Code Clone Matching: A Practical and Effective Approach to Find Code Snippets

Katsuro Inoue
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

Yuya Miyamoto
Osaka University
Osaka, Japan
yuy-mymt@ist.osaka-u.ac.jp

Daniel M. German
University of Victoria
Victoria, Canada
dmg@uvic.ca

Takashi Ishio
Nara Institute of Science and Technology
Ikoma-City, Japan
ishio@is.naist.jp

arXiv:2003.05615v1 [cs.SE] 12 Mar 2020

ABSTRACT

Finding the same or similar code snippets in source code is one of fundamental activities in software maintenance. Text-based pattern matching tools such as `grep` is frequently used for such purpose, but making proper queries for the expected result is not easy. Code clone detectors could be used but their features and result are generally excessive. In this paper, we propose Code Clone matching (CC matching for short) that employs a combination of token-based clone detection and meta-patterns enhanced with meta-tokens. The user simply gives a query code snippet possibly with a few meta-tokens and then gets the resulting snippets, forming type 1, 2, or 3 code clone pairs between the query and result. By using a code snippet with meta-tokens as the query, the resulting matches are well controlled by the users. CC matching has been implemented as a practical and efficient tool named `ccgrep`, with `grep`-like user interface. The evaluation shows that `ccgrep` is a very effective to find various kinds of code snippets.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

code snippet search, pattern matching, grep, code clone types

1 INTRODUCTION

Finding and locating code snippets in source code files are fundamental activities to understand and maintain software systems[9, 35]. When we identify a bug in a code snippet, we would search the same or similar snippets in the same or other systems to check if they contain the same bug[16, 17, 30]. When we modify a code snippet for feature change or enhancement, we would find the same or similar snippets for preventing unintentional inconsistent bug[26, 44]. When we identify a code snippet with a bad coding practice, we would search the same or similar snippets for possible refactoring[34].

To find the same or similar code snippets effectively, various kinds of software engineering tools or IDE's have been proposed and implemented[9, 42]; however, it has been reported that the character-based pattern matching tool `grep`[11] is still widely used to find code snippets, due to its simplicity, trustworthiness, speed, and availability[21, 38]. Although `grep` is very convenient to find lines with specific keyword, it is not easy to make a proper query for a code snippet that ignores comments and white spaces, and might span multiple lines.

Code clone is a pair of code snippets those are identical or similar each other[6]. A large body of scientific literature on clone detection has been published and various kinds of code clone detection tools (detectors) have been developed[33, 36]. Most of these detectors are designed to detect all of the code clone pairs in the target source files, and thus, the resulting code clone pairs become generally huge[22] and they contain a lot of code clone pairs that might not be of interest (including false positives). There are a few tools that find similar code snippet for a query code snippet, but their performance and usability are limited[15, 24, 32].

In this paper we propose a method, which we call *Code Clone matching* (CC matching), to find clones of specific code snippets by using a combination of clone detection and pattern matching. Search queries can be simply code snippets, or code snippets enhanced with meta-patterns (with meta-tokens having leading $) that can provide flexibility to narrow or broaden the search query. An example of a query that uses meta-patterns would be searching for for-loops in which the control variable of the for-loop is the variable `index` and which contains a `return` statement inside the body of this for-loop, represented like

```
for($index=$$){$$ return $$}
```

As a consequence, CC matching is programming language aware, and able to properly tokenize the source code (ignoring whitespace and comments). In clone-detection terms, CC matching retrieves, given a code snippet (that potentially includes some meta-patterns), the clones (type 1, 2 and 3) of this snippet that also satisfy the meta-patterns (e.g. that use a specific variable name in some specific locations).

We will also present *ccgrep*, that is current implementation of CC matching for C, C++, Java, and Python. `ccgrep` works as a handy but reliable pattern matching tool with a grep-like and easy-to-understand user interface. An evaluation of `ccgrep` shows that it is capable of representing the queries for all of type 1, 2, and a part of type 3 clones. In addition, `ccgrep` accurately performs CC matching, and the speed is slower than `grep` but faster than other similar code finders.

As far as we know, little has been studied on the pattern matching based on the notion of code clone. The contributions of this paper are twofold:

- We propose CC matching, a new concept of matching code snippets based on token-based code clone detection and enhanced pattern matching.
- We present a practical, efficient, and easy-to-use tool called `ccgrep` that implements CC matching, with its evaluation.

2 BACKGROUND

2.1 Motivating Example

Some uses of the ternary operator (e.g., `exp1 ? exp2 : exp3` meaning the result of this entire expression is *exp2* if *exp1* is true, otherwise the result is *exp3*—available in C, C++ and Java) are, arguably, considered bad practice[41]. For example, the use of $a < b ~?~ a : b$ is arguably harder to read than using *min(a,b)*. Therefore, it might be desirable to replace the ?: operator with a function or macro that returns the minimum value. The following is an example found in the file `drivers/usb/misc/adutux.c` in the Linux kernel (v5.2.0).

```
amount = bytes_to_read < data_in_secondary ?
                  bytes_to_read : data_in_secondary;
```

This line of code should be replaced with a more readable expression (note that the macro `min` in Linux guarantees no side effects):

```
amount = min(bytes_to_read, data_in_secondary);
```

Finding all occurrences of such usage of the ternary operator using `grep` is not easy. For example, simply executing "`grep '<'`" for all 598 files (total 51,6394 lines in C) under /drivers/usb produces 16335 matching, including many unwilling patterns such as "`if (a<b)`", "`for (i=0; i<x; ...)`", or "`#include <linux/...>`". We could narrow the matches by concatenating `grep` like,

```
grep '<' -r . | grep '?' | grep ':'
```

However, it still produces 149 matches. Perhaps more problematic is that the expressions could span multiple lines. While it is possible to create a complex regular expression to find these expressions, it would be time consuming and potentially error prone.

Another alternative would be to use a clone detector to detect uses of the ?: operator. The clone detector NiCad[7] detects 646 block-level clone classes for the `drivers/usb` files by the default setting, but no snippet with the ternary operator case is included in the result because it is too small to be detectable.

Ideally we would like to be able to specify simple and easy-to-create-and-understand query to find these types of snippets. Therefore in this paper we propose CC matching, which is based on the notion of code clone detection. Using CC matching, the query is written simply as:

```
a < b ? a : b
```

In a nutshell, this query specifies that a variable (represented by a) should be followed by < and then a second variable (represented by b), followed by a `?`, followed by the same first variable found, followed by `:`, followed by the second variable. Also, whitespace and comments should be ignored. This query would match `x<y?x:y` but it would not match `x<y?x:z`.

Using this query, we used our implementation of CC matching (which we call `ccgrep`) to find the occurrences of this type Linux's `drivers/usb`. We identified 3 instances, and submitted patches to replace them with `min`. Two of those patches have been accepted already into Linux.

2.2 Pattern Matching with grep

`grep` takes a query pattern in a form of a regular expression (or an extended regular expression), and reports the matched lines in the target files. Developers specify a keyword, or a short idiom as the query, and get the result output composed of the file paths and matched lines. `grep` is generally fast and effective, but sometimes a simple query pattern generates a large output which is hard to analyze further. A complex query pattern might reduce the size of the output, but making a proper query is not an easy task even for an expert of the tool.

Also, `grep` is designed to work with any text file, and it is not specifically designed to explore source code files. Unless a complex query is written, it reports matches in comments (which users might want to have ignored) and it is difficult to find matches that spawn multiple lines. Furthermore, because grep is based on finite state machines, it is not capable of dealing with matching parenthesis, brackets or braces (e.g. one might want to match the entire block of code inside a for-loop, including other blocks inside it). There have been various proposals for extending `grep` to do code matching[1, 5, 8]; however, none of these have been successful as `grep`.

2.3 Code Clone Detection and Search

A code clone is broadly defined as a code snippet having the same or similar code snippets in the target software collection[3, 20, 36], and a code clone pair is classified into 4 types, type 1 (syntactically the same snippets except for comment or whitespace differences), type 2 (type 1 with identifier or literal differences), type 3 (type 2 with addition, deletion, or change of statements within the clone), and type 4 (semantically equivalent snippets)[36].

Code clone detector is a promising method for finding code snippets with unique characteristics. A user would want to simply provide a snippet and then find all the clones of this specific snippet. However, most of those clone detectors report all of code clone pairs in the target files; these results are generally huge and mostly irrelevant for finding specific code snippets. Further, most clone detector tend to ignore small-size clones[28] and so they would miss small code snippet for search as shown in the motivating example.

Some code clone detector, such as `ccfinderX` [19], have an option to find clones between a specified file and all other files; however, we still need to prepare a query file and also need to tune various parameters to the specifics of the query (such as its length), which heavily affect the detection result[36].

There are several tools dealing with code search for code clone pairs, such as CBCD[24], NCDSearch[15], and Siamese[32]. CBCD is a PDG (Program Dependency Graph)-based code matching tool by graph isomorphism testing. NCDSearch is a block-based code search tool using normalized compression distance of two code snippets. Siamese is token-based code retrieval tool with inverted index. These tools would show good accuracy as code clone finder for certain conditions, but they would still have issues on performance or usability as daily-used practical software engineering tool.

Another issue is usability of the tools. Most of those tools are stand-alone in the sense that they take their specific input format and generate proprietary output. Integration with other tools needs

to transform their input/output formats, so it becomes hard to collaborate with others.

3 CC MATCHING

3.1 Design Policy

Our design policies of CC matching are as follows.

- CC matching is a process of finding code snippets in the target source code for a query snippet. The query and matched result can be seen as a code clone pair of either type1, 2, or 3. The query is a code snippet or a code snippet enhanced with meta-patterns that can widen or narrow the potential matches.
- The matching is made at a granularity of a sequence of language-dependent tokens (white space and comments are removed) and not as simply sequences of characters. .
- CC matching can also precisely control how tokens in the query are matched. Using command line options, the query pattern can match type 1, type 2 (P-match), type 2 (non-P-match) and a part of type 3 clone snippets effectively.

3.2 Formulating CC Matching

The input of CC matching is the query q, the target T of source code files in a programming language L, and matching option o. The output is a matched code snippet t in T[1]. We refer to reserved words, delimiters (operators, brackets, ; ...), identifiers, and literals in L as *regular tokens*. Other tokens starting with meta symbol $ are called *meta-tokens*. q is a sequence of regular tokens and the meta-tokes, and each matched result t is a sequence of the regular tokens. These token sequences do not contain comments, white spaces, or line breaks. We always consider the matching on the token sequence level, not on the character level.

In Tab.1, we define a token-level matching for various kinds of tokens in CC matching, with simple examples. The basic ideas of these matches are as follows.

- A language-defined token such as reserved words or delimiters matches the exact token.
- A user-defined token such as identifier or literal can match same kind of token with a possibly different name or value. To pin down them to a specific identifier name or literal value, $ is used before the token. For example, `$count` would match only the token `count`.
- Wildcard tokens $., $#, and $$ are introduced for the matches to any single token, any token sequence, or any token sequence discarding paired brackets, respectively.
- Popular regular expression operators for choice, repetition, and grouping are introduced to enhance the expressiveness.

Consider that query q is a token sequence $q_1, ..., q_m$ $(1 \leq m)$, and a target t is a token sequence $t_1, ..., t_n$ $(0 \leq n)$. From q_1 to q_m, if each token in the query matches tokens in the target from t_1 to t_n as defined in Tab.1 without overlapping or orphan tokens, then we say q matches t by CC matching.

[1] Note that in an actual implementation of CC matching, the location of all the matched code snippets in T will be the output, but for simplicity of the explanation here, we take one of the matched snippets itself t as the output.

3.3 Type 1 Matching

The most simple type of query in CC matching—named *type 1 matching*—is the case that $m = n$ and $q_i = \$t_i$ for each identifier or literal and $q_i = t_i$ for other tokens. Type 1 matching is performed to find the exact code snippet, discarding comments, white spaces, or line breaks. This type of query does not use wildcard nor regular expression tokens. These are examples of queries and targets:

q1: `$a = $0; $b = $10;`
t1: `a = 0; b = 10;`

q1 matches t1 as type 1 matching. Note that annotating all identifiers and constants with $ would be sometimes bothersome; for this reason our implementation has an option for automatic annotation of these tokens to find any type-1 clone of the query.

The next example is a case of "no match".

q2: `$a = $0; $b = $10;`
t2: `a = 0; b = 20;` (no match)

Literal `10` does not match `20`, thus overall q2 does not match t2.

3.4 Type 2 Matching, P-Matching, and Pinning Down

For the query token sequence $q_1, ..., q_m$ and the target token sequence $t_1, ..., t_n$, if $n = m$ and $norm(q_i) = norm(t_i)$ for each i, then q matches t as *type 2 matching*. Here *norm* is a normalization function to flat the distinction of identifiers (or literals), defined below.

$$norm(x) \equiv \begin{cases} \#id & \text{if } x \text{ is an identifier} \\ \#li & \text{if } x \text{ is an literal} \\ x & \text{otherwise} \end{cases}$$

In type 2 matching, an identifier in the query can match any identifier in the target, and also a literal in the query can match any literal in the target.

q3: `a = 0; b = 10;`
t3: `x = 10; y = 200;`

q3 matches t3, because the sequences of the normalized tokens are both $[\#id, =, \#li, ;, \#id, =, \#li, ;]$.

A special case of type 2 matching, with a constraint such that for any identifier or literal q_i if $q_i = q_j$, then $t_i = t_j$, is called *Parameterized match* or *P-matching*. This is sometimes referred to consistent or aligned matching, meaning the same identifiers (or literals) in the query are mapped into the same ones in the target. P-matching is formally defined with a specialized normalization function $norm_p()$, as follows.

$$norm_p(x) \equiv \begin{cases} \#id_{pos(x)} & \text{if } x \text{ is an identifier} \\ \#li_{pos(x)} & \text{if } x \text{ is a literal} \\ x & \text{otherwise} \end{cases}$$

Here, $pos(x)$ is a function returning position i such that identifier (or literal) x is the i-th identifier (literal) newly appeared in the token sequence[2].

q4: `a = 0; a = a + b;`
t4: `y = 0; y = y + c;`

For q4, $pos(a) = 1$ and $pos(b) = 2$, and for t4, $pos(y) = 1$ and $pos(c) = 2$. q4 matches t4 as P-matching, because the P-normalized sequences

[2] Note that any meta-token starting with $ in the query and their matched tokens in the target are out of consideration of $pos()$.

Table 1: Token-Level Matching

token(s) in query	matched token(s) in target	simple example of match	
		query	target
reserved word†	exact reserved word	`while`	`while`
delimiter	exact delimiter	`(`	`(`
identifier	any identifier‡	`myname`	`abc`
literal	any literal‡	`1`	`100`
$identifier	exact identifier	`$myname`	`myname`
$literal	exact literal	`$1`	`1`
$.	any single token	`$.`	`if`
$# X	any shortest token sequence ending with X	`$# +`	`while(f(a+`
$$ X	any shortest token sequence ending with X, excluding X inside well-balanced bracket {...}, [...], or (...)	`$$ +`	`while(f(a+1))+`
X $| Y	either X or Y	`+ $| -`	`-`
X $*	repeated sequence of X zero or more times	`( $*`	`(((`
X $+	repeated sequence of X one or more times	`( $+`	`((`
X $?	X or none	`( $?`	`(`
$(X1 X2 ... $)	X1, X2, ... (group for further regular expression operations)	`$( a++ $| ++a $)`	`a++`

†Type names are treated as identifiers.
‡Identifier and literal may match only the exact one by an option.
Tokens starting with $ are meta-tokens and others are regular tokens.
Wildcard meta-tokens $# and $$ match in reluctant way, and $*, $+, and $? match in possessive way[14].
X, Y, X1, X2, ... are any regular token or a group with $(... $).

are both $[\#id_1, =, \#li_1, ;, \#id_1, =, \#id_1, +, \#id_2, ;]$. The following case is type 2 matching but not P-matching.

q5: `a = 0; a = a + b;`
t5: `y = 0; y = z + c;` (type 2 matching but not P-matching)

At t5, `z` cannot be matched by `a` because $norm_p(a) = \#id_1$ is not equal to $norm_p(z) = \#id_2$. As a default of CC matching, P-matching is assumed[3].

Sometime type 2 matching, even P-matching might match many targets, and we would want to narrow them by pinning down an identifier (or literal) to a specific one. To do this, we also use the annotation of $ at the beginning of the identifier (literal) in the query, so that the identifier (literal) headed by $ is not normalized and it matches only the exact token in the target.

q6: `$cat = 1`
t6-1: `cat = 1` (match)
t6-2: `dog = 1` (no match)

q6 matches t6-1, but it does not match t6-2 since identifier `cat` is not normalized and is pin down to `cat`.

We can mix normalized identifiers (literals) and non-normalized ones in the query as follows.

q7: `$cat = cat+1`
t7: `cat = dog+1`

q7 matches t7, because `$cat` in q7 and its corresponding target `cat` in t7 are excluded from consideration of *pos*(), and so both `cat` in q7 and `dog` in t7 are treated as the first appeared identifiers and both are normalized to $\#id_1$.

[3]It can be changed by the tool's option

3.5 Wildcard Tokens and Type 3 Matching

There are three wildcard tokens, $. (any token, but only one), $# (any token sequence), and $$ (any token sequence with bracket discarding). $. matches any single token in the target, and $# X matches any shortest token sequence ending with X. $$ X is similar to $# X but it does not end at X inside balanced brackets {...}, [...], and (...). In this case, the match continues until the first X outside the balanced brackets; thus X's inside the brackets are not seen.

For the query token sequence $q_1, ..., q_m$ and the target token sequence $t_1, ..., t_n$, if a wildcard token q_i in the query matches some target tokens $t_j, ..., t_{j+k-1}$, and other query tokens match properly the target tokens (as defined in Tab.1), preserving the order of the query and the matched target tokens without any overlapping or orphan tokens, we call it *type 3 matching*. Here, $k \geq 0$, meaning that the matched token sequence may be none($k = 0$), a single token($k = 1$), or more($k \geq 2$). Here we present several examples of wildcard tokens.

q8: `$a = $. ;`
t8-1: `a = b ;`
t8-2: `a = 10 ;`

In this case, query q8 matches both t8-1 and t8-2, by the replacement of $. with `b` and `10`, respectively.

q9: `$a $# ;`
t9-1: `a = b+c ;`
t9-2: `a++ ;`

Query q9 matches `a` and any tokens before '`;`', and $# matched '= b+c' in t9-1 and '++' in t9-2.

q10: `$a = $$ $b`
t10-1: `a = b`

t10-2: `a = 10+c+b`
t10-3: `a = f(b,10)+b`

t10-1 is the case that the wildcard $$ matches none, t10-2 is '`10+c+`', and t10-3 is '`f(b,10)+`' where the first `b` is inside the bracket (...) and it is not matched by `$b`. The next is a more complex example.

q11: `a= f(p); if($$){a=-a;}`
t11-1: `b= g(q); if(c<=0){b=-b;}`
t11-2: `b= g(q); if(h(c)==0){b=-b;}`

In this query, the conditional expression of `if` statement can be any token sequence followed by the closing bracket `)`. In t11-1, $$ matches `b<=0`, and in t11-2 it matches `h(b)==0` where the first closing bracket is balanced with the open bracket so that the matching for $$ continues after `h(b)` until another closing bracket appears.

With these wildcard tokens, we can specify possible changes of the target token sequence, i.e., variants of the target code snippet.

3.6 Regular Expression Extensions

To effectively represent queries for largely different variants, CC matching employs regular expressions in its query with meta symbols |, *, +, ?, (, and) preceded by $.

Selection: $p1$ $| $p2$ means a matching by either patterns $p1$ or $p2$.
Iteration: $p$$*, $p$$+, and $p$$? are the repetition of p zero or more times, one or more times, and zero or one time, respectively.
Grouping: $(p $) defines the scope and precedence of meta symbols in pattern. For example, $($p1$ | $p2$ $)$+ means any repeated pattern of $p1$ or $p2$.

For example, we can find nested if-else clauses as follows.

q12: `$( if $$ else $) $+`
t12: `if(i<10) {a=0;} else if(i<20) {a=5;} else`

In this case, conditional expressions and then-clauses are matched by $$, and if-else are sought until else is not followed by if.

3.7 Finding Various Code Patterns

Combining the regular tokens and meta-tokens in the query, we can find various kinds of code snippets in the target, from simple to complex code patterns. The following are examples in Java.

- Method XYZ with no parameter
q13: `$XYZ( )`

- Method XYZ with 0 or more parameters
q14: `$XYZ($$)`

- Method $print$ with variable buf as the 1st parameter
q15: `$print($buf, $$)`

- Any method definition
q16: `T f($$){$$}`
Note that type names are treated like identifiers and T matches any type name.

- Getter method
q17: `T f(){return this.v;}`

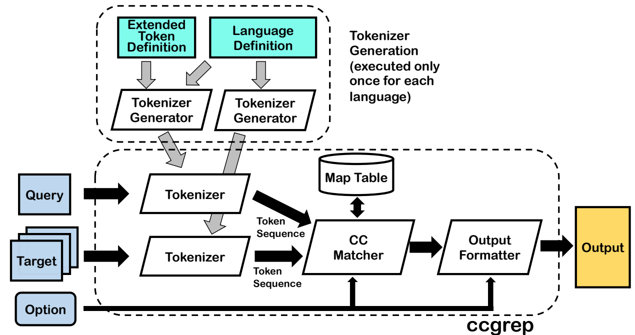

Figure 1: Architecture of ccgrep

- Setter method
q18: `T1 f(T2 v1){this.v1=v2;}`

- if statement
q19: `if ($$){$$}`

- for statement using control variable
q20: `for(T i=0; i<$$; i++){$$}`

These queries can be narrowed by restricting to specific identifiers or constants, by using $id, or $0 to match `id` or `0` exactly.

4 IMPLEMENTATION OF CC MATCHING

We have implemented our proposed CC matching approach for finding code snippets in a tool named `ccgrep`[4]. The target languages of `ccgrep` at this moment are C, C++, Java, and Python3. We have chosen a `grep`-like input/output interface to facilitate its adoption.

4.1 Architecture of ccgrep

The architecture of the process of `ccgrep` is presented in Fig.1. Each component is described below.

Tokenizer Generators: Parser generator ANTLR[5] is used to generate two kinds of tokenizers. For the target tokenization, only the language definition is used to recognize the regular tokens, but for the query tokenization, the definition of the meta symbol extension for the meta-tokens explained in Sec.3.2 and that of regular tokens are used. This process has been executed only once for each target language.
Tokenizers: Each tokenizer removes white spaces and comments from the input text, and decomposes the code into tokens. The query tokenizer accepts the meta-tokens starting with $ and the regular tokens defined by the language, but the target tokenizer accepts only the regular tokens. The tokenizer for the target files are executed in parallel for each file, along with following CC Matcher.
CC Matcher: This performs a naive sequential pattern matching algorithm between two token sequences for the query and the target[13]. For the case of the selection pattern in the query, simply each case is tested one by one. For type 2 code

[4]https://github.com/yuy-m/CCGrep
[5]https://www.antlr.org/

```
$ccgrep 'catch($IOException $$){$$ $toolError($$);}' -r .
./parse/TokenVocabParser.java: catch (IOException ioe) {
./Tool.java:                catch (IOException ioe) {
./Tool.java: catch (IOException ioe) {
./Tool.java:                catch (IOException ioe) {
./Tool.java: catch (IOException ioe) {
./codegen/CodeGenerator.java: catch (IOException ioe) {
./codegen/target/SwiftTarget.java:      catch (IOException ioe) {
$
```

The target is ANTLR V.4, `~antlr4/tool/src/org/antlr/v4/`.

Figure 2: An Example of `ccgrep` Output

clone matching, we record $norm_p()$ values for each identifier and literal in Map Table so that we can check if correspondence of identifiers in the query and target is consistent or not. Note that the table contents are flushed for each query. The option controls the normalization level, input language, output form, and many others discussed in the next section.

Output Formatter: This process constructs the output for the successful matching result. Based on the input option, we can view the match result, like `grep`, in the form of the file name associated with the matched top line as the default, or as many other styles such as full matched lines, only the number of lines, or so on. Fig.2 is an example output of `ccgrep` where `catch` statement followed by identifier `IOException` and a specific call `toolError(...)` are sought in java files of ANTLR Ver.4, and the top lines of the matched results with their file paths are listed.

`ccgrep` is written in Java associated with the ANTLR output, and it is easily installed and executed in various environments such as Unix and Windows (a single JAR is provided that contains all the required libraries needed to run it).

4.2 Input/Output and Options of ccgrep

`ccgrep` has a character-based user interface, and it takes similar options and standard input/output treatment to `grep` so that developers familiar with `grep` can easily use `ccgrep`. The options are as follows.

Target Language: Currently, Java, C, C++, and Python3 are the target languages we have implemented. The default target language is Java.

Target Files: The target files are specified by the command line or they are sought recursively from the specified directory. By connecting `ccgrep` with pipe '|', the target becomes the result of the previous command, and therefore we can combine `ccgrep` with other shell commands or `ccgrep` repeatedly effectively.

Query Pattern: The default setting requires the query pattern at the command line. This can be changed to a specified file or the standard input.

Normalization (Blind) Level: We can choose the normalization level (blind level) of user defined identifiers and literals, such as none for non normalization of type 1 clone detection, P-matching for type 2 with consistent name change, non-P-matching for all type 2 clone. The default is P-matching.

Output (Print-Out) Form: Various kinds of output forms can be chosen. The default is a familiar form of `grep`, and this can be changed to different styles. The print out is usually made to the character-based standard output, but it can be changed to a JSON or XML format file.

5 EVALUATION

Goal of the evaluation is to show that our proposed approach, CC matching and its implementation `ccgrep`, can find various kinds of intended code snippets effectively and efficiently, compared to other approaches. This goal could be decomposed into following three research questions.

RQ1:Query Expressiveness Are various kinds of queries expressed by CC matching (and also its implementation `ccgrep`)? Also, Are the queries written more easily than `grep` ?

RQ2:Accuracy of ccgrep Is `ccgrep` accurately find various types of code clones already detected by other approaches?.

RQ3:Performance of ccgrep What it the execution time of `ccgrep`? Is the token-based naive sequential pattern matching approach fast enough in practice, compared to other tools such as `grep` or code-clone search tool NCDSearch?

5.1 RQ1: Query Expressiveness and Effectiveness

RQ1 explores how easily we can make the queries by our approach. Here we discuss the way of creating queries of each matching type. Since expressiveness of CC matching is equivalent of that of `ccgrep` we only mention `ccgrep` here.

5.1.1 Various Queries Classified with Matching Types. As shown in previous sections, it is obvious that our approach can easily create various query patterns for type 1 matching, type 2 matching with P-match, and type 2 matching with non-P-match, by specifying a code snippet associated with appropriate options. In addition, we can specify the name of an identifier or literal, if we place $ before the name.

A type 3 code clone is one with a few statement addition, or deletion, or change for a seed snippet. Thus the query for type 3 matching could be made from the seed by adding meta-tokens such as $., $$, or $*, deleting some regular tokens in the seed, or modifying some regular tokens with $., $$, or other meta-tokens, if we could specify how to modify the seed. We call such a type 3 matching *specified type 3 matching*, here.

On the other hand, we might want to make a broad query that matches all the code snippets of type 3 code clone for the seed with similarity higher than a threshold. We call it *unspecified type 3 matching*. Currently, CC matching (and `ccgrep`) does not have a feature for the unspecified type 3 matching. We will discuss further on this issue in Sec.6.2.

Type 4 clones, which are semantically the same but syntactically different, are not considered here, because those cannot be found by syntactical pattern matching approach like ours and we consider that they are out of scope in this research.

5.1.2 Comparison with `grep`. **[Type 1 Matching]**

For finding type 1 clones with `ccgrep`, we can place a code snippet

Table 2: Various Queries for kmalloc in usb drivers

	query	#found
1	`kmalloc($$)`	109,101
2	`$kmalloc($$)`	333
3	`$kmalloc(sizeof($$),$$)`	84
4	`$kmalloc(sizeof(struct x),$$)`	29
5	`struct x *p = $kmalloc(sizeof(struct x),$$)`	9
6†	`struct u132_endp *endp = kmalloc(sizeof(struct u132_endp), mem_flags);`	3

†With command option for non normalization (type 1 matching)

as the query with non-normalization option (or adding $ to all identifiers and literals). Note that query strings below are surrounded by a box to clearly separate from other descriptions.

q21(`ccgrep`**):** `int a = b;` with non-normalization option

Instead, `grep` needs to care about white spaces between tokens.

q22(`grep`**):** `\s*int\s+a\s*=\s*b\s*;`

As seen here, q21 is simpler and more straightforward than q22. Furthermore, additional complication should be needed for q22 if we would want to eliminate comments possibly located between tokens.

[Type 2 Matching]

Following are type 2 matching with any `long` variable declaration.

q23(`ccgrep`**):** `$long a;`

q24(`grep`**):** `long\s+[a-zA-Z][a-zA-Z_0-9]*\s*;`

Both are equivalent but `grep` requires a much more complex representation for any possible identifier.

[Specified Type 3 Matching]

q25(`ccgrep`**):** `$time($$)` Find *time* with any parameter

q26(`grep`**):** `time(` Find string '*time*('

In this case, we are searching type 3 clones of a function call `time()`, and `ccgrep` can easily specify it directly. However, in `grep` case, if we simply give `time`, it matches many variable names and comments, so we add '(', a partial string of function call, which might reduce such unwanted matches. We have applied these two queries to linux's tools[6]. `ccgrep` matched 47 `time()` function calls. On the other hand `grep` matched 382 lines that include `strftime()`, `get_time()`, and many other function calls[7]. Therefore, we would say that our approach is straightforward to locate specific identifiers or function calls, and that creating proper queries for `grep` is more comprehensive and difficult, compared to our our CC matching approach with `ccgrep`.

5.1.3 Narrowing Matching Results. An advantage of `ccgrep` is interactive and repeated matching trials such that the users can try various queries immediately after a result is not sufficient. As an example, we show in Tab.2 a process of investigating a system call function `kmalloc()` in Linux usb driver sources /drivers/usb/*.

[6]~ linux/tools, rev. 4.20.0.

[7]If we add -w option (word-based matching) to `grep`, the result is reduced to 7 matches but it misses the cases immediately followed by a word such as `time(NULL)`.

Table 3: Checked Clones in BigCloneBench

Clone Type	Clone Pairs	Found	Not Found
Type 1	48116	48111	5*
Type 2	4234	4232	2*
Total	52350	52343	7*

* indicates faulty clone pairs in BigCloneBench.

In this example, we start at query 1 with the type 2 matching of `kmallock()` with any parameter list denoted by $$. This query matches any function calls, thus it produces a large number of the result(#found). In query 2, we pin down function name to kmalloc, so that the result is reduced to a few hundred. We can further narrow the matches by specifying the first parameter with sizeof at query 3, and with any struct name x at query 4. At query 5, we specify an assignment to a pointer variable with the same struct name x, producing nine matches which are easily checked by hand. We take one of these and give it as query at 6 as it is with non normalization option, resulting in three type 1 clones including the query snippet itself.

As shown by this example, we can interactively and effectively narrow or widen the matching with adding normal or meta-tokens in queries.

5.2 RQ2: Accuracy of ccgrep

For evaluation of query-matching (or information retrieval) systems, recall and precision values, computed by comparing the matched results with the oracles for the queries, are popularly employed[2]. Here in our approach, however, the query to CC matching has no ambiguity and it reports the matching result rigorously as expected and specified by the query with options. In such sense, the result is always the same as the oracle, i.e., the recall and precision are always one. Thus, instead of using recall and precision, here we simply investigate if `ccgrep` works accurately in the sense that code clones already reported by other approaches could be found by our approach.

For this purpose, first we have employed BigCloneBench[43] that is a huge collection of various kinds of code clones. We have extracted all pairs classified as type 1 and type 2 code clones from BigCloneBench, and for each clone pair ($sp1, sp2$), we have checked if $sp2$ is successfully found in the result of `ccgrep` for $sp1$ as query with appropriate options, and vice versa. Tab.3 shows the numbers of type 1 and 2 clones, accurately found and not.

As we can see in Tab.3, most type 1 and 2 clones are found accurately. There were several cases of not-found clones, and we have investigated further those cases and recognized that those cases are faults of the classification of BigCloneBench, some of which should be classified into type 3, and some others are not clones. Thus, we can say that all of proper type 1 and 2 clones in BigCloneBench were perfectly found by `ccgrep`.

For type 3 clones, since BigCloneBench contains a lot of faulty type 3 data, we have instead used CBCD data[24], that contains 11 type 3 clone sets taken from the source code of Git, the Linux kernel, and PostgreSQL. We have crafted type 3 queries from one of code snippets in each clone set as the seed and have checked

ccgrep query

```
$( $map_write($map, $$, $$); $) $+
$( $chip->state = FL_ERASING; $) $+
```

Snippet 1

```
map_write(map, cfi->sector_erase_cmd,
        chip->in_progress_block_addr);
chip->state = FL_ERASING;
chip->oldstate = FL_READY;
```

Snippe 2

```
map_write(map, CMD(0xd0), adr);
map_write(map, CMD(0x70), adr);
chip->state = FL_ERASING;
chip->oldstate = FL_READY;
```

Snippe 3

```
map_write(map, CMD(LPDDR_RESUME),
        map->pfow_base + PFOW_COMMAND_CODE);
map_write(map, CMD(LPDDR_START_EXECUTION),
        map->pfow_base + PFOW_COMMAND_EXECUTE);
chip->state = FL_ERASING;
chip->oldstate = FL_READY;
```

Figure 3: An Example of Crafted Query and Type 3 Clones for CBCD Data

if those queries accurately match the other snippets in the same clone set. We have confirmed that all the crafted queries accurately match other snippets in each clone set. Fig.3 shows an example of a crafted query and its type 3 clone set. In this example, function `map_write(...)` is repeatedly invoked followed by repeated assignments to variable `chip`'s elements.

As far as our investigation, all the matches are controlled by the query, and are performed accurately as we have expected.

5.3 RQ3: Performance of ccgrep

It is interesting to know that our approach, i.e., token-based and naive sequential pattern matching, can be implemented fast enough for practical use. We have examined various queries for `ccgrep` with the target source files of Antlr and Ant in Java, and CBCD data (Git, PostgreSQL, and Linux Kernel)[8] in C, and have measured the performance of `ccgrep`. Following are employed queries. All execution was made with default setting of `ccgrep` except for the language option.

qA: `a < b? a: b`

Find ternary operation to give a smaller value.

qB: `T1 f(T2 a) { return $$; }`

Find function definition immediately returning a value.

qC: `f($$, $$, $$);`

Find three parameter function.

[8]For comparison to CBCD, we have used the same data set, but currently those projects are enhanced 2 to 4 times larger, and the results for applying to the current projects grow almost linearly in both #found and time.

Table 4: Target and Execution Result by `ccgrep`

Target		Antlr	Ant	Git	PgSQL	Linux
Lang.		Java	Java	C	C	C
#file		678	1,272	339	904	15,123
#line		59,511	138,396	90,495	177,174	3,756,212
qA	#found	0	2	8	3	48
	time(sec.)	1.12	1.32	1.11	1.43	9.46
qB	#found	159	161	7	27	543
	time(sec.)	1.15	1.33	1.10	1.47	10.15
qC	#found	1,710	2,487	5,717	10,603	187,653
	time(sec.)	1.20	1.38	1.13	1.55	12.01
qD	#found	1	13	442	621	10,754
	time(sec.)	1.19	1.52	1.10	1.49	11.06

Antlr: Antlr4 v.4.7.2, Ant: Apache Ant v.1.10.5, Git: v.1.6.4.3, PgSQL: PostgreSQL v.6.5.3, Linux: Linux kernel v.2.6.14rc2

qD:

```
for(a = 0; a < $$; a++) { $$ }
$|
a = 0; while(a < $$) { $$ a++; }
```

Find `for` or `while` statement with a control variable.

Tab.4 shows the size metrics of the target, the number of found snippets, and the execution time of each query on a workstation with Intel Xeon E5-1603v4 (@2.8GHz × 4), 32GB RAM, and Windows 10 Pro for WS 64bit.

As we can see from Tab.4, the execution times are about 10 sec. even for a few million lines of Linux kernel target. This might not be very fast, but we would think that they are acceptable speed as a daily-used development or maintenance tool.

The execution times for qA to qD are very stable for each target. For example, in the case of Linux, they are about 10 sec. even for the small #found case (48 for qA) and the large #found case (187,653 for qC). Thus, we would say that the execution time is not heavily affected by the result size (#found) but mainly affected by the target size (#line).

Targets Ant in Java and PgSQL in C have similar sizes around 140-180K lines, and the execution times are also similar around 1-1.5 sec. This would show that the execution time is not strongly affected by the target language.

For comparison to other tools, we have used a code snippet finder NCDSearch[15] and `grep`. NCDSearch finds similar code blocks in the target file for the query code block, by checking the normalized compressed distance of the query and target blocks, and it reports sometimes false positive results. For NCDSearch, we have used qA as its query, and for `grep` we have used GNU grep[11] with following qA'.

qA'(`grep`):

```
([a-zA-Z_][a-zA-Z_0-9]*)\s*<
([a-zA-Z_][a-zA-Z_0-9]*)\s*\?\s*
\1\s*:\s*\2
```

This qA' is to find ternary expression using an extended regular expression[9], and it is almost equivalent query to qA for `ccgrep` except that qA' does not allow the new line between tokens. In order to allow the new lines for `grep`, the query expression becomes too complex to present here.

[9]We have used options `-w` and `--include='*.[ch]'`. Some version of `grep` might require -E option or `egrep` command for the extended regular expression.

Table 5: Comparison of `ccgrep` with NCDSearch and `grep`

Target		Antlr	Ant	Git	PgSQL	Linux
qA	#found	0	2	8	3	48
(ccgrep)	time(sec.)	1.12	1.32	1.11	1.43	9.46
	time ratio	1.0	1.0	1.0	1.0	1.0
qA	#found	3	21	22	80	21,047
(NCDSearch)	time(sec.)	5.55	12.00	8.41	16.54	366.39
	time ratio	4.96	9.09	7.58	11.57	38.73
qA’	#found	0	1	8	2	44
(grep)	time(sec.)	0.24	0.30	0.12	0.24	2.62
	time ratio	0.21	0.23	0.11	0.17	0.28

Time ratio means the ratio of the execution times of each tool to `ccgrep`.

Tab.5 shows the execution result of `ccgrep`, NCDSerach, and `grep`. Since NCDSearch contains false positives, the number of found (#found) is larger than that of `ccgrep` that contains no false positives. Also, since qA’ could sometimes miss the snippets with new lines, #found for `grep` is sometimes smaller than that of `ccgrep`.

`ccgrep` is faster than NCDSearch for all targets, and `grep` is faster than `ccgrep`. Since `grep` is known to be very fast[10], we would think that the speed of `ccgrep` is acceptable as a practically usable software engineering tool. We will also discuss on further improvement of the performance in Sec.6.3.

6 DISCUSSIONS

6.1 CC Matching Approach

We have proposed and formulated CC matching as a method of finding code snippets for the user’s interest. The approach is based on the notion of finding code clones in the target, and the query is a simple code snippet or its extension with meta symbols representing specified or wildcard tokens, regular expressions, and so on.

We would think that our approach is a very good support for software maintainers who need to look around huge source code, due to various reasons such as bug fixing, feature locating, refactoring and so on. Compared to `grep` CC matching (and `ccgrep`) generally can make rich queries more easily and compactly, in the sense that white spaces, new lines, or comments are not considered, and matching identifier or literals can be controlled with meta-tokens.

In fact, during our testing and evaluation of `ccgrep` we scanned the source code of Linux for specific clones. We discovered that the pattern `a<b?a:b` had been removed over time, yet some instances persisted. As described in the Motivating Example, we fixed 3 of them in the `usb/drivers` modules and submitted patches for them. To this date, two of the three have been accepted and are already integrated into Linux.

There are many other approaches proposed to make rich queries for code matching[1]; however, those are not easy to use for many software engineers due to their own query forms. Our approach relies on the notion of finding code clones, which is straightforward and easy to understand and to use for many people. In addition, we have adopted a `grep`-like interface for `ccgrep`, which would greatly reduce the burden of using a new tool.

[10] https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html

Issues of code clones have become popular and acknowledged by not only by software engineering researchers but also industry people[23], along with prevalence of various clone detectors such as stand-alone tools, CCfinderX[19], NiCad[7], and SourcererCC[37], or IDE’s with clone detection features such as Eclipse[45], or Visual Studio[27]. However, those are basically large systems, and their proper installation and operation are not easy in general. We would encourage the creators of these tools to create simpler interfaces that make it easy to find specific instances of clones of small snippets, as ccgrep does.

6.2 Extending Matching for Unspecified Type 3 Matching

As discussed in former sections, CC matching allows type 3 queries as the specified type 3 matching, where we have to predict and specify the variable parts of the seed snippet of the clone pair, with some wildcard tokens. For the unspecified type 3 matching, i.e., if we do not know the variable parts clearly, but would want to find just ’similar’ code snippets with a similarity metric value higher than a threshold, the current CC matching approach might be insufficient.

To solve this issue, we could extend CC matching to allow unspecified type 3 queries, by introducing, say, similarity-based matching or error-allowable matching[29]. However, this introduction would be far away from the current policy of CC matching such that the matches are rigorously controlled by the query with options and no ambiguity in the result is allowed. In such sense, although we are interested in this extension, it would be accomplished as another matching framework and a different tool.

6.3 Performance of ccgrep

As shown in the evaluation, `ccgrep` is not as fast as `grep` but we think that it is acceptable as a practical and useful tool for finding code snippets. Currently, `ccgrep` employs a simple and naive sequential matching algorithm. The mismatch information is not used for the following process, but the algorithms using the mismatch information such as Knuth-Morris-Pratt algorithm or Boyer-Moore algorithm[40] could be used for further performance improvement. In addition, in the current implementation, a sequential trial process for the selection of regular expressions is employed, but it could be improved by introducing the parallel processes.

7 RELATED WORKS

7.1 Pattern Matching Tools

`grep` was originally developed as a simple pattern matching tool for Unix, and has been enhanced with many regular-expression features and other fast pattern matching algorithms for GNU grep. Variants of grep, such as context grep cgrep, approximate grep agrep, and many others had been proposed and implemented to meet various requirements[1]. However, there is no one for clone-based matching like ours.

Semantic-based matching tool sgrep[5] relates to our work in the sense of matching based on the program contexts, and the logic-based query pattern capturing language is proposed in [39]. However, compared to these approaches, our approach is much

closer to the original program syntax and code snippet, than their proprietary query patterns. We would think, the learning cost for our approach is smaller than those for special and proprietary matching patterns.

Formalization of abstracted pattern matching over various languages had been proposed by Dekel et al.[8]. They have defined an abstract code pattern language CPL focusing on semantics rather than syntax. Paul et al. had proposed a formal data model with an algebraic expression-based query language[31]. Those approaches are aiming at generalization and formalization of query patterns, and our approach, on the other hand, focuses on language-depend easily-created query patterns with notion of code clone and meta-pattern.

awk is a pattern matching and text processing tool for general text handling[12]. Though it provides flexible pattern matching with regular expressions and powerful actions associated with the matches, no mechanism for matching based on the notion of code clones is provided.

7.2 Code Clone Detectors

There are numerous number of publications on code clone detection methods and their tools[33, 36]. Token-based approach is a very popular process for clone detection, and one of early works was Dup[3], where important notions such as P-match and suffix tree matching algorithm were used. CCFinder extended this idea to strengthen practical use by normalization and other transformation[20]. CP-Miner used a frequent subsequent mining method to find similar sequences[25].

Textual-based approach is another major method, pioneered by Johnson[18], and it has been used by many others[10]. NiCad used both text and tree-based approach to detect near-miss clones[7]. SourcererCC employed block-based matching and heuristics for filtering out redundant comparison to scale analysis[37].

Most of these approaches focus on finding all code clone pairs in the target file collection. They report all code clones or similar code snippets with similarity higher than certain threshold. Precisely controlling the matches with meta-symbols like CC Matching cannot be accomplished by those approaches.

7.3 Code Snippet Search Tools

There are several tools specialized for finding code snippet. CBCD has been designed for finding related code snippets from a buggy code snippet, by using matching of Program Dependence Graph (PDG)[24]. It can be used to find type 1, 2, and 3 clones; however, the matching generally requires long pre-processing time to construct PDG, and so this approach would not fit to the handy clone finding that we are interested in. For example, it is reported that pre-processing time for a part of the Linux kernel with 170K LOC required 32 minutes, which is far slower than our approach, even though it is claimed that the pre-processing is a one-time overhead and repeatedly used for many queries.

NCDSearch has been designed to find similar code snippets in the pile of source code files for the analysis of code reuse and evolution[15]. The query and the target snippets in block level are compressed together to see the similarity. The approach would be unique and interesting, but the speed is slower than ours as shown in Sec.5.3. In addition, the user cannot predict properly the similarity to be matched by the approach, which produces false-positive and unexpected matching result.

Siamese has been developed for finding code clone pairs for a query method or file using multiple representation of n-gram token sequences with inverted index[32]. It reports the ranked result of type 1, 2, and 3 clones, and it shows good accuracy and scalability. However, although its query response time is small, it requires fairly long indexing time (e.g., about 10 minutes indexing time for 10,000 method target). Thus its application and usage would be different from ours.

8 THREAT TO VALIDITY

We have started with the issue on the current pattern matching tools and code clone tools. For highly experienced users of those tools, our issue might not be applicable due to their deep knowledge and skill of the tools. We are interested in an approach widely applicable to many software engineers including not highly experienced ones for those tools. Controlled experiment and/or feedback from various users might help to understand the issue deeply.

Our evaluation in this paper was mainly focusing on finding code snippets with well expected and specifiable patterns, as discussed in Sec.6.2. Experiment with broader or vaguer queries including unspecified type 3 matching might lower performance, but we would think that handling such queries will be a different research topics including a different tool implementation.

Expressiveness for the specified type 3 matching might be non-exhaustive. However, as long as we have investigated the CBCD clone data, the queries for all type 3 clones were created without any difficulty and we strongly think that we can easily extend them to many other type 3 clones.

9 CONCLUSIONS

We have presented and formulated a new approach to find code snippets in the target files with notion of code clone and meta-pattern, as CC matching, associated with its implementation `ccgrep`. It is a practical and effective pattern matching method, and the tool is efficient and easy-to-use to many software engineers.

As a future direction, we are interested in further performance improvement by using more efficient pattern matching algorithms as discussed in Sec.6.3. Also, we are trying to spread use of `ccgrep` to industry and academia. We believe that `ccgrep` is a very good tool to explore similar bugs to prevent the late propagation[4], and some companies have already shown their interest to their industry applications.

REFERENCES

[1] Tony Abou-Assaleh and Wei Ai. 2004. Survey of Global Regular Expression Print (grep) Tools. In *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.95.3326.*

[2] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. 1999. *Modern Information Retrieval.* ACM Press, Addison-Wesley, New York.

[3] Brenda S. Baker. 1992. A Program for Identifying Duplicated Code. *Proc. of Computing Science and Statistics: 24th Symposium on the Interface 24* (1992), 49–57.

[4] Liliane Barbour, Foutse Khomh, and Ying Zou. 2011. Late propagation in software clones. In *2011 27th IEEE International Conference on Software Maintenance (ICSM).* 273–282. https://doi.org/10.1109/ICSM.2011.6080794

[5] R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey. 2002. Semantic grep: Regular Expressions + Relational Abstraction. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* 267–276. https://doi.org/10.1109/WCRE.2002.1173084

[6] S Carter, RJ Frank, and DSW Tansley. 1993. Clone Detection in Telecommunications Software Systems: A Neural Net Approach. In *Proc. Int. Workshop on Application of Neural Networks to Telecommunications.* 273–287.

[7] J. R. Cordy and C. K. Roy. 2011. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension.* 219–220. https://doi.org/10.1109/ICPC.2011.26

[8] U. Dekel, T. Cohen, and S. Porat. 2003. Towards a Standard Family of Languages for Matching Patterns in Source Code. In *Proceedings 2003 Symposium on Security and Privacy.* 10–19. https://doi.org/10.1109/SWSTE.2003.1245311

[9] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: a Taxonomy and Survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.

[10] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings IEEE International Conference on Software Maintenance (ICSM'99).* IEEE, 109–118.

[11] Free Software Foundation. 2018. GNU Grep 3.3 Manual. https://www.gnu.org/software/grep/manual/grep.html

[12] Gnu. 2015. The GNU Awk Userś Guide. https://www.gnu.org/software/gawk/manual/gawk.html

[13] Dan Gusfield. 1997. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, New York, NY.

[14] Mehran Habibi. 2004. *Java Regular Expressions: Taming the Java.util.regex Engine.* Apress. https://doi.org/10.1007/978-1-4302-0709-2

[15] Takashi Ishio, Naoto Maeda, Kensuke Shibuya, and Katsuro Inoue. 2018. Cloned Buggy Code Detection in Practice Using Normalized Compression Distance. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018.* 591–594.

[16] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12).* IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2012.13

[17] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based Detection of Clone-related Bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07).* ACM, New York, NY, USA, 55–64. https://doi.org/10.1145/1287624.1287634

[18] J. Howard Johnson. 1993. Identifying Redundancy in Source Code Using Fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1 (CASCON '93).* IBM Press, 171–183. http://dl.acm.org/citation.cfm?id=962289.962305

[19] Toshihiro Kamiya. 2010. CCFinderX Manual. http://www.ccfinder.net/

[20] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Eng.* 28 (2002), 654–670.

[21] Brian W. Kernighan and Bob Pike. 1999. *The Practice of Programming.* Addison-Wesley, Boston.

[22] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 187–196.

[23] Rainer Koschke, Ira D. Baxter, Michael Conradt, and James R. Cordy. 2012. Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071). *Dagstuhl Reports* 2, 2 (2012), 21–57.

[24] J. Li and M. D. Ernst. 2012. CBCD: Cloned Buggy Code Detector. In *2012 34th International Conference on Software Engineering (ICSE).* 310–320. https://doi.org/10.1109/ICSE.2012.6227183

[25] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.

[26] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta. 2016. JDeodorant: Clone Refactoring. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C).* 613–616.

[27] Microsoft. 2013. Finding Duplicate Code by using Code Clone Detection. https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2012/hh205279(v=vs.110)

[28] M. Mondai, C. K. Roy, and K. A. Schneider. 2018. Micro-clones in evolving software. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* 50–60. https://doi.org/10.1109/SANER.2018.8330196

[29] Gonzalo Navarro. 2001. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.* 33, 1 (March 2001), 31–88. https://doi.org/10.1145/375360.375365

[30] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring Bug Fixes in Object-oriented Programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10).* ACM, New York, NY, USA, 315–324. https://doi.org/10.1145/1806799.1806847

[31] Paul and Prakash. 1994. Querying Source Code Using an Algebraic Query Language. In *Proceedings 1994 International Conference on Software Maintenance.* 127–136. https://doi.org/10.1109/ICSM.1994.336782

[32] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: Scalable and Incremental Code Clone Search via Multiple Code Representations. *Empirical Software Engineering* 24, 4 (2019), 2236–2284. https://doi.org/10.1007/s10664-019-09697-7

[33] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software Clone Detection: A Systematic Review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[34] Jeff Gray Robert Tairas. 2012. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology* 54, 12 (2012), 1297–1307.

[35] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How Do Professional Developers Comprehend Software?. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12).* IEEE Press, Piscataway, NJ, USA, 255–265. http://dl.acm.org/citation.cfm?id=2337223.2337254

[36] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 74, 7 (2009), 470 – 495. https://doi.org/10.1016/j.scico.2009.02.007

[37] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16).* ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[38] Janice Singer and Timothy C. Lethbridge. 1997. Whatś so Great about 'grep'? Implications for Program Comprehension Tools. In *Tech. Rep., National Research Council, Canada.*

[39] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2019. Active inductive logic programming for code search. In *Proceedings of the 41st International Conference on Software Engineering.* IEEE Press, 292–303.

[40] William Smyth. 2003. *Computing Patterns in Strings.* Addison-Wesley, New York.

[41] Q. D. Soetens and S. Demeyer. 2010. Studying the Effect of Refactorings: A Complexity Metrics Perspective. In *2010 Seventh International Conference on the Quality of Information and Communications Technology.* 313–318. https://doi.org/10.1109/QUATIC.2010.58

[42] M. Storey. 2005. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *13th International Workshop on Program Comprehension (IWPC'05).* 181–191. https://doi.org/10.1109/WPC.2005.38

[43] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating Clone Detection Tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 131–140.

[44] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. 2015. Assessing the Refactorability of Software Clones. *IEEE Transactions on Software Engineering* 41, 11 (Nov 2015), 1055–1090. https://doi.org/10.1109/TSE.2015.2448531

[45] Minhaz F. Zibran and Chanchal K. Roy. 2012. IDE-based Real-Time Focused Search for Near-Miss Clones. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012.* 1235–1242.