Identification of Program Similarity in Large Populations

G. WHALE
Department of Computer Science, University of New South Wales, PO Box 1, Kensington, NSW, Australia 2033

Various techniques for detecting similar programs in large classes have been proposed previously, but research in this area is hampered by the lack of a means for evaluating their performance. To address this deficiency, new concepts are introduced that permit the effectiveness of competing systems to be quantified and enable realistic comparisons to be made. Using these criteria, popular approaches to plagiarism detection based on counting program attributes are shown to be inadequate. A two-stage method of identifying similar pairs based on structural features is proposed, and the superior performance of this technique is established.

Received October 1988

1. INTRODUCTION

Like any creative human activity, programming promotes diversity. Under the influence of competition, however, programs may begin to exhibit similarities that suggest that little originality is present. The two contexts where program similarity is particularly important are in the commercial software field, where many similar products compete for a share of the market, and in educational institutions, where students compete for grades in programming courses. In both cases, programs that are attributed to nominated authors sometimes represent the work of other, uncited originators. Reliably detecting instances where programs closely resemble other assignments or products is essential if both eductational and professional standards are to be maintained.

An important distinction between the commercial and educational environments lies in their population sizes. The number of products that accomplish a specified task is typically orders of magnitude smaller than the number of students attempting a programming exercise. Commercial plagiarism detection is largely a matter of establishing a link between a pair of products that perform identically. Similarity detection amongst student programs compounds this problem with the more difficult task of firstly identifying the dependent pairs or groups from a population of several hundred.

The need for plagiarism detection software has been questioned by Hwang and Gibson[10], who argue that grading methods alone are sufficient to discourage the practice. In large classes encompassing a wide range of abilities and motivations, however, immediate pragmatic considerations are likely to overcome the fear of failing some future exam. If only to monitor the level of original work within a class, similarity detection systems have a place in the administration of large student populations.

2. ORIGINS OF PROGRAM SIMILARITY

Program similarity may arise through a variety of mechanisms, from mere coincidence through to premeditated plagiarism. Eliminating coincidence as a possible cause is the ultimate goal for the user of a similarity detection system. The task may be simplified by finding a distinctive characteristic such as a misspelled identifier or paraphrased comment, though such a capability is very difficult to build into any automated system. The likelihood that coincidence alone is responsible for observed similarity is substantially reduced if the problem is non-trivial and is solvable in a variety of ways. A lower bound therefore exists on the size of a problem for which automated similarity detection is practical.

Co-operation between authors is the next mechanism that promotes similarity. Students working closely together will produce variants of a single solution. Whether or not this is acceptable is a matter of policy for the department concerned with administering the courses. A common practice[16] is to prohibit collusion beyond discussion of the problem to be solved. While this is a reasonable restriction in introductory and intermediate classes, a more relaxed approach is preferable for senior students, who, as pointed out by Doris K. Lidtke in Dodrill[3], should be encouraged to develop the teamwork skills they will need as professional programmers.

Co-operation is often one-sided, as where a weaker student seeks the assistance of a friend. It is expedient for the person who has completed the work simply to replace the original but unworkable effort of the first student with part or all of their own. Unlike staff members, students are not expected to be able to show other students *how* to obtain a feasible solution without divulging all the details of their own answer. Nor are students likely to appreciate the range of implementation possibilities.

It is also necessary to consider the possibility that instructors or teaching assistants might promote similarities by providing too much detailed asistance to weaker students. Fortunately, only if the problem were trivial would a substantially complete program be obtainable this way.

The most serious cause of program similarity is plagiarism. Among students, two classes of this practice are evident. A student may ask to 'borrow' a copy of a friend's working program with a view to gleaning new ideas or to compare techniques. Almost inevitably the program is substantially copied. The other mechanism by which a deliberate copy is made relies on the use of common computing facilities. Poor security measures can allow students access to one another's accounts, and unsupervised printers and waste paper bins are a source of program listings in various stages of completeness.

In the remaining instances of non-original assignment submission, the program originates from outside the class entirely. Senior students can be prevailed upon

to supply solutions to corresponding assignments from earlier classes, or to generate a new solution. In extreme cases, even professional programmers may provide their services.

3. SIMILARITY DISGUISES

If plagiarism always consisted of copying a static work, the task of detecting these actions would be simple. Unfortunately, very often a period of time elapses between the transfer or distribution of a (possibly incomplete) solution and the submission of the derived programs. During this period modifications may be made that disguise their common origin. The changes are made either because the original solution was not fully developed, or in a deliberate attempt to conceal the source.

Techniques used to disguise programs include the following, listed in increasing order of sophistication, with examples in brackets:

- Changing comments or formatting.
- Changing identifiers.
- Changing the order of operands in expressions.
- Changing data types.
  [real for integer; exploding data structures]
- Replacing expressions by equivalents.
  ["**while** found = false **do** . . ." for "**while not** found **do**. . ."]
- Adding redundant statements or variables.
  [unnecessary initializations; additional output statements]
- Changing the order of independent statements.
  [rearranging Prolog clauses and reordering independent goals]
- Changing the structure of iteration statements.
  [using **repeat** for **while**, or **while** for **for**]
- Changing the structure of selection statements.
  [linearizing nested **if**s; using **if**s for **case**]
- Replacing procedure calls by the procedure body.
- Introducing non-structured statements.
- Combined original and copied program fragments.

Many of these changes occur 'naturally', as refinements or corrections made by the original author. It is therefore unwise to assume that particular differences are inevitably the result of an attempt at deception. Accordingly, automated systems cannot be expected to attribute originality, but only to flag pairs of probably dependent programs for further investigation.

The influence of these alterations on the performance of a similarity detector depends critically on the means of comparison. Direct comparison of source text is able to be defeated easily, while differences in object code may be difficult to relate to source changes. Dynamic comparisons (using methods for program testing) can determine only functional similarity. Carefully designed representations of the source program provide the best compromise between sensitivity (ability to detect similarities) and selectivity (ability to reject differences).

4. EVALUATING DETECTION PERFORMANCE

Many schemes have been devised to detect similar programs within a group implementing the same task. Authors of each method support their approach by often sketchy reports of the success of their method compared with some of the previously published techniques. Unfortunately, these results are difficult to put in perspective because of the lack of a common basis for comparison that spans the whole range of techniques.

The success of a similarity detection system is dependent on an analysis of the set of *nominations*, or program pairs alleged by the system to be dependent. The proportion of nominations that are subsequently verified by manual inspection as related, and the (estimated) number of dependencies that remain undetected, are key factors in system evaluation.

4.1 Classification of Matches

The list of close submission pairs produced by a comparison system is analogous to a set of edges comprising an undirected graph. Submissions correspond to vertices, and each edge represents a detected similarity. Edges may be weighted by a distance value, some quantitative estimate of how dissimilar the associated programs are. Distance metrics are often functions of the differences between the values of characteristic properties of the programs. Other systems may identify groups of related submissions directly. Each group corresponds to a connected component of the graph that has more than one vertex. Single-vertex components are of no interest, as they denote submissions distinct from the rest of the class.

Groups of 2 are established by a single edge. In general, a group of n submissions may be related by up to ${}^{n}C_{2} = (n/2)(n-1)$ edges, which form a complete subgraph of n vertices. However, the same n vertices may be identified by a spanning tree containing only $n-1$ edges. Because of this range of edge counts, the number of matched pairs reported does not uniquely determine the number of submissions under suspicion.

In order to remove the ambiguity it is proposed that matched pairs be divided into two classes: *essential* and *redundant* matches. Essential matches are defined as members of a set of $n-1$ matches that spans a group of n submissions. If the detection system defines an ordering of the matches (based on the distance measure), the essential matches will correspond to a minimum spanning tree for the group. Matches that are not essential are redundant, as they do not identify additional suspicious submissions. Since matching can be coincidental, however, redundant matches provide additional bindings within a group that increases confidence in the identification.

The *sensitivity* of the system determines how many of the $\frac{1}{2}(n-1)(n-2)$ possible redundant matches in a group of n are detected. A threshold distance will exist within which programs are nominated as being similar, and beyond which no pairing is made. An increase in sensitivity occurs when the threshold is extended, possibly resulting in additional nominations. In a tightly bound group it is possible for all distances to be within the threshold value; for a group forming a linear chain of submissions, redundant matches may not be detectable at all. Figure 1 illustrates an intermediate situation. Five hypothetical programs are shown as vertices in 2-dimensional space, where Euclidean distance is the measure of dissimilarity. The indicated threshold estab-

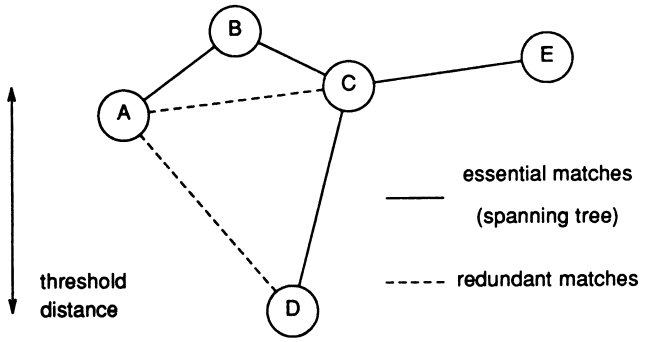

Figure 1. Essential and redundant matches.

lishes the group with a minimum spanning tree of 4 essential matches. Only two of the remaining 6 possible pairings are within the threshold distance – these are the redundant matches. The configuration might have arisen from students denoted by A, B and C working together, D having been assisted by A and/or C, and E having obtained help (or copied!) from C. These suppositions must obviously be supported by other evidence.

4.2 Detection Effectiveness

When analysing the performance of plagiarism detection, only essential matches are counted. Where groups are identified instead of pairs, the equivalent number of essential matches is substituted (one less than the cardinality of the group). Direct group identification is possible when a Boolean distance measure is applied, a match being registered only if submissions are identical in some respect. For example, identifying groups of submissions with equal line counts is simply a matter of scanning a list sorted on this attribute.

Detection reliability can be estimated in different ways. One practical measure is the proportion of *positive detections*, essential matches that manual inspection has verified as dependent. If the proportion is low, instructors' time will be wasted examining spurious pairs. The instructor must be confident, however, that the essential matches include the bulk of the plagiarism cases in the class. Analogous concepts occur in text-retrieval systems, which select documents from a large collection in response to a query. Due to indexing limitations, only a subset of all documents in the collection relevant to a subject may be selected by the query: the proportion of relevant documents retrieved is called *recall*. Conversely, the proportion of relevant documents in the retrieved group is the *precision*[15]. The analogy is established by equating documents with program pairs. As in text retrieval, there is a tendency for recall and precision to be inversely related, but both rates must be high for reliability to be achieved.

The proportion of plagiarisms uncovered (recall) is impossible to measure directly without a perfect similarity detector, though the effectiveness of a particular method can be evaluated relative to a range of other methods. Some indirect techniques, such as seeding the population with crafted plagiarisms and measuring their rate of detection, are, like program testing, only able to establish the presence of deficiencies. Fortunately, other aspects of the behavior of a system can suggest that a high proportion of plagiarisms have been exposed.

A reliable similarity detector must possess adjustable sensitivity, the threshold where a given submission pair is nominated as being similar (sensitivity in text retrieval is adjusted by narrowing or broading the query). If the sensitivity is reduced, only very close pairs will be reported; as it is increased, more and more spurious matchings occur. A useful parameter is the *limiting sensitivity*, the sensitivity that produces the last positive essential match. Ideally, the precision p at this point should be 1, which would mean that once a few negative matches were found, further sensitivity adjustments would be unnecessary. Figure 2 shows two ways in

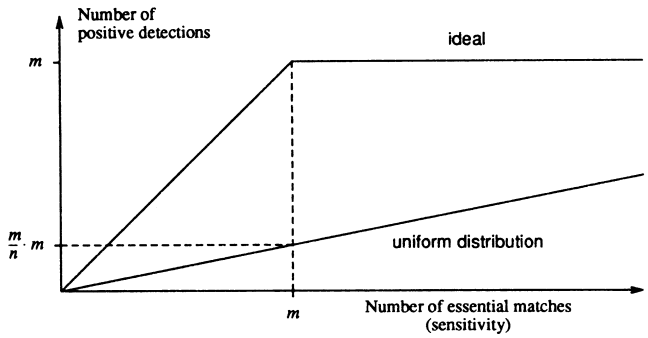

Figure 2. Detection responses of an idealized similarity detector.

which an idealized similarity detector could successfully nominate all m positive detections from a population of N submissions. Depending on the grouping relationship inferred by the system, the total number of essential matches n nominated at maximum sensitivity is bounded by $(N/2) \leq n \leq N - 1$. The ultimate precision of such a system is m/n; however, the two responses differ in their behaviour over the whole sensitivity range. In the case of the upper curve, the first m nominated essential matches are all positive detections, while the uniform-distribution response produces positive detections at equal intervals over the sensitivity range. The precision of the linear response is fixed at $p = (m/n)$, which is impractical unless plagiarism is endemic.

4.3 Selectivity and Performance

The two features that characterize the ideal response of Fig. 2 are its shape and its vertical dimension. The ability to maintain high precision with increasing sensitivity is a system's *selectivity*. It can be viewed as the degree to which a system is able to prevent spurious matches from filtering to the top of a list ordered on sensitivity. The uniform-distribution response exhibits zero selectivity; maximum selectivity is obtained by the ideal response. However, high selectivity alone does not guarantee superior performance, since it describes only the shape of the response. Unless the system also approaches a level close to the ideal (i.e., has high recall), the benefits of high selectivity are not realised.

Representation methods that preserve large amounts of information from the source program are potentially highly selective. For example, a system based on textual comparisons will easily nominate literal copies, but will rarely detect even slightly modified versions At the other extreme, a simple measure such as program length enables many dependent pairs to be identified, but they will be embedded in a long list of pairs with coincidentally similar lengths.

An overall measure of similarity detection performance will be influenced by both selectivity and estimated recall. A simple indicator derived directly from a response plot is the maximum vertical distance D between the plot and the line representing the uniformly-distributed case. D represents the *excess detections* made by a real system over an idealized system with maximum recall but zero selectivity. The slope of the reference line must be estimated first, but can be approximated by the ratio of two estimated values: the total number of positive detections made by all methods ($\hat{m} \leq m$) and half the population size ($\hat{n} = (N/2) \leq n$). Since a performance index is relevant only when compared to that of another system participating in the same experiment, uncertainties in the slope estimate have little effect.

Two refinements to the excess-detection measurement are necessary to produce a performance index that reliably measures system quality. The first is a restriction on the region where D is measured. Unless the precision at this point is acceptably high, the value of D is unrepresentative of the utility of the system. Only points on the response curve lying above a contour denoting some minimum precision are candidates. A reasonable (though arbitrary) value is $p_{\min} = 0.6$, which is a mid-range value in text retrieval. An example of this measurement revision is shown in Fig. 3. Experimentally-obtained response plots are presented in Section 6.

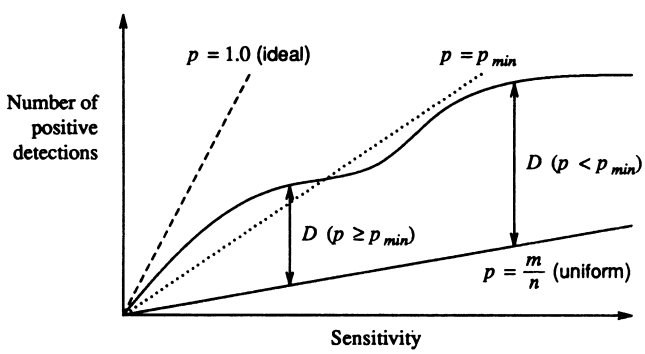

Figure 3. Measurement of excess detections.

The final performance index is obtained by the second refinement, that of normalizing the precision-limited excess detection. Normalization reduces the dependence of the performance index on the particular experiment that furnished the data. The reciprocal normalization coefficient is $\hat{m}\ (1 - (\hat{m}/\hat{n}))$, and the resulting performance index ranges from 0 (poor selectivity-recall product) to 1 (ideal).

5. SIMILARITY DETECTION TECHNIQUES

The roots of many of the systems developed to detect plagiarism lie in program complexity theory. If a program can be reduced to its essentials for the purpose of analysing complexity, a similar reduction should permit programs to be compared.

Complexity metrics can be classified in many ways,[17] but fall into two broad groups – ranking measures and non-ranking measures. Ranking measures seek to represent the complexity of a program module by a single value. A collection of programs is easily ordered using such a method, and programs with (nearly) identical complexity values could well be dependent. Non-ranking methods represent program features using some form of structured notation, such as a list of values, a polynomial, or a regular expression. Non-ranking methods are potentially highly selective.

5.1 Attribute Counts

Ranking measures which have been used for complexity estimation include *cyclomatic complexity*,[12] *scope number*[7] and several of the parameters of Halstead's *software science*.[6] While these measures are easy to derive and compare, profiles incorporating several ranking metrics have always been preferred to single values for automated plagiarism detection. Single values, unless they encode a large amount of detail, do not possess sufficient selectivity to be practical. The earliest attempts[13,5] have used the 4 parameters on which software science is based – counts of total and unique operators and total and unique operands. While constructing the profile for each student submission is a simple counting task, several approaches to matching the profiles are possible. Ottenstein[13] required profiles to match exactly, while Grier[5] permitted a small amount of variation between profiles. Neither would appear to cope with non-trivial program alterations.

The reliability of the Halstead metric as a representation of a program's important characteristics has been questioned by Berghel and Sallach.[1] They developed an alternative profile comprising counts of assignment statements, code lines, variables, and keywords. Both profiles were used to identify possible plagiarisms in a selection of FORTRAN programs spanning 100 students and 4 assignments. Coarse sensitivity adjustment was provided by two types of comparison criteria – one narrow (tuples must be identical) and one broader (corresponding elements differ by at most one).

Berghel and Sallach report that their alternative metric was consistently more reliable than the Halstead metric, which tended to detect non-existent similarities. This 'reliability' is measured by the proportion of matched pairs (presumably essential matches) that are found by inspection to be likely copies. In the narrow comparison, all 8 matches made using the alternative metric were positive, yet so were 10 of the 12 matches selected by the Halstead profile. The former, highly favoured method actually discovered fewer plagiarisms! At the broad comparison level the respective figures were 22 from 56 (Alternative) and 14 from 48 (Halstead), but the authors consider both these precision values far too low to be useful. These results are

plotted in Fig. 4, with the first part of the ideal response from Fig. 2. The gradual roll-off of precision confirms that, even if the limiting sensitivity were reached, too many program pairs would have to be inspected.

5.2 Other Approaches

A transition between the attribute-counting and structural approaches is shown by the work of Donaldson *et al.*[4] They propose two methods to be applied together. The first is a conventional attribute-count profile; the second denotes each statement in the program by a token according to its type. The statement token strings must be compared literally, while the profile comparison allows for some degree of difference. Only general results are reported.

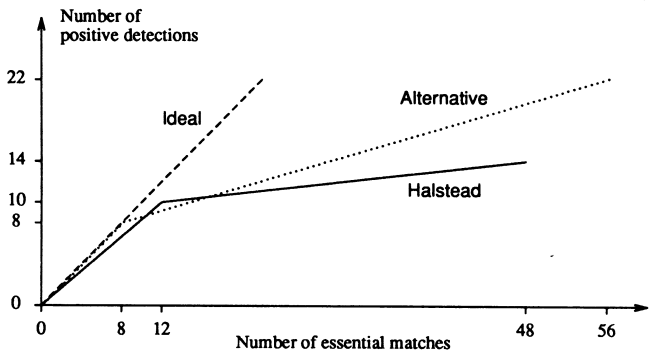

Figure 4. Detection responses for Halstead and Alternative profiles.

The final method, dispensing completely with attribute counts, is that described by Robinson and Soffa.[14] Their system uses code optimization techniques to extract details of the program's structure for comparison. Comparisons are made by examining the numbers of statements in each program's basic blocks. This amount of detail implies high selectivity, and since only programs with equal numbers of basic blocks are compared, recall is likely to be low.

Results are reported by Robinson and Soffa in some detail. Using programs from 44 students over 3 problems, the system identified 3 groups of 3 suspicious submissions and 11 groups of 2. This corresponds to 17 essential matches, of which 8 were classed as 'quite similar' or better ($p = 0.47$). The sensitivity of this method is not easily adjusted, so it is impossible to tell if the limiting sensitivity has been reached. It clearly out-performs Ottenstein's method, however, as the latter could identify only one match from the entire population.

The comparative success of the high-selectivity structured approach to similarity detection suggests that non-ranking complexity measures might be a basis for suitable representations. One such method[11] represents the structure of a program by a regular expression; another[2] by a 'characteristic polynomial'. They exhibit different degrees of selectivity, and each has its drawbacks. On the one hand, regular expressions (representing all possible execution sequences) are sensitive to minor alterations in structure. On the other, characteristic polynomials are unable to discriminate between some dissimilar statement combinations. Nevertheless, these schemes are pointers towards the development of a suitable representation tailored to the task of similarity detection.

5.3 Plague

The (unstated) aim of researchers into techniques for plagiarism detection has been to develop a single representation which balances selectivity and recall. In order to avoid this compromise, a composite method has been designed that incorporates two independent representations with different characteristics. It features a low-selectivity filtering stage followed by a highly selective matching phase. As a reminder of the insidious nature of the practice of plagiarism, the system is called Plague[18, 19].

The first stage represents program control structure using a notation resembling a generalized regular expression. Selection and iteration (or analogous constructs) are highlighted, and non-recursive procedure calls are effectively replaced by the contents of the called procedure. In order to overcome the difficulty in performing comparisons using this highly structured notation, a further transformation is made into an ordered sequence of quantitative terms called a *structure profile*. The profile terms encode statement type sequences together with the type of statement in which the sequence is nested. The number of terms in the profile is related to the global complexity of the program, while the magnitudes of individual terms are estimates of localized complexity.

The purpose of the first phase is to select a limited number of program pairs that includes all or most of the dependent essential pairs in the population. This is achieved by nominating for each submission its nearest neighbour using one of several parametrized metrics for comparing structure profiles. To defeat the procedure a plagiarist must make a copy structurally resemble some other program more than the original. In practice several near neighbors are retained, but the total number of selections is always linear in the number of submissions.

The representation used for the final selection is a sequence of tokens denoting syntactic and semantic entities, such as a statement boundary, an operator class or the type of an operand. The sequence resembles a flattened parse of the program. Procedures are ordered by a depth-first traversal of the call graph, using the textual ordering of calls within a procedure.

Token sequence matching is performed by identifying common subsequences. The longest common subsequence is not an ideal indicator of the degree of commonality between two such sequences, as it does not account for relocated blocks and its determination is relatively inefficient.[9] However, it is possible to perform an approximate match in $\mathbf{O}(n)$ time and space using an algorithm due to Heckel.[8] This algorithm can also cope with reordering of subsequences, and in this application requires only that some short (3-token) sequences occur uniquely in each token sequence.

The potential for high selectivity is assured by the detailed representation, while the robustness of the common-subsequence length calculation results in a

good recall value. That is, small changes reduce the common length only marginally, while the intersection between unrelated programs is typically half the maximum. A relative metric is used to further improve discrimination between the related and unrelated cases: $M = k/(n - k + 1)$, where k is the common-subsequence length estimate and n is the length of the shorter sequence. M ranges from 0 to n.

The final list of nominations, ordered by decreasing matching index M, provides a variable-sensitivity source of potential plagiarisms. Only pairs towards the top of this list are likely to be related, and the limiting point can be estimated from the roll-off in the index values. With experience, the number of source programs that need to be fetched and manually inspected can be tightly controlled.

6. COMPARISON OF SIMILARITY DETECTION METHODS

Assessing different techniques for similarity detection is possible only on a relative scale. An assignment requiring a Pascal program with between 4 and 8 procedures provided the testing environment in which to evaluate a range of methods. Data structures (2-dimensional arrays) were suggested but not mandatory, and several different algorithms were possible. The population consisted of 245 syntactically correct programs submitted on-line for assessment.

In implementing each reported plagiarism detector, it was necessary to incorporate adjustable sensitivity. This was achieved for the fixed 4-tuple methods (Halstead, and Berghel & Sallach's 'Alternative') by applying a Euclidean distance metric. For Donaldson's statement strings, a measure based on the lengths of the common prefix and suffix was substituted for the Boolean match/nomatch metric. The variable component in Robinson's method was the proportion of matched statements in equally-sized basic blocks (replacing the fixed threshold of 0.5). Donaldson's method of matching profiles ('sum of differences') already provides variable sensitivity.

The combined efforts of the six systems positively identified one group of 4 students, two groups of 3, and 17 pairs, totalling 24 positive detections. All but two of these were nominated by Plague. One of the recall failures (missed pairs) was a partial copy; the other involved unusual control structure changes. Verification of the hundreds of nominations was limited to the first 50 for each method. However, matching indices were determined for every nominated pair. All 24 positive detections scored highly on this scale, including the two pairs that escaped the filtering stage of Plague. Conversely, all manually inspected pairs with low values of M were found to be unrelated, even those scoring highly on other scales.

While every method produced some positive matches (the one blatant copy was detected easily by all except Halstead), precision values were often impractically low. Three representative detection responses are plotted in Figure 5. The two counting methods not shown (Halstead, Donaldson) produced responses resembling that of the Alternative Metric, while Donaldson's statement-string response approximated the middle plot.

Figure 5 demonstrates the clear superiority of structure-based similarity detection over attribute-counting methods. The high selectivity of both upper curves is evident, the primary difference being the poorer recall of the middle curve. The performance of Robinson's method might improve if matches were sought between basic blocks with non-identical sizes. The simplest of the structured systems, Donaldson's statement strings, scored marginally better than Robinson on overall performance (see Table 1), but there is little scope for improvement. The performance of the attribute-counting schemes on this experiment was consistently poor. Although all three methods ultimately managed to identify one-third of the total positive detections, they did so with impractically low precision.

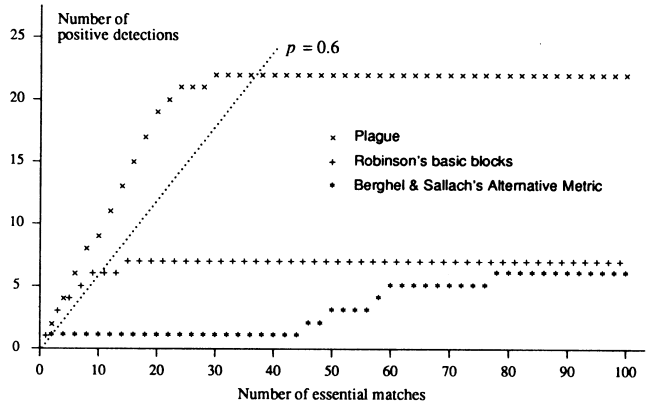

Figure 5. Comparison of similarity detectors.

Table 1. Performance Indices

First author	Method	Detections @$p = 0.6$	Performance index
various	Halstead	0	0.00
Berghel	Alternative	1	0.04
Donaldson	profiles	2	0.07
Robinson	basic blocks	6	0.23
Donaldson	strings	6	0.25
Whale	Plague	22	0.84

The clear distinction between two-stage structural, single-stage structural, and attribute-based approaches is apparent from the groupings in Table 1. The wide separation between groups suggests that fine-tuning of the methods to compensate for changes in testing environment (most systems were developed with FORTRAN in mind) is unlikely to alter the group rankings. While it is unwise to generalize from a single experiment, these results support the hypothesis that the calculated indices are reliable estimates of intrinsic quality.

7. CONCLUSION

Plagiarism detection historically has been a hit-and-miss affair, with the misses quietly ignored in favour of a few triumphant hits. For too long, the selection of similar programs has been pursued without any serious attempt at quantifying the results. The introduction of the con-

cepts of essential matches, recall, precision, and selectivity represents a first step towards placing the discipline on a scientific footing. By combining these ideas, it is possible to produce a performance index that estimates the relative effectiveness of competing systems.

Using this means of evaluating detection performance, it is easily established that the traditional representations based on attribute counters are incapable of detecting sufficient similar programs to be considered effective. Better performance is possible if the number of counters is determined by the size of the program. However, the most effective approach has been to utilize a two-stage selection process, using structural features to select each program's most likely copy, and ordering the resultant matches according to a reliable similarity index.

The commercial plagiarism problem is largely one of quantifying the degree of similarity between easily-nominated program pairs, rather than selecting these pairs from a large group. For this purpose the matching index described here would serve as a first approximation, particularly if its value is related to the range of similarity indices among all comparable products. To cope with sophisticated plagiarism techniques, and to demonstrate similarities to non-specialists, the index could be expanded from a single value to an ordered-pair sequence that records similarity module-by-module or statement-by-statement. Such a development parallels the trend towards more detailed measures in the related field of program complexity.

Acknowledgement

I am grateful to David Carrington for his encouragement and guidance in completing this research.

REFERENCES

1. H. L. Berghel and D. L. Sallach, Measurements of Program Similarity in Identical Task Environments. *ACM SIGPLAN Notices* **19** (8), 65–76 (August, 1984).
2. G. Cantone, A. Cimitile and L. Sansone, Complexity in Program Schemes: The Characteristic Polynomial. *ACM SIGPLAN Notices* **18** (3), 22–31 (March, 1983).
3. W. Dodrill *et al.*, Plagiarism in Computer Science Courses. *ACM SIGCSE Bulletin* **21** (1), 26–27 (February, 1980).
4. J. L. Donaldson, A. Lancaster and P. H. Sposato, A Plagiarism Detection System. *ACM SIGCSE Bulletin* **13** (1), 21–25 (February, 1981).
5. S. Grier, A Tool that Detects Plagiarism in Pascal Programs. *ACM SIGCSE Bulletin* **13** (1), 15–20 (February, 1981).
6. M. H. Halstead, *Elements of Software Science*. North-Holland, New York, 1977.
7. W. A. Harrison and K. I. Magel, A Complexity Measure Based on Nesting Level. *ACM SIGPLAN Notices* **16** (3), 63–74 (March, 1981).
8. P. Heckel, A Technique for Isolating Differences Between Files. *Commun. ACM* **21** (4), 264–268 (April, 1978).
9. D. S. Hirschberg, Algorithms for the Longest Common Subsequence Problem. *J. ACM* **24** (4), 664–675 (October, 1977).
10. C. J. Hwang and D. E. Gibson, Using an Effective Grading Method for Preventing Plagiarism of Programming Assignments. *ACM SIGCSE Bulletin* **14** (1), 50–59 (February, 1982).
11. K. Magel, Regular Expressions in a Program Complexity Metric. *ACM SIGPLAN Notices* **16** (7), 61–65 (July, 1981).
12. T. J. McCabe, A Complexity Measure. *IEEE Trans. on Software Engineering* **SE-2** (4), 308–320 (December, 1976).
13. K. J. Ottenstein, An Algorithmic Approach to the Detection and Prevention of Plagiarism. *ACM SIGCSE Bulletin* **8** (4), 30–41 (December, 1977).
14. S. S. Robinson and M. L. Soffa, An Instructional Aid for Student Programs. *ACM SIGCSE Bulletin* **12** (1), 118–129 (February, 1980).
15. G. Salton, Another Look at Automatic Text-Retrieval Systems. *Commun. ACM* **29** (7), 648–656 (July, 1986).
16. M. Shaw *et al.*, Cheating Policy in Computer Science Department. *ACM SIGCSE Bulletin* **12** (2), 72–76 (July, 1980).
17. J. Szentes and J. Gras, Some Practical Views of Software Complexity Metrics and a Universal Measurement Tool. *Proc. First Australian Software Engineering Conference*, pp. 83–88 (Canberra, 14–16 May 1986).
18. G. Whale, Detection of Plagiarism in Student Programs. *Proc. Ninth Australian Computer Science Conference*, pp. 231–241 (Canberra, 29–31 January 1986).
19. G. Whale, 'Plague: Plagiarism Detection using Program Structure', Dept. of Computer Science Technical Report 8805, University of NSW, Kensington, Australia, 1988.

Announcements

3–5 OCTOBER 1990

UNIVERSITY OF KEELE, UK

A Working Conference On Cooperating Knowledge Based Systems

The objective of the conference is to bring database and AI researchers together, from both academia and industry, in order to discuss the issues of the next generation of knowledge based systems. The themes to be covered will include:

Interaction of data and knowledge bases
Distributed knowledge base systems
Modular knowledge base systems
Cooperating expert systems
Mixed-mode systems

It is hoped that the presence of both database and AI researchers will help to bring about a synthesis of ideas for fruitful industrial applications. The conference is being organised by the Data and Knowledge Engineering (DAKE) Centre at the University of Keele. A number of organisations, including the EEC ESPRIT programme, are supporting this event.

Call for Papers

Papers to be presented at the conference will be preselected on the basis of extended abstracts; the full papers to be submitted at the conference which will be referee-ed after the conference for the final selection and publication. Papers are invited from workers active in this broad general area. Idea papers and papers from incomplete projects are also welcome. Abstracts of not more than 2000 words should be submitted – in six copies, or *preferably one copy by email in troff or TEX format* – to the organisers by 30th June, 1990.

There will be only a limited number of places for participants without abstracts. *Please note that we do not intend to issue a separate call for registration* (in order to keep mailing cost and hence registration fee low), and therefore if you wish to receive further information, please contact

Prof. S. Misbah Deen, DAKE Centre, University of Keele, Keele, Staffs. ST5 5BG, England. Tel: (0782) 621111. Fax: (0782) 613847. Telex: 36113 UNKLIB G.
E-mail Addresses:
JANET: deen@uk.ac.kl.cs
INTERNET: deen@cs.kl.ac.uk
BITNET: deen%cs.kl.ac.uk@ukacrl
UUCP:..!mcvax!ukc!kl-cs!deen