IDE-based Real-time Focused Search for Near-miss Clones

Minhaz F. Zibran Chanchal K. Roy
Department of Computer Science, University of Saskatchewan
Saskatoon, SK, Canada S7N 5C9
{minhaz.zibran, chanchal.roy}@usask.ca

ABSTRACT

Code clone is a well-known code smell that needs to be detected and managed during the software development process. However, the existing clone detectors have one or more of the three shortcomings: (a) limitation in detecting *Type-3* clones, (b) they come as stand-alone tools separate from IDE and thus cannot support clone-aware development, (c) they overwhelm the developer with all clones from the entire code-base, instead of a focused search for clones of a selected code segment of the developer's interest.

This paper presents our IDE-integrated clone search tool, that addresses all the above issues. For clone detection, we adapt a suffix-tree-based hybrid algorithm. Through an asymptotic analysis, we show that our approach for clone detection is both time and memory efficient. Moreover, using three separate empirical studies, we demonstrate that our tool is flexibly usable for searching exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones with high precision and recall.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, and reverse engineering*

General Terms

Algorithm, Design, Management

Keywords

clone detection, clone search, maintenance, reengineering

1. INTRODUCTION

Code clone is a well-known code smell, and duplicate or near-duplicate code fragments are regarded as code clones. Programmers' copy-paste-modification practice is regarded as one of the main reasons for the *intentional* clones that are beneficial in many ways [8]. Practically, *unintentional* clones also appear due to a number of reasons. For example, the use of design patterns, frameworks, and similar APIs may result in unintentional code clones. Previous studies reported that software systems might have 9%-17% [26] duplicated code, up to 50% [15]. However, such clones may cause fault propagation, inflate the code-base, and increase maintenance effort [24].

Over the past decade several techniques and tools for detecting code clones have been proposed. While most of them are capable of detecting *Type-1* (exactly similar code fragments except for white-spaces and formatting) and *Type-2* (syntactically similar code snippets, where identifiers/variables can be renamed) clones, only a few of them are reported to detect *Type-3* (where one or more lines of code can be added/modified/removed) clones.

Moreover, most clone detectors are developed as separate tools that facilitate 'postmortem' approach of clone detection after code development is complete [11]. While such tools are beneficial in the analysis and investigation of code clones and their evolution, they fail to provide necessary clone management support [25] for clone-aware development process, as they are not IDE-based. Those few tools that are integrated with IDEs (Integrated Development Environments) are mostly focused on detecting *Type-1* and *Type-2* clones, and typically report all the clones in the entire code-base. Such flooding of information may overwhelm the developer, who in practice, is likely to be interested in only the clones of a certain portion of code she deals with at a time.

To address these issues, we have developed an IDE-based clone search engine (as a plugin to the `Eclipse` IDE) to facilitate *focused* search of both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones during the *real development time*. By *"focused clone search"*, we mean searching and reporting the clones of a selected code segment only, something similar to that shown in Figure 1. This selected code segment is called the *seed fragment* for the search, and it encompasses one or more consecutive lines of code ignoring the comments. The *search space* may be the entire code-base, or a portion of it (e.g., set of directories/packages, projects, files) according to the user's choice. Thus, such focused clone search avoids the unnecessary computation overhead that the traditional clone detectors would perform in finding all the clones from the entire code-base.

Upon identifying the desired code clones, the developer then may decide to perform clone refactoring, or make reference to existing code instead of introducing a new clone fragment. Beside support for such clone-aware development activities, the focused clone search can also be useful in finding similar code fragments from different projects indicating potential reuse, or examples of usage of the concerned APIs.

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

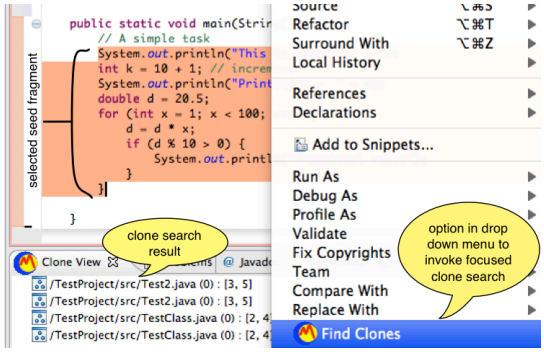

Figure 1: Focused clone search facility

2. CLONE DETECTION

Clone detection techniques can broadly be categorized as token-based, text-based, tree-based, graph-based, and metric-based [19], which have their advantages and weaknesses. For clone detection, we adopt a hybrid approach combining strengths of multiple techniques. Figure 2 presents a schematic diagram of the major algorithmic modules and the process of clone detection used in our approach.

Given a *seed fragment* our technique finds all of its near-miss clones residing in the bodies of the functions within the user-defined search space. For clone identification, each function within the search space is examined with respect to the *seed fragment*. In the following subsections we describe our clone detection procedure with an illustrative example. First, consider the top two code fragments in the Figure 3.

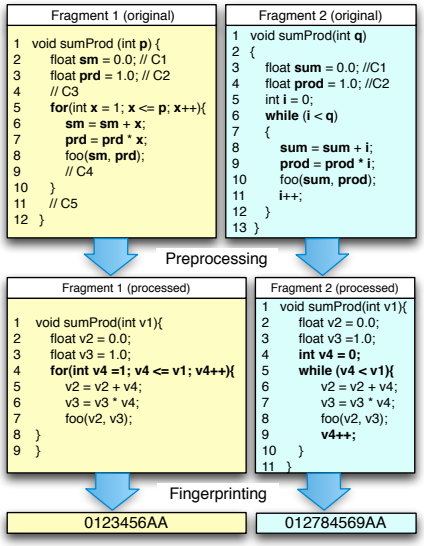

Figure 3: Preprocessing and fingerprinting

These two fragments represent the editing scenario $S4(d)$ of the clone detection technique evaluation framework proposed by Roy and Cordy [19]. They showed that most of the existing clone detection tools did not perform well to detect clones in this scenario. However, we will now show how our multi-phase technique can efficiently detect them as clones.

2.1 Code Preprocessing

At the very first phase, using Eclipse's JDT API's, we generate ASTs (Abstract Syntax Trees) for the source code within the user-defined search space. At this point, we filter out the comments and blank lines, and extract all the functions/methods. Then using the Eclipse's refactoring API, we further *normalize* the code by consistently renaming identifiers and variable names. Such normalization is applied to the *seed fragment* as well. The normalized code fragments are then uniformly formatted with the help of Eclipse's code formatter API. Thus, upon completion of the preprocessing, the original code segments are transformed to fragments free from variations in variable names, comments and layout, as shown in the Figure 3. For our current example, the original code snippets (top two in the Figure 3) are preprocessed to the transformed fragments (the two fragments in the middle of the Figure 3). The preprocessed code segments are then fed to the next module, `LIME` (Language Independent Matching Engine) [25], which performs further computation and comparison.

2.2 Source Code Fingerprinting

Using Rabin's fingerprinting algorithm [14], `LIME` computes fingerprints for each line of all the preprocessed code fragments, and thus generates unique integer value for every distinct line of code. By using fingerprints instead of original lines of source code, we avoid the overhead of comparatively expensive pairwise string comparisons. Suppose, for the preprocessed code fragments shown in Figure 3, the computed fingerprints are as presented in Table 1. The actual fingerprints are very different from what we show in the table. But for the clarity of description, here we show fingerprints having single digit hexadecimal integer values.

Table 1: Hypothetical fingerprints for lines of code

Line of Code	Fingerprint
void sumProd(int v1){	0
float v2 = 0.0;	1
float v3 = 1.0;	2
for (int v4 = 1; v4 <= v1; v4++){	3
v2 = v2 + v4;	4
v3 = v3 * v4;	5
foo(v2, v3);	6
int v4 = 0;	7
while (v4 < v1){	8
v4++;	9
}	A

Having the lines source codes 'fingerprinted', for each code fragment, we get a sequence of fingerprints. For the preprocessed fragments of Figure 3 the sequence of fingerprints we get are "0123456AA" for the fragment 1, and "012784569AA" for the fragment 2.

2.3 Creation of Suffix Tree

Upon fingerprinting the preprocessed code fragments, we prepare a generalized sequence of fingerprints by concatenating fingerprint-sequences from all the fragments. To separate fingerprint-sequences from subsequent fragments, we use a distinct terminator (for the current example, say the '$' and '#' symbol) after each of the fingerprint-sequences. So, for the preprocessed fragments of Figure 3, the generalized fingerprint-sequence we get is "0123456AA$012784569AA#".

Next, for the generalized fingerprint-sequence, we construct a generalized *suffix tree* using Ukkonen's online algorithm [22], which runs in linear time. A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m

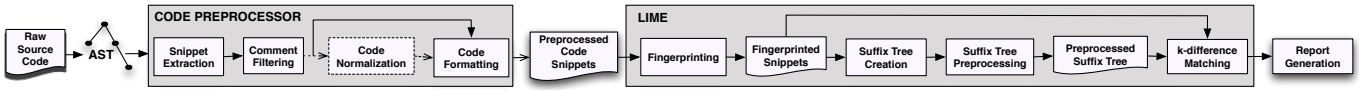

Figure 2: Schematic diagram of our approach for near-miss code clone detection

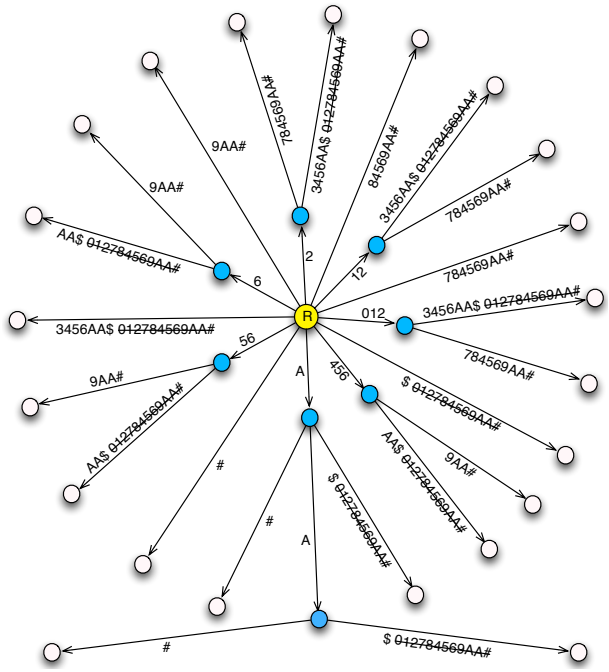

Figure 4: The suffix tree for the generalized sequence "0123456AA$012784569AA#"

leaves. Each internal node of the suffix tree, other than the root, has at least two children, and each edge is labeled with a nonempty substring of S. Moreover, no two edges out of a node have edge-labels beginning with the same character. Each leaf corresponds to a suffix of S that can be obtained by concatenation of the edge-labels on a path from the root to the leaf. A direct edge from a non-leaf node to a leaf node is called a *leaf edge*. In the Figure 4, we present the suffix tree for the sequence "0123456AA$012784569AA#" having the root at the center marked with 'R'. Upon creation of the suffix tree, we discard from each of the edge-labels the portion following the first (left-most) terminator symbol. Those discarded portions are shown in the figure with strike-through text. The label of each leaf edge thus ends with a terminator symbol.

2.4 Finding Largest Common Subsequences

The concatenation of edge-labels on paths from the root to the non-leaf internal nodes (filled circles in the figure) of the suffix tree gives a set of all sub-sequences common in the fingerprint-sequences. Which fragments participate in a certain common sub-sequence can be determined by looking at the terminator symbols on the labels of the leaf edges on the path from corresponding internal node. For our current example, {2, 12, 012, 6, 56, 456, A, AA} is the set of common sub-sequences we find. We further filter out this set by removing all those sequences that are subsumed by any other sequences and those which are smaller than a user-defined minimum size ω. For our example, the resulting set of the largest common sub-sequences becomes {012, 456} for $\omega \geq 3$ SLOC (Source Lines of Code).

Table 2: Global alignment of *0123456AA* and *012784569AA* (here, '_' denotes a space)

0	1	2	**7**	**8**	4	5	6	**9**	A	A
0	1	2	**3**	_	4	5	6	_	A	A

From this set of fingerprint-sequences we trace back the actual lines of the original code fragments, and thus we find that the first three lines of fragment 1 is a *Type-2* clone of the first four lines of the fragment 2. Similarly, lines six through eight of the fragment 1 and the lines eight through ten are also *Type-2* clones of each other. To the best of our knowledge, `CCFinder`, `CloneDigger`, `Dup`, `RTF` and the rest of the suffix-tree-based clone detectors including that of Tairas and Gray [20] exploit suffix trees up to this level with or without fingerprinting the source code [19]. But, the usage of suffix trees up to this point can facilitate the detection of *Type-1* and *Type-2* clones only. The detection of *Type-3* clones requires further computations.

2.5 Approximate Matching

For detecting *Type-3* clones, we invoke a *k-difference hybrid algorithm* (described later in Section 3) on pairs of fingerprinted code fragments. The approximate matching algorithm finds all the occurrences of one sequence inside another allowing at most k differences. Based on a user-defined *dissimilarity threshold*, we compute k for each pair of code fragment as, $\lceil k = \frac{l \times threshold}{100} \rceil$, where, l is the number of lines in the smaller fragment. The threshold signifies what percentage of different lines of code be allowed in the approximate matching. For example, consider two fragments of source code having 100 and 120 lines respectively, and the user's given threshold is 10%. Then the value of $k = \frac{100 \times 10}{100} = 10$, which means in the approximate matching, difference of at most 10 lines will be tolerated. Thus our computation of k is sensitive to the size of the code fragments.

For our current example, the corresponding fingerprint-sequences "0123456AA" and "012784569AA" approximately match (with $k \geq 3$) having three differences, yielding a global alignment, for instance, as shown in Table 2. Thus, given either of the original fragments of Figure 3 as seed, we accurately detect the other as its *Type-3* clone. All the detected clones are then sorted according to their similarity with the given seed fragment, and the result is then displayed in the Eclipse's interactive Tree View. Choosing a clone from the Tree View highlights the corresponding code fragment in the editor.

3. K-DIFFERENCE HYBRID ALGORITHM

We adapted the k-difference hybrid algorithm from the work of Landau and Vishkin [10]. The algorithm combines the advantage of further preprocessed suffix tree with dynamic programming (DP). Readers interested in the details of the algorithm are referred to elsewhere [5, 10]. In this section, we provide a brief description of how we adapted the algorithm for near-miss clone detection.

3.1 Preprocessing of Suffix Tree

We enumerate all the r nodes of the suffix tree $\mathcal{T}$ according to the standard *depth-first (preorder)* numbering as shown in Figure 5. Thus each node v of $\mathcal{T}$ is numbered with a $\lceil log_2 r \rceil$ bit integer.

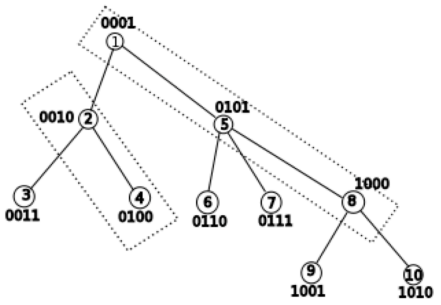

Figure 5: Enumerated suffix tree having nodes partitioned into Runs marked with dotted rectangles [5]

Then, we map the enumerated suffix tree $\mathcal{T}$ to a hypothetical rooted complete binary tree $\mathcal{B}$, whose vertices are enumerated according to *inorder* numbering, as shown in Figure 6. Throughout the remainder of the paper, we will simply use "node v", to refer to the node in the enumerated rooted tree having node number v.

DEFINITION 1 (LOWEST COMMON ANCESTOR). *The lowest or least common ancestor (LCA) of any two nodes u and v in a rooted tree Υ, denoted as $LCA_\Upsilon(u, v)$, is the deepest node x in the tree, which is an ancestor of both u and v [5].*

DEFINITION 2. *For any node k, $h(k)$ is the position (from right) of the least significant 1-bit in the binary representation of k. For a node k in $\mathcal{B}$, $h(k)$ equals the height of the node in $\mathcal{B}$ [5].*

DEFINITION 3. *For a node v of $\mathcal{T}$, $I(v)$ is the node w in $\mathcal{T}$ such that $h(w)$ is the maximum over all nodes in the subtree of v including v itself [5].*

DEFINITION 4. *A run in $\mathcal{T}$ is a maximal subset of nodes of $\mathcal{T}$, denoted as T_r, such that, $\forall \langle u, v \rangle \in T_r, I(u) = I(v)$. The* head *of a run is the node closest to the root [5].*

We map each node v in $\mathcal{T}$ to a node $I(v)$ in $\mathcal{B}$. Figure 5 presents an example of partitioning the nodes of suffix tree $\mathcal{T}$ into seven different runs. Figure 6 shows a complete binary tree $\mathcal{B}$, which the nodes of the suffix tree $\mathcal{T}$ of Figure 5 are mapped to. In the Figure 6, the binary representation of the number a node v in $\mathcal{B}$ is shown if there is a node in $\mathcal{T}$ that maps to v.

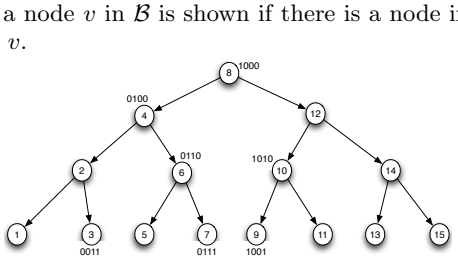

Figure 6: Binary tree with nodes enumerated according to inorder numbering [5]

Moreover, for each node v in $\mathcal{T}$, we create an $O(log_2 r)$ bit number A_v. Bit $A_v(i)$ is set to 1 if and only if node v has some ancestor in $\mathcal{T}$ that maps to height i in $\mathcal{B}$, i.e., if and only if v has an ancestor u such that $h(I(u)) = i$.

3.1.1 Computing LCA in Binary Tree

Having the inorder numbering of the nodes of binary tree $\mathcal{B}$, we compute $LCA_\mathcal{B}(u, v)$ in constant time using shift and XOR operations. We first determine if u is an ancestor of v, or vice versa. Using $h(u)$ and $h(v)$ we determine which of the nodes u and v is higher (closer to the root). If u is the higher node, we find the number of edges on the path from the root to node u. Then we take the XOR of u and v, and find the position k of the left-most 1-bit (counting from the left). Node u is an ancestor of node v if and only if k is larger than the number of edges on the path from the root to node u. In this case node u is the LCA of nodes u and v.

In the cases, where none of u and v is an ancestor of the other, we XOR u and v, find the left-most 1 bit in the k^{th} position in the result. Then we right shift u by $d - k$ places (where $d = \lceil log_2 r \rceil$ is the number of bits used in numbering the nodes of $\mathcal{B}$), set the right-most bit to 1, and left shift it back by $d - k$ places.

3.1.2 Computing LCA in Suffix Tree

Using the following procedure [5], we find z in constant time, where $z = LCA_\mathcal{T}(x, y)$ for the suffix tree $\mathcal{T}$ preprocessed as described before.

1. Find $b = LCA_\mathcal{B}(I(x), I(y))$.
2. Find the smallest position $\rho \geq h(b)$ such that both numbers A_x and A_y have 1-bits in position ρ.
3. Find x', the node closest to x on the same run as z (unknown yet) as follows.
   (a) Find location ρ_r of the right most 1-bit in A_x.
   (b) If $\rho_r = \rho$, then set $x' = x$, and go to step 4. Otherwise continue to next steps.
   (c) Find the position ρ_l of the left-most 1-bit in A_x, which is to the right of position ρ. Form the number i_w consisting of the bits of $I(x)$ to the left position of ρ_l, followed by a 1-bit in position ρ_l, followed by all zeros. Then find the head w of the run containing node i_w. Set node x' to be the parent of node w in $\mathcal{T}$.
4. Find y', the node closest to y on the same run as z, using the same approach as in step 3.
5. If $x' < y'$ then $z = LCA_\mathcal{T}(x, y) = x'$, otherwise, $z = LCA_\mathcal{T}(x, y) = y'$.

3.1.3 Finding Longest Common Extension

Suppose, $S = [s_1\ s_2\ s_3 \ldots s_m]$ is a sequence of length m, and $[s_i\ s_{i+1}\ s_{i+2} \ldots s_j]$ (where, $1 \leq i \leq j$ and $1 \leq j \leq m$) is a subsequence of S, which we denote as $S[i \ldots j]$. Thus, a suffix $[s_i\ s_{i+1}\ s_{i+2} \ldots s_m]$ starting from element s_i is denoted by $S[i \ldots m]$, and $S(i)$ refers to the i^{th} element in sequence S. Given a pair of sequences S_1 (of length m) and S_2 (of length n), and an index pair $\langle i, j \rangle$ where i and j refer to element-positions in S_1 and S_2 respectively, the *longest common extension (LCE)* between the sequences (with respect to the given index pair) is the longest subsequence of S_1 starting at position i that matches a subsequence of S_2 starting at the j^{th} position; in other words, the longest prefix of $S_1[i \ldots m]$ that matches a prefix of $S_2[j \ldots n]$.

Using a preprocessed generalized suffix tree $\mathcal{T}$ for the sequences, given an index pair $\langle i, j \rangle$ corresponding to sequences S_1 and S_2, the LCE can be computed in constant time. We first find the LCA, z of the leaves of $\mathcal{T}$ that correspond to $S_1[i \ldots m]$ and $S_2[j \ldots n]$. The concatenation of the edge-labels on the path from the root to z yields the LCE we want.

3.2 Hybrid Dynamic Programming

A classical way to represent a sequence matching problem is to express it in terms of *global alignment*. Given a couple of sequences S_1 and S_2, their global alignment is obtained by first inserting spaces in chosen places (between elements, at the end or beginning of S_1 and S_2) to make the resulting sequences S_1' and S_2' have equal length l, and then superimposing one above the other so that every element or space in either sequence is opposite a unique sequence or a unique space in the other sequence [5]. Table 2 presents an example of global alignment. Typically, a score $g(x, y)$ is associated with the alignment of each pair $\langle x, y\rangle$ of elements. The score of an alignment of S_1 and S_2 is computed as $\sum_{i=1}^{l} g(S_1'(i), S_2'(i))$, and the alignment with the optimal score yields the optimal global alignment.

Let $V(i, j)$ denotes the score of the optimal global alignment of $S_1[1 \dots i]$ and $S_2[1 \dots j]$. The optimal global alignment of the sequences having score $V(m, n)$ can be computed in $O(mn)$ time by traditional dynamic programming (DP) using the following recurrence.

$$V(i,j) = min \begin{cases} V(i-1, j-1) + g(S_1(i), S_2(j)), \\ V(i-1, j) + g(S_1(i), \_), \\ V(i, j-1) + g(\_, S_2(j)) \end{cases}$$

where the base conditions are,

$$V(0,j) = \sum_{1 \le p \le j} g(\_, S_2(p)) \text{ and } V(i,0) = \sum_{1 \le 1p \le i} g(S_1(p), \_).$$

We formulate the focused clone searching problem as a variation of the optimal global alignment problem. Let S_2 and S_1 be the fingerprint-sequences of respectively the seed and another fragment in the search space. We define a scoring scheme as follows:

$$g(S_1(i), S_2(j)) = \begin{cases} 0, & \text{if } S_1(i) = S_2(j) \\ 1, & \text{if } S_1(i) \neq S_2(j) \end{cases}$$

$$g(S_2(i), \_) = 1$$

$$g(\_, S_1(j)) = \begin{cases} 0, & \text{if } \forall j' < j, S_1(j') \text{ aligned to '\_'} \\ & \quad \text{or, } j > m \\ 1, & otherwise. \end{cases}$$

Instead of finding their optimal global alignment, we need to find all the global alignments with scores no more than the k. The k-difference hybrid approach makes use of suffix trees to solve subproblem of computing the *LCE* queries within the framework of DP. Consider an $m \times n$ traditional DP table for S_1 and S_2. The solution to the optimal global alignment problem computed using traditional DP approach yields a path on the DP table specifying the computed optimal alignment as well as giving the number of pairwise character matches and differences. The same concept of path is used in the hybrid algorithm.

For the sake of the hybrid algorithm, we enumerate the diagonals of the DP table as follows. The main diagonal of the table consisting of cells $\langle i, i\rangle$, for $0 \le i \le n \le m$, is numbered diagonal 0. The diagonals above the main diagonal are numbered 1 through m; the diagonal starting at cell $\langle 0, i\rangle$ is diagonal i. The diagonals below the main diagonal are enumerated -1 through $-n$; the diagonal starting at cell $\langle i, 0\rangle$ is diagonal $-i$.

DEFINITION 5 (*d*-PATH). *A d-path in the DP table is a path that starts in the row zero and specifies a total of exactly d mismatches and spaces [5].*

DEFINITION 6 (FARTHEST REACHING *d*-PATH). *A d-path is farthest reaching on diagonal i, if it is a d-path ending on diagonal i, and the index of its ending column c along diagonal i is the maximum among all the d-paths ending on diagonal i [5].*

For $d > 0$, three distinct d-paths on diagonal i can be computed from $(d-1)$-paths on diagonals $i-1, i$, and $i+1$ [5]:

R_v-path is composed of the farthest-reaching $(d-1)$-path on diagonal $i+1$, trailed by a vertical edge (corresponding to a space in S_1) to diagonal i, and a maximal extension along diagonal i that corresponds to identical subsequences in S_2 and S_1.

R_h-path consists of the farthest-reaching $(d-1)$-path on diagonal $i-1$, trailed by a horizontal edge (corresponding to a space in S_2) to diagonal i, and a maximal extension along diagonal i that corresponds to identical subsequences in S_2 and S_1.

R_d-path is made up of the farthest-reaching $(d-1)$-path on diagonal i, trailed by a diagonal edge (corresponding to a mismatch between an element in S_2 and an element in S_1) along diagonal i, followed by a maximal extension along diagonal i that corresponds to identical subsequences in S_2 and S_1.

With these specifications, in Algorithm 1, we describe the k-difference hybrid algorithm.

Algorithm 1 : k-Difference Hybrid Algorithm [5]

for $i = 0$ *to* m **do**
  find the *LCE between* $S_1[i \dots m]$ *and* $S_2[1 \dots n]$ *by* LCE *query to suffix tree. This specifies the end column of the farthest reaching 0-path on diagonal* i.
end for
Any path reaching row n in column c, defines an exact match of S_2 *in* S_1 *ending at* $S_1(c)$.
for $d = 1$ *to* k ***do***
  for $i = -n$ *to* m ***do***
    Find the end on diagonal i of paths R_v, R_h, *and* R_d. *The farthest-reaching of these three paths is the farthest-reaching d-path on diagonal i.*
  end for
  Any path reaching row n in column c, defines an approximate match of S_2 *in* S_1 *ending at* $S_1(c)$ *with at most k differences.*
end for

4. EVALUATION

In this section, we present both theoretical and empirical evaluation of our approach in terms of performance, accuracy, and usability.

4.1 Algorithmic Complexity

Suppose, we have $\mathcal{F}$ code fragments in the search space including the seed fragment, and fragment f has l_f lines of code. The total number of lines over all code fragments is $l_{\mathcal{F}} = \sum_{f=1}^{\mathcal{F}} l_f$. Without loosing generality, we can assume that each line of code has c characters.

Code preprocessing is done in linear time. Using Rabin's fingerprinting algorithm (which runs in linear time [14]), we fingerprint a line of code in $O(c)$ time, and thus for computing fingerprints of all $l_{\mathcal{F}}$ lines it takes $O(c \times l_{\mathcal{F}})$ time. The number of characters in a nonempty line of source code typically vary between 1 and 20. So, we may consider c to be invariant, and thus the running time of Rabin's fingerprinting algorithm over all lines of source code becomes $O(l_{\mathcal{F}})$.

To construct the generalized suffix tree we use Ukkonen's online algorithm [22], which also runs in linear time. Hence the construction of the generalized suffix tree also takes $O(l_{\mathcal{F}})$ time. Exact matches are readily detected as we construct the generalized suffix tree. Since, the fragments are preprocessed to discard variations in variable names and formatting, all *Type-1* and *Type-2* clones across all code the fragments are also detected in $O(l_{\mathcal{F}})$ time.

Preprocessing of the generalized suffix tree also takes $O(l_{\mathcal{F}})$ time. Given a text S_1 of length m and a pattern S_2 of length n, in $O(km)$ time and $O(m+n)$ space, the k-difference hybrid algorithm can find all end locations in S_1 where S_2 matches with at most k differences [5, 10]. Based on the algorithm, our implementation takes $O(kl_{\mathcal{F}})$ time and $O(l_{\mathcal{F}})$ space to find all occurrence of k-difference near-miss clones of a chosen seed fragment in all other fragments in the search space.

4.2 Usability and Customization

One of the primary objectives of our work is to support clone-aware programming during the real development activities. To make the *focused clone search* facility flexibly usable, our tool enables the user to simply select a code fragment and invoke *focused search* for it's near-miss clones by choosing the appropriate option from a drop-down menu (Figure 1). The user can also define the search space (i.e., projects, directories, or files). Moreover, the tool offers a variety of customization options through an Eclipse-preferences user-interface (UI) as shown in the Figure 7.

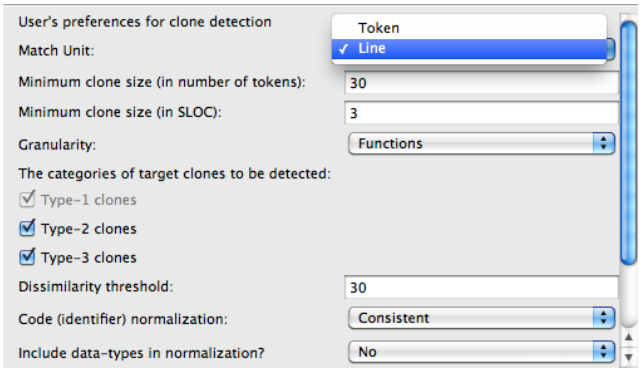

Figure 7: Customization options for clone detection

These customizations define a set of search options such as the type of the target clones (i.e., *Type-1*, *Type-2*, and/or *Type-3*), dissimilarity threshold, code normalization options, minimum clone size, and match-unit (i.e., lines or tokens). Though, in Section 2, we described our clone detection approach to have each line of code as a match-unit, the technique can also use each token/word as a match-unit, and thus can offer more rigorous search if needed. For the token-based operation, the code formatting phase is simply skipped, and the rest of the phases remain as they are. The code normalization phase is skipped if only the *Type-1* clones need to be identified, and the k-difference hybrid algorithm is invoked only when the *Type-3* clones have to be detected.

4.3 Empirical Evaluation

Since our tool is developed as a plugin to Eclipse, it is tedious and time consuming to use the Eclipse's editor UI to manually select code fragments, find their clones and then verify accuracy. Therefore, we created a variant of our tool by replacing the Eclipse-coupled "code preprocessor" module by a separate *TXL*[1] implementation. Such an implementations can handle any programming language, as long as the appropriate Grammar and the transformation rules are defined. Ours can handle C, C#, and Java. Also note that, except the "code preprocessor" module, the rest of our tool is language independent.

Table 3: Systems subject to the comparative study

Subject Systems	Total SLOC	# of Functions	# of Clone Groups	# of Clone Fragments
weltab	9936	123	20	68
PostGreSQL	154843	4689	203	519

The stand-alone variant of our tool can operate outside IDE, and we used this to empirically evaluate the accuracy of clone detection (Section 4.3.1 and Section 4.3.2). We also carry out a user-study (Section 4.3.3) to evaluate the usability of our tool and its usefulness in context. Due to limitation of space, we present abridged summaries of these studies, instead of providing very detailed descriptions.

4.3.1 Comparison with NiCad

Traditional clone detectors are not directly comparable to our tool, as the objective of our tool is to enable focused clone search instead of the detection of all clones from the entire code-base. The work of Lee at al. [11] could have been comparable, but their implementation was not available in public. So, we devised a technique for comparison with `NiCad` [16], a state-of-the-art tool for detecting exact (*Type-1*) and near-miss (*Type-2*, and *Type-3*) clones. We chose `NiCad` for two main reasons. First, the threshold schemes of our tool and `NiCad` are similar and comparable. Second, as a clone detector, `NiCad` was reported to have high precision [16, 18] and recall [17].

For both `NiCad` and our tool, we set the minimum clone size to three SLOC in the pretty-printed format [16] of the source code. Moreover, the UPIT (Unique Percentage of Items Threshold) for `NiCad` and the dissimilarity threshold for our tool, both were set to 30%. Since `NiCad` performs line-based comparison, the match-unit for our tool was also set to 'line'. Then we instructed `NiCad` to detect all near-miss *function* clones from the two subject systems, `weltab` and `PostGreSQL` (Table 3). These systems obtained from www.bauhaus-stuttgart.de/clones/, are two of those used in the clone detectors comparison framework of Bellon et al. [1]. `NiCad` reports the clone detection result having the clones clustered into clone-groups. The number of clone-groups and the total number of individual clone fragments as identified by `NiCad`, are presented in the right-most two columns of the Table 3.

From each of the clone-groups, we randomly picked a fragment, passed it to our tool as the *seed fragment*, and invoked the tool to perform a *focused search* for all the clones of the seed. Then we verified the search result to determine whether our tool detected all other members of the clone-group (as of the seed), and whether any false positives were reported. For each seed fragment obtained from every clone-group over both the subject systems, our tool did find all the remaining members of the corresponding group, and did not report any false positives.

[1]http://www.txl.ca/

Table 4: Accuracy of clone detection

Measurement	*Type-1*	*Type-2*	*Type-3*	Overall
precision	1.0	1.0	1.0	1.0
recall	1.0	0.96	0.90	0.94
f-score	1.0	0.97	0.94	0.96

4.3.2 Mutation-based Evaluation

The aforementioned comparative study evaluated our tool's accuracy in clone detection with respect to that of `NiCad`. Using a variant of the mutation framework proposed by Roy and Cordy [17], we further evaluated the precision and recall of our tool. As the code-base subject to this study, we chose `JHotDraw-5.4b1`, which had 20,613 SLOC Java code.

We selected, as *seed fragments*, 10 arbitrary functions ($f_1 \dots f_{10}$) of different sizes from different locations of the code-base. Then we made 15 copies ($f_i^1 \dots f_i^{15}$) of each of those 10 fragments, and modified them following a subset of mutation operators [17]. Thus, we produced five *Type-1* clones, five *Type-2* clones, and five *Type-3* clones of each *seed fragment* f_i, resulting a total of 150 synthetic function clones. Then we injected these clones into different locations of the original code-base, resulting a *mutated* code-base.

We configured our tool according to the specifications shown in the Figure 7. Then we invoked focused clone search on the mutated code-base, providing each of the 10 seed fragments, one at a time. For each of the seed fragments, we investigated whether all of its synthetic clones (*Type-1*, *Type-2*, and *Type-3*) were reported in the search result. Some additional fragments were also reported as clones, as they existed in the code-base but we did not know about them in advance. We manually verified those additional clones for any possible false positives.

With respect to the injected synthetic clones, we compute the precision (p), recall (r), and f-score ($\lrcorner$) of our tool as,

$$p = \frac{|c_s \cap c_d|}{|c_d|}, \quad r = \frac{|c_s \cap c_d|}{|c_s|}, \quad \lrcorner = \frac{2 \times p \times r}{p + r}$$

where c_s and c_d are respectively the sets of injected synthetic clones, and those that our tool detects. In the Table 4 we present the accuracy of our tool in detecting those synthetic clones. Our tool missed two of the *Type-2* and five of the *Type-3* synthetic clones. We manually investigated those clones, and found that while creating the *Type-2* clones, we altered the order of declaration of variables. The normalization based on "consistent renaming" of identifiers is sensitive to such orders, and this was set in the configuration. This is why our tool could not detect those as clones, but it was able to capture them when later we applied normalization using the "blind renaming" operation. The five *Type-3* clones escaped due to the fact that upon modification, those fragments simply went beyond the 30% dissimilarity threshold we used.

4.3.3 User Study

We carried out a user study involving eight programmers who were graduate students having more than five years of experience in programming under different IDEs including Eclipse. We provided our Eclipse plugin to them, and they used it for an extended period of time while doing some programming tasks. Then we interviewed them with a questionnaire comprising both open and closed questions. The closed questions included a Likart-scale query saying, "*How will you rate this tool's usability and usefulness?*". The participants had to choose one of the answers: very poor, poor, moderate, good, very good, or excellent. In response, five of the participants chose 'good', two chose 'moderate', and the other participant chose 'very good'. A noteworthy comment from one of the participants was, "*...this [tool] is intuitive!*".

5. RELATED WORK

There are many clone detection tools out there and a comprehensive list can be found elsewhere [19]. Among all those clone detectors, those which are integrated with IDEs are the most relevant to our work.

`CloneBoard` [3] and `CPC` [23] are Eclipse plugins that can detect and track clones based on clip-board (copy-paste) activities of programmers. Based on programmer' copy-paste activities, `CReN` [6] (another plugin to Eclipse), offers some sort of clone refactoring support by consistent renaming of identifiers. While such an approach based on programmers' copy-paste activities may be able to handle *intentional clones*, they cannot deal with *unintentional* clones. Moreover, such tools may not be suitable for distributed development, as they may fail to combine information about clones separately created by distinguished developers working in a distributed environment.

`CPD`[2] is a part of Java source code analyzer, `PMD. SDD` [12] and `Simian`[3] are plugins to Eclipse. Tairas and Gray [20] also developed a suffix-tree based clone detector as a plugin for the Microsoft Phoenix framework, which is also an inspiration to our work. All of these clone detectors can detect *Type-1* and *Type-2* clones only, but not *Type-3* [2]. Another Eclipse plugin, `CloneDR`[4], is an AST-based clone detector that can detect *Type-1* and *Type-2* clones, but it also fails to detect Type-3 clones in many scenarios [19]. `CloneDetective` [7] uses a suffix-tree-based algorithm to detect *Type-1* and *Type-2* clones but "probably not" *Type-3* [19]. However, `SimScan`[5], which is a parser-based tool available as plugin to Eclipse, IDEA, or JBuilder, can detect *Type-1*, *Type-2*, and possibly a subset of *Type-3* clones.

`SHINOBI` [9] is a plugin to the Microsoft Visual Studio. It internally uses `CCFinderX`'s preprocessor, and thus it can detect *Type-1* and *Type-2* clones only, but not *Type-3*. It was developed as a client(IDE)-server(CVS) application to relocate the the clone detection overhead from the client to a central server. A similar clone detection tool is `Clever` [13] that is available as an add-on to Subclipse/SVN, an Eclipse plugin software configuration management tool. However, such client-server configuration may not be suitable for those individual practitioners who work on their separate standalone machines.

`CloneTracker` [4] uses `SimScan` as the underlying clone detector, allows the developer to select individual clone groups, and supports simultaneous modifications of clone regions. `CeDAR` [21] can incorporate the results from different clone detection tools (e.g., `CCFinder`, `CloneDR`, `DECKARD`, `Simian`, or `SimScan`) and can display properties of the clones in an IDE. `DupMan`[6] is an Eclipse duplication management framework for clone detection and removal. It also works on top

[2]http://pmd.sourceforge.net/cpd.html
[3]http://www.harukizaemon.com/simian
[4]http://www.semdesigns.com/Products/Clone/
[5]http://blue-edge.bg/download.html
[6]http://sourceforge.net/projects/dupman/

of other clone detectors (e.g., `SimScan`, `CCFinder`, `CloneDR`, `Dup`, `Duploc`, `Duplix`). While all these three tools are developed as plugins to Eclipse, they suffer from the limitations of the underlying clone detectors they use internally.

All of the aforementioned tools adopt the traditional 'post-mortem' approach to detect clones from the entire codebase, which is overkill for a *focused search* of only the clones of an individual code fragment [11]. Besides, most of those tools have limitations in detecting *Type-3* clones. Ours is the first that enables efficient *focused search* not only for *Type-1* and *Type-2* clones, but also for *Type-3*. Moreover, our tool, unlike `SHINOBI` and `Clever`, can be conveniently adopted in both distributed or centralized development environment. Lee at al. [11] used an algorithm based on feature-vector computation over AST and finds the k most similar clones of a given code segment. But, their tool was not reported to have integration with IDE. On the contrary, integration with IDE was a main objective of our work. Moreover, using a suffix-tree-based hybrid algorithm, our tool finds *all* the clones of a given code segment for a given similarity threshold.

6. CONCLUSION

In this paper, we have presented an IDE-integrated *focused clone search* tool implemented based on a suffix-tree-based k-difference hybrid algorithm. Using an asymptotic complexity analysis, we have shown that our approach is efficient in terms of both time and memory. Performing a mutation-based evaluation, we have also demonstrated that our tool is highly accurate in terms of precision and recall. This was further confirmed by a case study comparing our tool with `NiCad`, a state-of-the-art near-miss clone detector. Moreover, from a user-study, we found that our tool is flexibly usable for clone-aware software development.

In the future, we plan to further evaluate our tool in the industrial context with larger code-bases. We have also been working to extend this towards a versatile clone management system [25] by incorporating features such as clone tracking and semi-automated clone refactoring support.

7. REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. on Softw. Engg.*, 33(9):577–591, 2007.

[2] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan. An empirical study on inconsistent changes to code clones at release level. *Science of Computer Programming*, 17 pages, 2010.

[3] M. de Wit. *Managing Clones Using Dynamic Change Tracking and Resolution*. M.Sc. thesis, Delft University of Technology, 2008.

[4] E. Duala-Ekoko and M. Robillard. CloneTracker: tool support for code clone management. In *ICSE*, pages 843–846, 2008.

[5] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Computer and Computational Biology. Cambridge University Press, 1st edition, 1997.

[6] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *ETX*, pages 16–20, 2007.

[7] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - a workbench for clone detection research. In *ICSE*, pages 603–606, 2009.

[8] C. Kapser and M. Godfrey. "Cloning considered harmful" considered harmful. In *WCRE*, pages 19–28, 2006.

[9] 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. In *WCRE*, pages 313–314, 2009.

[10] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.

[11] M. Lee, J. Roh, S. Hwang, and S. Kim. Instant code clone search. In *FSE*, pages 167–176, 2010.

[12] S. Lee and I. Jeong. SDD: high performance code clone detection system for large scale source code. In *OOPSLA*, pages 140–141, 2005.

[13] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone-aware configuration management. In *ASE*, pages 123–134, 2009.

[14] M. Rabin. Fingerprinting by random polynomials. Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[15] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.

[16] C. Roy and J. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[17] C. Roy and J. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *ICSTW*, pages 157–166, 2009.

[18] C. Roy and J. Cordy. Near-miss function clones in open source software: an empirical study. *J. of Softw. Maintenance and Evolution: Research and Practice*, 22(3):165–189, 2010.

[19] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, 2009.

[20] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *ACM-SE*, pages 679–684, 2006.

[21] R. Tairas and J. Gray. Get to know your clones with CeDAR. In *OOPSLA*, pages 817–818, 2009.

[22] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[23] V. Weckerle. CPC: an eclipse framework for automated clone life cycle tracking and update anomaly detection. Master's thesis, Freie Universität Berlin, Germany, 2008.

[24] M. Zibran and C. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *SCAM*, pages 105–114, 2011.

[25] M. Zibran and C. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *IWSC*, pages 75–76, 2011.

[26] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *ICECCS*, pages 295–304, 2011.