Plagiarism Detection Reduced to String Matching

Abstract. The number of students following programming courses is steadily increasing at the same time as access to computers and networks is readily available. There is a significant minority of students who – for a variety of reasons – take advantage of the available technology and illicitly copy other students' programming assignments and attempt to disguise their deception. Software that can help tutors to detect plagiarism is therefore of immense assistance in detecting – and so helping to prevent – such abuse. We design new and efficient algorithm for a basis to such software. Our algorithm is simple to implement, and provides very efficient means to detect plagiarized programs. The method is built over sparse suffix trees that allow efficient similarity queries of a new program file against all the files in database.

1 Introduction

The numbers of students following computer programming courses are increasing. One consequence of this increase in numbers is a corresponding increase in the difficulty of detecting isolated instances of students engaging in unacknowledged collaboration or even copying of coursework.

Assessment of programming courses typically involves students writing programs, either individually or in teams, which are then marked against criteria such as correctness and style. Unfortunately, it is very easy for students to exchange copies of code they have written. A student who has produced working code may be tempted to allow a colleague to copy and edit their program. This is discouraged, and is likely to be regarded as a serious disciplinary offense. However, it is easy for a lecturer to fail to detect plagiarism, especially when class sizes are measured in hundreds of students.

Automation provides a means with which to address these concerns [14]. Much of the program submission, testing and marking process has the potential to be automated, since programs are, by definition, stored in a machine-readable form.

We have developed a new algorithm for efficiently detecting instances of possible plagiarism. We use a program database that stores all the student program files seen so far. The database is indexed using sparse suffix trees that allow efficient similarity queries for a given new file.

1.1 Techniques for plagiarism

It is not feasible to classify *all* possible methods by which a program can be transformed into another one of identical (or similar) functionality. However, two common transformation strategies can be identified.

Lexical changes are those which could, in principle, be performed by a sophisticated text editor. They do not require knowledge of the language sufficient to parse a program. Typical approaches are e.g. rewording, adding or removing comments; changing the formatting; and modifying identifier names.

A *structural change* requires the sort of knowledge of a program that would be necessary to parse it. It is highly language-dependent. Some examples are replacing loops (e.g. `while..do` to `repeat..until` or to `for` or vice versa); nested `if` statements can be replaced by `case` or `switch` statements; in some case the order of some statements can be changed without affecting the meaning of the program; calls to subroutines may be inlined; and ordering of operands may be changed (e.g. `x < y` may become `y > x`). Our current solution do not handle structural changes.

Many of these techniques can be circumvented by simply removing all comments and white spaces and tokenizing the program source. The tokenizing process may e.g. replace all identifier names with a single token. This simple method have proved to be very effective in practice [10, 7].

1.2 Previous work

The ability to detect instances of similar programs can be (and usually is) distilled into being able to decide whether or not a *pair* of programs are sufficiently similar to be of interest. There are two principal comparison techniques.

First is to calculate and compare *attribute counts* [11, 4, 6]. This involves assigning to each program a single number or a tuple of numbers capturing a simple quantitative analysis of some program features. Programs with similar attribute counts are potentially similar programs. This is a very simple method, but the detection performs poorly in practice [16, 17, 14]. The second and much better approach is to compare programs according to their *structure* [12, 9], but these methods are very complicated.

A system which incorporates sophisticated comparison algorithms is, by its nature, complex to implement, potentially requiring the programs it examines to be fully parsed. In educational context, students will not necessarily use a single programming language throughout their degree course, and any detection software must be readily upgradeable to handle new languages and packages. There is, therefore, a need for a relatively simple method of program comparison which can be updated for a new programming language with minimal effort, and yet which is sufficiently reliable to detect plagiarism with a high probability of success.

There are some recent attempts for such methods [7, 10]. They rely on removing the comments and white spaces and tokenizing the program source code, and trying to find partial matches (substrings) between two given files. The substring matching based method is simple to implement, and the detection works well in practice. The tokenizing does not need full parsing, but simple lexical scan that transform e.g. all identifier and function names to single tokens, is enough. We take the same approach. Our algorithm uses suffix trees [15, 13] as an index

structure, or more specifically, *sparse* suffix trees [5, 8, 1]. This allows us to index a whole database of tokenized files for efficient queries.

The work by Baker [2, 3] also builds over (generalized) suffix trees, but the algorithms are more complex, they consider only pair-wise comparison of files, and the space requirement is larger (their suffix trees are not sparse).

2 Preliminaries

Our problem domain is a set $\mathcal{F}$ of *files*, $\mathcal{F} = F^i$, $i \in [1..f]$, which forms our database. We have also a *query* file Q that is compared against the database, to find similarities.

The database will also implicitly encode the similarities between the stored files, and inserting a new file into the database is almost the same operation as comparing the query file against the database.

We will assume that each file F^i is a tokenized program source code, and there is a distinct symbol marking the end of each (tokenized) statement of the program, and there is a unique file id at the end of each program, i.e. the files look like $s_1\#s_2\#s_3\#...s_r\#i$, where s_j is a statement, and # is a special symbol not appearing in any of the statements, and i is the file id. Let the total alphabet size, including the symbol # be σ. The total size, i.e. the number of symbols in F^i is denoted by $|F^i|$.

We will treat the files as plain *strings* of symbols, i.e. all the program structure is ignored. The string v is a *prefix* and the string w is a *suffix* of the string u, if u can be written as vw. We will say that the string v is #–prefix and the string w is #–suffix of u, if $u = vw$, and $v = x\#$, i.e. v ends a statement, and w starts a statement.

3 Preprocessing

In order to do efficient plagiarism detection, we build an index for the set of files $\mathcal{F}$. For the index we will use a variant of the well–known *suffix tree* [15, 13], namely the *sparse* version of it [5, 8, 1]. That is, we will index only the suffixes of the files that start a statement. We will denote the (sparse) suffix tree of $\mathcal{F}$ as $\mathcal{S}(\mathcal{F})$.

The sparse suffix tree for a file F of length $|F|$ can be built in $O(|F|+r \log_\sigma r)$ average time, where r is the number of #–suffixes. This is possible by a simple scan through the file; for each #–suffix search its prefix from the suffix tree in $O(\log_\sigma r)$ average time, and into the position of the first mismatch, add a new branching node, and a new leaf for the suffix. The final sparse suffix tree has size (number of nodes) $O(r)$. The leaves store the pair (i, j), where i is the file *id*, and j is the suffix *id*, i.e. the suffix starts at statement j. Note that there is a unique leaf for every suffix of every file.

Note that the above construction does not use or add suffix links, and in the worst case it requires time $O(r|F|)$. The suffix links can be added in $O(r \log_\sigma r)$

expected time, if needed. It is possible to use the normal suffix tree construction algorithm, which requires only time $O(|F|)$, and then prune the tree to preserve only the #–suffixes. There also exists an $O(|F| + r)$ time worst case time algorithm for sparse suffix tree construction [1]. However, these algorithms are more complicated.

The initial index could be built using this method. For the subsequent updates of the index we will use a slower method. Typically we want to insert each new query file in the database also.

4 Searching

We want to find files in the database similar to the query file Q, satisfying the following conditions:

- The matching file F in the database can be used to cover (tile) Q with *blocks* of statements.
- A block of statements is a contiguous sequence of γ statements.
- Blocks of code whose length is $M\gamma \geq \beta$ can be rearranged, for some integer M. I.e. the exact or relative locations of the matched blocks can be anything.
- Let N be the total number of blocks in the tiling. Then we require that $\gamma N/r \geq \alpha$, is the necessary condition that plagiarism has occurred.

This is not based on any standard metric (like edit distance), but intuitively grasps what is our idea of plagiarism.

The matching algorithm searches all the matching prefixes of some of the #–suffixes of Q from $\mathcal{S}(\mathcal{F})$. The prefixes must be at least γ statements long, to qualify as 'significant'. The resulting list of matches is then parsed to discover similarities between Q and $\mathcal{F}$.

4.1 Greedy search

We search the query file Q from the sparse suffix tree $\mathcal{S}(\mathcal{F})$. We search Q starting from the root as long as a whole statement matches, or we have matched up to γ statements, ending in node u. The matching information is entered in a list, and the search continues from the root with the rest of the query Q.

Let $Q = s_1\#s_2\#s_3\#...\#s_r\#$, and $Q_{\#(i)} = s_i\#s_{i+1}\#...\#s_r\#$, i.e. $Q_{\#(i)}$ is the ith sparse suffix of Q. Q_i means the ith single character of Q. Similarly we use the notation $Q_{\#(i,j)}$ to denote the string $s_i\#...\#s_j\#$. In the sparse suffix tree the path from node v to node u spell out a string, denoted by $\overline{v,u}$. The label of the edge (v, u) is denoted by $label(v, u)$.

Alg. 2 greedily searches the γ–prefixes of the #–suffixes of the given query Q from the sparse suffix tree. The γ–prefix of the suffix $Q_{\#(i)}$ is the string $Q_{\#(i,i+\gamma-1)}$. In other words, the algorithm searches all non-overlapping substrings of the form $Q_{\#(i,j)}$, starting with $i = 1$, such that $j - i + 1 = \gamma$. We collect the pairs (i, u) in list L, where u is the node where the match ended.

Note that this method may fail to detect plagiarism in some cases, as it does not search all possible substrings (only non-overlapping strings, selected greedily). In practice the method should work fairly well, if γ is not too large.

Alg. 1 Search-γ-prefix(v, q, j)

Input: Suffix tree $\mathcal{S}(\mathcal{F})$, i.e. node v, a query string q, and length j of matched suffix
Output: The node that matches γ-prefix of q and the length of the prefix.

```
1   w ← v
2   i ← 1
3   while i ≤ |q| do
4       v ← v' | label(v, v') = q_i
5       if v is undefined then return (w, j)
6       if q_i = # then
7           w ← v
8           j ← j + 1
9           if j = γ then return (w, j)
10      i ← i + 1
11  return (w, j)
```

Alg. 2 Greedy-Search(v, Q, γ)

Input: Suffix tree $\mathcal{S}(\mathcal{F})$, i.e. node v, query file Q, and minimum prefix length γ
Output: List describing the common substrings

```
1   L ← {∅}
2   i ← 1
3   while i ≤ r do
4       (u, j) ← Search-γ-prefix(v, Q_#(i), 0)
5       if j = γ then
6           L ← L ∪ {(i, u)}
7           i ← i + j
8       else
9           i ← i + 1
10  return L
```

Analysis of Alg. 2. The running time is dominated by the actual search process, the list manipulation obviously takes at most $O(|Q|)$ time. The worst case arises when $\gamma = O(r)$ (consider e.g. $\gamma = r/2$), and no matching prefix is found under this criterion. The loop in Alg. 2 therefore executes r times. Each call to Alg. 1 can take $O(|q|)$ time (but this does not guarantee a match, as $\gamma = O(r)$). The total time is therefore at most $O(r|Q|) = O(|Q|^2)$. In the best case each symbol of Q is inspected only once, yielding $O(|Q|)$ time. Clearly the average time depends on the parameter γ. We would like to minimize γ to allow fast searching, but on the other hand too small γ gives too many matches.

Let $E(\gamma)$ be the average length of the string $Q_{\#(i,i+\gamma-1)}$. Let the database consist of R #–suffixes. In the average case all the strings of length $O(\log_\sigma R)$ appear in the suffix tree. Hence, to find non–trivial matches we must set $E(\gamma) \geq O(\log_\sigma R)$.

On average the search ends in a node that has $O(R/\sigma^{E(\gamma)})$ children. If we set $E(\gamma) = \Theta(\log_\sigma R)$, then on average the search ends in a node that has $O(1)$

children. In this case the search takes $O(|Q|)$ time, because on average we find a γ–prefix in each iteration, but the ending node u has $O(1)$ children on average.

Faster algorithm. There is also another way to obtain $O(|Q|)$ query times, for any γ. This allows free choice of γ without sacrificing search efficiency. This algorithm uses suffix links. If the search ends in node u, but the length l of the matched prefix $Q_{\#(i,j)}$ was $l < \gamma$, we follow the suffix link for node u to go directly in $O(1)$ time to the node *suffixlink*(u) that matches $Q_{\#(i+1,j)}$, and resume the search from there. Alg. 3 shows the pseudo-code.

Alg. 3 Greedy-Search-with-suffix-links(v, Q, γ)

Input: Suffix tree $\mathcal{S}(\mathcal{F})$, i.e. node v, query file Q, and minimum prefix length γ
Output: List describing the common substrings

```
1   L ← {∅}
2   i ← 1; k ← 1; l ← 0
3   w ← v
4   while i ≤ r do
5       j ← l
6       (u, l) ← Search-γ-prefix(w, Q_#(i), l)
7       j ← l − j
8       if l = γ then
9           L ← L ∪ {(k, u)}
10          w ← v
11          l ← 0
12          k ← k + l
13          i ← k
14      else
15          w ← suffixlink(u)
16          k ← k + 1
17          l ← l − 1
18          if l < 0 then l ← 0
19      if j = 0 then j ← 1
20      i ← i + j
21  return L
```

Analysis of Alg. 3. Alg. 3 is similar to Alg. 2, except it inspects each symbol of Q only once, and therefore runs in $O(|Q|)$ time.

4.2 Expanding and pruning the match list

The actual plagiarism detection is the postprocessing of the output of Alg. 2 or Alg. 3, i.e. the list L.

Expanding. The list entries are of the form (i, u), where u is the node in $\mathcal{S}(\mathcal{F})$ such that $Q_{\#(i,i+\gamma-1)} = \overline{v,u}$, and v is the root node. We first expand the list to associate each i with *all* the strings contained in the children of u. If $E(\gamma) \geq \Omega(\log_\sigma R)$, then u is a leaf on average, and therefore corresponds to only one string. We construct a new list L' whose entries are of the form $(i, (k, l))$, where i is as above, and (k, l) is the pair obtained from one of the leaves of the children of u, i.e. l is the file id, and k is the suffix id. Therefore $Q_{\#(i,i+\gamma-1)} = F^l_{\#(k,k+\gamma-i)}$. The construction of the list L' can be done trivially in time $O(|L'|)$.

Alg. 4 Expand(L)

Input: List L
Output: List L'

```
1   J ← {∅}
2   while L ≠ {∅} do                          // Expand with the children of u
3       (i, u) ← remove-first(L)
4       W ← the set of leaves of u
5       while W ≠ {∅} do
6           (k, l) ← remove-first(W)
7           J ← J ∪ (i, (k, l))
8   J ← sort(J)                               // Sort into ascending order
9   (i', (k', l')) ← remove-first(J)
10  L' ← (i', (k', l'))
11  while J ≠ {∅} do                          // Remove overlapping blocks
12      (i, (k, l)) ← remove-first(J)
13      if ℓ ≠ ℓ' then unmarkall
14      if (ℓ ≠ ℓ') or (i = i' + γ and k = k' + γ) or (k > k' + γ) then
15          if ℓ = ℓ' and unmarked(i) then
16              L' ← L' ∪ (i, (k, l))
17              (i', (k', l')) ← (i, (k, l))
18              mark(i)
19  return L'
```

We sort the entries of the list L' into ascending order using the l values (file id) as primary, and the k values (suffix id) as secondary comparison keys. This can be done in $O(|L'| \log |L'|)$ time using standard algorithms. The expanding process is given in Alg. 5. The code first expands the the list, then sorts it, and finally removes the overlap.

Pruning. The sorted entries $(i, (k, l))$ of L' are scanned to find blocks of contiguous increasing values of i that match with the values (k, l). That is, we want to find a sequence:

$$(i, (k, l))$$

$$
\begin{aligned}
&(i+\gamma, (k+\gamma-1, l)) \\
&(i+2\gamma, (k+2\gamma-1, l)) \\
&\vdots \\
&(i+(n-1)\gamma, (k+(n-1)\gamma-1, l))
\end{aligned}
$$

This in effect corresponds to finding all the maximal #–prefixes, length of modulo γ, of the #–suffixes of the given query Q that match one of the files in $\mathcal{S}(\mathcal{F})$. The matching #–prefix $q\#$ is said to be maximal if it matches some suffix in $\mathcal{S}(\mathcal{F})$, but $q\#a$ does not match for any $a \in \Sigma$. In other words, the algorithm searches all non-overlapping substrings of the form $Q_{\#(i,i+n_l\gamma-1)}$, trying to maximize integer n_l at each step.

We require that the length n of the sequence must be at least β. Upon finding such a sequence, the file F^l gets $(n+1)\gamma$ *votes*, denoted by $V(F^l)$. Finally, after the whole list is processed, we compute a *similarity ratio* $V(F^l)/r$ for all files F^l, where r is the number of suffixes (i.e. 'statements') in Q. All files that satisfy $V(F^l)/r \geq \alpha$ are declared to be similar to the query file. The parameters α, β, and γ are supplied by the user.

There are two caveats in the scanning algorithm. Firstly, long matching sequences in L' could be broken by swapping just two statements in the middle, and hence the algorithm may fail to identify a block of similar code. This could be solved by matching the sequence only approximately. On the other hand, the parameters β and γ can be used to obtain the same effect. Secondly, any block of code in the query file should be matched at most once against any single file in the database (the rationale being that the code is not likely copied several times in the same file). Alg. 6 shows the pseudo-code (that also fixes the second problem).

Analysis. The length of the list L is at most $O(r/\gamma)$, because the matching block must be at least γ statements long, there are total r statements in Q, and the matched substrings are non-overlapping. The list L' has therefore length $O(Cr/\gamma)$, where C is the expected number of children for nodes u (where the greedy search terminated). The dominating part is the sorting process which takes $O(|L'| \log_2 |L'|)$ time. Everything else is linear in $|L|$, independent of the actual output, the (size of the) list of similar programs.

It is hard to estimate C for any real probability distribution. If all programs in the database are almost the same, then $C = O(R)$, total number of statements in $\mathcal{S}(\mathcal{F})$, on the other hand, if everything is unique, $C = O(1)$ (depending on γ also).

If we assume uniform Bernoulli model of probability, then on average the search terminates in a node that has $O(R/\sigma^{E(\gamma)})$ children, where $E(\gamma)$ is the expected length of a string of γ statements. This basically depends on the tokenization, but we can assume that $E(\gamma) = \gamma$. The length of L' is therefore $O(\frac{rR}{\gamma\sigma^\gamma})$. If we set $\gamma = \Theta(\log_\sigma R)$, then $|L'| = O(r/\log R)$. In this case all the searching, list processing and plagiarism detection takes only $O(|Q|)$ total time

Alg. 5 Prune(L')

Input: Sorted list L'
Output: Matches.

```
1   (i', (k', l')) ← remove-first(L')
2   n ← 1
3   e ← FALSE
4   while L' ≠ {∅} do
5       (i, (k, l)) ← remove-first(L')
6       if l' = l then
7           if i = i' + γ and k = k' + γ then
8               n ← n + 1
9           else
10              e ← TRUE
11      if l' ≠ l or e or L = {∅} then
12          if n ≥ β then
13              V(l') ← V(l') + n
14          n ← 1
15          e ← FALSE
16      (i', (k', l')) ← (i, (k, l))
17  report all files l that satisfy (V(l)/r_Q + V(l)/r_l)/2 ≥ α
```

on average. This bound holds for larger γ as well, but $|L'|$ decreases exponentially with γ, so we can regard $\Theta(\log_\sigma R)$ as an upper bound for useful values for γ.

5 Conclusions and future work

We have developed a new efficient algorithm for plagiarism detection. Our method is based on indexing the code database with sparse suffix trees, which allows efficient retrieval of blocks of code that are similar to the query file. The resulting algorithm appears to be functionally similar[1] to the (on-line) algorithm used in the already established JPlag system [10], but our (off-line) algorithm is orders of magnitude faster.

The algorithm is simple to implement, and the index is relatively small compared to the size of the original files. The index needs $O(R)$ additional space, where R is the total number of statements in all the program files. The high constant factor in the $O(R)$ bound can be reduced by substituting suffix trees with suffix arrays. In this case the search times grow by factor $O(\log_\sigma R)$.

The main motivation of this work was plagiarism detection. However, there are also 'positive' applications for the method. For example, the algorithm could detect similar blocks of code in some large software system (take e.g. the X-windows system). The similar code sequences could be substituted by a function

[1] Actually, this wasn't our goal, we designed our method from the scratch, but the result was accidentally very similar

that achieves the same effect. This would reduce the maintenance, and make the code more bug resistant. Several other application areas come from the educational technology and related fields. Are several people doing overlapping work (co-operative work)? How the work have evolved, as a series of original publications of the same authors, or which documents of different authors are related to each other, etc.

One short-coming of our current method is that it can be cheated with structural changes of the code (see Sec. 1.1). This problem could be solved by transforming the code in some sort of normalized form in preprocessing phase.

In the full paper we include experimental results of the performance of the new algorithm.

References

1. A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.
2. B. S. Baker. A program for identifying duplicated code. In *Proceedings of the 24th Symposium on the Interface: Computer Science and Statistics*, pages 18–21, College Station, TX, 1992. ACM Press.
3. B. S. Baker. Parameterized pattern matching: algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996.
4. J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity within a university programming environment. *Computer Education*, 11:11–19, 1987.
5. G. Gonnet and R. Baeza-Yates. Lexicographical indices for text: Inverted files vs pat trees. Technical Report TR-OED-91-01, University of Waterloo, 1991.
6. S. Grier. A tool that detects plagiarism in pascal programs. In *12th SIGCSE Technical Symposium*, pages 15–20, 1981.
7. M. S. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999.
8. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In J.-K. Cai and C. K. Wong, editors, *Proceedings of COCOON'96*, LNCS 1090, pages 219–230. Springer-Verlag, 1996.
9. K. Magel. Regular expressions in a program complexity metric. *ACM SIGPLAN Notices*, 16(7):61–65, 1981.
10. L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical report 2000-1, Fakultat fur Informatik, Universitat Karlsruhe, Germany, 2000.
11. G. K. Rambally and M. Le Sage. An inductive inference approach to plagiarism detection in computer programs. In *Proceedings of the National Educational Computing Conference*, pages 22–29, 1990.
12. S. S. Robinson and M. L. Soffa. An instructional aid for student programs. *ACM SIGCSE Bulletin*, 12(1):118–129, 1980.
13. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
14. K. L. Verco and M. J. Wise. Plagiarism á la mode: A comparison of automated systems for detecting suspected plagiarism. *The Computer Journal*, 39(9):741–750, 1997.

15. P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.
16. G. Whale. Identification of program similarity in large populations. *The Computer Journal*, 33(2):140–146, 1990.
17. G. Whale. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13:131–138, 1990.