2015 30th IEEE/ACM International Conference on Automated Software Engineering

An Automated Framework for Recommending Program Elements to Novices

Kurtis Zimmerman and Chandan R. Rupakheti
Department of Computer Science and Software Engineering
Rose-Hulman Institute of Technology
Terre Haute, Indiana 47803
{zimmerka, rupakhet}@rose-hulman.edu

***Abstract*—Novice programmers often learn programming by implementing well-known algorithms. There are several challenges in the process. Recommendation systems in software currently focus on programmer productivity and ease of development. Teaching aides for such novice programmers based on recommendation systems still remain an underexplored area. In this paper, we present a general framework for recognizing the desired target for partially-written code and recommending a reliable series of edits to transform the input program into the target solution. Our code analysis is based on graph matching and tree edit algorithms. Our experimental results show that efficient graph comparison techniques can accurately match two portions of source code and produce an accurate set of source code edits. We provide details on implementation of our framework, which is developed as a plugin for Java in Eclipse IDE.**

***Keywords*—Recommendation Framework, pq-Gram Algorithm**

I. Introduction

Learning *how to program* can be a challenging endeavor to many novices. Past research has exposed many barriers to learning programming languages, APIs, and frameworks [15, 21]. Novices often think about problems in the *situation model*, which is an imprecise mental model of the problems. Translation of problems in the *situation model* to a precise *system model* (represented by artifacts such as source code) often requires external help [7].

Given a programming problem, a novice may not know how to break it into meaningful pieces (*design barrier*). Even if he breaks the problem properly, he may not know which programming elements to choose to get the desired result (*selection barriers*). After selecting programming elements, he may not know how to make them work together (*coordination barriers*) or how to use them correctly (*use barriers*). Assuming he was successful in all of these steps, he may now wonder why it did not work as expected (*understanding barriers*). Even if he was able to form an idea as to why the algorithm did not work, he may not know how to check the internal properties of the program to validate his idea (*information barriers*). Ko et al. identify these learning barriers as some of the central challenges in programming software systems [15]. Fischer, on the other hand, argues for a software framework consisting of the necessary toolset to overcome the barriers between the *situation model* and the *system model* [7].

As instructors of introductory programming courses for several years, we have experienced that example-based learning [1] and few coaching sessions can help overcome the *design* and *understanding* barriers to some extent. There are debugging tools [13, 17] to help with *information* barriers that work well. The *selection*, *coordination*, and *use* barriers, however, present a unique challenge as they require constant guidance from instructors. Due to the lack of time, instructors may fail to provide such guidance to novices.

Search-based tools [3, 9, 10] may help novices to some extent, however, past research has shown that novices may not know how to formulate the right queries to get meaningful help from such tools [14]. An automated tool integrated with a novice's development environment that could read his code, recognize the pieces of algorithms being implemented, and recommend a set of source code edits to achieve the correct solutions iteratively seems ideal in this context. This work is an effort in that direction.

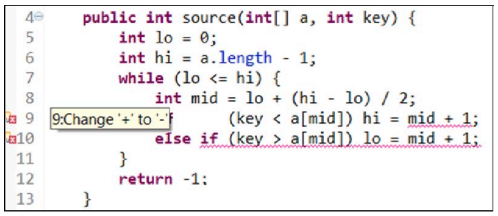
We make the following key contributions: i) given a source and multiple target implementations, we present algorithms for recognizing the best matching target implementation, ii) given a source and target implementation, we present algorithms for evaluating edit recommendations, and iii) we present a non-invasive visualization of edit recommendations integrated to a novice's development environment.

In the rest of the paper, we present a typical use case of the framework in Section II, discuss related works in Section III, present the framework's design in Section IV, discuss results in Section V, identify limitations and discuss future work in Section VI, and finally, conclude the paper in Section VII.

II. A Motivating Example

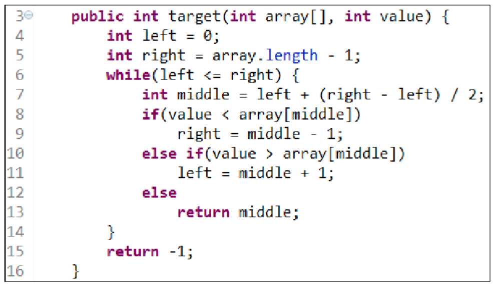
Let's assume that an instructor wants to use our framework and has all of her students install our plugin in their Eclipse IDE. The following is the typical use case:

1) The instructor will create programming problems.
2) She will create/find different implementations of the programming problems (Java files) and supply it to the students' IDEs as knowledge bases. Note that the current prototype stores files locally. This process can be improved by having a remote server host the encrypted version of target implementations to discourage students from directly copying the code.
3) Among other relevant help, she will provide students some guidance in designing the solutions.
4) A student writes some code for a problem in the set. When he gets stuck, he will press the help button of the framework. The framework will read his code, search for the best matching implementations in the knowledge base, and recommend the next set of edits towards the best matched solution using error markers (see Figure 1). In Figure 1a, two recommendations



```
4   public int source(int[] a, int key) {
5       int lo = 0;
6       int hi = a.length - 1;
7       while (lo <= hi) {
8           int mid = lo + (hi - lo) / 2;
9   9:Change '+' to '-'f   (key < a[mid]) hi = mid + 1;
10          else if (key > a[mid]) lo = mid + 1;
11      }
12      return -1;
13  }
```

(a) Source code written by a novice



```
3   public int target(int array[], int value) {
4       int left = 0;
5       int right = array.length - 1;
6       while(left <= right) {
7           int middle = left + (right - left) / 2;
8           if(value < array[middle])
9               right = middle - 1;
10          else if(value > array[middle])
11              left = middle + 1;
12          else
13              return middle;
14      }
15      return -1;
16  }
```

(b) Target code provided by an instructor

Fig. 1: The framework offering recommendations using markers based on the best matching target implementation.

are presented: one for substituting the '+' operator at line 9 and another for the missing return statement immediately after line 10.

5) Step 4 repeats until he gets the solution right.

In this way, novices will work independently while still getting help to overcome their learning barriers. It is possible that students may rely heavily on the framework rather than solving the problem themsleves. To discourage such situtation, as a future work, the framework will allow configurations that throttle recommendations based on the frequency of requests.

III. Related Work

Recommendation Systems: Recommendation systems for software engineering extract vital information from most often the source code or object code to provide help with software engineering tasks in a given context [24]. Several tutoring systems have been developed in the past that automate generation of hints to help novices based on their code [16, 18, 23, 27]. However, the generic hints produced by such systems can be hard to translate to actual code for novices. Unlike theirs, our framework provides code-specific edits.

Programming Tools: There are tools that focus on searching for API methods [3, 9], understanding example code [22], critiquing API client code [8, 25], and analyzing and debugging programs [17]. Other efforts include auto-completion tools within IDEs to recommend API methods [4, 12]. A related tool, Strathcona, uses structure of code and different matching heristics such as inheritance relation, method calls, and types of objects declared in a method body to find matching examples in a repository [11]. Our framework complements theirs as we focus on programming constructs rather than higher-level API methods.

Syntax Tree Matching: We use Abstract Syntax Tree (AST) provided by Eclipse's Java Development Tools (JDT[1]) to perform comparisons between source and target code. Note that in syntax trees matching, the use of different names for the same variable between source and target code can be an issue. A node mapping isomorphism technique can be used where, for example, all instances of the variable i in the source is mapped to the variable p in the target program [19]. The algorithm stops after finding the first structural difference, but our problem requires all differences.

Falleri et al. propose a method of source code differencing using AST called the Gumtree algorithm [6]. Given two ASTs, the algorithm uses two phases to produce a mapping of matching (or minimum difference) subtrees from one tree to another that can be fed into an algorithm to compute an edit script. Gumtree is similar to our pq Gram-based algorithm in that it is focused on producing fine-grained, realistic edit sequences. Gumtree has a worst-case complexity of $O(n^2)$, where n is the number of nodes in the larger tree and assumes the source and target are known. For a tool whose goal is to determine the target and determine edits, this algorithm could only feasibly be used after the closest target is determined.

Graph Matching: A method for fast exact graph matching is proposed by Etheredge, making use of adjacency matrices [5]. However, the given algorithm does not take node labels into account and only performs subgraph matching. We need the best overall match between two graphs. Seeking an approach to match source code and extract edits in one pass, we explored the VF graph isomorphism algorithm [20]. A drawback of the VF technique is that it does not ensure an optimal mapping between the two given trees. An objective function would have to be introduced for optimal mapping. Determining this objective value is the essential problem we are trying to solve, thus, making this method ineffective.

We took inspiration from the similarity matrix based graph matching technique [30] to develop a preliminary algorithm for comparing graphs. In the algorithm, we compare the label as well as total number of incoming and outgoing edges of each node in the two graphs. This algorithm runs in $O(n^3)$ time and takes $O(n^2)$ space. We switched to a more efficient pg-Gram based algorithm [2] discussed shortly.

Tree Edit Distance: Tree edit distance refers to the minimal number of node insertions, node deletions, and node relabelings required to transform a given tree T into a desired target T', as originally detailed by Tai as an extension of the string edit problem [28]. If V and V' are the number of nodes in trees T and T', respectively, and L and L' are their respective maximum depths, Tai presents a node-mapping approach which computes the edit distance in $O(V \cdot V' \cdot L^2 \cdot L'^2)$ time. Zhang and Shasha improved on the time complexity of Tai's approach by eliminating certain subtree distance calculations to $O(V^2 \cdot \log^2(V))$, where V is the number of nodes in the larger of the two trees [26].

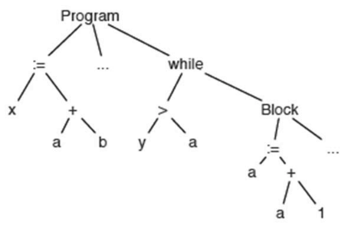
An approximate but faster version of the tree-edit distance algorithm called pq-Gram is developed by Augsten et al. [2]. The algorithm considers both labels and overall structure of the trees during comparison. It breaks the given trees into smaller structures and compares the two for similarities. We use pq-Gram to identify the best matching target code.

Figure 2 illustrates how pq-Gram profiles are created from the given source code. The granularity at which a tree can be broken down is given by the values p and q, which are the number of non-leaf and leaf nodes in the smaller structure, respectively (Figure 2c). The algorithm adds null (*) nodes

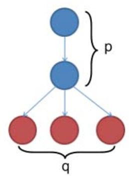
[1]http://www.eclipse.org/jdt/. All URLs verified on 5/13/2015.

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b;
}
```

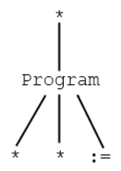
(a) Sample code



(b) AST for the sample code



(c) pq-Gram structure



(d) First pq-Gram for the root node

*	P	*	*	:=
*	P	*	:=	...
*	P	:=	...	W
*	P	...	W	*
*	P	W	*	*
P	:=	*	*	x
P	:=	*	x	+
P	:=	x	+	*

(e) Start of profiles for the AST

Fig. 2: Translation of source code to pq-Gram profiles (P = Program and W = while).

to break each tree into all of its subtrees of the same shape. Figure 2d is the first pq-Gram corresponding to the root node of the AST in Figure 2b. Similarly, Figure 2e is the beginning of the profile for the AST. Given two trees, the pq-Gram distance between the two is a value between 0 (identical trees) and 1 (no similarity). The algorithm traverses each tree by visiting each node just once taking $O(n)$ time and $O(n)$ space. The end result relies on a multiset intersection that can be computed in $O(n \log n)$ time, which is the bottleneck of the algorithm [2]. pq-Gram is much more efficient than the similarity matrix based approach in [30] and provides a good approximation of the distance between the two trees.

IV. Framework Design

Our framework can be divided into three primary modules: i) Target Recognition, ii) Edit Recommendation, and iii) Presentation via User Interface. We have already demonstrated the presentation part of the framework in Section II. We explore the former two modules in this section.

A. *Target Recognition*

For each target code in the knowledge base, the pq-Gram distance from the user's code to the target code is computed. The target code corresponding to the smallest pq-Gram distance is selected for further processing. If the minimum pq-Gram distance exceeds a threshold of certainty, the system cannot guarantee reasonable recommendations. Similarly, if multiple targets correspond to the same pq-Gram distance, then the framework non-deterministically chooses one.

We present a necessary size criterion for target code to narrow the search scope early. Only targets that satisfy this criterion are selected for computing pq-Gram distances. Assume that I_i corresponds to the pq-Gram profile of tree T_i.

Size Criterion Theorem: *Given trees T_1 and T_2 with $|I_1| \leq |I_2|$ and matching threshold r, if $d = dist(T_1, T_2)$ is such that $0 \leq d \leq r \leq 1$, then $|I_1| \geq \frac{1-r}{1+r}|I_2|$.*

Proof: In [2], the pq-Gram distance is defined as

$$dist(T_1, T_2) = 1 - 2\frac{|I_1 \cap I_2|}{|I_1 \cup I_2|}.$$

Note that in any case, using multisets means $|I_1 \cup I_2| = |I_1| + |I_2|$. In the best case scenario, to maximize $|I_1 \cap I_2|$, if $I_1 \subseteq I_2$, then $|I_1 \cap I_2| = |I_1|$, so we can make the simplification

$$dist(T_1, T_2) = 1 - 2\frac{|I_1 \cap I_2|}{|I_1 \cup I_2|} = 1 - 2\frac{|I_1|}{|I_1| + |I_2|}.$$

Then, if r is the threshold such that $0 \leq r < 1$, we want distance d to be such that $0 \leq d \leq r$, so we must have

$$\begin{array}{rcl} r & \geq & 1 - 2\frac{|I_1|}{|I_1|+|I_2|} \\ \Rightarrow \quad |I_1| & \geq & \frac{1-r}{1+r}|I_2| \text{ (after simplification).} \end{array}$$

Hence, pre-computing the size of pq-Gram profiles of targets in the knowledge base can significantly improve the performance of this step. At the conclusion of this step, ASTs T_s (source) and T_t (target) have been determined, where T_t corresponds to the closest matching target AST.

B. *Edit Recommendation*

Once the target code has been identified, the next step is to provide the user with a set of recommendations to transform their code into the target implementation. Existing solutions suffer from a debilitating runtime as they rely on a node-to-node mapping technique (graph isomorphism [20]), which in the worst case runs in exponential time.

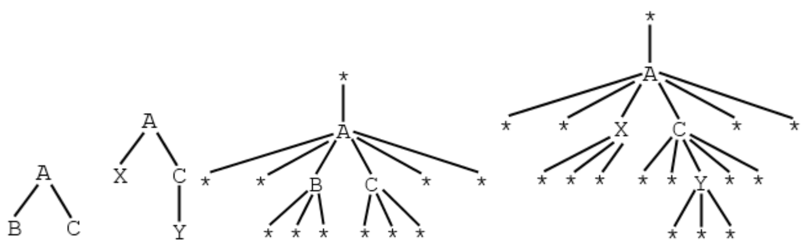
In order to provide a reasonable experience for the user and maintain computational efficiency, we developed a new recommendation algorithm based on the pq-Gram profiles obtained earlier. The algorithm has four steps: acquiring pq-Gram profiles (Section IV-B1), building common subtrees (Section IV-B2), identifying naive edits (Section IV-B3), and finally, minimizing insertions and deletions (Section IV-B4).

1) Acquiring pq-Gram Profiles: The edits rely on information from the pq-Gram profiles, I_s (source) and I_t (target), which are computed in the previous stage (Section IV-A). To help understand all of the steps involved in edit recommendations, let's use a running example of Figure 3. The algorithm will be carried out with T_s and T_t shown in Figure 3a and 3b, respectively. The pq-Gram algorithm begins by extending each tree with null nodes to ensure each node has the necessary number of siblings and children (Figure 3c and 3d). Then, using the algorithm described in [2], the profiles corresponding to T_s and T_t are computed (Figure 3e and 3f). At this point, we proceed to building common subtrees.

2) Building Common Subtrees: By building the common subtrees, a considerable number of nodes and edges are removed and not considered for further edits. Note that the pq-Gram tuples are of the form $(a_1, a_2, ..., a_p, c_1, c_2, ..., c_q)$, where a_i is the parent of a_{i+1}, and a_p is the parent of each c_i. Algorithm 1 takes advantage of this form by first, adding all nodes as roots of common subtrees and then, by removing them once they have been added as children to another common node. At the end, B_S holds the set of roots of the common subtrees of T_s and T_t.

For our example in Figure 3, $C_S = \{(*, A, C, *, *)\}$. Hence, $B_S = \{A\}$ and $A \rightarrow C$ is the only common subtree.

3) Identifying Naive Edits: The simplest set of edits is to delete all extra nodes (those not in the common subtrees) in T_s and subsequently insert all missing nodes (those not in the common subtrees) in T_t (Algorithm 2). Any nodes that show up in the extra tuples (E_S) and not in the common tuples (C_S)



(a) T_s (b) T_t (c) Extended T_s (d) Extended T_t

*	A	*	*	B
A	B	*	*	*
*	A	*	B	C
A	C	*	*	*
*	A	B	C	*
*	A	C	*	*

(e) I_s

*	A	*	*	X
A	X	*	*	*
*	A	*	X	C
A	C	*	*	Y
C	Y	*	*	*
A	C	*	Y	*
A	C	Y	*	*
*	A	X	C	*
*	A	C	*	*

(f) I_t

Fig. 3: A running example for explaining the edit recommendation algorithm.

Algorithm 1 Building common subtrees of T_s and T_t

```
Require: pq-Gram profiles $I_s$, $I_t$
Ensure: Set $B_S$ is roots of common subtrees of $T_s$ and $T_t$
  $C_S \leftarrow I_s \cap I_t$
  $B_S \leftarrow \{\}$
  for all $(a_1, a_2, ..., a_p, c_1, c_2, ..., c_q) \in C_S$ do
    $B_S \leftarrow B_S \cup \{a_1\}$
    $B_S \leftarrow B_S - \{a_2, ..., a_p, c_1, c_2, ..., c_q\}$
  end for
```

must be deleted, while all nodes that show up in the missing tuples (M_S) and not in C_S must be inserted. Note that this set of edits is a correct solution, but not an optimal one.

Algorithm 2 Identifying insertions and deletions

```
Require: pq-Gram profiles $I_s$ and $I_t$
Ensure: $A_S$ is the set of insertions, $R_S$ is the set of deletions
  $A_S \leftarrow \{\}$, $R_S \leftarrow \{\}$
  $C_S \leftarrow I_s \cap I_t$
  $M_S \leftarrow I_t - C_S$
  $E_S \leftarrow I_s - C_S$
  for all $(a_1, a_2, ..., a_p, c_1, c_2, ..., c_q) \in M_S$ do
    $A_S \leftarrow A_S \cup \{(a_{i-1}, a_i)\}$ for $2 \le i \le p$ if $a_i \notin C_S$
    $A_S \leftarrow A_S \cup \{(a_p, c_i)\}$ for $1 \le i \le q$ if $c_i \notin C_S$
  end for
  for all $(a_1, a_2, ..., a_p, c_1, c_2, ..., c_q) \in E_S$ do
    $R_S \leftarrow R_S \cup \{(a_{i-1}, a_i)\}$ for $2 \le i \le p$ if $a_i \notin C_S$
    $R_S \leftarrow R_S \cup \{(a_p, c_i)\}$ for $1 \le i \le q$ if $c_i \notin C_S$
  end for
```

Returning to our example in Figure 3, recall that $C_S = \{(*, A, C, *, *)\}$. Missing tuples ($M_S$) and extra tuples ($E_S$) are shown in Figure 4a and 4b, respectively. Notice that the missing components (in M_S) are the relationships $A \to X$ and $C \to Y$. We now add these to the set of insertions (A_S). Likewise, the extra component (from E_S) is $A \to B$. We add it to the set of deletions (R_S). At the end of Algorithm 2, $A_S = \{(A, X), (C, Y)\}$ and $R_S = \{(A, B)\}$.

4) Minimizing Insertions and Deletions: We now introduce relabelings, as shown in Algorithm 3, in an effort to minimize the number of insertions and deletions generated by Algorithm 2. There are two cases for which an insertion or deletion pair can be removed:

1) A node a is being deleted from x, and a node b is being inserted onto x. In this case, the insertion and deletion can be replaced by a single relabeling, mapping a to b.

*	A	*	*	X
A	X	*	*	*
*	A	*	X	C
A	C	*	*	Y
C	Y	*	*	*
A	C	*	Y	*
A	C	Y	*	*
*	A	X	C	*

(a) Missing Tuples (M_S)

*	A	*	*	B
A	B	*	*	*
*	A	*	B	C
A	C	*	*	*
*	A	B	C	*

(b) Extra Tuples (E_S)

Fig. 4: M_S and E_S, computed during Algorithm 2.

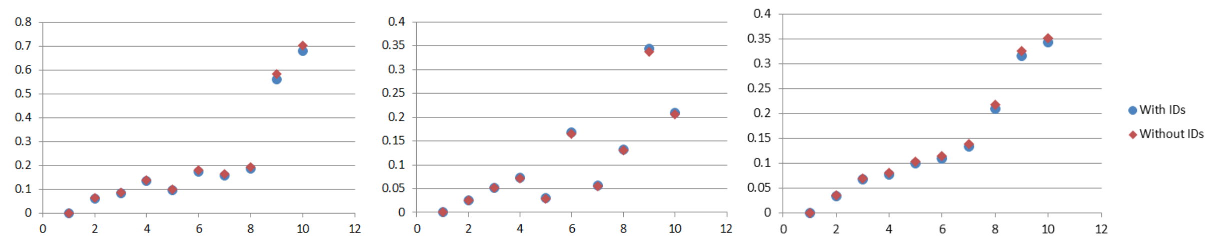
2) A node a is being deleted from x, and the node a is being inserted onto y, and x is being relabeled to y (i.e. (x, y) is in the set of relabelings).

Notice that in either case, the insertion and deletion can be removed, and only in the first case do we replace them with a relabeling. In the second case, the insertion and deletion are simply inverses of one another.

Algorithm 3 Minimizing insertions and deletions

```
Require: Set of insertions $A_S$ and deletions $R_S$
Ensure: $N_S$ is the set of relabelings, $A_S$ and $R_S$ the minimized sets of insertions and deletions
  $N_S \leftarrow \{\}$
  for all $(p_i, c_i) \in A_S$ do
    for all $(p_d, c_d) \in R_S$ do
      if $N_S(p_d) = N_S(p_i)$ then
        if $c_d \ne c_i$ then
          $N_S \leftarrow N_S \cup \{(c_d, c_i)\}$
        end if
        $A_S \leftarrow A_S - \{(p_i, c_i)\}$
        $R_S \leftarrow R_S - \{(p_d, c_d)\}$
      end if
    end for
  end for
```

At the end of Algorithm 3, we are left with three sets, one each for relabelings, insertions, and deletions. These three together represent the total set of edits. To finish our example in Figure 3, recall that $A_S = \{(A, X), (C, Y)\}$ and $R_S = \{(A, B)\}$. Because we must remove B from A and insert X onto A, those edits can be squashed into one by replacing them with a relabeling, $B \to X$, so $N_S = \{(B, X)\}$. By removing one insertion and one deletion, we are left with $A_S = \{(C, Y)\}$ and $R_S = \{\}$. Thus, the final set of edits can be described as follows: i) relabel B to X, and ii) insert Y onto C in Figure 3a. Starting from T_s, this sequence of edits will indeed match T_t. This concludes our discussion on recommendation



(a) Binary Search (b) Bogo Sort (c) Bubble Sort

Fig. 5: Graphs showing pq-Gram distances (vertical axis) for increasing code differences (horizontal axis).

algorithms. We now discuss an issue with variable names that needed special handling.

C. *Issue with Variable Names*

The source and target programs may use different names for the same variable. Since our approach utilizes node labels, we want to ensure the variable names have no (or minimal) impact on both the pq-Gram distance scores and the recommended edits. To address this issue, we slightly altered the AST parsing process. As the Java method is parsed, a mapping is constructed, where each unique identifier points to all of its nodes in the AST. We proceed through the recommendation algorithm as is. We then filter out the target code identifiers while maintaining the effect of the edits as follows. If i is a string identifier, let $M(i)$ represent the set of nodes corresponding to each use of i in the user's code, and let N be an empty map which will be used to keep track of equivalent variables between the source and target code. For each i, if there exists a relabeling of i corresponding to each node in $M(i)$, consider two cases:

1) If each relabeling maps i to the same i', then remove all of the relabelings associated with i and add $i' \to i$ to N.
2) Otherwise, for each relabeling $i \to i'$,
    a) If $i' \in N$, change the relabeling to $i \to N(i')$.
    b) Otherwise, if i' refers to a variable in the user's source code, keep the relabeling, as it refers to a variable that is in use.
    c) Otherwise, remove the relabeling, because the variable name is not of interest.

All of these algorithms work together to create an optimal set of edit proposals for the user. Since Eclipse provides support for mapping AST nodes to the source code, we are able to use the error notification markers of Eclipse to denote regions in the user's code where changes need to be made.

V. Results

Our framework relies heavily on the ability of the target recognition algorithm (Section IV-A) to react to the following situations appropriately:

1) As the differences between two fragments of code increases, the pq-Gram distance also increases.
2) The trend in pq-Gram distances should have minimal impact due to variable name differences (Section IV-C) in the source and target programs.

As a preliminary evaluation of our prototype, we focus on validating these two key technical assumptions. A thorough evaluation of edit recommendations (Section IV-B) needs user studies and is out of the scope of this paper.

To test these assumptions as well as the recommendation algorithm itself, a knowledge base of 32 different searching and sorting algorithms was built, each featuring a test suite of perturbed implementations. The test suites were divided into four complexity classes as follows: i) *Semantically equivalent:* syntactic differences, but semantically equivalent methods; ii) *Minimal:* edits are minimal, including at most a few relabelings; iii) *Moderate:* edits are moderate, including at least one insertion or deletion, but no more than one of each; and finally, iv) *Severe:* edits are more severe, including a pair of insertions or a pair of deletions.

For each test suite, two sets of the pq-Gram distances between the source and target ASTs were computed: one including identifiers and the other excluding identifiers. In Figure 5, we sample the three representative cases out of 32. The result shows that, in general, the pq-Gram distances increase as complexity increases. Secondly, the distance trends for ASTs with identifiers and without are similar, with small adjustments due to the number of AST nodes being reduced. The results are consistent across all 32 test suites.

Note that the trend in Figure 5 is not monotonically increasing due to one of the following two factors. First, the placement of edits relative to one another can impact their effect on the distance between trees. For example, two adjacent deletions can disrupt significantly more sibling relationships than removing two nodes far from one another. Second, the level of placement of edits affects their impact on the distance score. An edit at a lower level (particularly leaves) affects fewer pq-Grams than a higher-level, non-leaf edit. Due to the varying nature of the algorithms under test, the edits cannot be placed uniformly across all algorithms, resulting in non-uniform trends. Overall, our preliminary evaluation shows promising results as it validates our key technical assumptions.

VI. Limitations and Future Work

We now present some of the limitations of the framework and opportunities for future improvements.

Program Analysis: We use AST for our analyses. AST-based analyses work well even under compilation errors, which is critical for novices. Also, our framework can be quickly adapted to a new language that can be parsed into ASTs. However, AST-based analyses are intraprocedural (analysis of one method at a time). Use of interprocedural analyses (spans multiple methods) remains a future work.

Semantic Equivalence: One major pitfall of AST representations of source code is the loss of behavioral information.

Because syntactically different programs can behave equivalently, the AST-based approach toward finding differences will incorrectly classify many pieces of source code as different even though they accomplish the same task. A few semantic equivalences are reduced to a canonical form in our current prototype, such as normalizing the *for* loop to the *while* loop and normalizing operators in the form *x [op]= y* to *x = x [op] y*, but there are others that need special care. We have explored the possibility of using Control Flow Graphs (CFG) built on top of Jimple Intermediate Representation (IR) of SOOT [29] instead of AST. SOOT, to a certain extent, optimizes IR to have similar flow structures for semantically equivalent programs. However, compilation and de-compilation are asymmetric operations, and the recommendations based on IR do not map well with the original code. Semantic equivalence remains a key area to be explored in the future.

Usability Testing: So far our evaluation efforts have been directed towards validating the technical aspects of the framework. As a future work, we plan to conduct user studies to assess the quality of recommendations in a controlled environment. We also plan to evaluate the framework by using it as a part of our introductory programming courses.

VII. Conclusion

Novices may face several challenges while learning programming. Tooling efforts in software engineering have been more focused on helping programmers with higher-level programming tasks than with basic language-specific constructs. We present a framework that can help novices in their learning process by recommending specific code edits relevant to their problems within the Eclipse IDE. Our preliminary results establish technical feasibility of the framework. In the future, we plan to conduct user studies to test the usefulness of our prototype. We also plan to release the framework as an open source tool for both instructors and students.

References

[1] R. Anderson, R. Anderson, K. M. Davis, N. Linnell, C. Prince, and V. Razmov, "Supporting active learning and example based instruction with classroom technology," in *SIGCSE*, 2007, pp. 69–73.

[2] N. Augsten, M. Böhlen, and J. Gamper, "Approximate matching of hierarchical data using pq-grams," in *VLDB*, 2005, pp. 301–312.

[3] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *CHI*, 2010, pp. 513–522.

[4] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *FSE*, 2009, pp. 213–222.

[5] M. Etheredge, "Fast exact graph matching using adjacency matrices," in *PCG*, 2012, pp. 10:1–10:6.

[6] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014, pp. 313–324.

[7] G. Fischer, "Cognitive view of reuse and redesign," *IEEE Softw.*, vol. 4, no. 4, pp. 60–72.

[8] G. Fischer, A. C. Lemke, T. W. Mastaglio, and A. I. Mørch, "Using critics to empower users," in *CHI*, 1990, pp. 337–347.

[9] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *ICSE*, 2010, pp. 475–484.

[10] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: finding and leveraging implicit references in a web search interface for programmers," in *UIST*, 2007, pp. 13–22.

[11] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE*, 2005, pp. 117–125.

[12] D. Hou and D. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion," in *ICSM*, 2011, pp. 233–242.

[13] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *ICSE*, 2008, pp. 301–310.

[14] A. J. Ko and Y. Riche, "The role of conceptual knowledge in API usability," in *VL/HCC*, 2011, pp. 173–176.

[15] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *VL/HCC*, 2004, pp. 199–206.

[16] T. Lazar and I. Bratko, "Data-driven program synthesis for hint generation in programming tutors," in *ITS*, 2014, pp. 306–311.

[17] R. Lencevicius, U. Hölzle, and A. K. Singh, "Dynamic query-based debugging of object-oriented programs," *Automated Software Engg.*, vol. 10, no. 1, pp. 39–74.

[18] A. Mitrovic, "An intelligent sql tutor on the web," *Int. J. Artif. Intell. Ed.*, vol. 13, no. 2-4, pp. 173–197.

[19] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5.

[20] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372.

[21] J. Pane and B. Myers, "Usability issues in the design of novice programming systems," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-96-132, August 1996.

[22] D. F. Redmiles, "Reducing the variability of programmers' performance through explained examples," in *CHI*, 1993, pp. 67–73.

[23] K. Rivers and K. Koedinger, "Automating hint generation with solution space path construction," in *ITS*, 2014, pp. 329–339.

[24] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Soft.*, vol. 27, no. 4, pp. 80–86.

[25] C. R. Rupakheti and D. Hou, "CriticAL: A Critic for APIs and Libraries," in *ICPC*, 2012, pp. 241–243.

[26] D. Shasha and K. Zhang, *Pattern Matching in Strings, Trees and Arrays*. Oxford University Press, 1995, ch. 14.

[27] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *PLDI*, 2013, pp. 15–26.

[28] K.-C. Tai, "The tree-to-tree correction problem," *J. ACM*, vol. 26, no. 3, pp. 422–433.

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *CASCON*, 1999, pp. 125–135.

[30] L. A. Zager and G. C. Verghese, "Graph similarity scoring and matching," *Applied Mathematics Letters*, vol. 21, no. 1, pp. 86–94.