Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems

ARTICLE · AUGUST 1996

DOI: 10.1145/369585.369598 · Source: CiteSeer

Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems[1]

Kristina L. Verco and Michael J. Wise
Department of Computer Science
University of Sydney, F09
N. S. W. 2006
Australia

michaelw@cs.su.oz.au

ABSTRACT

The early automated systems for detecting plagiarism in student programs employed attribute-counting techniques in their comparisons of programs. It has been claimed that the more recent, automated systems, which encode structural information, are more effective in their detection of plagiarisms than systems based on attribute-counting. To explore the validity of these claims, a comparison is presented of two systems based on attribute counting and a structure-metric system. The major result of this study is that the systems based on structural information consistently equal or better the performance of systems based on attribute-counting metrics. A distinction is also made between plagiarisms due to novice programmers and those of more experienced programmers.

1. INTRODUCTION

The aim of this study is to compare a number of different systems for detecting potential plagiarisms in program texts. Plagiarism is a persistent problem in many university courses, particularly those with programming assignments. This is due to the relative ease with which a student may cosmetically alter another student's program, while not understanding what the program does. Parker and Hamblen [9] define software plagiarism as: *a program which has been produced from another program with a small number of routine changes.* In practice, if one program can be transformed into another simply through use of editor operations (such as global substitutions) or by exploiting synonymous expressions provided by the programming language, then a prima facie case of plagiarism has been found and should be examined further. (Note that neither ploy requires a knowledge of the problem being solved by the source program.)

Whale [11] lists techniques used to hide plagiarism. These range from changing comments, identifiers or formatting to shuffling independent code segments or turning in-line code into procedure calls. On the other hand, as students are attempting the same assignment question using the same implementation language, some similarities are inevitable. It is therefore important that systems attempting plagiarism detection should distinguish between chance similarities and similarities that might give rise to reasonable suspicion, and report all of the latter. However, as the task of comparing a group of assignments is inherently $O(n^2)$, it is also important that few false positives be recorded, i.e. reports of suspected plagiarism that turn out to be unfounded.

1.1 History of Plagiarism Detection Systems

The earliest automatic plagiarism detectors used Halstead's *software science metrics* [7] to determine the level of similarity between program pairs. The specific quantities suggested by Halstead were:

$$\eta_1 = number\ of\ unique\ operators$$
$$\eta_2 = number\ of\ unique\ operands$$
$$N_1 = number\ of\ operator\ occurrences$$
$$N_2 = number\ of\ operand\ occurrences$$

These four metrics were employed by Ottenstein [8] in the first automated plagiarism detection system that we have found in the literature. In this system, pairs of programs with identical values for each of η_1, η_2, N_1 and N_2 are assumed to be similar and therefore deserving closer inspection.

The systems which followed experimented with these, and other metrics such as the number of loops and the number of procedures, to find more accurate measures of similarity between pairs of programs (Berghel and Sallach [2]; Grier [5]; Rinewalt, Elizandro et al. [10]). Collectively, these systems have been called *attribute-counting-metric* systems. The attribute-counting-metric systems became quite complex, for example by including counts reflecting flow control structures, such as in Faidhi and Robinson's system [4].

1. Presented at the First Australian Conference on Computer Science Education, Sydney, Australia, July 3-5, 1996, John Rosenberg (Ed), ACM.

A second type of automatic plagiarism detection systems also used attribute-counting metrics but in addition compared representations of the program structure to give an improved measure of similarity. One example of this type is Donaldson, Lancaster and Sposato's system [3] which uses eight attribute-counting metrics and also generates a string representation of the program text. Each letter in the string represents single or multiple adjacent occurrences of program structures such as variable declarations, assignment statements and procedure calls. If the string representations match exactly or the metric counts are similar, then the pair of programs is flagged as a plagiarism. Systems which compare representations of the program structure are said to employ *structure metrics*, so the Donaldson, Lancaster and Sposato system uses both structure and attribute-counting metrics.

The most recent detection systems use *structure metrics*, i.e. they compare string representations of program structure. These systems do not require exact matches but assess the similarity of token strings. Three examples of this class are Plague [11], Sim [6] and YAP3 [12]. The present study focuses on comparisons between YAP3 and certain attribute-counting-metric systems; comparisons with Sim and Plague will be the subject of a future paper.

In the literature to date only Whale [11] has provided a comparison' between attribute-counting-metric systems and structure-metric systems (specifically Plague). The attribute-counting-metric systems included in the comparison where those of Ottenstein [8] and Berghel and Sallach [2]. It could be argued that neither of these systems, developed for FORTRAN programs, give a full account of the possible performance of attribute-counting-metric systems, particularly when running Pascal programs. Unfortunately, no attribute-counting-metric systems were available to Whale so he was forced to reconstruct them from descriptions in the literature, leaving open the suggestion that different attribute-counting-metric systems may achieve better results. For this reason it was decided that comparisons would involve the construction of *tuned* attribute-counting-metric systems, in that the metrics have been tuned for optimal performance, particularly on Pascal programs. (Once again this has meant reconstructions based on descriptions in the literature.) The current study also focuses on the detection of potential plagiarism in Pascal program texts because Pascal is the only programming language supported by all the systems under consideration.

Another deficiency in the previous literature has been the lack of any comparison of the performances of these systems in detecting novice plagiarisms versus their performance on more sophisticated plagiarisms. A novice programmer will generally make far less complex alterations than an experienced programmer, so novice plagiarisms should be easier to detect. Therefore, the texts on which experiments are done should be representative of programmers with varying levels of experience.

2. COUNTING-METRIC SYSTEMS

Based on the descriptions in the literature, two attribute-counting-metric systems were singled out for reimplementation: Grier's Accuse system [5] and Faidhi and Robinson's system [4].

2.1 Accuse

Accuse was chosen for reimplementation because it is intended for Pascal programs and because, starting with twenty parameters similar to those used in most attribute-counting-metric systems, Grier tested the parameters in different combinations to determine the most effective combination. The resulting system uses seven of the parameters:

1. Number of unique operators
2. Number of unique operands (such as identifiers, labels, integers, reals, strings and nil)
3. Total Operators
4. Total Operands
5. Code Lines (excluding blank or comment lines and declarations)
6. Variables Declared and Used
7. Total Control Statements

For a description of these parameters see [5].

During the first phase, values for the seven metrics are calculated for each program. In the second phase, pairs of 7-tuples are compared. The *correlation* scoring function described by Grier in [5] is used to score matches. The correlation scheme depends on two constants being assigned to each parameter, the *window size* and the *importance value*. The window size of a parameter represents the maximum difference between two program's values for that parameter consistent with a judgement of plagiarism. The importance value of a parameter can be seen as its significance.

- The correlation value for a pair of programs *A* and *B* is initially 0.
- For each of the metrics,
  - if A_i is the value of the i^{th} metric of program A and B_i is the corresponding metric in program B then let $diff_i = |A_i - B_i|$.
  - If $diff_i \le window_i$, i.e. the range of differences allowed for that metric, then $correlation_{AB} += importance_i - diff_i$.

Finally, all correlation numbers are sorted by size in decreasing order, so the program pairs which are most similar should appear toward the top of the output.

Window sizes and importance values are given in [5], but these values were tuned using a group of only forty-three programs. For this reason, the reimplementation of Accuse was retuned using several larger program groups as described in Section 2.3.

2.2 The Faidhi-Robinson System

In [4] Faidhi and Robinson argue that their system is more sensitive than previous plagiarism detectors because there are less frequent false positives. Specifically, they argue that the set metrics used by their system is *minimal*, in the sense that no metric in their set of 23 correlates with another. In addition, included in the set of metrics are *hidden measures* that should be difficult for a novice programmer to change.

The first ten metrics used in the Faidhi-Robinson system are more likely to be altered by a novice plagiarist. These are:

1. Average Number of Characters per Line
2. Number of Comment Lines
3. Number of Indented Lines
4. Number of Blank Lines
5. Average Procedure/Function Length
6. Number of Reserved Words
7. Average Identifier Length
8. Average Spaces Percentage per Line
9. Number of Labels and Gotos
10. Count of Unique Identifiers

The next fourteen metrics attempt to measure intrinsic and hidden features of a program's structure:

11. Number of Program Intervals – This uses Cocke and Allen's algorithm for partitioning a program control-flow graph into *intervals* [1].
12. Number of Vertices Coloured with Basic Colours – When a program's control-flow graph vertices are coloured using a graph colouring algorithm (with a maximum of four colours), the *basic colours* are the colours 1 and 2.
13. Number of Vertices Coloured with Colour 3
14. Number of Vertices Coloured with Colour 4
15. Program Expression Structure Percentage – Based on the conventional terminology for describing Pascal syntax, this is the number of expressions as a percentage of all expressions, simple expressions, terms and factors.
16. Program Simple Expression Structure Percentage
17. Program Term Structure Percentage
18. Program Factor Structure Percentage
19. Number of Program Impurities – For example, extraneous semicolons and BEGIN/END pairs.
20. Module Contribution Percentage – That is, the number of code-lines inside functions or procedures.
21. Number of Modules
22. Conditional Statement Percentage
23. Repetitive Statement Percentage
24. Number of Program Statements

Because [4] does not specify a method for scoring the levels of similarity found between two program texts, a slightly modified version of the correlation scoring function in [5] was used for scoring pairs of matches in the Faidhi-Robinson system reconstruction. Window sizes were determined by observing the *difference* values, for each parameter, over a small group of programs. Grier's correlation method requires the importance values to be larger than the window sizes, which is not appropriate in the Faidhi-Robinson system, where the window sizes vary greatly. In the new scoring phase, the *increment* is calculated using the *importance* value to scale the *increment*:

$$increment_i = \frac{(window_i - diff_i)}{window_i} \times importance_i$$

for the i^{th} parameter (rather than simply: $importance_i - diff_i$). Except for the first ten parameters, equal importance values were initially chosen.

2.3 Tuning the Systems

It is important to tune the attribute-counting-metric systems to ensure that each parameter is given the appropriate weighting. In the case of two systems described above, this has been done by altering the importance values. To decide the scaling of particular importance values, the two systems were run on student submissions for three Pascal assignments. These assignments were done by students early in their first year, at the end of first year, and early in second-year, respectively. These assignments were chosen because they illustrate a range of programming skills and increasingly sophisticated attempts to cover plagiarism. The results of these, and the other tests, are in the RESULTS section.

The parameters of two plagiarism detectors were split into groups of related parameters which would eventually have equal weights, with the metrics in Accuse being influenced by the values given by Grier in [5]. For the Accuse and the Faidhi-Robinson system the groups were:

Group	Accuse Metric No.	Faid-Rob Metric No.
1	1,2	1-10, 24
2	3,4	11-14
3	5-7	15-18
4		19-21
5		22,23

Although the importance values provided in [5] were derived from tests on only forty-three programs from an introductory course, they still could serve as a starting point. On the other hand, [4] does not describe which parameters are most important though it does explain that the first ten are the ones most susceptible to change. These, therefore, were given an initial weighting less than that of the other parameters. The number of program statements was included in this grouping as it too is easily changed, even by a novice programmer.

To find the weightings of the metric groups, the two systems were run on each set of assignments, with one group's parameters not included in the calculation of the

correlation number. This was done for each group of parameters.

When using the structure-metric detectors on the three sets of assignments it was noticed that most of the obvious plagiarisms were detected in the first N/10 matches, where N is the total number of submissions for a particular assignment. For this reason, the first N/10 matches made by the systems on each run were checked and the results from these were used to establish the parameter weights. The parameters which were not included in the runs with the lowest score of verified plagiarism detections were then given the largest weightings.

For Accuse, the final importance values were:

Group	Value	
1	5	
2	7	
3a	6	Code Lines
3b	4	Declared and used variables
3c	3	Control statement

Notice that during the tuning, group 3 split.

For the Faidhi-Robinson system, the final importance values were:

Group	Value	
1	4	
2	10	
3	9	
4a	6	Number of program impurities
4b	7	Module contribution percentage
4c	7	Number of modules
5	9	

These values were used for the final comparison run of the systems on submissions for two assignments, from first-year and second-year Computer Science courses.

3. COMPARING SYSTEM PERFORMANCE

The method described by Whale in [11] to evaluate the performance of plagiarism detection systems was also used in this study to compare the different systems. Using the terminology employed by Whale, matched pairs can be classified as either *essential* or *redundant*. Essential matches are the first $N-1$ matches to span a group of N submissions. All other matches, which do not identify any new suspicious programs or join previously unrelated subgraphs, are classified as redundant matches.

Two measures of detection reliability are provided by [11]. The first is the *precision* of a system, i.e. the proportion of *positive detections* in the group of plagiarisms retrieved by the system. Positive detections are essential matches that have been examined and verified as probable plagiarisms. High precision is important in preventing an instructor from wasting time in comparing improbable matches. The second measure is the *recall* of the system. The recall is the proportion of plagiarisms retrieved from the full set of assignment submissions. High recall is necessary to ensure that not too many plagiarisms go unnoticed.

Most plagiarism detectors give as output a set of pairs of suspected plagiarisms, ranked from most likely plagiarism to least likely. The number of essential matches included in the retrieved group can be seen as the *sensitivity* of the system. In order to allow valid comparisons to be made between systems, in each experiment the sensitivity has been held constant, i.e. equal numbers of essential matches are checked for each system.

An ideal plagiarism detector has a limiting sensitivity equivalent to a precision of 1. That is, plotting the cumulative incidence of positive detections against the essential matches, all m plagiarisms (i.e. positive detections) are reported in the first m essential matches. The other, lower limit is where the m plagiarisms are uniformly distributed among the n essential matches, equivalent to a precision of m/n. (An upper limit for the number of essential matches is $N/2 \le n \le N-1$, which arises if every submission is a plagiarism. In the case represented by the lower bound, every assignment matches exactly one other, while in the second case all n assignments are members of a single group.)

Any final measure of system performance must be influenced by both recall and precision. (Ideally one would want high recall and high precision, but in practice increasing recall must be traded off against a lowered precision.) After plotting the cumulative incidence of positive detections against the essential matches, the maximum vertical distance, D, between the detector's response and the line representing the uniformly-distributed case measures the success of the system. Specifically, it represents the *excess detections* made by the detector over pure chance.

In practice, m and n must be estimated. The uniform distribution line can be estimated by using the total number of positive detections made by all the systems, $\hat{m}$, instead of the total number of true plagiarisms, m, that would be found by an idealized detector. Secondly, $\hat{n} = N$ is used as a estimate for n, the number of essential matches that would be nominated at maximum sensitivity.

Two further refinements are suggested in [11]. First, D may only be measured above a certain line $p_{\min}$, to ensure acceptably high precision is maintained; $p_{\min} = 0.6$ is suggested. In the second refinement, D is normalized to create a *performance index* that lessens the dependence on the actual experiment. This is done by multiplying D by the reciprocal of $\hat{m}(1-\hat{m}/\hat{n})$, which gives a value in the range range between 0 (poor) to 1 (ideal). Both recommendations were adopted in the current study. (The only deviation from the

methodology described in [11] is that $\hat{n} = N$, rather than $N/2$. The change has been dictated by one of the experiments in which the number of postive detections is greater than $N/2$, which would make $\hat{m}/\hat{n} > 1$.)

3.1 Manual Verification

It is generally agreed that, while automated systems are useful for sifting through a large number of pairs, the matches should be verified by the lecturer. Manual inspection is not foolproof, but there are often small similarities, such as shared idiosyncracies which lead to a strong suspicion of plagiarism. Examples of these are: useless statements, e.g. `X := X div 1`, duplicated comments or identically misspelt comments, unusual data structures, or algorithmic choices.

4. RESULTS

The submissions for five Pascal assignments, drawn from the first-year and second-year Computer Science courses, were used to tune or test the performance of the three systems. The assignments were divided into two groups:

- Three assignments, Calendar, Loader and Simulator, were used to tune the reimplementations of the attribute-counting-metric systems. Nothing more will be said of them.
- A second pair of assignments, Ethernet and Formatter, which were used to compare performance of the tuned attribute-counting-metric systems with that of YAP3. The results are presented below.

The experiments were only attempted on syntactically correct programs; unparsable files were removed from the set.

4.1 Ethernet Results

Of the two assignments used to test the three detectors, this assignment is, in programming terms, the simpler. Specifically, a skeleton containing all the type-definitions, most of the global-variable definitions and a number of procedures (including the main program) is supplied by the lecturer. The remaining procedures are indicated by stubs, which are to be completed by the students. In other words, it was more an exercise in code reading and comprehension, than in code writing. In this experiment the line representing the minimum precision (i.e. $cum\_pd = 0.6esm$) falls below the line representing uniform distribution of postive detections. This is due to the large number of almost identical programs discovered. The response plot, this time involving all five detectors, is in Appendix A.1.

Number of assignments	126
Essential matches checked for each system	50
Number of positive detections	40

System	Positive Detections	Excess Detections	Performance Index
Accuse	25	1.40	0.17
Fai-Rob	31	4.00	0.50
YAP3	34	5.60	0.70

4.2 Formatter Results

This is an example of a somewhat more complex assignment, which contains enough scope for variability in the solutions to make the adjudication of plagiarisms more clear-cut. Closer examination reveals that there were relatively few plagiarisms in this assignment, and that they were relatively well disguised. Notice that Accuse was unable to detect any plagiarisms and those that the Faidhi-Robinson system found appeared rather far down the list of essential matches, resulting in a negative value for the excess detections. The response plot is in Appendix A.2.

Number of assignments	299
Essential matches checked for each system	71
Number of positive detections	14

System	Positive Detections	Excess Detections	Performance Index
Accuse	0	0	0
Fai-Rob	2	-1.56	-
YAP3	9	6.42	0.57

5. DISCUSSION AND CONCLUSIONS

It can be seen that the attribute-counting-metric systems performed better in the detection of plagiarism where very close copies were involved. This would usually be the case for a less-experienced group of students. On the other hand, the attribute-counting-metric systems were unable to detect partial plagiarisms, e.g. where a student has only copied part of another student's program. The systems also have problems detecting a plagiarism which involves the rewriting of in-line code as a procedure (or the reabsorption of a procedure into the calling code).

The performance indices for the attribute-counting-metric systems were considerably lower than those of the structure-metric systems in the Formatter assignment and in the Ethernet assignment. On this evidence, attribute-counting-metric systems could therefore not be recommended for use on any but the most inexperienced classes. While it can be argued that an optimal attribute-counting-metric system might outperform the structure-metric systems, the evidence presented above does reinforce Whales' conclusions, obtained on a different sample of attribute-counting-metric systems. Certainly, the metrics used in Accuse could not be recommended in any new attribute-counting-metric system due to their poor performance on both assignments, and while the Faidhi-Robinson metrics showed more promise, particularly the control-flow and

expression-percentage metrics, these have the disadvantage of being more difficult to implement. Overall, the evidence points to the fact that no single number, or set of numbers, can adequately capture the level of information about program texts that a structure-metric system is able to achieve.

6. FURTHER WORK

One avenue for further experimental work would be to explore more fully the differences between the plagiarisms of novice and more experienced programmers, to catalogue the devices they use and the effect of those ploys on different detectors. A second avenue for further work is based on the realization that establishing the actual set of positive detections will always be a matter of guesswork as long as data from actual student assignments are used. What might therefore be attempted is to solicit solutions to a typical assignment from readers of Internet newsgroups such as comp.edu. Specifically, a request would be broadcast over the net asking respondents to register with an independent adjudicator. Respondents will then be given the specifications of a programming assignment to be done in Pascal. Most respondents will simply return their solutions. However, the adjudicator will also arrange a percentage of "cheats" who will be given other respondents' solutions and asked to use these as the basis for their solutions.

REFERENCES

[1] ALLEN, F. E. AND J. COCKE, ''A Program Data Flow Analysis Procedure'', *CACM* **19**(3), pp. 137-147 (March 1976).

[2] BERGHEL, H. L. AND D. L. SALLACH, ''Measurements of Program Similarity in Identical Task Environments'', *SIGPLAN Notices* **19**(8), pp. 65-75 (August 1984).

[3] DONALDSON, JOHN L., ANN-MARIE LANCASTER, AND PAULA H. SPOSATO, ''A Plagiarism Detection System'', *Twelfth SIGCSE Technical Symposium*, St Louis, Missouri, pp. 21–25 (February 26-27, 1981) (SIGCSE Bulletin Vol. 13, No. 1, February 1981).

[4] FAIDHI, J. A. W. AND S. K. ROBINSON, ''An Empirical Approach for Detecting Program Similarity within a University Programming Environment'', *Computers and Education* **11**(1), pp. 11-19 (1987).

[5] GRIER, SAM, ''A Tool that Detects Plagiarism in Pascal Programs'', *Twelfth SIGCSE Technical Symposium*, St Louis, Missouri, pp. 15–20 (February 26-27, 1981) (SIGCSE Bulletin Vol. 13, No. 1, February 1981).

[6] GRUNE, DICK, *Concise Report on Algorithms in Sim*, (Report distributed with Sim software), June 1991.

[7] HALSTEAD, MAURICE HOWARD, *Elements of Software Science*, Elsevier (1977).

[8] OTTENSTEIN, KARL J., ''An Algorithmic Approach to the Detection and Prevention of Plagiarism'', *SIGCSE Bulletin* **8**(4), pp. 30-41 (1977).

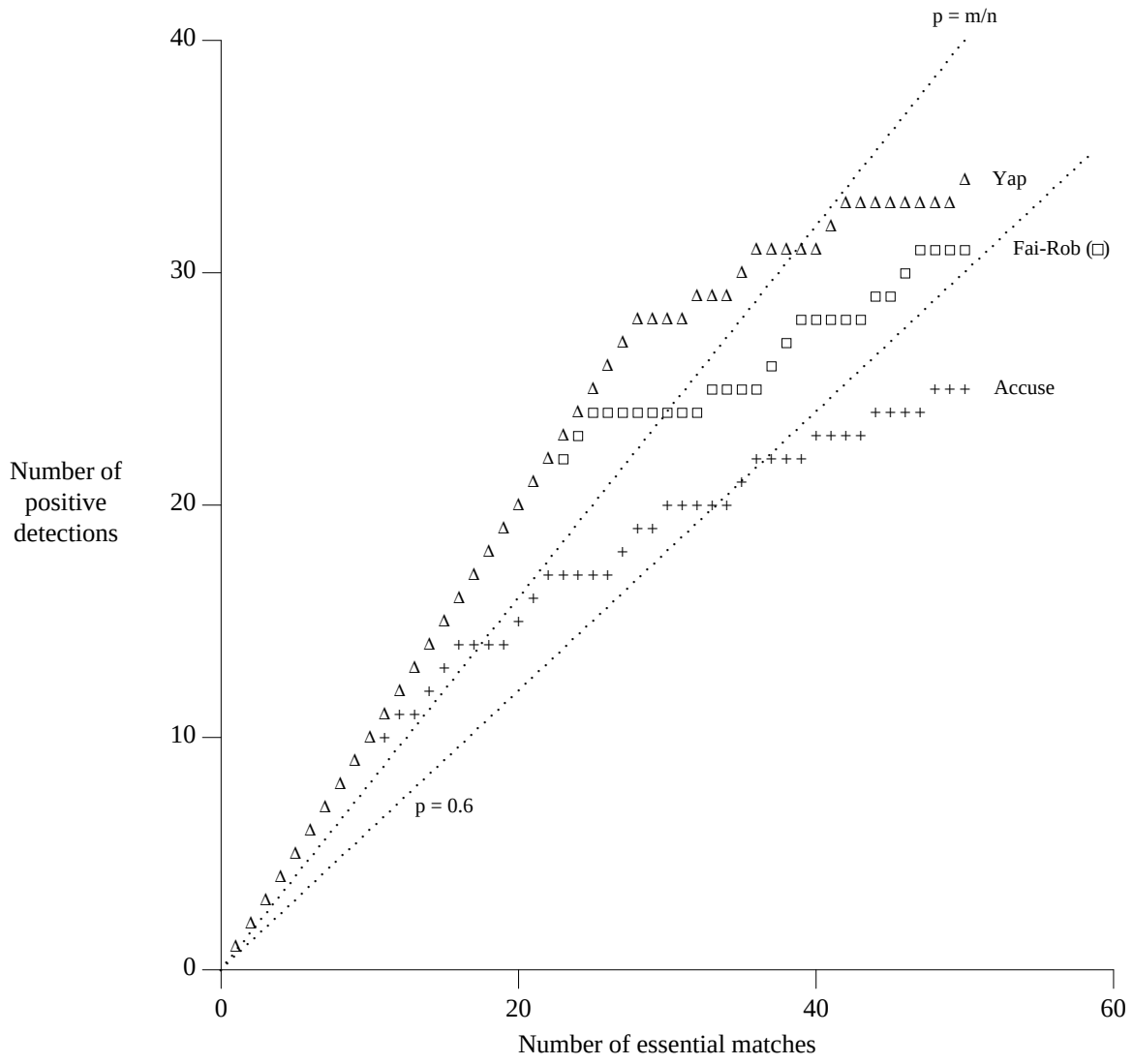
[9] PARKER, ALAN AND JAMES O. HAMBLEN, ''Computer Algorithms for Plagiarism Detection'', *IEEE Transactions on Education* **32**(2), pp. 94-99 (May 1989).

[10] RINEWALT, J. D., D. W. ELIZANDRO, R. C. VARNELL, AND S. A. STARKS, ''Development and Validation of a Plagiarism Detection Model for the Large Classroom Environment'', *Computers in Education (CoED)* **6**(3), pp. 9-13 (1986).

[11] WHALE, G., ''Identification of Program Similarity in Large Populations'', *The Computer Journal* **33**(2), pp. 140–146 (1990).

[12] WISE, MICHAEL J., ''Improved Detection of Similarities in Computer Program and other Texts'', *Twenty-Seventh SIGCSE Technical Symposium*, Philadelphia, U.S.A., pp. 130-134 (February 15-17, 1996).

Appendix A.1. Response plot for Ethernet assignment



Appendix A.2. Response plot for Formatter assignment