Fast Plagiarism Detection System

Maxim Mozgovoy[1], Kimmo Fredriksson[1,⋆], Daniel White[2], Mike Joy[2], and Erkki Sutinen[1]

[1] Department of Computer Science, University of Joensuu,
PO Box 111, FIN–80101 Joensuu, Finland
{Maxim.Mozgovoy, Kimmo.Fredriksson, Erkki.Sutinen}@cs.joensuu.fi
[2] Department of Computer Science, University of Warwick, Coventry CV4 7AL, U.K.
{D.R.White, M.S.Joy}@warwick.ac.uk

Introduction. The large class sizes typical for an undergraduate programming course mean that it is nearly impossible for a human marker to accurately detect plagiarism, particularly if some attempt has been made to hide the copying. While it would be desirable to be able to detect all possible code transformations we believe that there is a minimum level of acceptable performance for the application of detecting student plagiarism. It would be useful if the detector operated at a level that meant for a piece of work to *fool* the algorithm would require that the student spent a large amount of time on the assignment and had a good enough understanding to do the work without plagiarising.

Previous Work. Modern plagiarism detectors, such as Sherlock [3], JPlag [5] and MOSS [6] use a tokenization technique to improve detection. These detectors work by pre-processing code to remove white-space and comments before converting the file into a tokenized string. The main advantage of such an approach is that it negates all lexical changes and a good token set can also reduce the efficacy of many structural changes. For example, a typical tokenization scheme might involve replacing all identifiers with the `<IDT>` token, all numbers by `<VALUE>` and any loops by generic `<BEGIN_LOOP>`...`<END_LOOP>` tokens. Our algorithm also makes use of tokenised versions of the input files and we use suffix arrays [4] as our index data structure to enable efficient comparisons.

While all the above-mentioned systems use different algorithms to each other, the core idea is the same: a many-to-many comparison of all files submitted for an assignment should produce a list sorted by some similarity score that can then be used to determine which pairs are most likely to contain plagiarism. A naïve implementation of this comparison, such as that used by Sherlock or JPlag, results in $O(f(n)N^2)$ complexity where N is the size (number of files) of the collection, and $f(n)$ is the time to make the comparison between one pair of files of length n. Without loss of detection quality, our method achieves $O(N(n+N))$ average time by using indexing techniques based on suffix arrays. If the index structure becomes too large, it can be moved from primary memory to secondary data storage without significant loss of efficiency [2].

The approach we describe can be also used to find similar code fragments in a large software system. In this case the importance of fast algorithm is especially

⋆ Supported by the Academy of Finland, grant 202281.

M. Consens and G. Navarro (Eds.): SPIRE 2005, LNCS 3772, pp. 267–270, 2005.

Algorithm 1. Compare a File Against an Existing Collection

```
1   p = 1 // the first token of Q
2   WHILE p ≤ q − γ + 1
3     find Q[p...p + γ − 1] from the suffix array
4     IF Q[p...p + γ − 1] was found
5       UpdateRepository
6       p = p + γ
7     ELSE
8       p = p + 1
9   FOR EVERY file F_i in the collection
10    Similarity(Q, F_i) = MatchedTokens(F_i)/q
```

high due to large file collection size. The Dup tool [1] uses parametrized suffix trees to solve this task, but the algorithms are relatively complex compared to our approach.

Algorithms and Complexity. Our proposed system is based on an index structure built over the entire file collection. Before the index is built, all the files in the collection are tokenized. This is a simple parsing problem, and can be solved in linear time. For each of the N files in the collection, The output of the tokenizer for a file F_i is a string of n_i tokens. The total number of tokens is denoted by $n = \sum n_i$.

We use suffix array as an index structure. A suffix array is a lexicographically sorted array of all suffixes of a given string [4]. The suffix array for the whole document collection is of size $O(n)$. We consider the total memory requirements to be acceptable for modern hardware. A suffix array allows us to rapidly find a file (or files), containing any given substring. This is achieved with a binary search, and requires $O(m + \log_2 n)$ time on average, where m is the length of the substring (it is also possible to make this the worst case complexity, see [4]). The array can be constructed in time $O(n \log n)$, assuming atomic comparison of two tokens.

Algorithm 1 is intended for finding all files within the collection's index that are similar to a given query file. It tries to find the substrings of the tokenised query file, $Q[1..q]$, in the suffix array, where q is the number of tokens. Matching substrings are recorded and each match contributes to the similarity score. The algorithm takes contiguous non-overlapping token substrings of length γ from the query file and searches all the matching substrings from the index. These matches are recorded into a 'repository'. This phase also includes a sanity check as overlapping matches are not allowed.

The similarity between the file Q being tested and any file F_i in the collection is just a number of tokens matched in the collection file divided by the total number of tokens in the test file (so it is a value between 0 and 1), i.e.

$$Similarity(Q, F_i) = MatchedTokens(F_i)/q,$$

In Algorithm 2, we encounter two types of collisions. The first one appears when more than one match is found in the same file. If several matches that are found correspond to the same indexed file, these matches are extended to

Algorithm 2. Update the Repository

```
1   Let S be the set of matches of Q[p...p + γ − 1]
2   IF some of the strings in S are found in the same file /* collision of type 1 */
3     leave only the longest one
4   FOR every string M from the remaining list S
5     IF M doesn't intersect with any repository element
6       insert M to the repository
7     ELSE IF M is longer than any conflicting rep. element /* collision of type 2 */
8       remove all conflicting repository elements
9       insert M to the repository
```

Γ tokens, $\Gamma \geq \gamma$, such that only one of the original matches survives for each indexed file. Therefore, for each file in the index, the algorithm finds all matching substrings that are longer than other matching substrings and whose lengths are at least γ tokens. The second one is the reverse of the first problem: we should not allow the situation when two different places in the input file correspond to the same place in some collection file. To resolve the difficulty we use 'longest wins' heuristics. We sum the lengths of all the previous matches that intersect with the current one, and if the current match is longer, we use it to replace the intersecting previous matches.

The complexity of Algorithm 1 is highly dependent on the value of the γ parameter. Line 3 of Algorithm 1 takes $O(\gamma + \log n)$ average time, where n is the total number of tokens in the collection (assuming atomic token comparisons). If we make the simplifying assumption that two randomly picked tokens match each other (independently) with fixed probability p, then on average we obtain np^γ matches for substrings of length γ. If Q was found, we call Algorithm 2. Its total complexity is, on average, at most $O((q/\gamma \cdot np^\gamma)^2)$. To keep the total average complexity of Algorithm 1 to at most $O(q(\gamma + \log n))$, it is enough that $\gamma = \Omega(\log_{1/p} n)$. This results in $O(q \log n)$ total average time. Since we require that $\gamma = \Omega(\log n)$, and may adjust γ to tune the quality of the detection results, we state the time bound as $O(q\gamma)$. Finally, the scores for each file can be computed in $O(N)$ time. To summarize, the total average complexity of Algorithm 1 can be made $O(q(\gamma + \log n) + N) = O(q\gamma + N)$. The $O(\gamma + \log n)$ factors can be easily reduced to $O(1)$ (worst case) using suffix trees [7] with suffix links, instead of suffix arrays. This would result in $O(q + N)$ total time.

Note that we have excluded the tokenization of Q and that we have considered the number of tokens rather than the number of characters. However, the tokenization is a simple linear time process, and the number of tokens depends linearly on the file length.

To compare every file against each other, we can just run Algorithm 1 for every file in our collection. After that, every file pair gets two scores: one when file a is compared to file b and one when the reverse comparison happens, as the comparison is not symmetric. We can use the average of these scores as a final score for this pair.

Summing up the cost of this procedure for all the N files in the collection, we obtain a total complexity of $O(n\gamma + N^2)$, including the time to build the suffix array index structure. With suffix trees this can be made $O(n + N^2)$.

Evaluation of the System. It is not feasible in the nearest future to compare our system's results with a human expert's opinion on real-world datasets as a human would not have the time to conduct a thorough comparison of every possible file pair. However, we can examine the reports that are produced by different plagiarism detection software when used on the same dataset. The systems used for the analysis include MOSS [6], JPlag [5] and Sherlock [3]. Every system printed a report about the same real collection, consisting of 220 undergraduate student's Java programs.

The simple approach (to consider only detection or rejection) allows us to organize a 'voting' experiment. Let S_i be the number of 'jury' systems (MOSS, JPlag and Sherlock), which marked file i as suspicious. If $S_i \geq 2$, we should expect our system to mark this file as well. If $S_i < 2$, the file should, in general, remain unmarked.

For the test set consisting of 155 files marked by at least one program, our system agreed with the 'jury' in 115 cases (and, correspondingly, disagreed in 40 cases). This result is more conformist than the results obtained when the same experiment was run on the other 3 tested systems. Each system was tested while the other three acted as jury.

Conclusions. We have developed a new fast algorithm for plagiarism detection. Our method is based on indexing the code database with a suffix array, which allows rapid retrieval of blocks of code that are similar to the query file. This idea makes rapid pairwise file comparison possible. Evaluation shows that this algorithm's quality is not worse than the quality of existing widely used methods, while its speed performance is much higher. For the all-against-all problem our method achieves $O(\gamma n)$ (with suffix arrays) or $O(n)$ (with suffix trees) average time for the comparison phase. Traditional methods, such as JPlag, need at least $O((n/N)^2 N^2) = O(n^2)$ average time for the same task. In addition, computing the similarity matrix takes $O(N^2)$ additional time, and this cannot be improved, as it is also the size of the output.

References

1. B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.
2. D. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, 1996.
3. M. S. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999.
4. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of SODA '90*, 319–327. SIAM, 1990.
5. L. Prechelt, G. Malpohl, and M. Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Fakultat for Informatik, Universitat Karlsruhe, 2000. `http://page.mi.fu-berlin.de/~prechelt/Biblio/jplagTR.pdf`.
6. S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of SIGMOD '03*, 76–85. ACM Press, 2003.
7. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.