A Similarity Ranking of Python Programs

Jonathan Wardell Avery

under supervision of

Kourosh Neshatian
and
Walter Guttmann

October 16, 2015

Abstract

Detection of similar programs is a highly studied problem. Detecting similar code is an important strategy for detecting badly modularized code, finding vulnerabilities due to error prone copy-paste programming methodologies, and detecting academic dishonesty in online code assignment submissions following the copy-paste-adapt-it pattern. The latter is the impetus for this work.

A novel system is presented that is specifically adapted to programs that may be small, and similar by virtue of being written to solve the same problem. The system is also adapted toward specific expected behaviors of plagiarists, making use of algorithms custom built to both recognize these behaviors while satisfying hierarchical properties. A defining and novel property of the proposed method is the categorical information it provides. A hierarchy of categories with an implication relationship are leveraged in the production of descriptive, rank-able results.

Contents

Chapter 1

Introduction

1.1 Overview of The Field

Determining code similarity is not a new concept. Many systems have been proposed and implemented with this aim in mind. Examples include *DECKARD* [10] and *MOSS* [17].

Detecting code similarity can be optimized for different goals. Systems such as *DECKARD* detect code clones with the general aim of finding similar portions of code, whether this be to detect areas in need of refactoring or to detect areas that are possible security risks. *MOSS* is a system designed with an aim in line with our own: to detect clones in the context of finding plagiarism.

Different approaches have been taken to this problem. Some approaches focus on differences at the textual level, others detect differences at the token level, some detect differences at the abstract syntax tree level, and still others examine differences in lists of properties of the program.

It is important to note that systems working with different representations of the program have different information available to them. Many systems sacrifice some information in order to get to a different representation.

1.2 Research Goals

Our research aims to detect similar programs to help in recognizing plagiarism. The developed system must be optimized to work with programs that are expected to be somewhat similar by virtue of being written to solve the same problem. It must also be sensitive enough to work with programs that may be small in size. As such, it is essential that any useful information a program provides is not discarded prematurely, as by moving to a more general representation of the program.

It is desired that this system be effective (accurate in determining program similarity) and scalable. Effectiveness is valued more than scalability as this system's intended use is on datasets of limited size from Moodle.

There are a list of common types of plagiarism that we would like this system to be able to detect. It is important that the system be designed with these expected forms of plagiarism in mind. There are specific changes to a program that should not disguise the program from being matched to a clone. These changes are as follows.

Table 1.1:

Goals of The Clone Detector	
Changes That Must Not Deter Detection	Annotations/Notes
order of functions	the order of function definitions
order of statements	the order of statements within the program
order of parameters	the order of parameters to a function
changing identifier names	changing variable names
changing whitespace	addition or deletion of non-meaningful tabs, spaces, and newlines
changing layout	indentation level or tabs vs spaces
changing comments	deletion, insertion, or editing of comments (including docstrings)
replacing statements with equivalent statements	for example, a += 1 and a = a + 1
replacing expressions with equivalent expressions	for example, a = 1 + 1 and a = 2

1.3 Research Methodology

A system will be designed such that the expected forms of plagiarism can be detected and descriptively reported. This system will then be tested with real student submissions to analyze its effectiveness.

Chapter 2

Background

Two similar pieces of code are referred to as *code clones* [9, 16]. The goal of the proposed system will be to detect such code clones. A system for detecting code clones is referred to as a *clone detector*.

It has been suggested that a typical *clone detector* follows six phases in its operation [16]. These phases are *pre-processing*, *transformation*, *match detection*, *formatting*, *post-processing*, and *aggregation*.

In the *pre-processing* phase, the parts of the code that are not of interest in the comparison are removed. This is referred to as *normalization*, and some may also occur after the code has already in some way been transformed. The code is then separated into *source units*. These are the largest pieces of the source that are deemed to be comparable for similarity with each other. The source units may be further separated into *comparison units* of a differing size depending on the intended granularity of the comparison.

In the *transformation* phase, the code of the comparison units is then changed into the preferred representation for comparison. If the preferred representation is text, then a transformation may not be necessary. This transformation may also not be necessary if a transformation was already performed because the separation of the code into *comparison units* required it.

In the *match detection* phase, transformed comparison units are then used as input to the comparison algorithm to determine similarity.

In the *formatting* phase, the code clones are then mapped back to the original code by location.

In the *post-processing* phase, the code clones are visualized and the operator of the program is allowed to manually discern false positives. A false positive in this context is something the clone detector detects as a code clone, but is not.

In the *aggregation* phase, groups of code clones are gathered into *clone classes* or *families*.

2.1 Related Work

Previous work has described a wide range of approaches to similarity detection, including the following: evolutionary programming based [3], string-based [10], token-based [10], tree-based [10], semantics-based [10], instruction-based [1], bytecode-based [11], and metric-based [4, 7, 16]. These methods can be loosely divided into three categories: *sequence-based*, *graph-based*, and *metric-based* methods.

Sequence-based methods include: string-based, token-based, bytecode-based, and instruction-based methods. All of these methods retain the advantage of working with a sequence of values instead of more complex data structures. These methods are more amenable to a plethora of techniques deriving from well understood principles of information theory. It should be noted that detecting code similarity in intermediate code representations is less widely studied than working with the source code itself [11].

Graph-based methods include: tree-based and semantics-based methods. These methods sacrifice simplicity for a richer representation of the code. In methods working with sequences, metrics such as the *Kologmorov distance* and concepts such as *fast dynamic time warping* [1], *normalized information distance* [12], and *tf-idf* (term frequency–inverse document frequency) can be immensely useful. When working with trees, finding edit distances can be computationally demanding. In fact, efficient tree similarity detection still remains an open problem [10]. This can lead to a lack of scalability [2].

Metric-based methods find characteristics or metrics of code, then compare these metrics instead of the code itself [16]. Metric-based methods do not fit well into either of the general categories as these metrics could be attributes of a sequence or a graph.

2.2 Types of Clones

There is a hierarchy of clone types found in the literature. These categories can be seen to have a rough correspondence to our goals.

Roy [16] describes 4 types of clones. I will assign each a short name.

Table 2.1: Categories from Roy [16] with short names assigned

#	Name	Description
1	Near Identical	identical code fragments except for variations of whitespace, layout, and comments
2	Similar	syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout, and comments
3	Less Similar	copied fragments with further modifications as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments
4	Arguably Similar	two or more code fragments that perform the same computation but are implemented by different syntactic variants

Rattan [15] describes the following types of clones by name. Many of these types can be described as subcategories of the types listed by Roy [16]. As such, they will now be numbered by which group they appear to belong to in the previous table.

Table 2.2: Categories from [15] mapped to categories from [16]

#	Type	Annotations/Notes
1	exact	identical
2	renamed/parameterized	program fragments which are structurally/syntactically similar except for changes in identifiers, literals, types, layout, and comments
3	near miss clones	program fragments that have been copied with further modifications like statement insertions/deletions in addition to changes in identifiers, literals, types and layouts
4	semantic clones	functionally similar without being textually similar
4	structural clones	patterns of interelated classes emerging from design and analysis space at architecture level
4	model based clones	overlaps in models (graphical languages?)
4	function clones	clones which are limited to the granularity of a function/method or procedure
	file clones	n/a
4	contextual clones	clones that can only be found by augmenting code fragments with related information referenced by the fragment to give its context

Being as the clone categories of Rattan [15] can be characterized as subcategories of the clone categories from Roy [16], we will refer to clone categories according to Roy [16], but will use the short names I assigned them in *Figure 3*.

With these categories in mind, it can be seen that most of our goals are within categories 1-3 (Near Identical, Similar, and Less Similar), with only a few goals falling within category 4 (Arguably Similar).

2.3 Summary

Many of our goals seem to be most gracefully handled by *graph-based* clone detectors at the abstract syntax tree level.

Whereas a *sequence-based* clone detector can ignore variable names (tokens are categorized by Python), it is more difficult to be insensitive to change in order of parameters, and order of functions.

Some information is lost in the process of parsing which can be valuable in detecting collaboration, such as comments. Thus it would seem logical that our approach might be a hybrid method.

Chapter 3

Theory and Design of the Proposed System

The program space this system targets differs in several significant ways from the program space targeted by other systems. Firstly, the programs this system is designed to deal with are often small. They may consist of only one or two functions. Secondly, the programs this system will be used with are written to solve the same task, and as such are expected to be more similar to each other than programs taken from other problem spaces.

These two characteristics of this program space emphasize the need of the system to retain information as much as possible. This is important because information stripped from a program may be the only information that distinguishes it from another program.

If the system were to rank pairs of programs perfectly, this information could be more or less useful depending on how closely the programs match. If programs tend to match very closely, then it is possible that only programs that have exactly the same text or exactly the same AST are of interest. For this reason, this information is useful to the user of the system.

3.1 Hierarchy

It was realized that there are certain equivalence classes of programs that form a hierarchy that could be useful in providing information to the user of the system. For example, if two programs have an exact textual match, then they will also have an exact abstract syntax match. The use of this hierarchy, to the knowledge of this author, unique to this system.

Let $\boldsymbol{e}$ represent a comparison of the text of two programs and let $\boldsymbol{a}$ represent a comparison of the abstract syntax trees of two programs. We expect that either

two comparisons will yield a boolean result. If one comparison/relationship is an ancestor of another in the hierarchy of match categories, then that relationship implies the other. If we let $\boldsymbol{E}$ be the set of all pairs of programs that satisfy $\boldsymbol{e}$ and we let $\boldsymbol{A}$ be the set of all pairs of programs that satisfy $\boldsymbol{a}$, then all elements of $\boldsymbol{E}$ should be also in $\boldsymbol{A}$.

$$
\begin{aligned}
e(pair(x,y)) &: bool \\
a(pair(x,y)) &: bool \\
ancestor(e,a) &\Leftrightarrow (e(pair(x,y)) \rightarrow a(pair(x,y))) \\
E &= \{pair(x,y) : e(pair(x,y))\} \\
A &= \{pair(x,y) : a(pair(x,y))\} \\
thus & \\
ancestor(e,a) &\rightarrow E \subseteq A
\end{aligned}
$$

Figure 3.1: Mathematical Explanation of Heirarchical Relation

The lattice structure of this hierarchy is convenient in that if two programs satisfy an ancestor equivalence relation in the hierarchy, then the programs will also satisfy all progeny equivalence relations. If comparisons between programs are performed in order of most specific to least, then we can stop when a comparison is satisfied because the satisfaction of all less specific categories is guaranteed. The most expressive relationship obtainable is the most specific, and as such we should report this category as the category in which the match is found.

The properties these categories have in relation to each other are intentional, and not derived from a previous work.

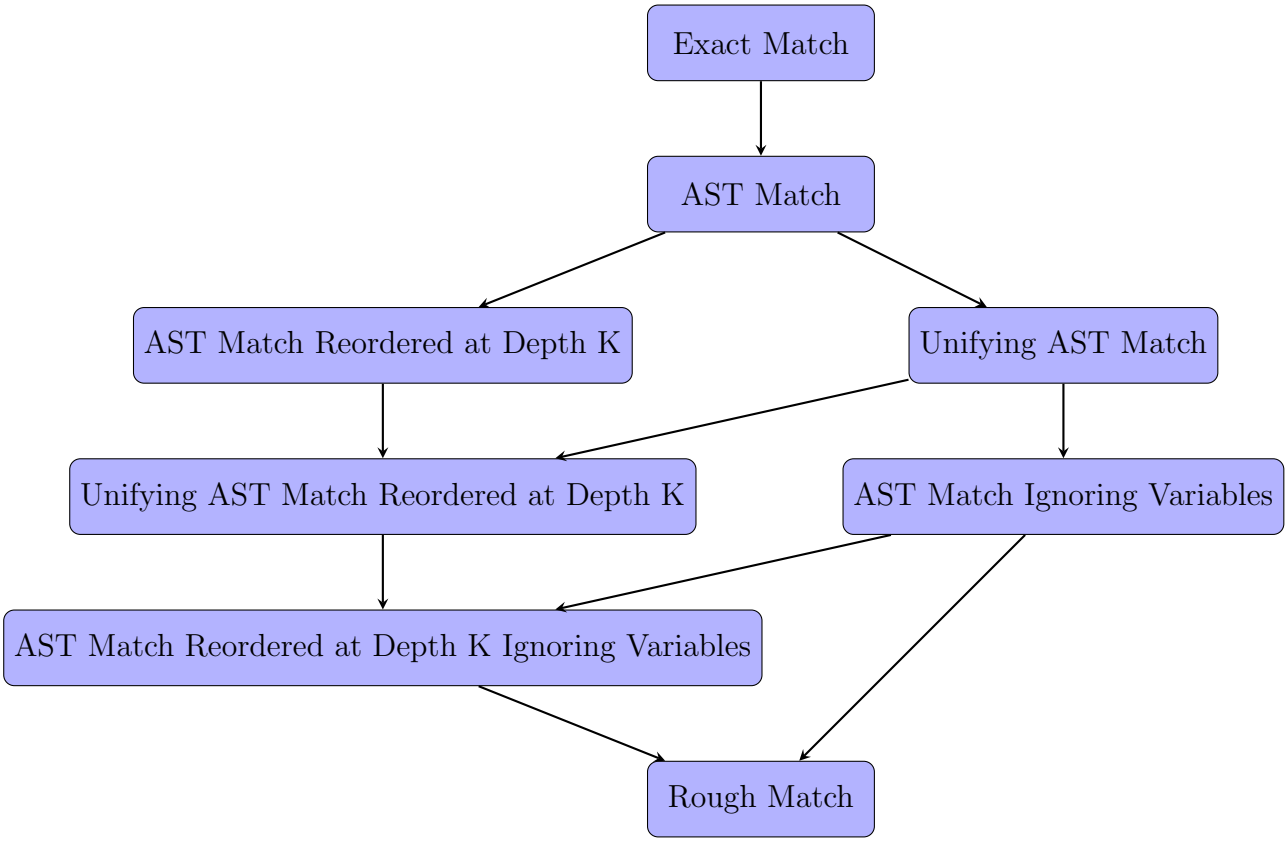

Figure 3.2: Hierarchy of Match Categories

3.1.1 Exact Match

The *exact match* is an exact match on the text of two programs. This is the most similar any two programs can be, and as such this is the most specific category of the hierarchy.

3.1.2 AST Match

The *AST match* is a match between the abstract syntax trees of two programs. This category disregards the whitespace differences that are not syntactically meaningful and comments of the two programs. Any two programs that are exact matches will also be AST matches. Thus the implication implied between the two relations in the hierarchy holds.

```
def dist(p1,p2):
  # distance between
  # two x,y points
  r1=(p1.x   -p2.x)**2
  r2=(p1.y-  p2.y)**2
  return sqrt(r1+r2)
```

```
def dist(p1,p2):
  r1=(p1.x-p2.x)**2
  r2=(p1.y-p2.y)**2
  return sqrt(r1+r2)
```

Figure 3.3: two programs sharing the same AST

Let $children(x)$ be the children of a node x of an abstract syntax tree. The children of a node x are ordered and denoted $x_0, x_1...x_n$. Let $x.type$ be the type of a node x in an abstract syntax tree, for example: "If", "$FunctionDef$", or "$Assign$". $x.id$ is the identifier, if any, of a node x of an abstract syntax tree. Identifiers are the names of variables and literals. The following pseudocode presents the algorithm for finding if there is an abstract syntax tree match between two abstract syntax trees. It takes two ASTs (abstract syntax trees) as arguments, and returns a boolean value.

procedure ASTMATCH(a, b)
  if $a.type \neq b.type$ **or** $a.id \neq b.id$ **or** $|children(a)| \neq |children(b)|$ **then**
    return $False$
  if $\forall_{0 \leq i < |children(a)|} ASTmatch(a_i, b_i)$ **then**
    return $True$
  else
    return $False$
return $True$

Figure 3.4: AST Match pseudocode

3.1.3 Unifying AST Match

```
def fact(x):
  if x==0:
    return 1
  else:
    return x*fact(x)
```

```
def factorial(nmb):
  if nmb  ==       0:
    return 1
  else:
    return nmb*factorial(
       nmb)
```

Figure 3.5: two programs matching under a Unifying AST Match

The *unifying AST match* is a match between the abstract syntax trees of two programs that finds a one-to-one mapping between the variable names of the two programs. This category can then allow programs in which all variables have been renamed in a naive fashion. The naming of this category is based on the unification algorithm [13].

The algorithm for determining this mapping involves the recursive walking of both ASTs and comparison of nodes while keeping track of previous variable mappings. If two nodes do not match in type, do not have the same number of children, or do not make use identifier names that map to eachother, then the unifying AST match fails. Also if any of the children children of the two nodes fail this test, then the node fails. Otherwise there is a unifying AST match between the two nodes.

From this recursive definition, we have the following pseudocode. Let *mapping* be a hash table/dictionary provided empty at the invocation of the function.

```
procedure UnifyingASTmatch(a, b, mapping)
    if a.type ≠ b.type or |children(a)| ≠ |children(b)| then
        return False
    if a.id ≠ b.id then
        if a ∉ mapping then
            mapping[a.id] ← b.id
        else
            if mapping[a.id] ≠ b.id then
                return False
    if ∀_{0≤i<|children(a)|} UnifyingASTmatch(a_i, b_i, mapping) then
        return True
    else
        return False
    return True
```

Figure 3.6: Unifying AST Match pseudocode

Note that this *Unifying AST Match* is more restrictive than a typical unification algorithm.

3.1.4 AST Match Ignoring Variables

An AST match where variables are ignored allows for any renaming of variables. This category is more general than the unifying AST match. This category is useful as it is invariant to the reordering of variable arguments to commutative

operations and the reordering of function arguments. The two programs below illustrate this, and would match under this category.

```
def mult(a,b):
  return a*b
```

```
def mult(a,b):
  return b*a
```

Figure 3.7: two programs where commutivity allowed reordering of variables

procedure ASTMATCHIGNORINGVARIABLES(a, b)
  if $a.type \neq b.type$ **or** $|children(a)| \neq |children(b)|$ **then**
    return $False$
  if $\forall_{0 \leq i < |children(a)|} ASTmatchIgnoringVariables(a_i, b_i)$ **then**
    return $True$
  else
    return $False$
  return $True$

Figure 3.8: AST Match Ignoring Variables pseudocode

3.1.5 AST Match Reordered at Depth K

It is often the case that a student may reorder statements or functions in a copied program [4]. An equivalence relation allowing for such reorderings at all depths in the AST would lead to a computationally prohibitive algorithm [18].

By selecting a maximum depth at which these reorderings may be allowed, this computational hurdle can be overcome. This category retains much of its power to express a type of program match.

In the all pseudocode that follows, the funcion f is a function mapping indices to another permutation of indices. It is one-to-one and onto.

procedure ReordDepthK(a, b, k)
  if $a.type \neq b.type$ **or** $a.id \neq b.id$ **or** $|children(a)| \neq |children(b)|$ **then**
    return $False$
  if $k = 0$ **and** $\forall_{0 \leq i < |children(a)|} reordDepthK(a_i, b_i, k)$ **then**
    return $True$
  else
    if $\exists_{\{f:f(x) \to y\}} \forall_{0 \leq i < |children(a)|} reordDepthK(a_i, b_{f(i)}, k-1)$ **then**
      return $True$
  return $False$

Figure 3.9: AST Match Reordered at Depth K pseudocode

Even at depth one this is a very useful relationship, as then it might detect reordered functions.

```
def square (x):
  return x * x

def halve (x):
  return x/2
```

```
def halve (x):
  return x/2

def square (x):
  return x * x
```

Figure 3.10: two programs that are reordered at depth 1

3.1.6 Unifying AST Match Reordered at Depth K

This category allows for both reordering the AST up to a certain depth and renaming of variables in a one-to-one fashion.

```
procedure UnifReordDepthK(a, b, k, mapping)
    if a.type ≠ b.type or or |children(a)| ≠ |children(b)| then
        return (False, None)
    if a.id ∈ mapping then
        if mapping[a.id] ≠ b.id then
            return (False, None)
    else
        mapping[a.id] = b.id
    if k = 0 and ∀_{0≤i<|children(a)|} UnifReordDepthK(a_i, b_i, k, mapping) then
        return (True, mapping)
    else
        newmap = copy(mapping)
        if k > 0 then
            for all f ∈ {f : f(x) → y} do
                for all i ∈ {i : 0 ≤ i < |children(a)|} do
                    (bool, newmap) = UnifReordDepthK(a_i, b_{f(i)}, k − 1, newmap)
                    if bool == False then
                        Break
            mapping = newmapping
            return (True, mapping)
    return (False, None)
```

Figure 3.11: Unifying AST Match Reordered at Depth K pseudocode

3.1.7 AST Match Reordered at Depth K Ignoring Variables

This category allows for both reordering the AST up to a certain depth and renaming of variables in general. Note the lack of the *id* check.

procedure P(a, b, k)
  if $a.type \neq b.type$ **or** $|children(a)| \neq |children(b)|$ **then**
    return $False$
  if $k = 0$ **and** $\forall_{0 \leq i < |children(a)|} P(a_i, b_i, k)$ **then**
    return $True$
  else
    if $k > 0$ **and** $\exists_{\{f:f(x) \to y\}} \forall_{0 \leq i < |children(a)|} P(a_i, b_{f(i)}, k-1)$ **then**
      return $True$
  return $False$

Figure 3.12: Unifying AST Match Reordered at Depth K Ignoring Variables pseudocode

3.2 Rough Match

The *rough match* is a novel technique for determining the similarity of two programs that can't be compared and categorized by its parents in the hierarchy. The rough match borrows from the technique utilized by DECKARD [10]. In contrast to DECKARD, *rough match* keeps track of the subtrees associated with each characteristic vector to do finer comparisons of similar subtrees.

3.2.1 Characteristic Vectors

Characteristic vectors capture structural information of trees without regard for the ordering of the children of a tree. The figure below illustrates characteristic vectors and their associated ASTs. Note that the characteristics to be measured are chosen in the vector template in the upper left of the figure.

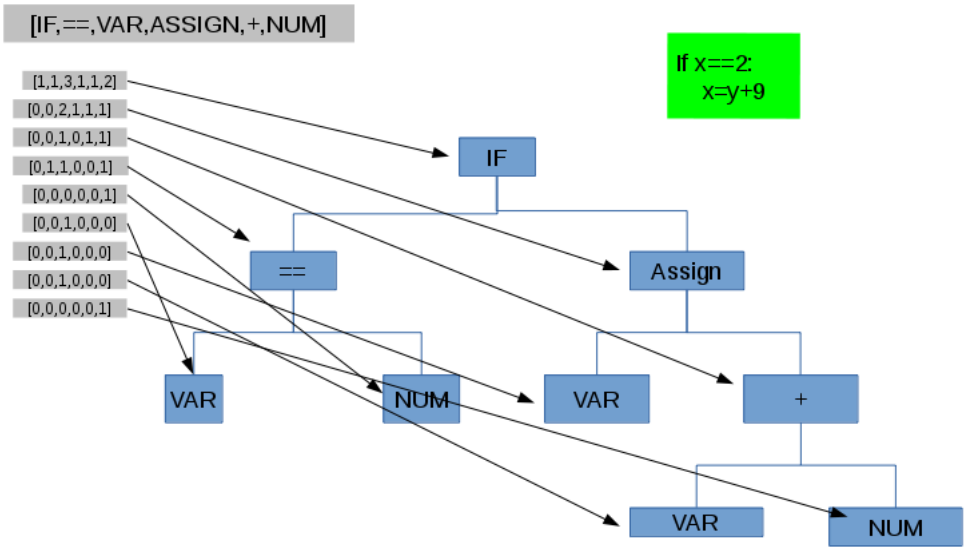

Figure 3.13: A sample parse tree with generated characteristic vectors [10].

The characteristic vector of a node of the AST is a vector representing the counts of nodes of each type represented in the template vector within the subtree rooted at that node. The *weight* of a vector is the sum of its constituents.

3.2.2 Filtering Characteristic Vectors For Significance

Determining that two programs each contain a variable is not likely to be significant. Time can be saved in the processes that follow by filtering out characteristic vectors that do not represent significant subtrees of a program's AST.

The significance of a characteristic vector is proportional to its weight. The significance of a characteristic vector is also inversely proportional to the total size of the program it came from. A subtree corresponding to one line of a program might be important in a program of only three lines, but not important in the context of a program comprised of 80 lines.

From this we might construct a test for the significance of a vector, *vector*. Note that the weight of the vector corresponding to the root node of the AST of a program is proportional to the size of a program. Let c be a real factor. *rootvec* is the characteristic vector corresponding to the node at the root of the abstract syntax tree from which the subtree corresponding to *vector* comes.

$$weight(vector) \geq c * weight(rootvector) \tag{3.1}$$

There is more to be considered here. A vector corresponding to a subtree from 50 lines might be significant in both a 50 line program and a 2000 line program,

whereas a vector corresponding to 1 line might be significant in a one line program but not in a 40 line program. Because $\frac{50}{2000} = \frac{1}{40}$, no constant c could be chosen to make the previous equation a consistent satisfaction criterion.

To allow for this desired non-linear behavior, non-linearity must be introduced into the formula. One possible choice of heuristic is shown below.

$$\frac{weight(rootvector)}{weight(vector)} \leq c * \log weight(rootvector) \tag{3.2}$$

Figure 3.14: significance heuristic for characteristic vectors

Thus the weight of the root vector is allowed to be a factor larger than the weight of the vector in question, where the factor is proportional to the log of the weight of the root vector.

3.2.3 Matching of Characteristic Vectors

Similarly we would rather matches where the characteristic vectors differ wildly not be reported as significant. A number of different metrics can be used to find a distance between two numerical vectors. We will denote the chosen distance norm between two vectors by δ. The formula used for determining whether two vectors, a and b, are significantly similar is as follows.

$$\delta(a, b) < c * \log average(weight(a), weight(b)) \tag{3.3}$$

Figure 3.15: heuristic for determining if two vectors are similar

The justification for the non-linearity of the heuristic is similar to the justification for the non-linearity of the heuristic for determining if a vector is important in the context of its containing program. With small vectors it is important that the vectors be more near identical, as we would like to prevent the matching of small matches unless extremely similar. Large matches can afford to be slightly less similar to allow for the insertion of useless statements or slight program rearrangement by a determined plagiarist.

Abstract syntax trees are trees, and the weight of a characteristic vector of a tree is proportional to the number of nodes. The depth of the tree is proportional to the log of the weight of the characteristic vector. From this it can be roughly thought that this heuristic allows for differences (δ) proportional to the depth of the trees being compared.

3.2.4 Mapping of Characteristic Vectors to AST Subtrees

If care is taken in keeping all characteristic vectors associated with their respective AST subtrees, then after two vectors are found to be both significant and similar, their respective AST subtrees can be compared. There is useful structural information in the AST subtrees that is lost in the associated vectors, as is demonstrated by the vectors' child subtree ordering invariance.

3.2.5 Categorization and Scoring

By comparing the two subtrees, it can then be seen if the subtrees satisfy any of the equivalence properties seen higher up in the categorical hierarchy. The subtrees can be compared to see if they are exact AST matches, unifying AST matches, if they differ only in variable names, etc.

Matches can be penalized based on how loose of an equivalence relationship they satisfy. The size of the subtrees is also an importance characteristic when determining a score for these matches.

In general, we have the following formula for calculating a score of a match.

$$\frac{average(weight(a), weight(b))}{average(weight(rootvecof(a)), weight(rootvecof(b)))} * \prod penalties \tag{3.4}$$

Figure 3.16: equation for scoring of rough match

A penalty is incurred at each level of depth required in the category hierarchy, and a further penalty is required for subtrees that do not fit any equivalence relation. This further penalty is the inverse of the tree edit distance of the two subtrees.

3.2.6 Tree Edit Distance

Tree edit distance is a metric for the distance between two trees. Abstract Syntax trees are trees and thus amenable to this metric. There are three operations on trees that are allowed in the algorithm to change one tree into another. Each operation is assigned a cost, and the algorithm seeks to minimize cost using dynamic programming.

The tree edit distance algorithm utilized by *rough match* is the *Zhang Shasha* [19] tree edit distance algorithm. The code for this algorithm has been ported from a Python2 package to Python3.

3.3 Reporting of Clone Pairs

Results are reported with most specific categories before more general categories. That is, clones within one category are reported after the clones of that category's ancestor categories. The order clone pairs are reported from each equivalence category is of no consequence. Rough matches are reported in order of descending score.

An essential characteristic of the clone reporting of this system is that the categories of the pairs is also reported. The user is not artificially deprived of useful information. Even in the rough match category, the type of match that the AST subtrees achieved is reported along with the score.

Sample output follows.

```
BEGIN REPORT
  EXACT MATCH
    program1
    program2
  AST MATCH
    program3
    program4
  AST MATCH
    program4
    program5
  UNIFYING AST MATCH
    program6
    program7
  SIMILAR FRAGMENTS
  SCORE: 7.13
  TYPE: EXACT FRAGMENT
    program11
    program12
  SIMILAR FRAGMENTS
  SCORE: 3.584
  TYPE: APPROX FRAGMENT
    program8
    program9
END REPORT
```

Figure 3.17: sample clone reporting output

3.4 Graphical Interface

The text-based form of reporting does not facilitate the easy verification of clone pairs. To simplify the verification of clone pairs, it is desired that a clone pair display both programs side by side, and that the list of clones be easily navigable. A GUI (graphical user interface) was developed for this purpose.

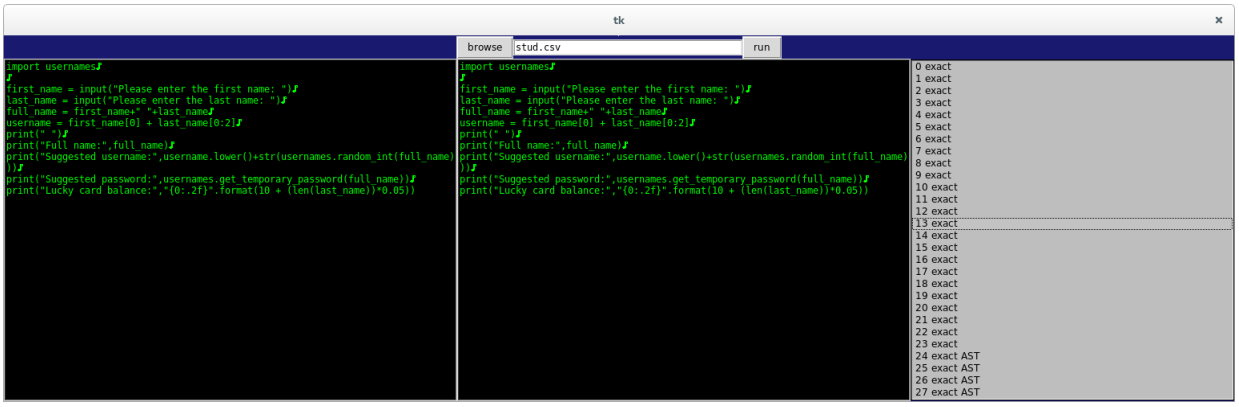

Figure 3.18: A Session with the Graphical Interface

3.4.1 File Selection

A text field at the top allows a user to input the path to file of student submissions from Moodle in CSV (comma separated values) format to be used. If the user is unsure of the exact path to the file, the "browse" button opens a new window allowing the user to find the file in a file manager.

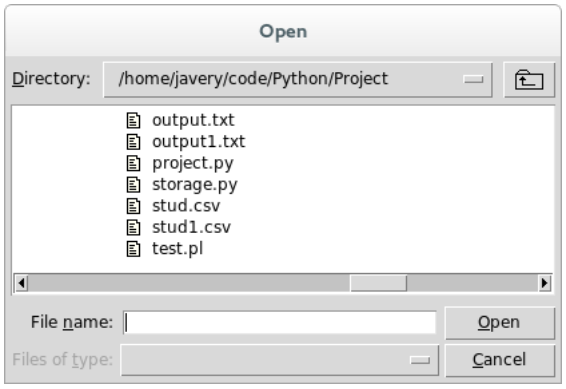

Figure 3.19: The File Selection Interface

After a file has been selected, the user must then click the "run" button for the detector to begin processing the programs from the selected file.

3.4.2 Pair List

On the right of the display there is a list area that displays a list of all clone pairs found. On the left of each line is that clone pair's ranking. After that follows the category of the match, and if the matching was found using *rough match*, then the type of rough match and the score of the match is also displayed.

3.4.3 Side by Side Display

On the left of the display are two text areas that allow for the display of two programs side by side. If the user clicks a clone pair from the pair list on the right, then the two programs corresponding to the match are displayed in the two text areas of the side by side display.

3.5 Interpretation of Reported Pairs

The pairs are reported in order of suspicion. It is intended that a user of the system would look at the pair listing, and verify clones by hand going down the list until the proportion of the pairs that are not clones becomes high enough that further analysis is not productive or determining with certainty that two programs are clones becomes unclear.

3.6 Maintaining Anonymity

It is important to remove the biases of the user from the process of verifying if two programs are similar. To aid in this, the proposed system assigns a unique identifier to each of the programs it compares. It is these unique identifiers that are reported.

3.7 Evaluation

There are two components to consider in the evaluation of the efficacy of this experiment.

precision
: the ratio of the number of reported clones that are clones to the total number of clones reported

recall

the ratio of the number of reported clones to the number of clones that exist

Determining recall is not feasible because of the nature of the interpretation of the reported clone pairs. To determine recall, at the very least the ground truth of all clone pairs would need to be known. Knowing the ground truth of all clones pairs would involve analysis by hand of every pair of programs. This would be a manual quadratic process and not reasonable by hand.

Similarly, because it is the onus is on the user to determine at what point in the pair listing to cease looking at matches, it is difficult to quantify precision. Different choices of when to stop looking at reported pairs may result in different precision. It would be expected, however, that if the clone detector is effective, then the farther down the list the user decides to stop verifying results, the lower the precision will be.

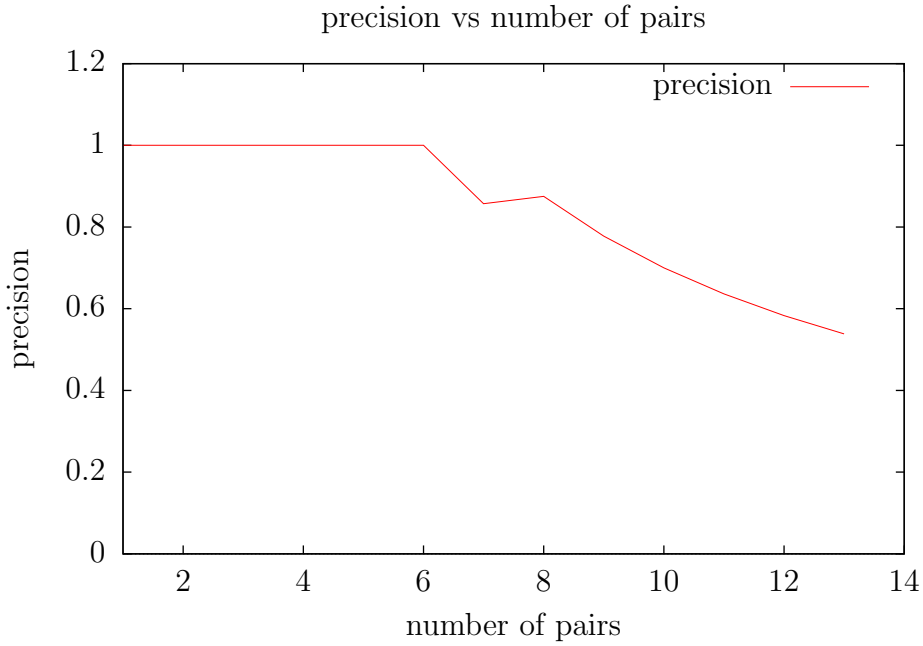

Figure 3.20: demonstration of an expected plot

From this expectation a scheme for quantifying the effectiveness of the clone detector can be derived. The precision of the matches can be plotted against the number of pairs (starting from the most certain) that are considered by the user. It is assumed that if the system is effective, such a plot should have non-positive slope. It is sensible that the more effective the system is, the less pronounced the

slope of such a plot should be. This is because in an ineffective system, precision will fall more markedly with decreasing reported certainty.

Additionally, as not all programs are clones of one another, it is expected that at some point moving down the list of reported clone pairs, the amount of programs reported that are clones should fall markedly.

Chapter 4

Improving Performance

Performance has been a recurring issue in the implementation of this system. Intolerable performance has been improved through several measures.

4.1 Heuristics

Heuristics as previously detailed (both the similarity and the importance heuristics) have been used to filter data considered, eliminating some data from consideration in expensive comparative operations.

4.2 Parallelization

The results of the comparison of two programs has no affect on the result of comparison of another two programs. Taking this into account, it is logical that the comparison of different pairs should be parallelized [5, 14].

Parallelization of program comparisons has been accomplished through the use of **concurrent.futures** and the **concurrent.futures.ProcessPoolExecutor**. Creating a new process for each pair of programs is not ideal [8]. Thus a process is only created for each group of at most 100 pairs.

4.3 Minimizing Use of Memory

The number of pairs of programs to be considered scales quadratically with the number of programs considered. This can be a prohibitive number of pairs and can use a lot of memory. To prevent this undesirable behavior, the groups of programs to be fed to the **concurrent.futures.ProcessPoolExecutor** are created in a lazy (on-demand) fashion. These groups are taken from a lazy generator of all

pairs created using **itertools.combinations**. The lazy grouping function is shown below.

```
def group(numberPerGroup,source):
   result = []
   for thing in source:
     result += [thing]
     if len(result) >= numberPerGroup:
       yield result
       result = []
   if result != []:
     yield result
```

Figure 4.1: lazy grouping function

A result is created for every pair of programs that either matches a higher match category or passes the heuristic filters and achieves a rough match score of more than zero. Many programs will fit this description. In general, the number of pairs fitting this description will scale quadratically with the number of programs considered. The number of pairs that is not in the rough match category is generally small, and does not tend to be a problem in practice.

Before modifications, the number of rough match pairs would often be so high that the computer would run out of memory and the program would crash. To correct for this, a naive method of limiting the number of rough match pairs was implemented. If a rough match is found, and the number of rough matches is currently 400 or higher, then the current rough match is only added if its score exceeds the minimum rough match score in the list of rough matches. If it is added, then the rough match with minimum score is removed from the list of rough matches. A limit of 400 rough matches was chosen because a user of this system is unlikely to manually verify more than 400 matches.

Chapter 5

Results

5.1 Performance

Table 5.1:

Experimental Setup	
Processor	Intel(R) Core(TM) i3-3217U
Processing Speed	1.80GHz
op-mode	64-bit
ram	3843
Operating System	Fedora release 20 (Heisenbug)

The plots below show the running time of the system vs the number of programs considered. The programs used in the first plot generally have a shallow AST and are small.

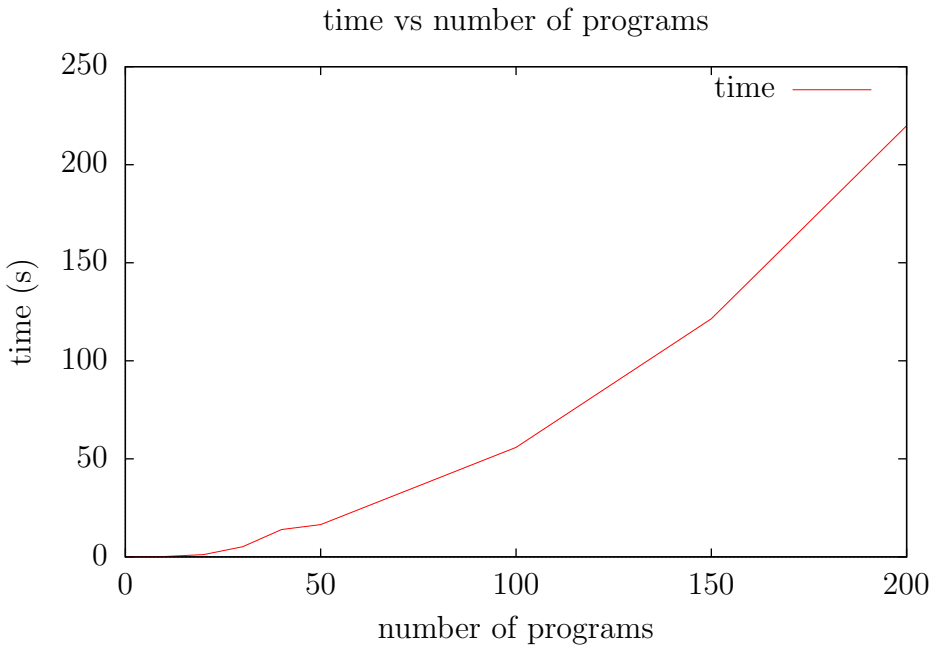

Figure 5.1: performance with small shallow programs

The polynomial nature of this graph is clear, but running times for large amounts of programs remain manageable. The programs in the following plot are generally larger with a deeper AST.

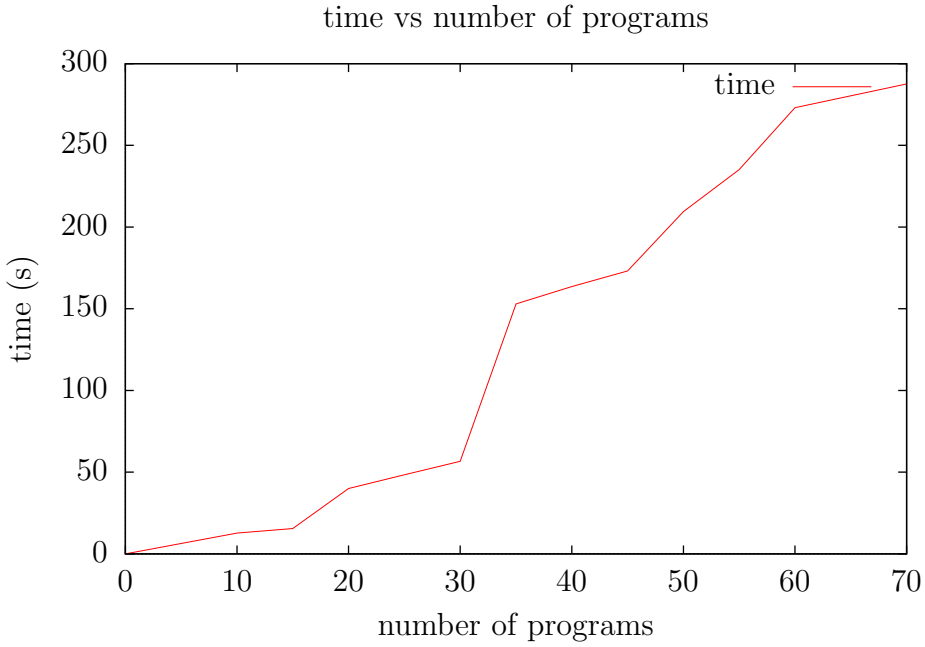

Figure 5.2: performance with large deep programs

The polynomial nature of this graph is less clear. This is sensible, as the increased depth and variance of the ASTs involved makes the performance of the tree edit distance algorithm more variable.

5.2 Validity

Below is a plot showing the precision of the system when working with 100 student submissions for the same question. A difficulty with this problem is that many of the students were using the same pseudocode as a model for a large part of their program. This lead to some confusion of the system.

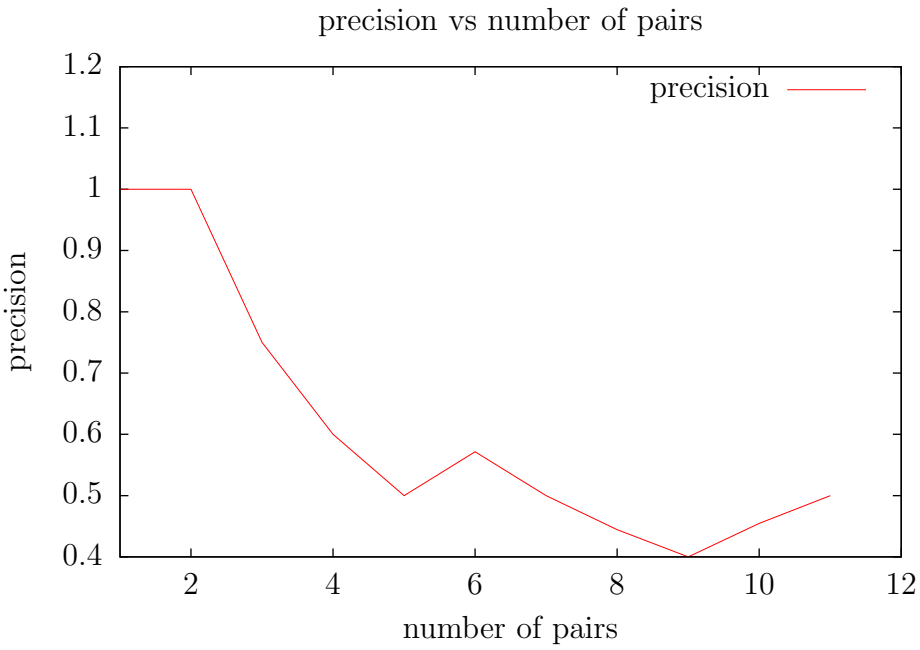

Figure 5.3: plot for a question with 100 student submissions

As can be seen from the plot, there are not many clones before the certainty drops dramatically. This is expected as there were only 100 programs being analyzed. This run of the program took 9 minutes and 12 seconds.

Another plot follows showing the precision of the system when working with 444 student submissions for a question in which the programs are smaller with shallower ASTs.

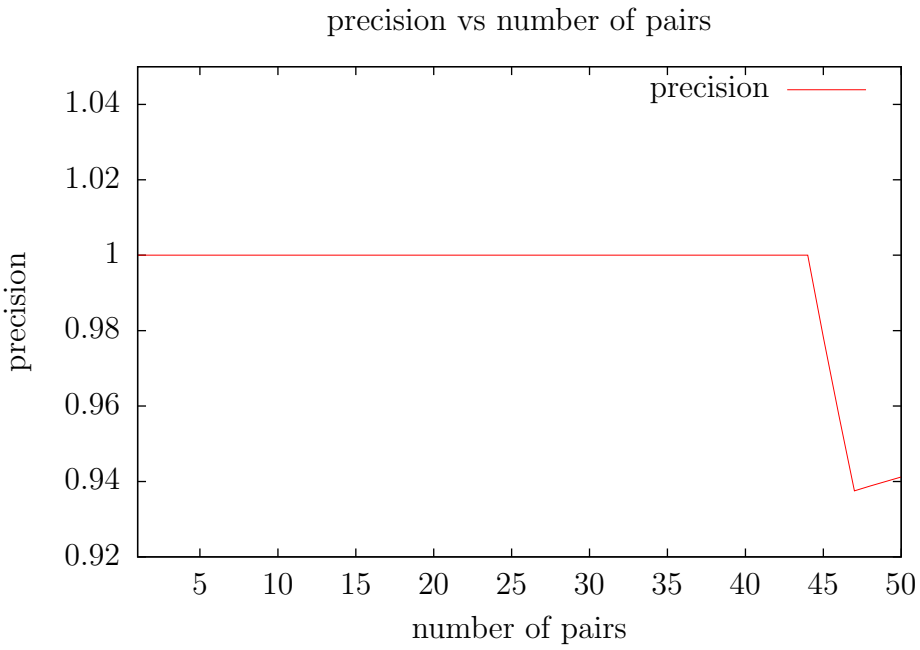

Figure 5.4: plot for a question with 444 student submissions

A surprising amount of programs here were obviously based on each other when compared to the previous example. This report included 24 exact matches, 7 matches sharing abstract syntax trees, 13 unifying matches, and one match that was the same when variable names were ignored. This goes a long way in showing the usefulness of these very restrictive categories. This run of the program took 18 minutes and 39 seconds.

Chapter 6

Conclusion

6.1 Strengths of this System

The categorical approach of this system supplies more information to the user. This better enables a user to make decisions about whether a reported clone is in fact a clone.

The categorical hierarchy of this system allows for greater granularity in clones reported. Other systems immediately discard potentially useful information in the very beginning during the *pre-processing phase*. Our system attempts to only discard useful information when it has no way to utilize it.

6.2 Suggestions for Future Work

6.2.1 Improve Performance

While much has been done to improve the performance of the system for the purposes of usability, there is still room for dramatic improvement. Barring any real performance improvement, the addition of a loading bar and/or time estimation would be considerate. This is especially important to prevent a user from thinking the program has frozen.

Two possible avenues for improved performance without additional insight would be to port the program to a faster language, or to increase parallelization using distributed computing. Heuristics could also be tightened with the choice of new constants.

6.2.2 Production of New Categories

The categories of matches presented in the match category hierarchy are not exhaustive. Any categories with the same implicative relationship could be used. Examples of category not utilized in this system include "programs that have the same text excluding leading and trailing whitespace" and "programs that have the same text when non-syntactically significant blank lines are excluded". Finer granularity of results can be achieved with the addition of new categories of matches.

6.2.3 Improvement of Template Characteristic Vector and Weighting

The characteristic vector template used in this project was chosen via intuition with some common language constructs, all of which are equally weighted in future calculations. With directed effort, the features that are measured by the characteristic vector could be more thoughtfully chosen and weighted. With a large corpus of ground truth provided, efforts in this direction would be drastically simplified.

6.2.4 Alternative Reporting Strategies

The system here, as presented, reports clone pairs in a list of decreasing certainty. While this is a valid choice, it should be noted that the upper categories of the category hierarchy form equivalence relations. This allows for the separation of equivalence classes from the results [6], where each equivalence class contains a number of programs that match under a given matching category.

Reporting of this kind can aid a teacher in finding students who work together frequently, or alarmingly large groups of students with very similar solutions. Such reporting would be most optimally handled with an altered graphical user interface (as it might then be desirable to see more than two programs next to each other).

The logic needed for separating equivalence classes has been implemented as this angle was explored. This approach was abandoned in favor of a normal pair listing in the interest of data collection as it was unclear how to quantitatively evaluate results in that form.

6.2.5 Automated Identification of Evidence of Plagiarism

Speaking to some professors provided insight on typical methods of proving plagiarism. If two suspicious programs are not exactly identical, and even in some cases if they are and the programs are very generic, a professor will often search for distinguishing features the two programs share that are unlikely to occur aside

from in the case of plagiarism. Examples of such distinguishing features include trailing whitespace on certain lines of a program, or similar comments.

In the case of similar trailing whitespace or identical comments, the process of finding such evidence of plagiarism can be automated. This would be a nice extension to the current system in that instead of just reporting suspected clones, suspected clones and the evidence needed to pursue disciplinary action could be provided in tandem.

6.2.6 Utilizing more Information in Rough Matching

Similarly such atypical distinguishing features could be taken into consideration in the matching process. While not all such features would fit nicely into the presented match category hierarchy, they could conceivably be used to affect scoring in a rough match.

6.2.7 User Experience Considerations

The system this report details has a user interface that was designed mainly with the researcher in mind. As such it has facilities for data collection and testing, but does not have features for recovering student names or IDs from matches or marking matches as verified. Such extensions would be trivial to implement, but meaningful for a user of the system.

Additionally, the categories of clones reported, while descriptive and precise, are not ideal for non-technical users. A renaming of the categories within the graphical user interface would make the system more accessible for non-technical users. It is not always the case that the person operating a clone detection system is a technical user, and ultimately if results are to be brought for disciplinary action, it is unlikely that the authority consulted will be a technical user.

It would also be nice if the areas of code corresponding to a rough match were highlighted for easy identification. The abstract sytax trees of Python programs hold line number information. This information could be used to this end.

6.3 Final Remarks

A novel method for code clone detection and reporting has been presented that is specifically tailored to the problem of detecting plagiarism in student submissions. By design it offers more information to the user of the system than many alternatives. Also by design many of its component algorithms specifically recognize common techniques of plagiarists.

It was effective in tests with real student program submissions. It is also extensible with multiple ways in which it might be improved. A graphical interface was implemented and used in the collection of data for this report.

Bibliography

[1] M. M. Bernal, H. Estrada, and J. F. Nazuno. Code similarity on high level programs. *arXiv preprint arXiv:0710.5547*, 2007.

[2] S. Burrows, S. M. Tahaghoghi, and J. Zobel. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, 37(2):151–175, 2007.

[3] V. Ciesielski, N. Wu, and S. Tahaghoghi. Evolving similarity functions for code plagiarism detection. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1453–1460. ACM, 2008.

[4] J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato. A plagiarism detection system. In *ACM SIGCSE Bulletin*, volume 13, pages 21–25. ACM, 1981.

[5] D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. *Computers, IEEE Transactions on*, 100(4):289–296, 1978.

[6] J. Gallian. *Contemporary abstract algebra*. Cengage Learning, 2012.

[7] S. Grier. A tool that detects plagiarism in pascal programs. In *ACM SIGCSE Bulletin*, volume 13, pages 15–20. ACM, 1981.

[8] F. Hao, K. Wilson, R. Fujimoto, and E. Zegura. Logical process size in parallel simulations. In *Proceedings of the 28th conference on Winter simulation*, pages 645–652. IEEE Computer Society, 1996.

[9] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Information and Software Technology*, 49(9):985–998, 2007.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[11] I. Keivanloo, C. K. Roy, and J. Rilling. Sebyte: Scalable clone and similarity search for bytecode. *Science of Computer Programming*, 95:426–444, 2014.

[12] M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitányi. The similarity metric. *Information Theory, IEEE Transactions on*, 50(12):3250–3264, 2004.

[13] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[14] K. J. Ottenstein. A note on the relationship between the expression and the automatic detection of parallelism in programs. *Applied mathematics and computation*, 20(1):37–40, 1986.

[15] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

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

[17] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[18] D. Shasha, K. Zhang, and F. Shih. Exact and approximate algorithms for unordered tree matching. *Systems, Man and Cybernetics, IEEE Transactions on*, 24(4):668–678, 1994.

[19] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.