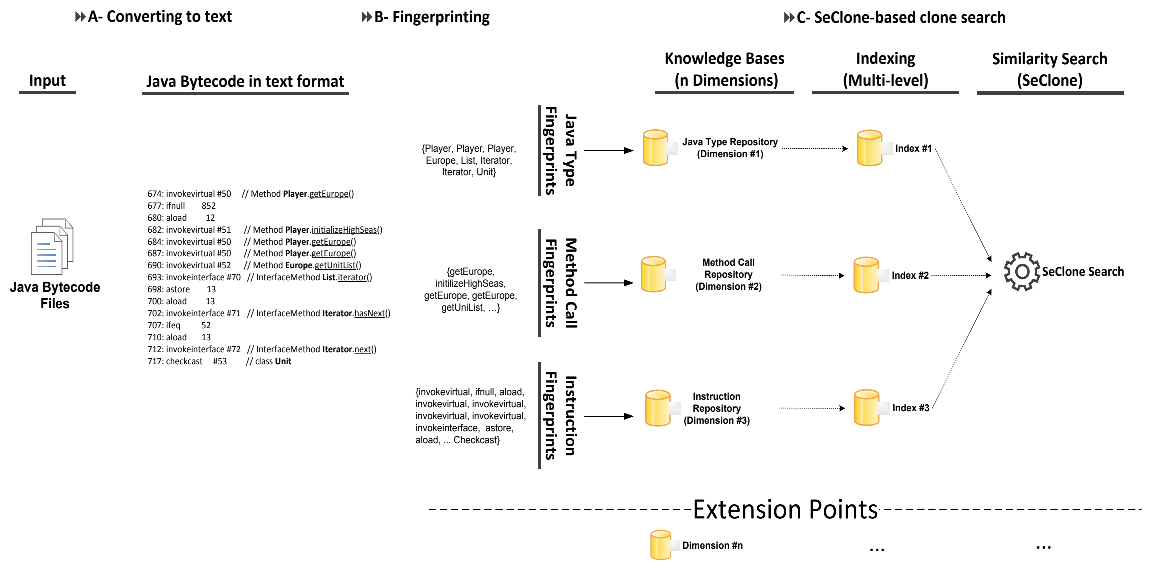


Figure 38. Bytecode Ontology overview highlighted with the most popular families[6]

[6] Created by http://gephi.org/

9.5. SeByte – a Java bytecode clone search approach

In what follows, we provide a more detailed implementation overview of SeByte and its major processing steps (Figure 39). During the first processing step (converting to text), a conversion of bytecode content to plain text takes place. The plain text constitutes the input data for the later phases. The plain text content is used by the SeByte parser for dimension population, using relaxation on code fingerprints. In our current implementation, SeByte maintains three dimensions: type, method call, and instruction fingerprints. This three-dimensional model is then used by SeClone multi-level indexing approach to create an index for each dimension. Finally, we take advantage of the clone search functionality provided by our SeClone search model to search for bytecode clones. In order to support the search requirements for bytecode content, we extended the SeClone core algorithm with the semantic search capability. Our heuristic-based semantic search implementation takes into consideration both existence and degree of similarity, which are modeled by the ontology.



A- Converting to text
B- Fingerprinting
C- SeClone-based clone search
Input
Java Bytecode in text format
Knowledge Bases (n Dimensions)
Indexing (Multi-level)
Similarity Search (SeClone)
Java Bytecode Files
{Player, Player, Player, Europe, List, Iterator, Iterator, Unit}
Java Type Fingerprints
{getEurope, initilizeHighSeas, getEurope, getEurope, getUniList, …}
Method Call Fingerprints
{invokevirtual, ifnull, aload, invokevirtual, invokevirtual, invokevirtual, invokevirtual, invokeinterface, astore, aload, ... Checkcast}
Instruction Fingerprints
Java Type Repository (Dimension #1)
Method Call Repository (Dimension #2)
Instruction Repository (Dimension #3)
Index #1
Index #2
Index #3
SeClone Search
Extension Points
Dimension #n
...
...

```
674: invokevirtual #50    // Method Player.getEurope()
677: ifnull       852
680: aload         12
682: invokevirtual #51     // Method Player.initializeHighSeas()
684: invokevirtual #50     // Method Player.getEurope()
687: invokevirtual #50     // Method Player.getEurope()
690: invokevirtual #52     // Method Europe.getUnitList()
693: invokeinterface #70   // InterfaceMethod List.iterator()
698: astore        13
700: aload         13
702: invokeinterface #71   // InterfaceMethod Iterator.hasNext()
707: ifeq        52
710: aload       13
712: invokeinterface #72   // InterfaceMethod Iterator.next()
717: checkcast    #53       // class Unit
```

Figure 39. Clone search approach for Java bytecode (token-level)

9.6. SeByte performance evaluation

As discussed in the previous chapter, performance evaluation of clone search engines differs from clone detection evaluation. A key requirement for evaluation was not only to have a sufficient large dataset, but that the dataset must also contain (1) a few highly similar clones, (2) several relatively similar clones, and (3) a large number of irrelevant fragments. A dataset that meets these requirements allows us to evaluate our search approach in situations where, for each query, the number of irrelevant fragments (noise) will be considerably larger than the number of actual clones, which makes the resulting ranking an even more challenging task. Therefore, for the case study, we have created a dataset consisting of bytecode (including all bytecode dependencies) from the latest six major versions of Eclipse IDE (2007 – 2012). Table 19 summarizes the dataset details and the processing time.

Table 19. The Eclipse dataset overview and processing time report

Feature	Value
Total #Jar (library) files	3,900
Total #file (Java class)	482,768
Total #LOC (bytecode level)	73 M
Total #method	3,898,475
Total #significant method (min 2 token)	~1,780,000
Total #significant method (min 5 token)	~780,000
Processing time (seconds)	
Jar file bytecode extraction (unzipping + disassembling)	3422
Crawling (local file system)	0.802738268
Fact processing	267
Index construction (+fact processing)	755

9.6.1. SeByte search schema

For our performance evaluation, we use three parameters (dimensions), which are represented in our search schema by a triple (I, M, T), where *I* indicates the weight of the instruction dimension, *M* the weight of the method, and *T* the weight of the type dimensions, indicating whether a dimension is considered to be "*leveraged*" or "*regular*". In the case of leverage, its similarity score is given a higher priority during the final ordering (when search result sets of all three dimensions are being merged) compared to regular similarity scores. Furthermore, our search schema does not restrict the number of dimensions that can belong to a particular group (leveraged or regular), therefore allowing all dimensions to belong to the same group (and therefore have an equal weight). We use $+$ to denote that a dimension is leveraged, and $-$ to indicate that a dimension has a regular weight. For example, in the context of our case study, the triple $- + -$ indicates that the instruction and type dimensions have a regular weight, while the method dimension will be leveraged. Throughout our case study, we evaluated all seven possible combinations and their effect on the performance of our clone search model.

9.6.2. First False Positive measure

From a clone search viewpoint, our search model deals with two major challenges: first, being able to detect the few relevant fragments, and second, assigning a higher priority to these true positive results than to the false positives in the result sets. On average in the corpus used for our case study, only 6 out of ~1.7 million code fragments (for each search) were highly relevant code fragments, whereas almost all of the remaining ones were non-relevant. We assessed the quality of our search and ranking approach using the First False Positive measure, which returns the position of the first false positive hit in the result set. For our evaluation, we randomly selected 20 queries that we tested across all 7 possible search combinations. We believe this measure is one of the strictest measures when evaluating the performance of the clone search system, especially in cases such as ours, where the corpus contains lots of noise (irrelevant code fragments). We manually evaluated the top 30 hits of the 140 result sets (~4200 clones/hits) to determine the true and false positives. Figure 40 summarizes the results from our manual evaluation in terms of the position of the first false positive within the top 30 hits.

The analysis of SeByte's performance results (Figure 40) shows that the schemata perform quite differently when placing the first false positive in the ordered result set. In addition, we can observe that a few schemata almost consistently outperform the other schemata. The overall best performance was achieved with the $- + -$ search schema, which leverages the method dimension over the other two dimensions. This schema places the first false positive at 6th position in the worst case (excluding the single exception). This is in contrast to the $- - +$ search schema, which placed the first false positive within its top 3 answers for 12 out of 20 queries, and therefore can be considered as a poor configuration.

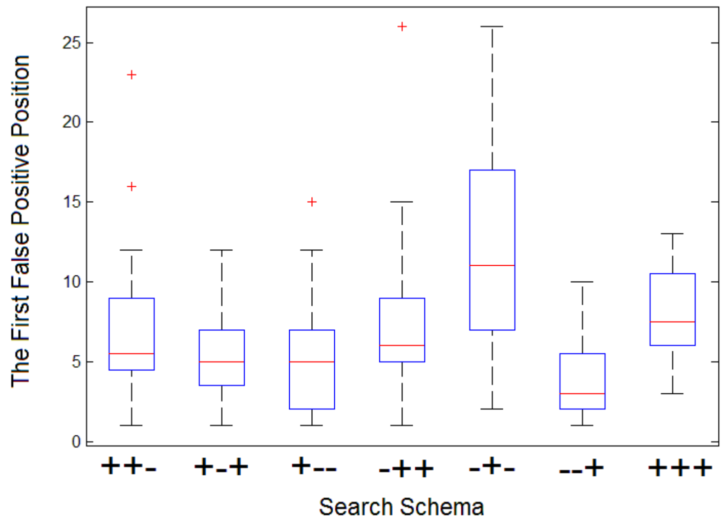


Figure 40. Summary of the First False Positive measure study

9.6.2.1. Precision at K measure

Precision at K can be considered as a complementary measure for the first false positive evaluation. However, the major limitation of this measure is its query dependency. For example, in order to provide a fair evaluation using "Precision at 10" measure, at least 10 cloned fragments (true positives) must exist in the corpus for all executed queries. We therefore had to split our candidate queries into two subsets: (1) queries with less than 10 actual cloned fragments in the whole corpus, and (2) queries with more than 10 cloned fragments. We selected a "Precision at 5" measure for the evaluation of our queries with less than 10 clone fragments. For the second query subset, we used the standard "Precision at 10". We manually evaluated the top K hits of 40 queries, which we executed across all seven schemata (2100 fragments in total) to calculate their precision at K.

Figures 41 and 42 summarize the results from our manual evaluation, which showed that for both sets, the $- + -$ schema provides the best overall results, achieving at least a 90% precision (excluding certain outliers which are tagged in Figures 41 and 42). For the outlier cases, the precision for the $- + -$ schema drops to 40%. Figures 43 and 44 provide a more detailed analysis of the different schemata based on the individual queries. It should be noted that there is no pre-defined order among the queries in Figures 43 and 44. We added the curves to improve the result interpretation for each schema. From Figures 43 and 44, one can further observe that the $- + -$ schema achieves the best overall performance. Some schemata, such as $+ - -$, show a significant fluctuation in their performance, with their precision being between 100% and 0%.

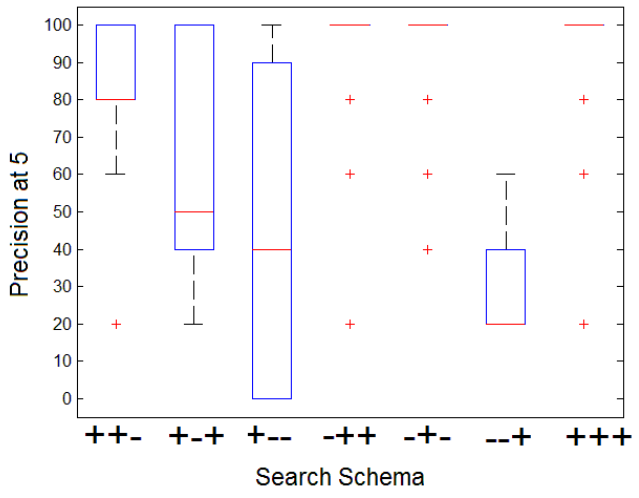


Figure 41. Summary of the Precision at 5 measure study

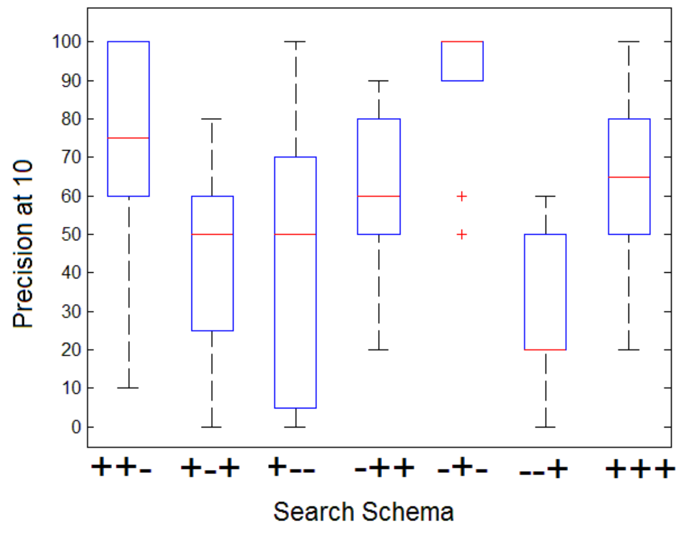


Figure 42. Summary of the Precision at 10 measure study

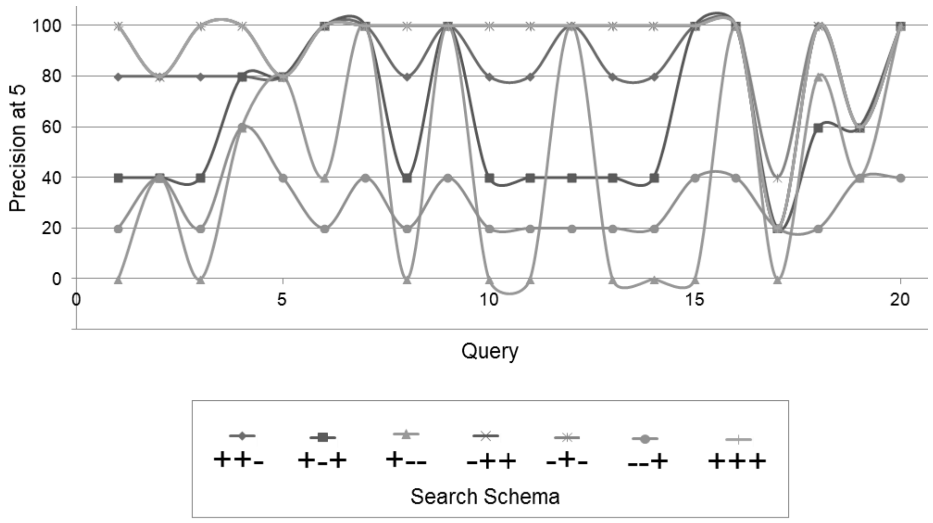


Figure 43. Details of the Precision at 5 measure study

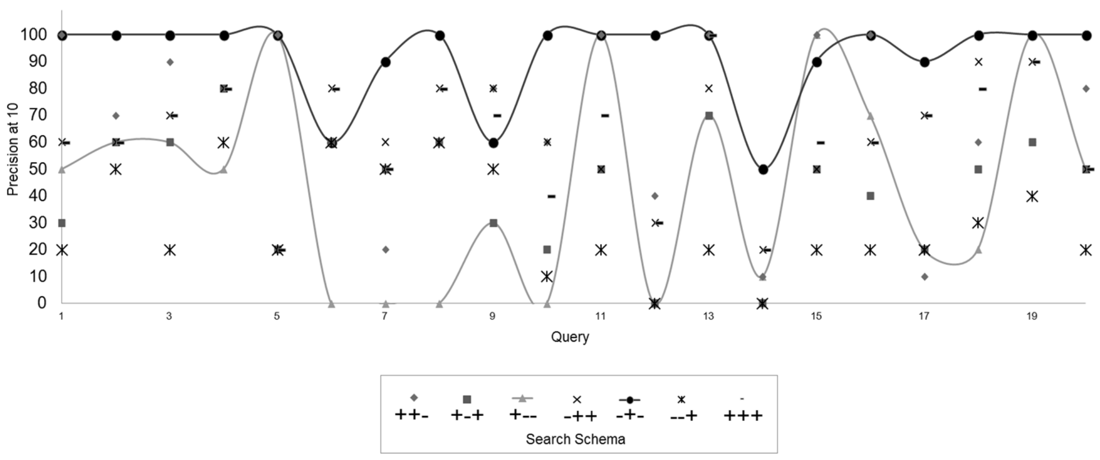


Figure 44. Details of the Precision at 10 measure study

9.6.3. NDCG measure

This measure has been used to provide a fine-grained evaluation of the quality and ordering of result sets. However, the measure should only be applied when the average value or evaluation of fine-grained ordering is required. Otherwise, measures such as Precision at K are preferred. Nevertheless, NDCG is one of the state of the art search engine measures commonly used in the IR domain. For our evaluation, we again selected 20 queries and their clone results, with each query returning at least 30, but fewer than 100 matches. In order to create an oracle for each query (required by NDCG), we manually evaluated a total of 1481 candidate clone pairs and assigned them a similarity score between 0 and 3. We used a similarity score of 0 to indicate totally irrelevant pairs (100% False Positive), whereas similarity scores of 1, 2, and 3 denote the presence of a clone pair with some degree of similarity. The 20 queries and their manually tagged set constitute the oracle that we used for our study. In total, we retrieved 10,367 hits after executing the 20 queries across all seven search schemata. Figure 45 presents the NDCG value for all query-search schema pairs. Again, there is no ordering among queries, and the lines in the figure are only added to improve the readability. Figure 45 also includes the average NDCG for each schema across all queries.

In summary, while some of the schemata achieve either close or slightly above average value (e.g., $-++$ with 0.87 NDCG), the $-+-$ search schema again outperforms the other schemata by achieving, on average, a 0.88 NDCG (Figure 45). Overall, considering the result of all measures altogether (i.e., First False Positive, Precision at K and NDCG), $-+-$ was the most reliable search schema for the bytecode clone search problem.

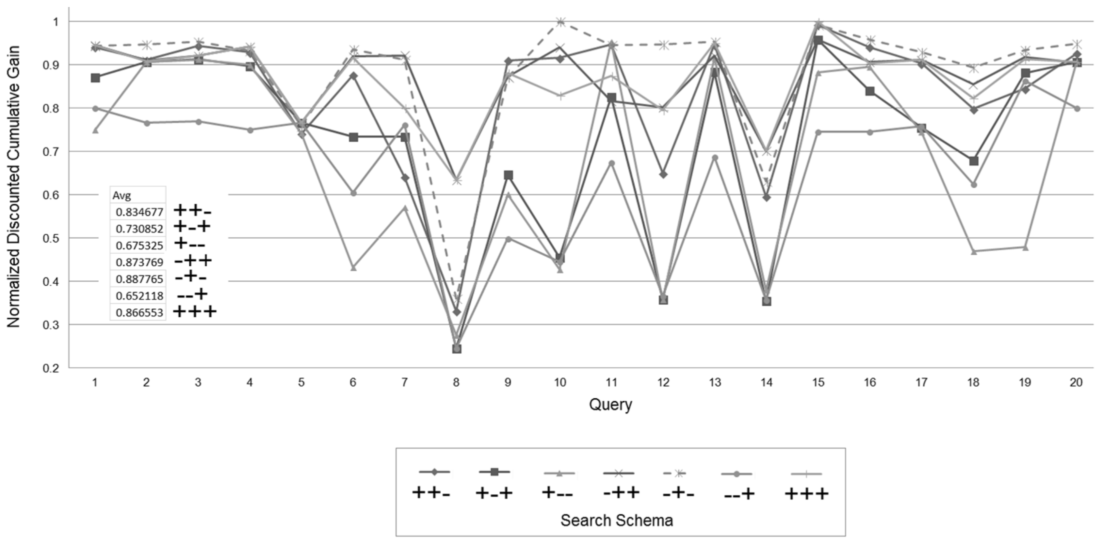


Figure 45. Details of the NDCG measure study presenting the averaged behaviour of schemata

9.7. Summary

In this chapter, we introduced SeByte a concrete solution for adaptation of our core clone search model (SeClone) for Java bytecode. Our solution extends SeClone based on the observed characteristics of the bytecode language. Using the provided performance evaluation, we can conclude: (1) SyByte can be used for clone search applications on bytecode, as some of the search schemata provide acceptable results, and (2) there is at least one schema (i.e., $-+-$) which performs well considering all three measures.

10. Adaptation of the clone search model for pragmatic reuse

Source code examples play a major role in programming, and provide an intrinsic resource for learning [NYK02] and re-using [ROS96][JON92]. A lack of available source code examples has been considered to be a major drawback of learning and improving coding [ROB11] during software development, as code examples can accelerate the development process [MAN05], and increase the product quality [MAR09]. Since it is not common in software development to explicitly document code examples [HOL05][SIN98][WAN13], programmers have to rely on manually searching through previously written projects (e.g., [WAN13]) and code repositories (e.g., sourceforge.net) for code examples. However, not every code fragment that meets a query criteria should be considered a good code example, as a good example should also be concise, self contained, and easy to understand and integrate [HOL05][MIS12][WAN13]. Throughout this chapter, we refer to such a code fragment as a *working code example*. Such working code examples can spawn a wide range of application context, varying from API usage (e.g., how to use JFreeChart library to save a chart) to basic algorithmic problems (e.g., bubble sort).

In this chapter, we discuss how clone search models can be adapted as an alternative solution to the current approaches (Chapter 2) to the problem of detecting concrete working code examples (i.e., spotting) for pragmatic reuse and program synthesis. Spotting these examples is challenging, since tradeoffs among a variety of criteria, such as popularity, conciseness, and completeness of a code example must be taken into consideration [HOL05][MIS12][WAN13]. The spotting process itself consists of two phases: (1) finding some abstract solutions that satisfy a given query, and (2) locating the code fragments that satisfy the solutions. Both steps are considered challenging, as it is often the case that hundreds of potential matches are found in a large-scale corpus. In this chapter, we demonstrate how a clone search model can satisfy these two search problems. Furthermore, the clone search-based approach supports (1) different types of code examples that are not limited to API usage, and (2) free-form querying, where, for a $query = \{term_1, term_2, \ldots, term_n\}$, each term can be a data type, method name, or concept (e.g., download or bubblesort). This is different from most of the earlier work (e.g., [BUS12]), where it was necessary to write either a partial code fragment, or to provide the API names and data flow information (e.g., $term_{n-1} \rightarrow term_n$) when formulating a query.

10.1. Characteristics of the working code examples

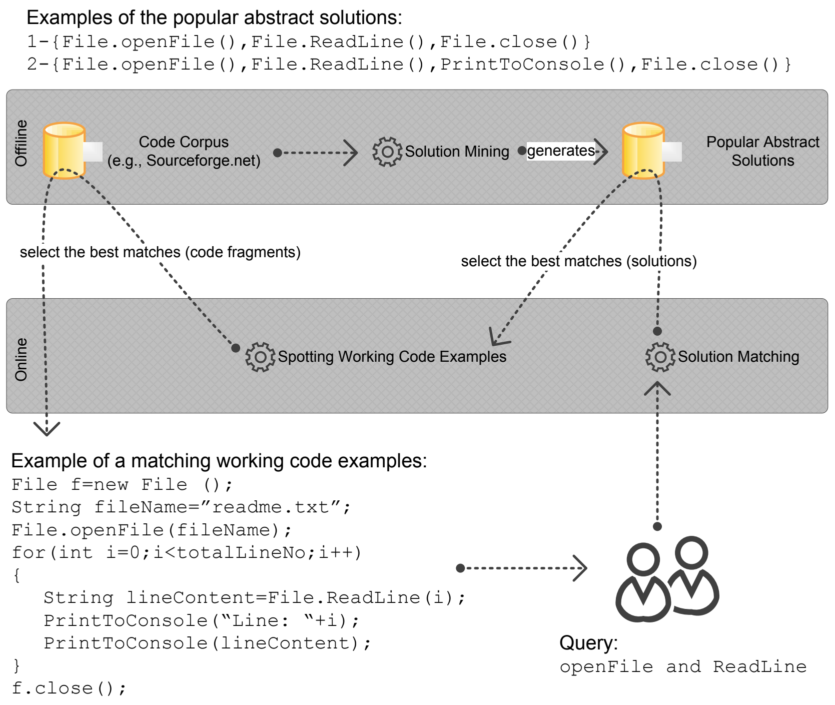
Although there is no formal definition of what constitutes a good query result, several features of a working code example are discussed in the literature. Table 20 provides a brief summary of the features and measures that are commonly used for evaluation purposes. The support for these features should lead to a search approach that can differentiate good matches from among the millions of potential matches (i.e., code fragments) available in repositories.

Table 20. Features and the related measures for identifying the working code examples

Feature	Measures and additional comments
Conciseness [BUS12] [MAN05] [THU07]	The fragment must focus on a given use case. It can be measured via: • size (LOC) • number of usage • irrelevant code (#other unnecessary tasks) [KIM10]
Correctness [KIM10]	-
Readability & self understanding	e.g., well-chosen variable name [BUS12]
Completeness	• Well-typed [KIM10] [BUS12] (including intermediate) • Variable initialization • Correct control flow [BUS12] • Exception Handling [BUS12]
Successful integration [HOL09]	The end-user should be able to successfully apply the recommended answer onto her code.
Result set qualtiy	• Succinct [WAN13] • High-coverage [WAN13] • Representativeness [KIM10] [BUS12]

10.2. Schematic approach and its challenges

As discussed in the literature (e.g., [HOL05]), a plain matching or standard relevance-based IR search system will fail to provide code examples that meet the features (requirements) described in Table 20. In this chapter, we describe and discuss the schematic approach for spotting working code examples (Figure 46). The key to this approach is the use of data mining approaches to extract popular abstract solutions from a comprehensive code corpus. These abstract solutions will then be used to recommend either the next potential steps (e.g., [WAN13]) or to complete code examples (e.g., [MIS12]). Since several solutions can be matched to a given query, the popularity of solutions has been exploited to reduce the risk of returning a poor quality result set (e.g., [WAN13][MIS12][BUS12]). The intuition is that the higher the popularity of a potential solution, the higher the chance of acceptance by the end-user. However, this approach is still subject to some threats, which are discussed in this section, specifically concerning spotting of the working code examples.


Examples of the popular abstract solutions:
Offline
Code Corpus (e.g., Sourceforge.net)
Solution Mining
generates
Popular Abstract Solutions
select the best matches (code fragments)
select the best matches (solutions)
Online
Spotting Working Code Examples
Solution Matching
Example of a matching working code examples:
Query:

```
1-{File.openFile(),File.ReadLine(),File.close()}
2-{File.openFile(),File.ReadLine(),PrintToConsole(),File.close()}
```

```
File f=new File ();
String fileName="readme.txt";
File.openFile(fileName);
for(int i=0;i<totalLineNo;i++)
{
   String lineContent=File.ReadLine(i);
   PrintToConsole("Line: "+i);
   PrintToConsole(lineContent);
}
f.close();
```

```
openFile and ReadLine
```

Figure 46. The schematic approach towards spotting working code examples

Popularity of a solution is a key criterion (Figure 46 – the solution matching process) that cannot be ignored (e.g., [MIS12][WAN13]) to avoid poor quality result set. However, there are other factors affecting the selection process. For example, relevance to the query cannot be completely ignored [HOL09]. For a free-form query, relevance is continuous (not binary), so solutions other than simple filtering and matching are required. In addition, conciseness and completeness (Table 20) are two important but often contradictory aspects when optimizing the result set. The tradeoff between these factors makes spotting the best matches a challenging task.

We provide a reasonably representative dataset including the source code of ~25,000 Java open source projects (Table 21 summarizes our corpus characteristics) that is essential for mining abstract programming solutions (e.g., Figure 46). The size and richness of our repository is the key to the success of the approach. However, the size of the corpus also introduces new challenges. Given the large number of potential matches (for search steps in Figure 46 - abstract solution and code fragments search), the size of our corpus not only provides a richer knowledge base, but also increases the noise level. From this point of view, the solutions (e.g., [WAN13]) for the schematic in Figure 46 suffer from the same challenges as traditional Web search.

Table 21. The characteristics of our corpus (both raw and processed data)

Aspect$^{\Delta}$		**Value**
Raw Data		
Java projects		24,824
Total Java files		12,104,499
Unique$^{\sim}$ Java files		2,882,458
LOC		~300 M
Selected fragments^{+}		5,436,638
Selected lines*		65,478,267 (LLOC)
Processed Data		
Unique encoded lines		13,945,442
Observed frequent abstract solutions$^{■}$	#Solutions	15,856,377
	Size (#encoded lines)	140,410,866 (encoded lines)
	#Unique items	77,905
	Min support	20
	Max observed support	2,412

$^{\Delta}$the table reports the number of encoded lines which is smaller than the actual #unique LLOC. For example int y=0; and int x=0; are counted only once since their encoded patterns are identical. $^{\sim}$duplicated files are eliminated via their shared fully qualified name $^{+}$fragments with at least 5 Logical Line of Code (LLOC) *LLOC after removing duplicated encoded lines within each fragment $^{■}$maximal frequent itemsets [BOR12] (min:4, max:30, support:20)

In addition to these explicit challenges (conciseness vs. completeness) and noise in the search space, there are further implicit issues reducing the success of popularity or relevance-oriented approaches (e.g., [WAN13][BUS12]). The following example illustrates such an implicit threat, based on hidden dependencies. In general, an ideal working code example should reflect a highly popular and concise abstract solution. In this section, we discuss the fact that satisfying both conditions is not trivial. As noted in Table 20, size is one of the measures used for evaluating the conciseness of a recommendation. Figure 47 summarizes the average size of frequent (i.e., popular) abstract solutions that we observed in our studies. The abstract solutions are grouped by their popularity degree, which is measured by the number of occurrences (Figure 47 - the support value) of the solution within the corpus. The result shows that although the size decreases as the popularity (i.e., the support value) increases, the changes are not considerable. If we ignore the first three groups (i.e., $support = \{20, 21, 22\}$), the size remains in the narrow range between 6 and 5. Since the size seems constant, one can argue that it can be ignored in favor of the popularity aspect. However, this is not always the case, as illustrated in the following example.

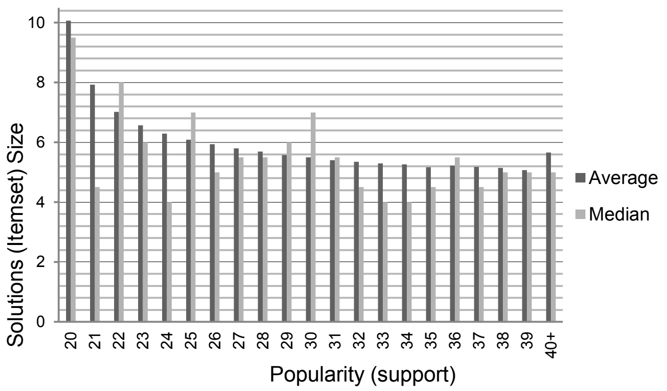


Figure 47. The average size of our corpus abstract solutions with certain popularity

In our corpus (Table 21), there are 6,836,738 relevant frequent abstract solutions[7] for the MD5 hash value generation problem[8]. The most popular solutions have 134(17), 175(10), and 195(10) occurrences, with the number in parenthesis indicating the size of the solution in LOC. None of these solutions are close to the satisfying answers for MD5 hash value generation. Although their popularity is highest, we observed that they spot false positive fragments. Among the true positive answers, Figure 48 (popularity=24 and size=7) and Figure 49 (popularity=65 and size=6) present two spotted answers. Even though Figure 49 is associated with a smaller and more popular solution, it provides a lower quality solution as a working code example, as it is neither self-contained nor complete. The lower quality is due to calling the convertToHex() method in the last line, which makes the returned solution less concise (Table 20). These examples highlight the presence of implicit challenges for the schematic approach (Figure 46), such as the popularity-size tradeoff.

```
MessageDigest md = MessageDigest.getInstance("MD5");
md.update(s.getBytes());
byte digest[] = md.digest();
StringBuffer result = new StringBuffer();
for (int i = 0; i < digest.length; i++) {
    result.append(Integer.toHexString(0xFF & digest[i]));
}
return result.toString();
```

Figure 48. A high quality true positive for the MD5 example (size=7 support=24 rank=1)

```
MessageDigest md;
md = MessageDigest.getInstance("MD5");
byte[] md5hash = new byte[32];
md.update(text.getBytes("iso-8859-1"), 0, text.length());
md5hash = md.digest();
return convertToHex(md5hash);
```

Figure 49. A low quality true positive for the MD5 example (size=6 support=65 rank=8)

[7] These solutions are identified via their association to 82 unique ep of 7,251 relevant lines of code.
[8] Using MessageDigest API ($query = \{md5, MessageDigest\}$)

10.3. Adaptation of clone search for the code search problem

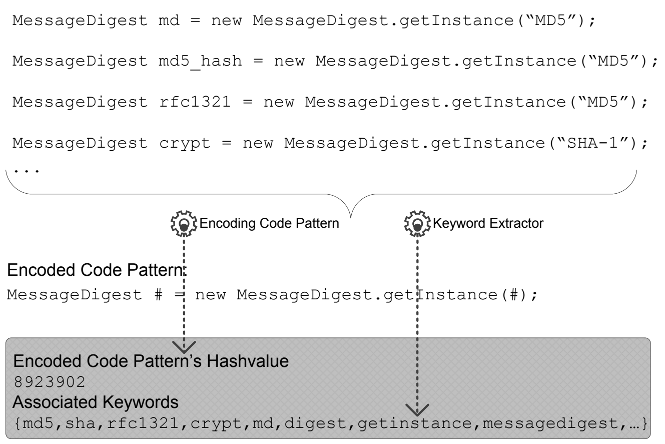
The discussion and observations made in the previous section illustrate that all characteristics (Table 20) of working code examples contribute to the filtering process of poor quality answers. Since achieving an optimum result set can be impractical [MIS12], the alternative is to provide *retrieval* and *ranking* models that are capable of producing high quality ranked result sets. A clone search model using the vector space model can be applied on both search steps of the schematic approach (Figure 46) to address the enumerated concerns and challenges in the previous section. For example, a proper SeClone schema is able to address the complications related to relevancy, completeness, and conciseness, with regard to the query as supported by our performance evaluation study (Chapter 8). Contrary to the other approaches (Chapter 2), where popularity is the main factor contributing to the ranking, this approach considers popularity as a necessary condition during search space deployment. This section describes how clone search can be adapted for the search problems available in the schematic approach (Figure 46) by providing a concrete solution as the motivating example for the research community.

10.3.1. Populating the search space

The schematic approach (Figure 46) requires at least two data families: (1) the code fragments and (2) the popular abstract programming solutions. While the code fragments can be extracted from extensive web crawling and data gathering (Table 21), the abstract programming solutions require different types of data abstraction and mining. The details of the abstraction and mining methods are described in this section, where both search steps at Figure 46 are realized using clone search models.

10.3.1.1. The initial search space - code abstraction

Creating *abstract programming solutions* requires modelling and transforming programming content (code fragments) to higher levels of abstraction. Creating these abstract programming solutions is essential for the performance, since it allows for the removal of unnecessary details from code content. In our illustrative solution, we adapt and extend the SeClone search space and the encoded code pattern (ep) approach (Section 4) by including the keywords. The encoded code patterns and their associated keywords (Figure 50) constitute the baseline search space.



```
MessageDigest md = new MessageDigest.getInstance("MD5");
MessageDigest md5_hash = new MessageDigest.getInstance("MD5");
MessageDigest rfc1321 = new MessageDigest.getInstance("MD5");
MessageDigest crypt = new MessageDigest.getInstance("SHA-1");
...
```

```
MessageDigest # = new MessageDigest.getInstance(#);
```

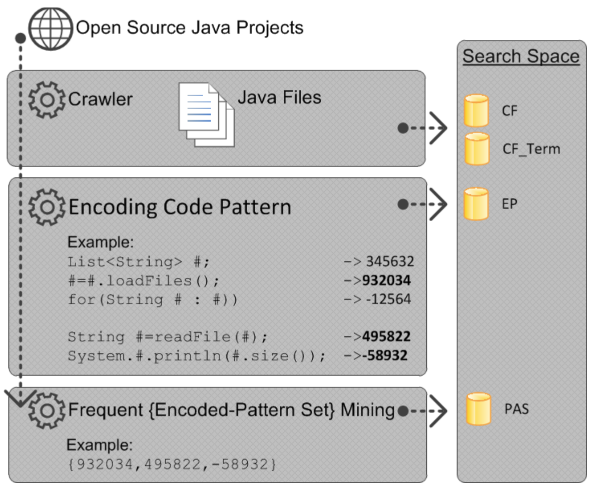
```
8923902
```

```
{md5,sha,rfc1321,crypt,md,digest,getinstance,messagedigest,…}
```

Figure 50. A sample encoded code pattern and its associated keywords

10.3.1.2. Complete search space - encoded pattern mining

Including only the encoded code patterns and associated keywords as part of the search space is not sufficient to support spotting working code example problems, as both of them (1) are too fine-grained to be considered code examples, and (2) lack of support for code popularity. In order to identify the popular abstract programming solutions (e.g., Figure 46), a maximal frequent itemset mining such as the FPgrowth algorithm [BOR05] can be employed. Since the input for the algorithm is made up of encoded code patterns (not the actual code), the output will be popular abstract programming solutions (or pas). Figure 51 illustrates the details of populating the search space and different processing steps involved, based on the following legend: Code Fragment (CF), Keywords (CF_Term), Encode Patterns (EP), Popular Abstract Solution (PAS).



```
List<String> #;                ->345632
#=#.loadFiles();               ->932034
for(String # : #))             ->-12564

String #=readFile(#);          ->495822
System.#.println(#.size());    ->-58932
```

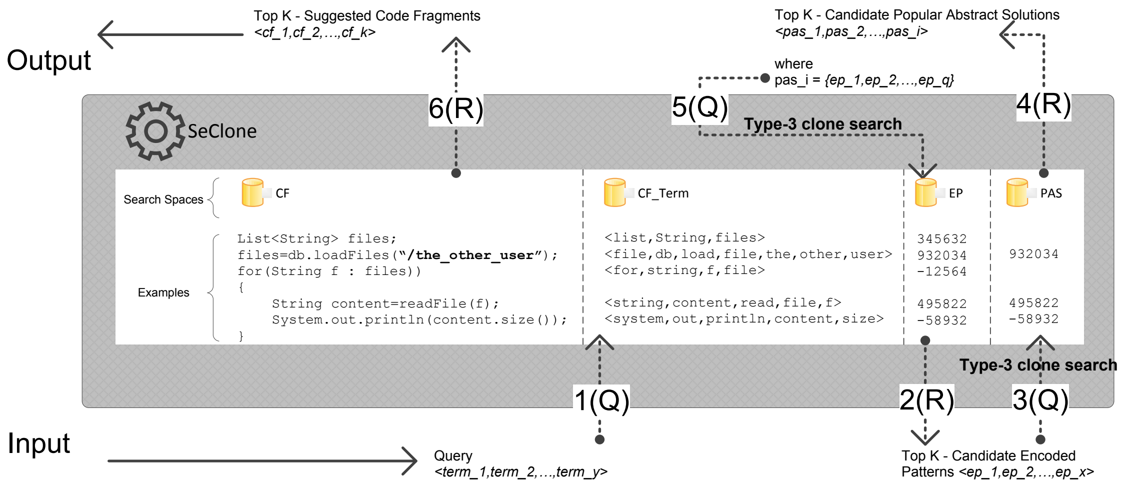
```
{932034,495822,-58932}
```

Figure 51. The search space population process

Frequent itemset mining algorithms [BOR12] are capable of extracting popular patterns within a provided record set, with a record being one or more items. In its most simplistic form, the algorithm requires a dataset and a *support* value that determine the minimum number of occurrences of a pattern in the whole record set before it can be considered a frequent item. Originally, the frequent itemset mining concept did not consider any ordering constraint between items. For a clone search-based spotting approach, a variation of the itemset mining concept referred to as maximal frequent itemset mining is required. This variation has two specific properties: (1) it considers maximal itemsets and (2) it has no ordering constraint. The omission of the ordering constraint provides us with a robust mining feature, where re-ordering of code statements does not interfere with the pattern mining process. The maximal property overcomes some of the challenges of the other itemset mining approaches, such as the possibility of producing an exponential number of frequent sub-itemsets. The occurrence of sub-itemsets in the search space is a threat when answer completeness is required. Therefore, we can define a maximal itemset as: given m possible elements (i.e., encoded code pattern) in the code base $E = \{e_1, e_2, \dots, e_m\}$ and n code fragments $C = \{c_i \mid c_i \subseteq E, i \in \{1, \dots, n\}\}$, $PAS^{y,x}$ is the set of all possible reputable code patterns defined as ${P_R}^{y,x} = \{e_l \mid \exists A \subseteq C, |A| = y, e_l \in \bigcap_{c_k \in A} c_k\}$ where $|{P_R}^{y,x}| \geq x$. A frequent itemset[9] ${P_R}^{y,x}$ is maximal if $\nexists {P'_R}^{y,x} \in PAS^{y,x}, {P_R}^{y,x} \subset {P'_R}^{y,x}$.

10.3.2. Search process

Queries for a source code search engine are usually a set of terms, which are used for retrieving and matching code fragments, as well as for ranking. To satisfy the schematic approach (Figure 46), the clone search-based solution requires, in total, three phases of querying to support the spotting problem at run-time. Figure 52 provides an overview of this three-phase querying process and the dataflow among these processes for a single search query.



```
List<String> files;
files=db.loadFiles("/the_other_user");
for(String f : files))
{
    String content=readFile(f);
    System.out.println(content.size());
}
```

```
<list,String,files>
<file,db,load,file,the,other,user>
<for,string,f,file>

<string,content,read,file,f>
<system,out,println,content,size>
```

```
345632
932034
-12564

495822
-58932
```

```
932034

495822
-58932
```

Figure 52. Our concrete solution - the three querying phases

[9] x is the minimum size and y is the support (i.e., min popularity of the pattern)

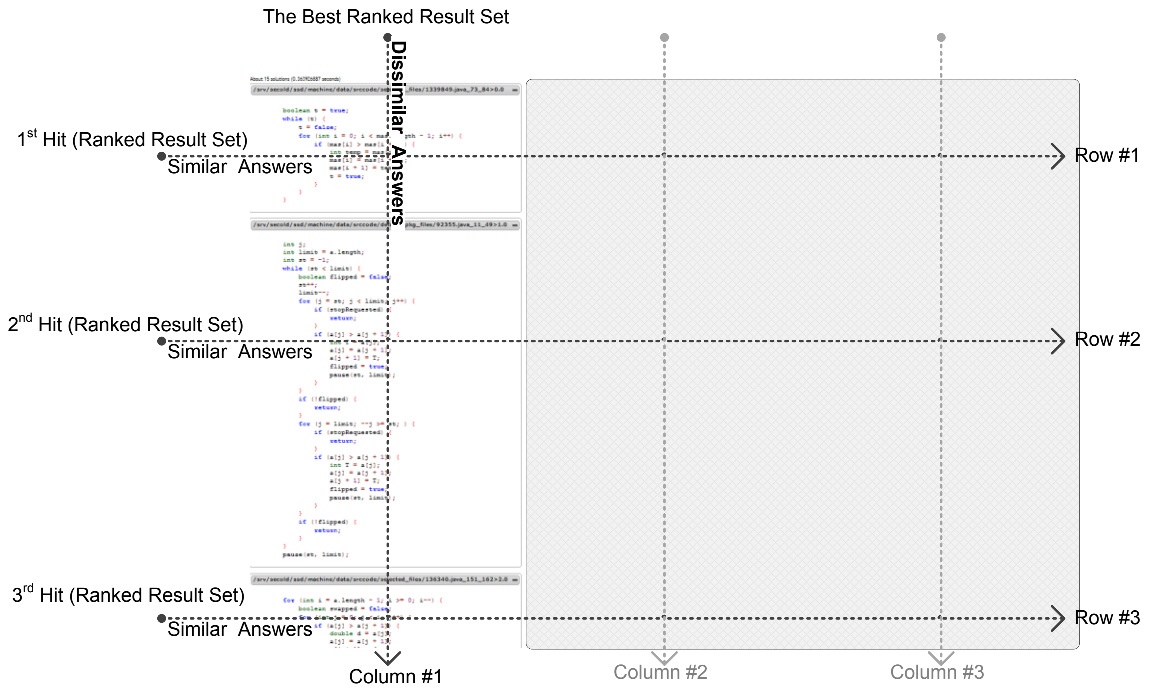
Three-phase querying process. For a given free-form query $q =< term_1, term_2, \ldots, term_y >$, the approach returns the most relevant code fragments by finding: (1) the most relevant encoded code patterns, (2) the most relevant popular abstract solution for selected encoded patterns, and (3) the most relevant fragments for a given solution.

Phase 1. The first querying process, Figure 52 - 1(Q), selects the $top\ K$ relevant encoded patterns, comparing their associated keywords to the query terms. That is, the data used in this search problem are query terms and ep keywords, while the output consists of encoded patterns. It should be noted that an encoded code pattern ep that shares a keyword with q is not automatically included in the candidate list. Only $top\ K$ hits are selected, in order to maintain the relevancy between the query and the final spotted code fragments, as query terms are no longer used explicitly in the search process after this phase.

Phase 2. In this phase, the $top\ K$ popular abstract solutions are identified using clone search, Figure 52 - 3(Q), where the query is made of the candidate encoded patterns from the last step output. Due to the clone search-based approach, the $top\ K$ popular abstract solutions are ranked based on their relevancy, completeness, and conciseness.

Phase 3. During the last querying phase, Figure 52 - 5(Q), the spotting of the best working code examples for each of the chosen abstract solutions takes place. Additionally, this step ensures that the output fragments are syntactically and semantically correct, which is crucial as our pas mining and querying model ignores the ordering of the statements.

The result of this search approach is a two-dimensional hit list for each free-form query. Figure 53 illustrates a graphical representation of such hit list. Each row contains the ranked code fragments matching a corresponding abstract solution (i.e., pas_i in Figure 52). Therefore, while the fragments in each row are highly similar, they look different from solutions in other rows, as they are satisfying different abstract solutions. The default presentation approach is to select the (final) $top\ K$ hits from the items of column #1 in Figure 53 to maximize the number of covered abstract solutions.



About 15 solutions (0.363926887 seconds)

```
/srv/secold/ssd/machine/data/srccode/se[illegible]_files/1339849.java_73_84>0.0

boolean t = true;
while (t) {
    t = false;
    for (int i = 0; i < mas[illegible]gth - 1; i++) {
        if (mas[i] > mas[i [illegible]]) {
            int temp = mas[illegible]
            mas[i] = mas[i [illegible]];
            mas[i + 1] = te[illegible]
            t = true;
        }
    }
}
```

```
/srv/secold/ssd/machine/data/srccode/de[illegible]pkg_files/92355.java_11_49>1.0

int j;
int limit = a.length;
int st = -1;
while (st < limit) {
    boolean flipped = false;
    st++;
    limit--;
    for (j = st; j < limit; j++) {
        if (stopRequested) {
            return;
        }
        if (a[j] > a[j + 1]) {
            int T = a[j];
            a[j] = a[j + 1];
            a[j + 1] = T;
            flipped = true;
            pause(st, limit);
        }
    }
    if (!flipped) {
        return;
    }
    for (j = limit; --j >= st; ) {
        if (stopRequested) {
            return;
        }
        if (a[j] > a[j + 1]) {
            int T = a[j];
            a[j] = a[j + 1];
            a[j + 1] = T;
            flipped = true;
            pause(st, limit);
        }
    }
    if (!flipped) {
        return;
    }
}
pause(st, limit);
```

```
/srv/secold/ssd/machine/data/srccode/selected_files/136340.java_151_162>2.0

for (int i = a.length - 1; i >= 0; i--) {
    boolean swapped = false;
    for (int j = 0; j < i; j++) {
        if (a[j] > a[j + 1]) {
            double d = a[j];
            a[j] = a[j + 1];
```

Figure 53. The two-dimensional ranked result set

10.4. Performance evaluation

In this section, we provide a summary of our performance evaluation study (the feasibility study) for the clone search-based approach. We evaluated the approach for its ability to spot working code examples by reporting the top K hits, where K is a relative small number (3 or 5). The summary of the corpus and output of the mining is presented earlier in Table 21. We determined the rank of the first true positive answer based on the five requirements we identified in Table 20 (excluding the result set feature, which is not applicable here). We then applied the suggested quantitative measures for these requirements, to evaluate the performance of the clone search-based approach with regard to correctness, conciseness, completeness, and readability. Since there is no explicit measure for ease of integration, we evaluated the spotted examples through an initial user study.

10.4.1. Performance result

As part of our performance evaluation, we adapted Mishne et al.'s query set [MIS12]. The dataset includes 7 queries from 6 Java libraries, with the first true positive rank as the performance evaluation measure. However, we extended it by including additional measures and queries. The additional queries are based on Java code search examples available in the literature (Chapter 2), or frequent programming questions posted on StackOverflow. We also extended the measure set by including normalized discounted cumulative gain (NDCG) in addition to the original measure (rank of the first best hit). Moreover, measures for correctness, completeness, readability, and conciseness features (Table 20) are exploited to identify the true positives amongst the hit list. These features are calculated via their quantitative measures (Table 20). Since this approach rarely reports false positives, we cannot consider precision, recall, or F-measures as representative measures. Finally, for purpose of comparison, we report the results of Koders[10].

10.4.1.1. Overall result

A summary is shown in Table 22, and is followed by a more detailed view on the measures in Table 23. Our observation shows that the clone search-based approach can successfully spot the working code example in the top 2 hits for free form querying. The order of query terms does not affect the result. Moreover, the query can be a mixture of class names, method names and general keywords (e.g., query# 9). While the corpus contains thousands of textual matches for each query, the clone search-based approach is capable of reducing the search space to a limited number (i.e., ~100) of eps (Table 23 within parentheses in the ep column). One of the reasons the approach returns fewer matches is that our search approach reduces the search space step by step (Table 23 shows the number of matches per step).

[10] http://www.koders.com

When comparing our results with Koders, the illustrative solution using clone search always returns fewer (but high quality) matches (i.e., first true positive rank between 1 or 2). In contrast, for the Koders code search engines, the best rank results (first true positive) fluctuate between 1 and 40+, and the returned results were often actual working examples (Table 22, the values within parentheses in the Koders column).

By comparing the quality of the spotted working code examples, the results in Table 22 show that the best hits always meet readability and correctness requirements. In terms of completeness, our approach spotted complete answers in all cases except query #8, where the exception handling statement was omitted. In terms of conciseness, our best hit size is always smaller than the average hit size (Table 23), however, conciseness (measured using irrelevant LOC) shows a fluctuation across the experiments. Table 22 summarizes the conciseness of the first best hits by High, Acceptable, or Low.

Our approach failed to spot any valid answer for query #6. Our further investigation revealed that query #6's expected solution is not a working code example. Table 23 also reports the NDCG values for two groups of top 5 hit, using two different result preview approaches. The vertical schema (the default presentation approach e.g., Figure 53) only shows the top-5 hits from the first column, whereas the alternative view generates the preview by selecting two hits from each row. In general, we observed that the vertical preview not only reports a higher number of true positives, but also higher quality hits based on the observation made by NDCG.

Table 22. The dataset and evaluation summary for the spotting problem

Query ID	Query	Query description	Query terms	Number of hits	Best hit rank	Correctness	Completeness	Conciseness	Readability understanding	Koders best answer's rank (working code example?)
1	Apache Commons CLI [MIS12]	Retrieve arguments from command line	{getOptionValue, CommandLine}	5+	1	Y	All	Acc.	yes	4 (no)
2	Eclipse UI [MIS12]	Check user selection	{ISelection, isEmpty}	5+	1	Y	All	High	yes	2 (no)
3	Eclipse GEF [MIS12]	Set up a ScrollingGraphicalViewer	{ScrollingGraphicalViewer}	2	1	Y	All	High	yes	32 (no)
4	Eclipse JDT [MIS12]	Create a project	{IProject,monitor}	5+	2	Y	All	High	yes	No (no)
5	Apache Commons Net [MIS12]	Successfully login and logout	{FTPClient}	5+	1	Y	All	High	yes	1 (no)
6	WebDriver [MIS12]	Click an Element	{WebElement}	0	-	-	-	-	-	8 (-)
7	JDBC [MIS12]	Commit and rollback a statement	{executeUpdate,rollback, PreparedStatement}	4	1	Y	All	High	yes	No
8	StackOverflow HTTP	send a HTTP request via URLConnection in Java	{response,URLConnection}	5+	2	Y	No	High	yes	22 (yes)
9	StackOverflow Runtime	Redirect Runtime exec() output with System	{read,Runtime}	2	1	Y	All	Low	yes	12 (yes)
10	StackOverflow Memory	Get OS Level information such as memory	{Memory}	5+	2	Y	All	High	yes	1(yes)
11	StackOverflow SSH	SSH Connection	{ssh}	5	1	Y	All	Low	yes	3 (no)
12	StackOverflow Download	Download and save a file	{download,URLConnection}	5+	1	Y	All	Acc.	yes	5(no)
13	StackOverflow MD5	Generate a string-based MD5 hash value	{md5}	5+	1	Y	All	High	yes	3(no)
14	HttpResponse	Read the content of a HttpResponse object line by line	{readLine,HttpResponse}	2	1	Y	All	Acc.	yes	40+ (no)
15	Lucene	Search via Lucene and manipulate the hits	{search,IndexSearcher}	5+	2	Y	All	High	yes	1(no)

Table 23. The details of the evaluation

Query ID	Best hi LOC	Avg. LOC top K	Irrelevant LOC	Scores for NDCG 2D preview	Scores for NDCG vertical preview (default)	#Textual matches	#EPs	#PASs
1	75	80	8	3-2-3-1-3	3-3-3-1-3	21718	2455 (129)	100+
2	13	25	0	3-1-3-1-1	3-3-1-2-2	43000+	201 (20)	100+
3	19	29	0	-	3	80	20 (1)	1
4	28	57	0	0-0-3-2-1	0-3-1-1-1	11851	608 (56)	100+
5	29	32	0	3-3-3-3-3	3-3-3-3-3	2410	725 (84)	100+
6	-	-	-	-	-	662	-	-
7	23	27	0	3-2-3-1	3-3	40000+	99 (34)	100+
8	16	22	0	1-1-3-3-3	1-3-3-1-3	6987	732(118)	100+
9	44	44	25	2	2-2	17223	386(23)	100+
10	8	40	0	1-1-3-2-2	1-3-3-3-2	30000+	6087(929)	100+
11	18	50	3	3-1-1	3-1-1-1-1	10045	858(201)	100+
12	36	47	2	3-2-3-1-1	3-3-1-1-2	6987	652(115)	100+
13	25	73	0	3-3-0-0-0	3-0-0-1-3	11358	2628(381)	100+
14	25	25	6	2	2-2	20000+	10(8)	100+
15	18	20	0	1-3-3-3-3	1-3-3-3-3	20000+	52(16)	100+

10.4.2. Initial user study

Since no specific measure for the ease of integration of working code examples was available (Table 20), we conducted an initial user study to evaluate this aspect. For our controlled study, we adapted a user study configuration (number of tasks, groups and people) for .NET framework that is proposed by Wang et al. [WAN13]. Table 24 summarizes the user study settings. We chose Koders and StackOverflow as alternative sources for spotting working code examples. The provided hints in Table 24 can be used as query seeds by the programmers, which are selected from three possible combinations: API names (e.g., class or method names), general keywords (e.g., MD5 or download), or a combination of both. The general keywords are not (neither completely nor partially) part of the participant class or the method names in the solution domain.

We replaced the C# tasks (derived from Wang's study [WAN13] using six developers identified by P1 to P6) by Java tasks using the queries listed in Table 24. The complete programming assignment, to be completed by the programmer, was to develop a software solution that retrieves a specific argument passed to the executed process via command line. The retrieved argument must be used to generate a MD5 hash value. The MD5 hash value determines the file UID on the web server. Finally, the target file on the Internet must be downloaded and saved on the local disk.

Task 1: Retrieve the argument. The goal is to read a specific argument (i.e., “n”) via Apache Commons CLI library. The name of the class from the library responsible for the given task is provided as the seed (i.e., hint) for the search process. The challenge is to handle the exceptional cases (e.g., null values) and errors carefully.

Task 2: Generate MD5 hash value. This task mandates the programmer to generate a string representation of the MD5 hash value for the Task 1 retrieved argument. The extra challenge here is the proper conversion of the value from binary format to string. The provided hint is “MD5”.

Task 3: Download and save file. The goal is to download and save a specific file from the Internet. The file name is equal to the generated MD5 hash value. Proper connection establishment, content encoding, and exception handling constitutes the major challenges of this task.

Tables 24 and 25 summarize the study configuration and the observation, respectively. In short, it shows the potential capabilities of a clone search-based approach in comparison to the other resources, as it either achieves equal result or outperforms the others. However, we are interested in the outcome of the study in terms of ease of integration. Table 25 provides initial evidence that the ease of integration feature is met by the code examples that are provided by the clone-search approach. Specifically, the tasks are completed successfully in less time using our approach, compared to the StackOverflow-based development study.

Table 24. The controlled user study configuration

	Seed Query (hint)	**StackOverflow**	**Koders**	**Our approach**
Task 1	CommandLine	P1 P2	P3 P4	P5 P6
Task 2	MD5	P3 P4	P5 P6	P1 P2
Task 3	URLConnection and download	P5 P6	P1 P2	P3 P4

Table 25. The controlled user study configuration

	StackOverflow	**Koders**	**Our approach**
#Successful integration	4/6	4/6	6/6
Time (avg. - minutes)	24	28	17
#Search activities per task	2	4	2

10.5. Discussion and promoting Examples

In this section, we describe three illustrative examples that highlight the capabilities and interesting features of clone search-based approach for the given problem.

10.5.1. Bubble sort example

Bubble sort is one of the classical code search queries used by programmers. Figure 54 shows the first hit that our spotting approach returns for the bubble sort query. The result is based on 5.5 million indexed code fragments that each has at least 5 lines of code. While the returned result is one of the possible implementations of a bubble sort algorithm, it also highlights one of the most interesting features of our clone search-based approach for code search. A matching answer might not necessarily have to contain the query terms. In this example, there is no occurrence of bubble, sort, or bubblesort keywords within the spotted fragment, while the code fragment is actually implementing a bubble sort. It should be pointed out that our search approach only uses the content of code fragment, and does not consider other sources of associated information such as inline comments, Java docs, and the signature of the owner method.

```
boolean t = true;
while (t) {
    t = false;
    for (int i = 0; i < mas.length - 1; i++) {
        if (mas[i] > mas[i + 1]) {
            int temp = mas[i];
            mas[i] = mas[i + 1];
            mas[i + 1] = temp;
            t = true;
        }
    }
}
```

Figure 54. The bubble sort example

10.5.2. MD5 example

Another example is related to the generation of MD5 hash values as string. This hash code generation is not a trivial programming task using Java native libraries. First, there is no method or class name existing within the Java libraries called MD5. The actual class and methods responsible for the MD5 Binary value generation are MessageDigest, getInstance() and update(). Second, the conversion of the binary representation to string, has special cases to be handled, which are highlighted by the programming community[11]. If the generated hash value starts with 0, this leading 0 will be omitted during the conversion from the original format to String (Binary → Numeric → String). This can be problematic, as all MD5 hash values must have an equal number of characters. Figure 55 presents a top rank hit that our approach returns for the MD5 query and addresses all of the discussed challenges.

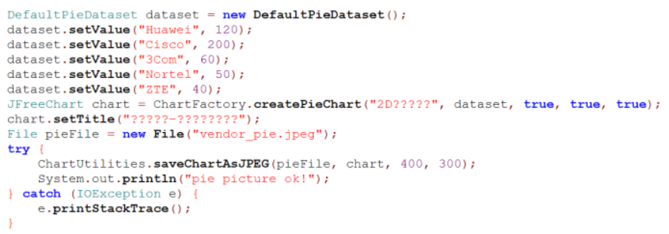
```
byte[] unencodedPassword = password.getBytes();
MessageDigest md = null;
try {
    md = MessageDigest.getInstance(algorithm);
} catch (Exception e) {
    log.error("Exception: " + e);
    return password;
}
md.reset();
md.update(unencodedPassword);
byte[] encodedPassword = md.digest();
StringBuffer buf = new StringBuffer();
for (byte anEncodedPassword : encodedPassword) {
    if ((anEncodedPassword & 0xff) < 0x10) {
        buf.append("0");
    }
    buf.append(Long.toString(anEncodedPassword & 0xff, 16));
}
return buf.toString();
```

Figure 55. The MD5 example

[11] http://stackoverflow.com/questions/415953/generate-md5-hash-in-java

10.5.3. Save chart as JPEG example

JFreeChart is a chart visualization library for Java. Saving a chart as a JPEG using JFreeChart library requires a query belonging to the API usage example identification problem (e.g., [MIS12][WAN13][BUS12]), which is different from the bubble sort example (i.e., algorithmic problems). Figure 56 illustrates the first hit returned by our approach. The fragment not only shows how to save the chart, but also includes all required steps (e.g., variable initialization) as a self-contained working code example. Note that holding the second property by the provided answer is necessary [KIM10] [BUS12] in such code search models (Table 20).



```
DefaultPieDataset dataset = new DefaultPieDataset();
dataset.setValue("Huawei", 120);
dataset.setValue("Cisco", 200);
dataset.setValue("3Com", 60);
dataset.setValue("Nortel", 50);
dataset.setValue("ZTE", 40);
JFreeChart chart = ChartFactory.createPieChart("2D?????", dataset, true, true, true);
chart.setTitle("?????-????????");
File pieFile = new File("vendor_pie.jpeg");
try {
    ChartUtilities.saveChartAsJPEG(pieFile, chart, 400, 300);
    System.out.println("pie picture ok!");
} catch (IOException e) {
    e.printStackTrace();
}
```

Figure 56. The save chart as JPEG example (JFreeChart Library)

In summary, the given examples highlight three major features for a clone search-based solution: (1) spotting working code example for API usage and algorithmic problems, (2) the ability to provide some form of self-contained examples, and (3) less dependency on term matching. Furthermore, our proposed illustrative solution requires only the code block content[12]. These features illustrate the potential of clone search for code search applications in the context of pragmatic reuse. These potentials can be exploited to either eliminate the limitations of earlier approaches, or for further improvements.

[12] Comments, Javadoc and the method signatures are excluded

10.6. Summary

In this chapter we have described how clone search models can be applied to improve Internet-scale code search for pragmatic reuse. The purpose of this chapter was not to provide a concrete solution limited to a specific research problem. Rather, we tried to show how clone search models can contribute to the actual code search problem at large by providing a sample solution. Such a clone search-based approach is in contrast to the earlier solutions (Chapter 2), which were based on ad-hoc code fingerprinting, pattern mining, and popularity-oriented solutions. Finally, our approach differs from the existing solution, since it is capable of taking into consideration formal code similarity definitions (e.g., Type1, 2, and 3) not only during the search space creation (detection of popular abstract solutions), but also during the final search and ranking steps (matching popular abstract solutions with working code examples).

11. Discussion

This dissertation has proposed a clone search model that can be adapted for applications that require a source code similarity search. The proposed model supports scalability, fast response time, ranking, and Type-1, 2, and 3 detection. This chapter provides a discussion on potential threats that must be taken into consideration. The chapter concludes with a list of immediate future work.

11.1. Threats to validity

11.1.1. Data characteristics study

Our data characteristics studies covered different aspects of the data in our research domain such as corpus growth rate, data outliers, and the strength of the hash function. However, the observations depend on three major factors: (1) the input, (2) the granularity of the study, (3) the selected encoded code patterns, and (4) the underlying hash function. Although we tried to consider a representative dataset for our studies, all conclusions drawn from our case studies remain highly dependent on our input data (dataset). For example, using a dataset from industrial or closed code systems, the conclusions will most likely differ, since the quality of the code might differ. Furthermore, our studies are limited to Java source code and Java bytecode. Additionally, the results are limited to line-level clone detection, and therefore our results and conclusions cannot be generalized to other granularities such as token-level clone detection. Finally, we have selected an encoded code pattern (Table 3 function m) that will result in high recall. Achieving high recall helped us to study the worst-case scenarios for our retrieval and ranking steps, as it resulted in a large number of candidates to be ranked when pattern similarity holds. Therefore, results will differ if different encoded code patterns are selected.

11.1.2. Performance evaluation study

Considering our evaluation approach, the quality of our benchmark plays an important role, since it has a direct impact on the outcome of the performance evaluation. Therefore, the following issues must be taken into consideration: (1) since no other benchmark that is applicable for the evaluation of clone search results and ranking performance exists, we created our own benchmark using a mutation framework to generate an oracle of known clones. A key challenge, as with any other benchmark, is how closely this benchmark reflects actual data. We address some of these threats by creating a dataset that we believe is representative enough in size (containing 25,000 different open source projects and approximately 356 MLOC). Furthermore, the mutation framework output (additional clones as our oracle) is injected to our corpus to ensure that a minimum number of clone instances are available for each query, to facilitate recall calculation. Moreover, for manually assigning the relevance scores, our tagging is biased towards Internet-scale code search and pragmatic reuse. Some of the results (e.g., Type-3 clones), which we considered as non-relevant for clone search, might be considered relevant in other application contexts, such as clone detection for software maintenance. In an attempt to reduce the subjectivity during the manual scoring process, we tried to keep the scoring process as transparent and objective as possible, by following a concrete pre-defined scoring guideline (Table 14) for the different clone types.

Implementation. We have implemented our clone search models and all of its processing components in Java. While we performed testing of our implementation, we did not consider a formal validation of our design nor of the implementation (including the programming heuristics). Moreover, we used implementation level heuristics in some cases to achieve scalability.

11.2. Limitations

11.2.1. The clone search model

Our study focuses on a clone search model for Java source code and bytecode. However, support for other programming languages (in particular OO languages) requires a substitution of the language parser in most cases. While our model can be applied to the other programming languages such as C, its performance might become completely different and unpredictable, since our encoded code pattern generation rules have been designed for Java after an experimental analysis (Appendix 1) on code search query logs.

11.2.2. Application for pragmatic reuse

In principal, adapting our clone search model for the pragmatic reuse problem might result in two major limitations. (1) In cases where there is a lack of reuse samples in the input corpus, the approach will fail to find a working code example. This is a general issue related to such approaches, and is discussed in more detail, with examples, in [MAN05]. Specifically, if one attempts to apply pragmatic reuse to new programming libraries or new programming paradigms, there is no guarantee that sufficient examples will be captured in the corpus. (2) Although the performance of our clone search-based approach is promising in finding the working code examples, by no means does it replace human judgment when it comes to the negative issues associated with pragmatic reuse [HOI08].

11.3. Future work

We believe that the outcome of this dissertation provides the first step towards the adaptation of the clone search models for source code similarity search problems. The following summarizes some of the problems that should be addressed as part of future work:

- Studying the data characteristics (e.g., outliers) for the other dataset types (e.g., industrial systems) and languages
- Studying the applicability of our clone search model as a core similarity function for classification algorithms in data mining (e.g., for clone classification)
- Finding a solution for soft breakdown of the ranked result set, instead of top K approach with fixed k values. This feature is interesting, as the number of actual relevant items varies considerably for each query in the clone search versus in text retrieval.
- Applying our bytecode clone search models for concrete applications, such as finding duplicated bytecode fragments within source code search engine indices (e.g., [BAJ12]).
- Applying our source code search models for specific search problems related to pragmatic reuse. We provided hints to show the potentials of clone search models for emerging code search problems by elaborating on the problem of spotting working code examples. The proposed ideas can be adapted for specific code search problems such as recommendation, completion, and synthesis.

12. Conclusion

Historically, clone detection as a research discipline has focused on devising source code similarity functions that will cancel out negative code reuse effects in software maintenance. However, it has been observed (Chapter 2) that identifying duplications and similar programming patterns can be exploited for pragmatic reuse. Identifying such patterns requires a source code similarity model for detection of Type-1, 2, and 3 clones. Due to the lack of such a model, ad-hoc pattern detection models have been devised as part of the state of the art solutions in order to support pragmatic reuse via code search.

In this dissertation, we presented a clone search model that satisfies the fundamental enumerated requirements. First, we studied the performance of the proposed model for both source code and bytecode content. Second, we demonstrated how such a clone search model could replace the ad-hoc similarity models of the code search. Our research presents a clone search model that not only supports scalability, short response times, and Type-1, 2 and 3 detection, but also emphasizes ranking as a key functionality. The ranking of result sets is used to place highly similar fragments (hits) higher than other hits within the result set. It takes advantage of a multi-level indexing (non-positional) approach to achieve a scalable and fast retrieval with high recall. Result sets are ranked using two information retrieval ranking approaches: Jaccard similarity coefficient and cosine similarity via the vector space model, which we combine with code patterns' (not token) local and global frequencies modeled by various combinations. Users can customize the search schemata based on their specific application requirements.

For the evaluation, we created a large corpus (356M LOC) which, in combination with 50 sample queries and a total of 650 seeded Type-1, 2, and 3 clones, form our benchmark dataset for the analysis of our approach. The creation of this benchmark includes an extensive manual tagging of relevance scores covering more than 117,000 hits, which were used to evaluate the clone search model retrieval and ranking quality. We selected 6 measures to study different quality aspects and to evaluate and identify schemata (configurations) that are consistently outperforming the other schemata. Overall, our studies showed not only that SeClone is scalable to very large datasets, but also that certain schemata, such as $c.ltcj.l1.m3$ and $j.bnnn.l1.m3$ can produce high quality results in near real-time.

12.1. Research approach and contributions

As part of our preliminary research [KLL10], we noticed that resolving ambiguity of the source code is not sufficient for structural code search (in the pragmatic reuse context), since duplications (i.e., clones) reduce the result set quality. Furthermore, using our shuffling framework [KLT12][SAV13], we have observed that inter-project cloning is common in Java and the open source community at large. Finally, in [KLX12] we discussed that while code duplication often results in negative effects on the code search performance quality, the duplications can also be controlled and exploited in other ways for result set improvement. This background study provided major motivation to propose a clone search model with certain features to be used for such applications where a function for source code similarity measurement and detection is required. The major contributions of this dissertation[13] are as follows:

- Proposing a novel clone search model [KLX11][KLZ11]
- Extending the clone search model for bytecode content [KLQ12][KLE13][KLP12]
- Providing a schematic approach to show how a clone search model can be employed for supporting pragmatic reuse via code search

[13] The content of this thesis is based on our cited publications

13. References

[ABR79] P. S. Abrams and J. W. Myrna, "Automatic control of execution: An overview," International Conference on APL (APL '79), 9(4): 141-147, 1979.

[ALO12] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, "Detecting clones across Microsoft. NET programming languages," Working Conference on Reverse Engineering (WCRE), 405-414, 2012.

[BAK92] B. S. Baker, "A program for identifying duplicated code," Computing Science and Statistics, 24:49-57, 1992.

[BAK98] B. S. Baker and U. Manber, "Deducing similarities in Java sources from bytecodes," USENIX Annual Technical Conference (ATEC '98), 1998.

[BAJ12] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for the large-scale collection and analysis of open-source code," Science of Computer Programming, 2012.

[BAR10] L. Barbour, H. Yuan, and Y. Zou, "A technique for just-in-time clone detection in large scale systems," International Conference on Program Comprehension (ICPC), 76-79, 2010.

[BAX98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees,", International Conference on Software Maintenance, 368-377, 1998.

[BEL07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Transactions on Software Engineering, 33(9): 577-591, 2007.

[BRU06] M. Bruch, T. Schäfer, and M. Mezini, "FrUiT: IDE support for framework understanding," OOPSLA Workshop on Eclipse Technology eXchange (Eclipse '06), 55-59, 2006.

[BAZ11] S. Bazrafshan, R. Koschke, and N. Gode, "Approximate code search in program histories," Working Conference on Reverse Engineering (WCRE), 109-118, 2011.

[BAS13] H. Bast and M. Celikik, "Efficient fuzzy search in large text collections," ACM Transactions on Information Systems, 31(2), 59 pages, 2013.

[BOR05] C. Borgelt, “An implementation of the FP-growth algorithm,” International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations (OSDM '05), 1-5, 2005.

[BOR12] C. Borgelt, “Frequent item set mining,” Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2(6):437-456, 2012.

[BRA10] J. Brandt, M. Dontcheva, M. Weskamp, S. R. Klemmer, “Example-centric programming: integrating web search into the development environment,” SIGCHI Conference on Human Factors in Computing Systems, 513-522, 2010.

[BRI98] S. Brin, L. Page, “The anatomy of a large-scale hypertextual Web search engine,” Computer Networks and ISDN Systems, 30(1):107-117, 1998.

[BRU09] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '09), 213-222, 2009.

[BBO99] B. Boehm, “Managing software productivity and reuse,” Journal Computer, 32(9):111-113, 1999.

[BUS12] R. P. Buse and W. Weimer, “Synthesizing API usage examples,” International Conference on Software Engineering (ICSE), 782-792, 2012.

[BUT00] G. Butler, R. K. Keller, and H. Mili, “A framework for framework documentation,” ACM Computing Surveys (CSUR), 32(1), 2000.

[CAR93] S. Carter, R. J. Frank, and D. S. W. Tansley, “Clone detection in telecommunications software systems: A neural net approach,” International Workshop on Applications of Neural Networks to Telecommunications, 273-287, 1993.

[CAU86] P. J. Caudill and A. Wirfs-Brock, “A third generation Smalltalk-80 implementation,” Conference on Object-oriented Programming Systems, Languages and Applications(OOPLSA '86), 21(11): 119-130, 1986.

[CHA11] S. Chaki, C. Cohen, and A. Gurfinkel, “Supervised learning for provenance-similarity of binaries,” Proceedings of the 17th ACM SIGKDD, 15–23, 2011.

[DAV10] I. J. Davis and M. W. Godfrey, “From whence it came: Detecting source code clones by analyzing assembler,” Working Conference on Reverse Engineering (WCRE), 242-246, 2010.

[DEW09] M. De Wit, A. Zaidman, and A. Van Deursen, “Managing code clones using dynamic change tracking and resolution,” International Conference on Software Maintenance, 169-178, 2009.

[DEE05] S. Deelstra, M. Sinnema, and J. Bosch, “Product derivation in software product families: a case study,” Journal of Systems and Software, 74(2):173-194, 2005.

[FRA05] W. B. Frakes and K. Kang, “Software reuse research: status and future,” IEEE Transactions on Software Engineering, 31(7): 529 -536, 2005.

[GAL09] R. E. Gallardo-Valencia and S. Elliott Sim, “Internet-scale code search,” ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE '09), 49-52, 2009.

[GRI81] S. Grier, “A tool that detects plagiarism in Pascal programs,” SIGCSE Technical Symposium on Computer Science Education (SIGCSE ‘81), 13(1):15-20, 1981.

[GLP13] S. Gulwani, Program Synthesis website, http://research.microsoft.com/en-us/um/people/sumitg/pubs/synthesis.html, visited Jan. 2013.

[GUH10] R. Guha, R. McCool, and E. Miller, “Semantic search,” International Conference on World Wide Web, 700-709, 2003.

[HEM11] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, “Finding software license violations through binary code clone detection,” Working Conference on Mining Software Repositories, 63-72, 2011.

[HIL04] R. Hill and J. Rideout, “Automatic method completion,” International Conference on Automated Software Engineering (ASE), 228-235, 2004.

[HOU10] D. Hou and D. M. Pletcher, “Towards a better code completion system by API grouping, filtering, and popularity-based ranking,” International Workshop on Recommendation Systems for Software Engineering (RSSE ‘10), 26-30, 2010.

[HOI08] R. Holmes, “Pragmatic software reuse,” Doctoral Dissertation, University of Calgary, 2008.

[HOL05] R. Holmes, R. J. Walker, and G. C. Murphy, “Strathcona example recommendation tool,” European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13), 237-240, 2005.

[HOL09] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger, “The end-to-end use of source code examples: An exploratory study,” International Conference on Software Maintenance (ICSM), 555-558, 2009.

[HUM10] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, “Index-based code clone detection: incremental, distributed, scalable,” International Conference on Software Maintenance (ICSM), 1-9, 2010.

[HUN77] J. W. Hunt and T. G. Szymanski, “A fast algorithm for computing longest common subsequences,” Communications of the ACM, 20(5), 350-353, 1977.

[JAC84] J. Jacobsen, “An automated management system for applications software,” ACM SIGUCCS Conference on User services (SIGUCCS '84), 173-175, 1984.

[JAC01] P. Jaccard, “Étude comparative de la distribution florale dans une portion des Alpes et des Jura,” Bulletin de la Societe Vaudoise des Sciences Naturelles, 37:547-579, 1901.

[JIA07] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” International Conference on Software Engineering (ICSE '07), 96-105, 2007.

[JON92] R. E. Johnson, “Documenting frameworks using patterns,” Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '92), 63-76, 1992.

[JUR11] V. Juričić, “Detecting source code similarity using low-level languages,” International Conference on Information Technology Interfaces (ITI), 597-602, 2011.

[KAP06] C. Kapser and M. W. Godfrey, “”Cloning considered harmful” considered harmful,” Working Conference on Reverse Engineering (WCRE), 19-28, 2006.

[KAM02] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” IEEE Transactions on Software Engineering, 28(7): 654-670, 2002.

[KAW09] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, “SHINOBI: A tool for automatic code clone detection in the IDE,” Working Conference on Reverse Engineering (WCRE), 313-314, 2009.

[KEL83] M. I. Kellner, "Ten years of software maintenance: progress or promises?," Conference on Software Maintenance (ICSM), 406-408 ,1993.

[KIM10] J. Kim, S. Lee, S. W. Hwang, and S. Kim, "Towards an intelligent code search engine," AAAI Conference on Artificial Intelligence, 1358- 1363, 2010.

[KLX11] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," Working Conference on Reverse Engineering (WCRE), 23-27, 2011.

[KLZ11] I. Keivanloo, J. Rilling, and P. Charland, "SeClone-a hybrid approach to internet-scale real-time code clone search," International Conference on Program Comprehension (ICPC), 223-224, 2011.

[KLX12] I. Keivanloo, "Leveraging clone detection for Internet-scale source code search," International Conference on Program Comprehension (ICPC), 277-280, 2012.

[KLZ12] I. Keivanloo, C. Forbes, and J. Rilling, "Similarity search plug-in: Clone detection meets internet-scale code search," 4th ICSE Workshop on Search-Driven Development: Users, Infrastructure, Tools and Evaluation (SUITE), 21-22, 2012.

[KLL10] I. Keivanloo, L. Roostapour, P. Schügerl, and J. Rilling, "SE-CodeSearch: A scalable Semantic Web-based source code search infrastructure," International Conference on Software Maintenance (ICSM), 1-5, 2010.

[KLT12] I. Keivanloo, C. K. Roy, J. Rilling, and P. Charland, "Shuffling and randomization for scalable source code clone detection," International Workshop on Software Clones (IWSC), 82-83, 2012.

[KLQ12] I. Keivanloo, C. K. Roy, and J. Rilling, "Java bytecode clone detection via relaxation on code fingerprint and Semantic Web reasoning," International Workshop on Software Clones (IWSC), 36-42, 2012.

[KLF12] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, and J. Rilling, "A Linked Data platform for mining software repositories," Working Conference of Mining Software Repositories (MSR), 32-35, 2012.

[KLE13] I. Keivanloo, C. K. Roy, and J. Rilling, "SeByte: A scalable clone search model for bytecode," Journal Science of Computer Programming, *submitted (second revision)*, 2013.

[KLP12] I. Keivanloo, C. K. Roy, and J. Rilling, "SeByte: A semantic clone detection tool for intermediate languages," International Conference on Program Comprehension (ICPC), 247-249, 2012.

[KON97] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," Working Conference on Reverse Engineering, 44-54, 1997.

[KOS12] R. Koschke, "Large-scale inter-system clone detection using suffix trees," European Conference on Software Maintenance and Reengineering (CSMR), 309-318, 2012.

[KRA08] N. Kraft, B. Bonds, and R. Smith, "Cross-language clone detection," International Conference on Software Engineering and Knowledge Engineering (SEKE), 2008.

[KRU92] C. W. Krueger, "Software reuse," ACM Journal Computing Surveys (CSUR) 24(2): 131-183, 1992.

[KZH10] H. D. Kim, C. Zhai, and J. Han, "Aggregation of multiple judgments for evaluating ordered lists," European Conference on Advances in Information Retrieval (ECIR), 5993:166-178, 2010.

[KNU77] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," SIAM Journal on Computing, 6(2): 323-350, 1977.

[KOS08] R. Koschke, "Frontiers of software clone management," Frontiers of Software Maintenance (FoSM), 119-128, 2008.

[LAP06] M. Lapata "Automatic evaluation of information ordering: Kendall's tau," Journal Computational Linguistics, 32(4):471-484, 2006.

[LAV11] T. Lavoie and E. Merlo, "Automated type-3 clone oracle using Levenshtein metric," International Workshop on Software Clones (IWSC), 34-40, 2011.

[LAV12] T. Lavoie and E. Merlo, "An accurate estimation of the Levenshtein distance using metric trees and Manhattan distance," International Workshop on Software Clones (IWSC), 1-7, 2012.

[LEM11] M. W. Lee, S. W., Hwang, and S. Kim, "Integrating code search into the development session," International Conference on Data Engineering (ICDE), 1336-1339, 2011.

[LER10] M. W. Lee, J. W. Roh, S. W. Hwang, and S. Kim, "Instant code clone search," ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10), 167-176, 2010.

[LOM83] J. V. Lombardi, “Computer literacy: The basic concepts and language,” Indiana University Press, 1983.

[LIN84] M. A. Linton, ”Implementing relational views of programs,” ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE), 19(5):132-140, 1984.

[LIZ05] Z. Li and Y. Zhou, “PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code,” European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), 30(5):306-315, 2005.

[MAN05] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, “Jungloid mining: helping to navigate the API jungle,” ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ‘05), 40(6):48-61, 2005.

[MAN08] C. D. Manning, P. Raghavan, and H. Schütze, “Introduction to information retrieval,” Cambridge University Press. 2008.

[MAR01] A. Marcus and J. I. Maletic, “Identification of high-level concept clones in source code,” International Conference on Automated Software Engineering (ASE), 107-114, 2001.

[MAR09] M. R. Marri, S. Thummalapenta, and T. Xie, “Improving software quality via code searching and mining,” ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE '09), 33-36, 2009.

[MAY96] J. Mayrand, C. Leblanc, and E. M. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” International Conference on Software Maintenance, 244-253, 1996.

[MCM12] C. McMillan, M. Grechanik, and D. Poshyvanyk, “Detecting similar software applications,” International Conference on Software Engineering (ICSE), 364-374, 2012.

[MEN13] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, “A machine learning framework for programming by example,” International Conference on Machine Learning (ICML), 2013.

[MIC99] A. Michail, “Data mining library reuse patterns in user-selected applications,” International Conference on Automated Software Engineering, 24-33, 1999.

[MIC00] A. Michail, “Data mining library reuse patterns using generalized association rules,” International Conference on Software Engineering (ICSE), 167-176, 2000.

[MIS04] G. Mishne and M. De Rijke, “Source code retrieval using conceptual similarity,” Conference on Computer Assisted Information Retrieval (RIAO), 539-554, 2004.

[MIS12] A. Mishne, S. Shoham, and E. Yahav, “Typestate-based semantic code search over partial programs,” International Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA), 997-1016, 2012.

[NAS12] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, “What makes a good code example?: A study of programming Q&A in StackOverflow,” International Conference on Software Maintenance (ICSM), 25-34, 2012.

[NYK02] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon, “What programmers really want: results of a needs assessment for SDK documentation,” International Conference on Computer Documentation (SIGDOC), 133-141, 2002.

[OTT76] K. J. Ottenstein, “An algorithmic approach to the detection and prevention of plagiarism,” ACM SIGCSE Bulletin, 8(4): 30-41, 1976.

[PAU94] S. Paul and A. Prakash, “A framework for source code search using program patterns,” IEEE Transactions on Software Engineering, 20(6): 463-475, 1994.

[PER12] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, “Type-directed completion of partial expressions,” ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’12), 275-286, 2012.

[POS07] D. Poshyvanyk and A. Marcus, “Combining formal concept analysis with information retrieval for concept location in source code,” International Conference on Program Comprehension (ICPC’07), 37-48, 2007.

[PER88] J. M. Perry, “Perspective on Software Reuse,” Technical Report, No. CMU/SEI-88-TR-22, Carnegie-Mellon Univ. Pittsburgh PA Software Engineering Inst., 1988.

[QUI67] M. R. Quillan, “Word concepts: A theory and simulation of some basic capabilities,” Behavioral Science, 12(5): 410-430, 1967.

[ROB08] R. Robbes and M. Lanza, “How program history can improve code completion,” International Conference on Automated Software Engineering (ASE), 317-326, 2008.

[ROB11] M. P. Robillard, “What makes APIs hard to learn? answers from developers,” IEEE Software, 26(6): 27-34, 2009.

[ROY09] C. K. Roy and J. R. Cordy, “A mutation/injection-based automatic framework for evaluating code clone detection tools,” International Conference on Software Testing, Verification and Validation Workshops (ICSTW'09), 157-166, 2009.

[ROS09] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Science of Computer Programming, 74(7): 470-495, 2009.

[ROS08] C. K. Roy and J. R. Cordy, “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” International Conference on Program Comprehension, 172-181, 2008.

[ROS96] M. B. Rosson and J. M. Carroll, ”The reuse of uses in Smalltalk programming,” ACM Transactions on Computer-Human Interaction (TOCHI), 3(3): 219-253, 1996.

[ROT03] M. A. Rothenberger, K. J. Dooley, U. R. Kulkarni, and N. Nada, “Strategies for software reuse: A principal component,” IEEE Transactions on Software Engineering, 29(9):825-837, 2003.

[RJC08] C. K. Roy and J. R. Cordy, “Towards a mutation-based automatic framework for evaluating code clone detection tools,” Canadian Conference on Computer Science and Software Engineering (C3S2E), 137-140, 2008.

[SAB09] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, “Detecting code clones in binary executables,” International Symposium on Software Testing and Analysis (ISSTA’09), 117-127, 2009.

[SAN11] A. Santone, “Clone detection through process algebras and Java bytecode,” International Workshop on Software Clones (IWSC), 73-74, 2011.

[SAN94] S. Paul, and A. Prakash, “A framework for source code search using program patterns,” IEEE Transactions on Software Engineering, 20(6): 463-475, 1994.

[SEL10] G. M. Selim, K. C. Foo, and Y. Zou, “Enhancing source-based clone detection using intermediate representation,” Working Conference on Reverse Engineering (WCRE), 227-236, 2010.

[SCH12] N. Schwarz, “Hot clones: a shotgun marriage of search-driven development and clone management,” Conference on Software Maintenance and Reengineering (CSMR), 513-515, 2012.

[SIN98] J. Singer, “Practices of software maintenance,” International Conference on Software Maintenance, 139-145, 1998.

[SMI09] R. Smith and S. Horwitz, “Detecting and measuring similarity in code clones,” International Workshop on Software Clones (IWSC), 2009.

[SOO12] Soot Framework, http://www.sable.mcgill.ca/soot/, (Dec 2012).

[SVK13] J. Svajlenko, I. Keivanloo, and C. K. Roy, “Scaling classical clone detection tools for ultra-large datasets: An exploratory study,” International Workshop on Software Clones (IWSC), 2013.

[SVJ13] J. Svajlenko, C. Roy, and J. Cordy, “A mutation analysis based benchmarking framework for clone detectors,” International Workshop on Software Clones (IWSC), 8-9, 2013.

[TAI09] R. Tairas and J. Gray, “An information retrieval process to aid in the analysis of code clones,” Empirical Software Engineering, 14(1):33-56, 2009.

[TAN87] A. S. Tanenbaum, “A UNIX clone with source code for operating systems courses,” ACM SIGOPS Operating Systems Review, 21(1):20-29, 1987.

[THU07] S.Thummalapenta and T. Xie, “Parseweb: A programmer assistant for reusing open source code on the web,” International Conference on Automated Software Engineering (ASE), 2007.

[UCI10] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, “UCI source code data sets,” http://www.ics.uci.edu/~lopes/datasets, 2010.

[UDD11] Md. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, “On the effectiveness of simhash for detecting near-miss clones in large scale software systems,” Working Conference on Reverse Engineering (WCRE), 13-22, 2011.

[UDD13] Md. S. Uddin, C. K. Roy, and K. Schneider, “SimCad : An extensible and faster clone detection tool for large scale software systems,” International Conference on Program Comprehension (ICPC), 236-238, 2013.

[WAH04] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, “Clone detection in source code by frequent itemset techniques,” International Workshop on Source Code Analysis and Manipulation (SCAM), 128-135, 2014.

[WAN13] J. Wang, D. Yingnong, Z. Hongyu, C. Kai, X. Tao, and Z. Dongmei, “Mining succinct and high-coverage API usage patterns from source code,” Working Conference on Mining Software Repositories (MSR), 2013.

[WES12] Wessa, Kendall tau Rank Correlation (v1.0.11) in Free Statistics Software (v1.1.23-r7), Office for Research Development and Education, URL http://www.wessa.net/rwasp_kendall.wasp/, visited 2012.

[WEH03] H. J. Webber, “New horticultural and agricultural terms,” Science, 18(459): 501-503, 1903.

[ZIB12] M. F. Zibran and C. K. Roy, “IDE-based real-time focused search for near-miss clones,” ACM Symposium on Applied Computing (SAC’12), 1235-1242, 2012.

[ZHO09] H. Zhong, T., Xie, L., Zhang, J., Pei, and H. Mei, “MAPO: Mining and recommending API usage patterns,” European Conference on Object-Oriented Programming (ECOOP), 318-343, 2009.

14. Appendix

14.1. Transformation function design issues

Several token types exist in source code such as method names, class names, primitive types, language keywords, variables, and constants. In general, apart from language keywords, which are consistent through the code, the token names can refer to different concepts. Additionally, despite having different names, the semantic of tokens can be similar (from algorithmic point of view). We refer to this case as tokens' semantic stability issue. Figure 57 provides an example where two code fragments are clones with high confidence even though they use different variable names (i.e., att and var).

```
…                                                  …
String msg="exit 0";                               String msg="exit 0";
for(AttributeEntity att : t.getAttributes())       for(AttributeEntity var : t.getAttributes())
{                                                  {
…                                                  …
```

Figure 57. Two code cloned code fragments that are using different variable names

It is a well-known practice (e.g., [KAM02]) in clone detection tools to replace such tokens with placeholders to reduce such syntactic and semantic dissimilarities. This practice is useful when the clone detection approach is not able to judge the *semantics* of the token based on its name and other available information (e.g., AST).

In our research, we proposed various transformation functions in order to be able to address different types of similarity. For example, the *c* function (Table 3) only preserves method names and class names. *c* replaces almost all other tokens with # (the placeholder). We defined 5 transformation functions (Table 3) covering different scenarios and requirements. However, all of them preserve the method name tokens.

For our approach, we decided to preserve method names, as we observed that method names have stable semantics in our research context (i.e., code search). Our observation is based on an analysis of a one-year query log of Koders [UCI10] (one of the state of the art code search engines). When analyzing the query log, we focused on 18 programming languages with method construct as part of their language. This log contains a total of approximately 10 million records that we analyzed. As part of that analysis, we observed that for Internet-scale code search, method names play an essential role. Our analysis showed that if a method name was present as part of the query, code download occurred 98% of the time (Figure 58 – *MCQ values*), whereas the overall download rate is 69% (Figure 58 – *All values*). Note that in Web search activity mining, downloads/clicks on search results are interpreted as the result of a successful search. This observation shows the importance of method names to the code search success rate, which can be used as an indicator for method tokens' semantics stability from end-users' point of view. That is, contrary to the other token types, the need for ignoring method names in order to achieve higher recall is low. Therefore, all encoded code patterns generated by our transformation functions preserve the method names, which also provides the added benefit of reducing the number of false positive rates during the matching.

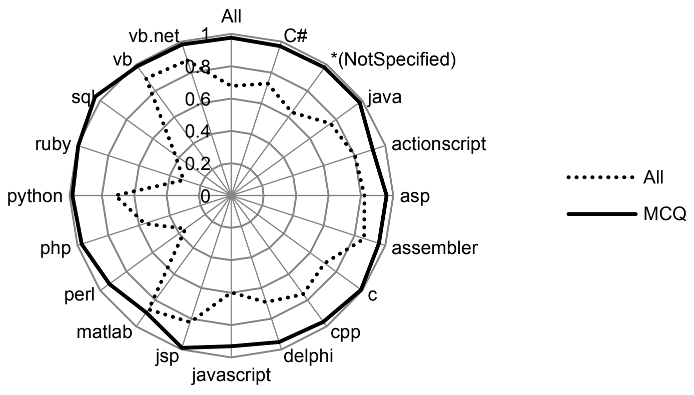


Figure 58. Importance of method names to the code search success rate – an indicator for method tokens' semantics stability from end-users' point of view.