Integrating Code Search into the Development Session

Mu-Woong Lee #, Seung-won Hwang #, Sunghun Kim *

#*Pohang University of Science and Technology (POSTECH), Korea, Republic of*
#{sigliel, swhwang}@postech.edu

**Hong Kong University of Science and Technology (HKUST)*
*hunkim@cse.ust.hk

***Abstract*—To support rapid and efficient software development, we propose to demonstrate our tool, integrating code search into software development process. For example, a developer, right during writing a module, can find a code piece sharing the same syntactic structure from a large code corpus representing the wisdom of other developers in the same team (or in the universe of open-source code). While there exist commercial code search engines on the code universe, they treat software as text (thus oblivious of syntactic structure), and fail at finding semantically related code. Meanwhile, existing tools, searching for syntactic clones, do not focus on efficiency, focusing on "post-mortem" usage scenario of detecting clones "after" the code development is completed. In clear contrast, we focus on optimizing efficiency for syntactic code search and making this search "interactive" for large-scale corpus, to complement the existing two lines of research. From our demonstration, we will show how such interactive search supports rapid software development, as similarly claimed lately in SE and HCI communities [1], [2]. As an enabling technology, we design efficient index building and traversal techniques, optimized for code corpus and code search workload. Our tool can identify relevant code in the corpus of 1.7 million code pieces in a sub-second response time, without compromising any accuracy obtained by a state-of-the-art tool, as we report our extensive evaluation results in [3].**

I. INTRODUCTION

Copy-and-paste is one of the most common software development activities [2] to support rapid development. Since it is known to help developers reuse existing code quickly, frequently used during software development, and there are commercial internet-scale code search engines, such as Koders.com[1] or Google code search[2] for locating the related code pieces.

Though these engines are scalable to very large code corpus, they have limited applicability to be used in development processes. These engines, by treating software as text, often fail to find the related code [4], [5]. More precisely, these engines are not quite suitable to be used to find "semantically" related code pieces, and this makes them less useful because referencing semantically related code is more desirable for development.

To address the problem of detecting semantically related code, in the software engineering community, recent research has developed clone detection tools, which detect and trace code clones to assist software development and maintenance tasks. As unmanaged code clones incur difficulty of software maintenance, and may cause *inconsistent clone changes* [6], [7], clone detection research has been actively studied [8], [9], [10], [11], [12], [13].

However, existing code clone detection tools are also insufficient to be "interactively" used during development processes, as they usually take a *post-mortem* approach of detecting clones "after" code development is completed. For example, developers run a code clone detector once per month, and based on the clone information, perform necessary maintenance work such as refactoring or fixing inconsistent clone changes.

In clear contrast, we propose to combine the strength of commercial code search engines and code clone research, by enabling fast similarity searches of semantically related code. This combination enables our application scenario of performing code search during development, as illustrated in Fig. 1.

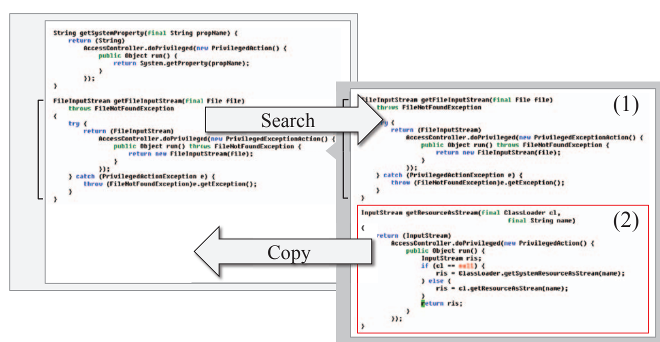

Fig. 1. Developers working on a large project may look up the code sharing the same syntactic structure as a reference and copy some code.

In our demonstration scenario, a developer working on a large project can search, during his development, for the code sharing the same syntactic structure, by hitting the search button. The developer can then, either use that as a reference or copy them into his own code. From our demonstration, we will show how such interactive search supports rapid software development. Our claim is also supported by recent tools

[1]http://koders.com/
[2]http://www.google.com/codesearch/

proposed in HCI community, integrating Web search into the development session [1]. While this work considers only the programming language and version as a context for the explicit keyword query provided by the user, we can leverage "richer" contexts of the code developed so far, to find the matches sharing the same syntactic structure. Another distinction is that while this work focuses on evaluating how the tool helps the development process, we focus on techniques for effectively and efficiently identifying clones, and their scalability to real-life large-scale corpus.

Specifically, as an enabling technology, we propose a search scheme to find syntactic clones of a given query code segment. Our proposed search algorithm adopts a multi-resolution vector abstraction of code, which is used by the existing clone search tool DECKARD [12], to enable efficient structural similarity comparisons. Meanwhile, to achieve instant search, we design index building and traversal algorithms, optimized for code data and code search workload. These techniques has enabled us to build a tool that can return the clones in sub-second, for a large-scale code corpus archiving 492 open source projects (54 million LOC, 1.7 million code pieces), achieving 40-fold speedup over the existing tool DECKARD [12], without compromising any accuracy. Full technical details and extensive evaluation results can be found at [3].

II. ALGORITHM DETAIL

This section discusses our indexing structure and algorithms supporting code clone searches. To support efficient structural similarity comparison between code pieces, we adopt the vector abstraction used in DECKARD [12] to abstract syntactic information of code, using Abstract Syntax Tree (AST) obtained from parsing. Specifically, DECKARD approximates the similarity notion of code by representing AST as multi-resolution numerical vectors, called *characteristic vectors*. Each tree node in AST is represented as a vector representing the frequency of syntactic elements in the code piece represented by its subtree. Definition 1 and 2 formally define code pieces and characteristic vectors.

Definition 1 (Code pieces): Given a code S, its AST T, and a threshold minT, if a subtree T_i of T contains at least minT nodes, then T_i's corresponding part in S is a code piece.

Definition 2 (Characteristic vectors): Given a code piece S_i and the AST T_i of S_i, the characteristic vector $v_i = \langle c_{i(1)}, c_{i(2)}, \cdots, c_{i(d)} \rangle$ of S_i consists of occurrence counters $c_{i(j)}$ of syntactic elements in T_i.

With this representation, we use L_2 norm as our distance metric between characteristic vectors.

Definition 3 (Distances between vectors): Given two d-dimensional vectors v_1 and v_2, the distance $\|v_1, v_2\|$ is their L_2-norm, $\|v_1, v_2\| = \sqrt{\sum_{i=1}^{d} (c_{1(i)} - c_{2(i)})^2}$.

Finding clones of a given query code piece can be viewed as finding k-nearest neighbors (kNNs) [14] of a query point, which abstracts the given code piece, as formally defined in Definition 4.

Definition 4 (Top-k code clones): Given a set $\mathcal{V}$ of characteristic vectors, a query vector q, and the retrieval size k, top-k clones $\mathcal{TC}_k(q) \subset \mathcal{V}$ is a set of vectors $\mathcal{TC}_k(q) = \{v_1, v_2, \cdots, v_m\}$, where v_i is the i^{th} closest vector from q, $m \geq k$, and $\|q, v_i\| = \|q, v_k\|$ for $\forall i$ satisfying $k < i \leq m$.

Then our goal is to retrieve a set $\mathcal{TC}_k(q)$ of a query code piece q, and $\mathcal{TC}_k$ is used as its shorthand. For notational simplicity, we use $\mathcal{TC}_k$ to represent both code clones and their corresponding vectors interchangeably.

However, these characteristic vectors, $\mathcal{V}$, are generally too high-dimensional to be indexed directly using a multidimensional index, *e.g.*, `R*tree`. To overcome this, we discuss a dimensionality reduction technique.

A. Dimensionality Reduction

For a given set $\mathcal{V}$ of $\mathcal{D}$-dimensional N characteristic vectors $\{v_1, v_2, \cdots, v_N\}$, our goal in dimensionality reduction is to generate lower-dimensional vectors $\mathcal{V}' = \{v'_1, v'_2, \cdots, v'_N\}$, which satisfy the *lower-bounding property* [15], and make our algorithm efficient.

As formally proved in [15], to ensure that we can retrieve candidates including all of the correct k results, by searching the reduced space only, it is important to preserve the lower-bounding property. Formally, for all v_i and $v_j \in \mathcal{V}$, and their corresponding reduced vectors v'_i and v'_j, the distances measured in the original space and the reduced space should satisfy $\|v'_i, v'_j\| \leq \|v_i, v_j\|$.

We can trivially show that selecting any $\mathcal{D}'$-dimensional subspace of the original $\mathcal{D}$-dimensional space ensures the lower-bounding property. However, not all such subspaces are equally effective. A desirable subspace should reflect the original distances between vectors, we thus choose a subspace, which preserves the distance relations between vectors as much as possible.

Formally, we minimize the sum Δ of differences $\delta_{i,j}$,

$$\Delta = \sum_{\forall i, \forall j, i \neq j} \delta_{i,j} = \sum_{\forall i, \forall j, i \neq j} \|v_i, v_j\| - \|v'_i, v'_j\|,$$

between two distances measured at the original space and the subspace respectively. As finding such subspace is known to be NP-hard [16], we use an approximation of selecting the top-$\mathcal{D}'$ dimensions with the highest variances, by computing the variances of all dimensions.

B. Filtering-then-Ranking Query Processing: `FrTCD`

After the dimensionality reduction, building an `R*tree` index of the reduced vectors and adopting a *filtering-then-ranking* approach, named `FrTCD`, could be a solution of our problem. To illustrate, let v be the characteristic vector of the given code segment, and v' be the reduced version of v. In the *filtering* phase, we traverse the `R*tree` index to obtain kNNs of v', in a best-first manner, and select the farthest one of these kNNs, in the original space. ε denotes the distance to the selected farthest vector, measured in the original space. We then retrieve all vectors C within range ε from v' through the `R*tree`. In the *ranking* phase, we rank the vectors in C

by their actual distances from v to find final k most similar clones. This candidate set C is guaranteed to have all kNNs of v, as formally proved in [15].

C. Interleaved Query Processing: `InTCD`

We propose another algorithm `InTCD`, enhancing `FrTCD` to reduce random access cost, by (1) "packing" vectors before building `R*trees` and (2) combining two index traversals of `FrTCD` into a single traversal.

1) Vector Packing & Index Building: Briefly, when building an index, we pack vectors into blocks, where each block contains a group of nearby vectors, more precisely, raw data records of them. We then build an `R*tree` of these blocks in the reduced space, which enables to reach multiple records with a single random access followed by cheaper sorted accesses, which incurs significantly lower cost than performing a random access per each record.

For one-dimensional data, this packing can be implemented straightforwardly, by storing raw data records in the same order as the index key. However, for multidimensional data, it is non-trivial to identify an effective one-dimensional sorted order to store records. We thus revised a data partitioning scheme proposed in [17], to apply to this packing process, as well as index building.

To build `R*trees`, we developed **workload-aware bulk loading** optimized for code data and search workload. Existing bulk loading algorithm, such as STR [17], recursively subdividing each dimension into the same number of slices. Meanwhile, for code data, the variance of each dimension differs significantly, *e.g.*, in one dimension, points are highly clustered in a small range, while in another, points are well scattered. In such dataset, baseline STR partitioning will generate "non-square" rectangles. This incurs high I/O cost for nearest neighbor search (of finding squared area). Our revised partitioning policy thus tries to partition the dataset to render more "squared" rectangles. Formally, for a D-dimensional dataset containing N points, we subdivide the i^{th} dimension into $s_i = \lceil r_i/R \rceil$ slices, where r_i is the value range, computed as the difference between the maximum and minimum values of the i^{th} dimension. In other words, a dimension with a high r_i is highly scattered. Assuming points are uniformly scattered, R is computed as $\prod_{i=1}^{D} \frac{r_i}{R} = \frac{N}{C}$, where C is a predefined parameter, the maximum number of items in one partition. This partitioning policy is used for bulk loading, as well as the vector packing process.

2) Single Index Traversal: Recall that `FrTCD` performs two traversals– first, to find the kNNs in the reduced space, and second, to perform a range search to find candidates. We now extend this algorithm to perform a **single traversal** combining these two traversals into one, by concurrently accessing raw records "during" the index traversal.

Specifically, `InTCD` traverses the index in the reduced space in a best-first manner, similar to `FrTCD`. However, during the traversal, when a leaf entry is reached, `InTCD` accesses the raw data block pointed by the leaf entry, without waiting for the index traversal to complete. Whenever data records are accessed from the leaf entry, `InTCD` updates a sorted list, $\mathcal{TC}_k$, of the current known top-k clones, and we denote the current k^{th}-NN in the list as tc_k. tc_k can be used as a pruning boundary, as we can safely prune out both non-leaf and leaf entries that are farther than tc_k. As more data records are accessed, $\mathcal{TC}_k$ converges to the actual top-k results. Algorithm 1 formally describes this process of `InTCD`.

Algorithm 1: `InTCD` (q, k, T)

Input : query vector q, retrieval size k, `R*tree` T
Output: set $\mathcal{TC}_k$ of vectors of top-k clones
/* $tc_i \in \mathcal{TC}_k$ denotes the i^{th} nearest vector in $\mathcal{TC}_k$, from q */
1 $q' \leftarrow$ the reduced vector of q
2 $\mathcal{TC}_k \leftarrow \{\}$; $\mathcal{Q} \leftarrow \{\}$;
$\mathcal{H} \leftarrow$ {entries within the root of T}
3 **while** $\mathcal{H}$ *is not empty* **do**
4 $\quad e \leftarrow \mathcal{H}.pop()$
5 $\quad$ **if** $|\mathcal{TC}_k| < k$ *or* `mindist` $(q', e) \leq \|q, tc_k\|$ **then**
6 $\quad\quad$ **if** e *is not a leaf* **then** $\mathcal{H}.push$(children of e)
7 $\quad\quad$ **else**
8 $\quad\quad\quad \mathcal{Q}.push(e)$
9 $\quad\quad\quad$ **if** $|Q| > \mathcal{W}$ **then**
10 $\quad\quad\quad\quad E \leftarrow$ pop block pointers from $\mathcal{Q}$
11 $\quad\quad\quad\quad$ **for** *each* $v \in$ *a block of* E **do**
12 $\quad\quad\quad\quad\quad$ `UpdateClones` $(\mathcal{TC}_k, k, q, v)$;
13 **while** $\mathcal{Q}$ *is not empty* **do**
14 $\quad E \leftarrow$ pop block pointers from $\mathcal{Q}$
15 $\quad$ **for** *each* $v \in$ *a block of* E **do**
16 $\quad\quad$ `UpdateClones` $(\mathcal{TC}_k, k, q, v)$;
17 **return** $\mathcal{TC}_k$

Algorithm 2: `UpdateClones` $(\mathcal{TC}_k, k, q, v)$

Input : set $\mathcal{TC}_k$, retrieval size k, query q, vector v
/* $tc_i \in \mathcal{TC}_k$ denotes the i^{th} nearest vector in $\mathcal{TC}_k$, from q */
if $|\mathcal{TC}_k| < k$ **then** $\mathcal{TC}_k \leftarrow \mathcal{TC}_k \cup \{v\}$
else if $|\mathcal{TC}_k| \geq k$ *and* $\|q, v\| \leq \|q, tc_k\|$ **then**
$\quad \mathcal{TC}_k \leftarrow \mathcal{TC}_k \cup \{v\}$
$\quad$ remove $\forall tc_i \in \mathcal{TC}_k$ farther than tc_k from q

To implement this single-scan best-first search, a min heap $\mathcal{H}$ of e is maintained in the ascending order of `mindist` (q', e), where q' denotes the reduced vector of query q, e is an entry of the `R*tree` index, and `mindist` (q', e) denotes the shortest distance between q' and e. At the beginning, the entries within the root of T are pushed into $\mathcal{H}$ (Line 2). Then iteratively, the entry e in $\mathcal{H}$ with the minimal `mindist` (q', e) is processed. If the `mindist` (q', e) is no farther than the distances of the current tc_k to q, we continue the iterations. Otherwise, we can safely ignore e (Line 5). If

$\texttt{mindist}\,(q', e) \leq \|q, tc_k\|$, we test if e is a leaf entry or not. If e is not a leaf, then the entries within its child node are pushed into $\mathcal{H}$ (Line 6). Otherwise, we process the raw data block pointed by e.

III. Demonstration

We will demonstrate our tool over a real-life `java` code repository of 492 Java open source projects hosted on SourceForge, Tigris.org and GoogleCode. This corpus repository contains contains 288,846 `java` files (54,709,384 lines). For technical details and extensive performance analysis, refer to our technical paper [3].

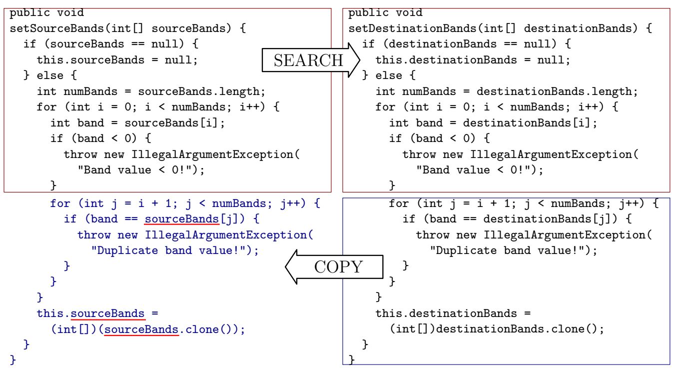

Fig. 2. Example 1: Exception throwing

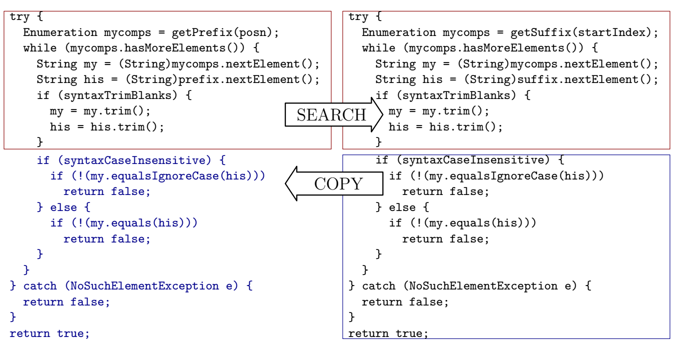

Fig. 3. Example 2: Try & catch exception handling

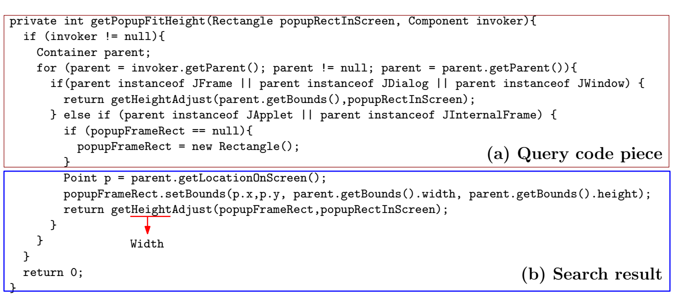

Fig. 4. Example 3: Applet user interface

The goal of this demonstration is to show how our proposed system can support rapid development. Toward this goal, we demonstrate real-life scenarios in Fig. 2, 3 and 4, using JDK code corpus. In Figure 2 and 3, users writing exception handling may hit search button, to find similar code pieces, to copy the rest (with only few modifications). In Figure 4, a user developing UI-related code finds a clone of the developed code piece (a), then copies the rest with only modifying a single word (b).

IV. Conclusion

In this paper, we propose to demonstrate our tool enabling instant code search during development editing sessions. As supporting technology, we design efficient index building and traversal techniques, which enable sub-second response time in a large-scale real-life code corpus with 1.7 million code pieces.

Acknowledgment

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2010-0001728).

References

[1] J. Brandt, M. Dontcheva, M. Weskamp, and S. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *SIGCHI*, 2010.

[2] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *ISESE*, 2004.

[3] M.-W. Lee, J.-W. Roh, S. won Hwang, and S. Kim, "Instant code clone search," in *ACM SIGSOFT/FSE*, 2010.

[4] J. Kim, S. Lee, S. won Hwang, and S. Kim, "Adding examples into java documents," in *ASE*, 2009.

[5] ——, "Towards an intelligent code search engine," in *AAAI*, 2010.

[6] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 187–196, 2005.

[7] E. Jürgens, B. Hummel, F. Deissenboeck, and M. Feilkas, "Static bug detection through analysis of inconsistent clones," in *Software Engineering (Workshops)*, 2008, pp. 443–446.

[8] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM*, 1998.

[9] V. Wahler, D. Seipel, J. W. v. Gudenberg, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *SCAM*, 2004.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[11] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder," in *ICSE*, 2007.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007.

[13] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[14] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM TODS*, vol. 24, pp. 265–318, 1999.

[15] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas, "Fast nearest neighbor search in medical image databases," in *VLDB*, 1996.

[16] H. Liu and H. Motoda, *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, 1998.

[17] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez, "STR: A simple and efficient algorithm for r-tree packing," in *ICDE*, 1997.