# JPlag: Finding plagiarisms among a set of programs

Lutz Prechelt (prechelt@ira.uka.de)
Guido Malpohl (malpohl@gmx.de)
Michael Phlippsen (phlipp@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/608-7343
http://wwwipd.ira.uka.de/EIR/

Technical Report 2000-1

March 28, 2000

**Abstract**

JPlag is a system that finds pairs of similar programs among a given set of programs. It has successfully been used in practice to detect plagiarisms among student Java exercise submissions. Support for the languages C, C++ and Scheme is also available. This report presents the design of JPlag, in particular the comparison algorithm, and carefully evaluates JPlag's performance on 12 rather different sets of Java programs. The results indicate that JPlag will find all plagiarisms with only very few exceptions. The execution time is less than one minute for submissions of 100 programs of several hundred lines each.

# Contents

# Chapter 1

# Introduction

Around the world, many millions of programming exercises are being turned in every year by students of Computer Science and other subjects. In most cases, their instructors have the uneasy feeling that a few of these programs are in fact copies — verbatim or with some modifications — of programs supplied by somebody else who is taking the same course at the same time. We call such copies *plagiarisms*.

Very few instructors have the patience to thoroughly search for plagiarisms; although finding plagiarisms is possible, it is much too time-consuming in practice. If any, instructors find duplicates only by accident, e.g., if a student has forgotten to replace the name of the friend in the head of the program source text or if two programs produce the same weird failure for a test input.

A powerful automated search that finds similar pairs among a set of programs would be most helpful — if, and only if, that search can discriminate well enough between incidental similarities on the one hand and actual plagiarisms on the other hand.

## 1.1 JPlag

We have built such a system, called JPlag. It analyzes program source text written in Java, Scheme, C, or C++. We will consider only the Java mode here. The program is publicly available as a WWW service. We have used the program very successfully in graduate level courses and in first-year CS courses with more than 100 participants each — the program scales to several hundred submissions easily.

JPlag takes as input a set of programs, compares these programs pairwise (computing for each pair a total similarity value and a set of similarity regions), and provides as output a set of HTML pages that allow for exploring and understanding the similarities found in detail.

JPlag works by converting each program into a stream of canonical tokens and then trying to cover one such token string by substrings taken from the other (string tiling).

## 1.2 Related work

JPlag is by far not the first system built for finding plagiarisms. This section discusses some of the previous ones.

**Feature comparison.** The early attempts at plagiarism detection are usually based on the notion of a feature vector. These systems compute for each program a number of different software metrics, so that each program

is mapped to a point in an $n$-dimensional cartesian space. The systems then consider sets of programs that lie close to each other to be possible plagiarisms.

The earliest such system we could find is Ottenstein's [9] from 1976. It uses only the basic Halstead [6] metrics (number of unique operators $\eta_1$, number of unique operands $eta_2$, total number of operators $N_1$, total number of operands $N_2$) on Fortran programs and considers only programs to be plagiarisms where all four values coincide.

Later systems such as those of Donaldson, Lancaster, and Sposato [3], of Grier [5], of Berghel and Sallah [2], or of Faidhi and Robinson [4] introduce a much larger number of metrics (up to 24) and notions of similarity for the resulting feature vector in order to improve performance.

Unfortunately, these efforts are only moderately successful [10]. Systems based on feature vectors can hardly have good performance, because summing up a metric across the whole program simply throws away too much structural information. Note that this deficiency cannot be removed by adding further dimensions to the comparison: Such systems will either be very insensitive (and hence easy to fool by simple program transformations) or will be sensitive and will come up with a large fraction of false positives.

**Structure comparison.** More recently, much more powerful computers made a better approach viable, namely the direct comparison of program *structure* rather than of just summary indicators. Some of these systems are hybrids between structure and metric comparison, e.g. [3, 7], others rely on structure comparison alone. In the latter case, the approach is usually to convert the program into a stream of tokens (thus ignoring easily changeable information such as indentation, line breaks, comments etc.) and then comparing these token streams to find common segments.

The most advanced other systems in terms of their plagiarism detection performance are probably Michael Wise's YAP3 [12] and Alex Aiken's MOSS [1].

JPlag uses roughly the same basic comparison algorithm as developed by Wise, but adds different optimizations for improving its run time efficiency. MOSS uses a slightly different approach for even much higher speed at the cost of some detection performance[1].

The distinguishing characteristic of JPlag is the unique combination of the following features:

- JPlag is available as a web service. The email service of MOSS is roughly comparable, but nothing similar is available for YAP3.
- JPlag has a powerful user interface for understanding the results. It has meanwhile inspired a similar interface for MOSS, no such interface is available for YAP3.
- JPlag is resource-efficient and scales to large submissions. MOSS is even better in this respect, but YAP3 is inferior.
- JPlag has very good plagiarism detection performance (as will be shown in Chapter 4). We cannot provide a direct quantitative comparison to MOSS and YAP3, since we were not able to get YAP3 to run and MOSS does not return the comprehensive pairwise similarity output required for such a study. Based on what we know about those systems, however, JPlag's performance ought to be as good as or slightly better than that of MOSS and YAP3. The results obtained in the present study should generalize fairly well to these other two systems.

The distinguishing characteristic of the present study about JPlag is the careful evaluation of JPlag's performance. Although several performance studies such as [10, 11, 12] exist for earlier systems, none of them is anywhere nearly as thorough as ours, neither with respect to the quality of the input data used, nor in the level of detail of the evaluation itself.

[1] However, MOSS is able to ignore base code present in almost all submissions automatically.

## 1.3 Structure of this report

Chapter 2 shortly describes the external appearance of JPlag, in particular the web service through which it can be accessed and the GUI by which it presents its search results. Chapter 3 describes the core algorithm that compares two programs and computes the similarity value and the set of pairs of corresponding regions.

Chapter 4 presents an empirical evaluation of JPlag on four real sets of student programs and on sets of plagiarisms specifically created to fool such a plagiarism search. The evaluation comprises measurements of the discrimination performance, a study of the sensitivity of the algorithm's free parameters, and a detailed analysis of the camouflaging strategies used by plagiarism authors.

The appendix supplies further detail: First, the individual results for each plagiarism submitted explicitly by an outside person as an answer to our call to "fool our plagiarism detector" and, second, a description of JPlag's Java token set.

# Chapter 2

# Using JPlag

Two interesting features of JPlag are its availability as a web service and its unique graphical user interface for understanding the results.

## 2.1 The JPlag WWW service

The JPlag core is a single Java program that is called with the name of a directory containing the program source files as its only argument and that produces a directory full of HTML files as output. The structure of the input directory is such that it contains a number of subdirectories, each of which supplies one program to be compared. Each subdirectory may contain an arbitrary number of source files that together represent the program.

For users of JPlag around the world, we have created a web service through which they can use JPlag without installing it on a local machine:

http://wwwipd.ira.uka.de/jplag/

Users apply for a JPlag account by email. They are sent a username and password and can then log in and use JPlag at any time. For using the service, the user supplies a directory of submissions as described above on his or her local machine. The submissions are uploaded to the JPlag server and the server applies JPLag to the submissions. It stores the resulting HTML pages to be viewed (via a normal HTTP session or after download) by a standard web browser. The web pages containing the results are visible for this particular JPlag user only. The web pages remain on the server until they expire after several days or are overwritten by a new submission of the same user. Multiple users do not interfere with each other.

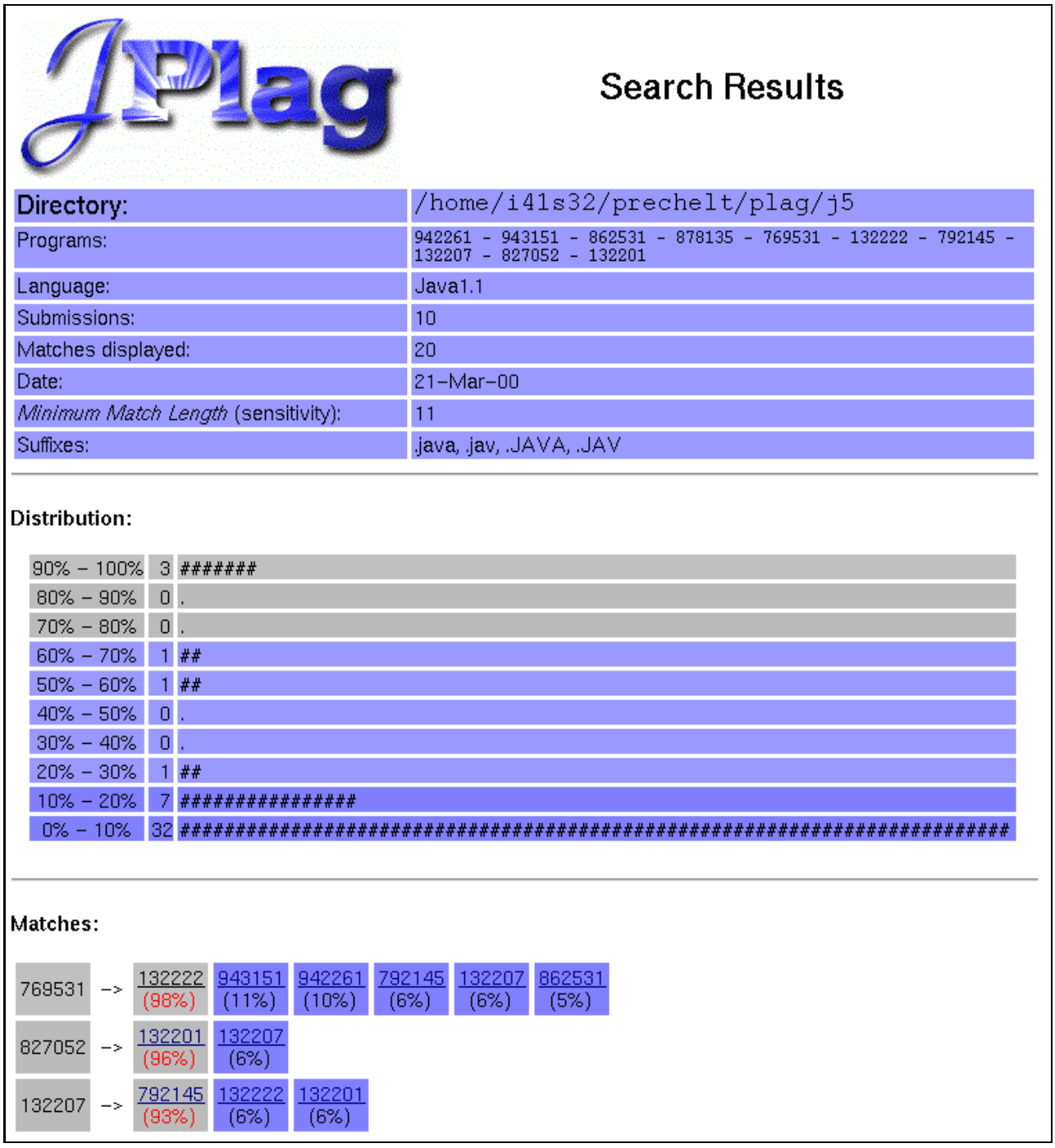
## 2.2 The user interface for presenting the results

An important question for plagiarism detection systems is the presentation of their results: Similarities of 0% or 5% can be represented by the similarity value alone — this clearly is no plagiarism. Likewise, similarities of 100% can also be represented by the similarity value alone — this clearly is a plagiarism. But what if the similarity is 40%? Such cases should usually be investigated by a human being for final judgement. JPlag supports this investigation by a unique user interface.

For a set of submitted programs, JPlag generates a set of HTML pages that present the results. At the top level, an overview page (see Figure 2.1 for a partial example) presents a histogram of the similarity values found

for all program pairs. Given this histogram one usually can identify a range of similarity values that doubtless represent plagiarisms and a range of values which doubtless represent non-plagiarisms. Those program pairs with similarities in between those ranges should be investigated further. A list of the highest-similarity pairs shown on the web page below the histogram allows for selecting these pairs.

For each pair selected by the user, a side-by-side comparison of the programs is then shown (see Figure 2.2 for a partial example). Ranges of lines that have been found to be corresponding in both files are marked with the same color. A hyperlink arrow at each region, when clicked, aligns the opposite half of the display such that the corresponding region is shown. Likewise, one can jump to each pair of corresponding regions from a correspondence table shown at the top which lists for each region its color code, position in each of the files,


JPlag

Search Results

| | |
|---|---|
| Directory: | /home/i41s32/prechelt/plag/j5 |
| Programs: | 942261 - 943151 - 862531 - 878135 - 769531 - 132222 - 792145 - 132207 - 827052 - 132201 |
| Language: | Java1.1 |
| Submissions: | 10 |
| Matches displayed: | 20 |
| Date: | 21-Mar-00 |
| *Minimum Match Length* (sensitivity): | 11 |
| Suffixes: | .java, .jav, .JAVA, .JAV |

**Distribution:**

| | | |
|---|---|---|
| 90% – 100% | 3 | ####### |
| 80% – 90% | 0 | . |
| 70% – 80% | 0 | . |
| 60% – 70% | 1 | ## |
| 50% – 60% | 1 | ## |
| 40% – 50% | 0 | . |
| 30% – 40% | 0 | . |
| 20% – 30% | 1 | ## |
| 10% – 20% | 7 | ################ |
| 0% – 10% | 32 | ################################################################################ |

**Matches:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 769531 | -> | 132222 (98%) | 943151 (11%) | 942261 (10%) | 792145 (6%) | 132207 (6%) | 862531 (5%) |
| 827052 | -> | 132201 (96%) | 132207 (6%) | | | | |
| 132207 | -> | 792145 (93%) | 132222 (6%) | 132201 (6%) | | | |

Figure 2.1: The top of an example JPlag results overview page

and length in tokens.

This unique presentation makes it very easy to judge whether or to which degree the pair of programs should in fact be considered plagiarized. For instance it will quickly become clear if the two programmers have shared part of the program but created the rest independently or if plagiarized program parts were camouflaged by artificial changes.

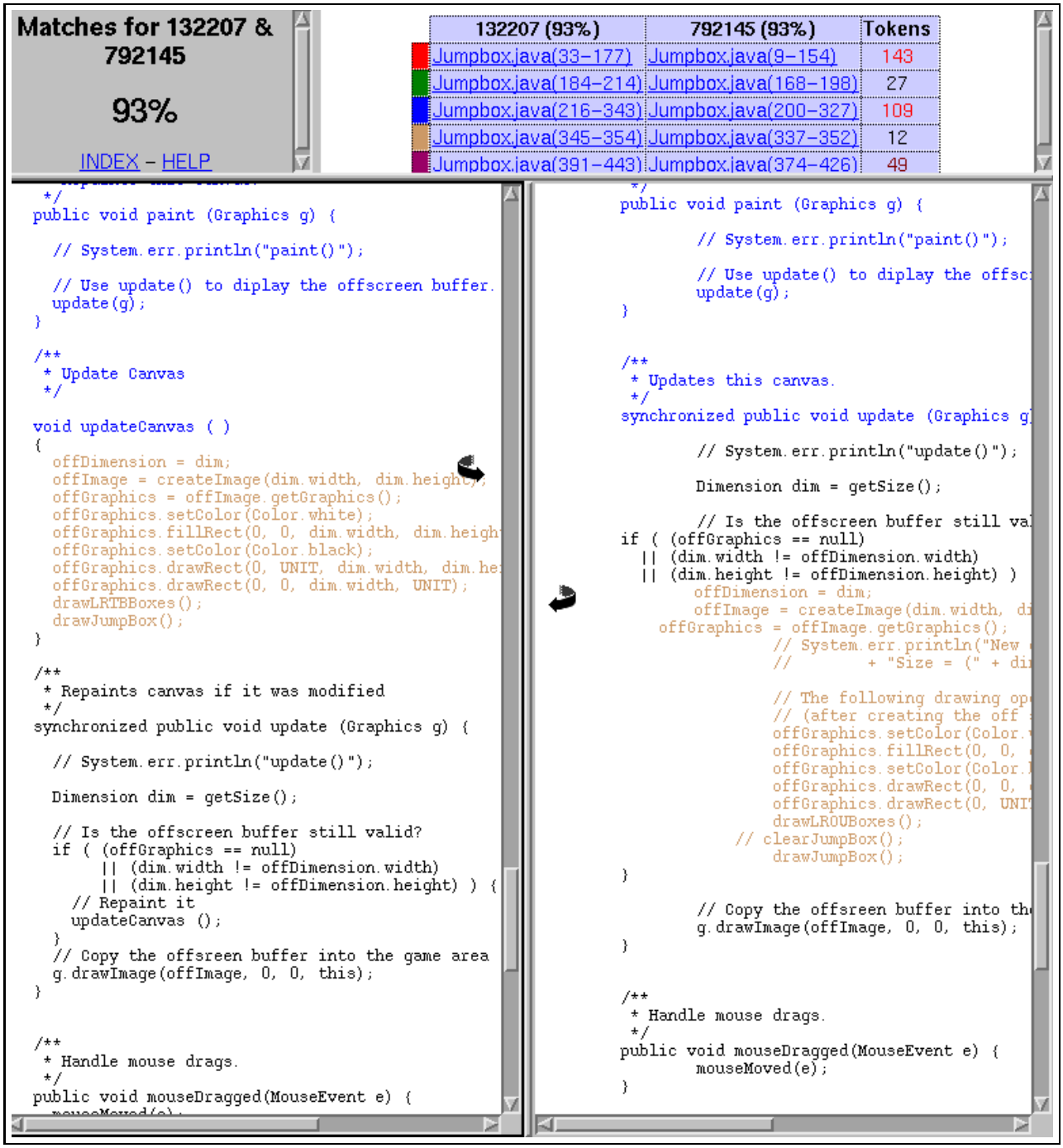



Figure 2.2: An example part of a JPlag results display page for one pair of programs

# Chapter 3

# The JPlag comparison algorithm

This chapter describes how the JPlag system computes the similarity of a pair of programs.

JPlag operates in two phases:

1. All programs to be compared are parsed (or scanned, depending on the input language) and converted into token strings.
2. These token strings are compared in pairs for determining the similarity of each pair. During each such comparison, JPlag attempts to cover one token stream with substrings ("tiles") taken from the other as well as possible. The percentage of the token streams that can be covered is the similarity value. The corresponding tiles are visualized by the interface described in Section 2.2.

## 3.1 Converting the programs into token strings

The front-end process of converting the programs into token strings is the only language-dependent process in JPlag. Three front-end implementations currently exist: The ones for Java and for Scheme both implement a full parser for the respective language. The third front end for C++ (or C) consists of only a scanner. A parser front-end has the advantage of allowing for more semantic information to be represented in the token string. For instance in Java, we can create a BEGINMETHOD token instead of just a OPEN_BRACE token.[1]

As a rule, tokens should be chosen such that they characterize the essence of a program (which is difficult to change by a plagiarist) rather than surface aspects. For instance, whitespace and comments should never produce a token, since they are the most obvious points of attack. Some sorts of tokens may or may not be considered useful. For instance, the default token set for Java completely ignores expressions except for assignments and method calls; see the example in Table 3.1. The complete token lists are printed in Appendix B.

The number of the line from which the token originates is stored within the token object so that it is possible to pinpoint any similarities between two files with respect to the original source files later. Note that internally each token is represented by a single character for the comparison.

## 3.2 Comparing two token strings

The algorithm used to compare two token strings is essentially "Greedy String Tiling" as introduced by Michael Wise [13]. When comparing two strings $A$ and $B$, the aim is to find a set of substrings that are the same and

[1] However, our evaluation suggests that JPlag is fairly robust against changes of the token set; see Chapter 4.

Table 3.1: Example Java source text and corresponding tokens.

| | Java source code | Generated tokens |
|---|---|---|
| 1 | `public class Count {` | BEGINCLASS |
| 2 | `  public static void main(String[] args)` | VARDEF,BEGINMETHOD |
| 3 | `            throws java.io.IOException {` | |
| 4 | `    int count = 0;` | VARDEF,ASSIGN |
| 5 | | |
| 6 | `    while (System.in.read() != -1)` | APPLY,BEGINWHILE |
| 7 | `      count++;` | ASSIGN,ENDWHILE |
| 8 | `    System.out.println(count+" chars.");` | APPLY |
| 9 | `  }` | ENDMETHOD |
| 10 | `}` | ENDCLASS |

satisfy the following rules:

1. Any token of $A$ may only be matched with exactly one token from $B$. This rule implies that it is impossible to completely match parts of the source text that have been duplicated in a plagiarized program.
2. Substrings are to be found independent of their position in the string. This rules implies that reordering parts of the source code is no effective attack.
3. Long substring matches are preferred over short ones, because they are more reliable. Short matches are more likely to be spurious.

Applying the third rule sequentially for each matching step leads to a greedy algorithm which consists of two phases:

**Phase 1:** In this phase, the two strings are searched for the biggest contiguous matches. This is done by 3 nested loops: The first one iterates over all the tokens in string $A$, the second one compares this token $T$ with every token in $B$. If they are identical, the innermost loop tries to extend the match as far as possible. These nested loops collect the set of all longest common substrings.

**Phase 2:** Phase 2 marks all matches of maximal length found in Phase 1. This means that all their tokens are marked and thus may not be used for further matches in Phase 1 of a subsequent iteration. This guarantees that every token will only be used in one match and thus satisfies the first rule from above. In the terminology of Wise, by marking all the tokens a match becomes a *tile*.

Additionally, some of the matches may overlap. In this case, the first match found is chosen and the others are ignored.

These two phases are repeated until no further matches are found. Since the length of the maximal matches decreases by at least 1 in each step, the algorithm is guaranteed to terminate.

If matches of any length were allowed, matches of just a few tokens would frequently occur by chance. Hence, for avoiding spurious matches, a lower bound for the length of a match is defined, called the “Minimum Match Length”.

Table 3.2 shows the pseudo-code of the algorithm. The $\oplus$ operator in line 12 adds a match to a set of matches if and only if it does not overlap with one of the matches already in the set. The triple $match(a, b, l)$ denotes an

Table 3.2: Greedy String Tiling

```
0   Greedy-String-Tiling(String A, String B) {
1       tiles = {};
2       do {
3           maxmatch = MinimumMatchLength;
4           matches = {};
5           Forall unmarked tokens A_a in A {
6               Forall unmarked tokens B_b in B {
7                   j = 0;
8                   while (A_{a+j} == B_{b+j} &&
9                           unmarked(A_{a+j}) && unmarked(B_{b+j}))
10                      j++;
11                  if (j == maxmatch)
12                      matches = matches ⊕ match(a, b, j);
13                  else if (j > maxmatch) {
14                      matches = {match(a, b, j)};
15                      maxmatch = j;
16                  }
17              }
18          }
19          Forall match(a, b, maxmatch) ∈ matches {
20              For j = 0 ... (maxmatch − 1) {
21                  mark(A_{a+j});
22                  mark(B_{b+j});
23              }
24              tiles = tiles ∪ match(a, b, maxmatch);
25          }
26      } while (maxmatch > MinimumMatchLength);
27      return tiles;
28  }
```

association between identical substrings of $A$ and $B$, starting at positions $A_a$ and $B_b$ respectively, with a length of $l$.

For every unmarked token $A_a$ in $A$, Phase 1 begins by looking for an identical token ($B_b$) in string $B$ (lines 5-8). Then succeeding tokens are compared in order to extend the match as far as possible (lines 7-10). None of these tokens must previously be marked. As soon as the two tokens ($A_{a+j}$ and $B_{b+j}$) differ or one of them is marked, a maximal match of length $j$ is found (line 11). Now the program checks whether this match is at least as long as the longest ones found before, whose length is stored in $maxmatch$. If $j$ equals $maxmatch$, another maximal match is found and added to the set $matches$ (line 12). If $j$ is bigger than $maxmatch$, the set $matches$ is re-initialized to contain only the new match (line 14) and then the new maximal length $j$ is stored in $maxmatch$ (line 15).

After this phase, $matches$ contains all the maximal matches. The set is empty if no matches of at least length $MinimumMatchLength$ were found.

The second phase begins in line 19. The set $matches$ is traversed and each match is marked (lines 20-23) and added to the result set $tiles$.

Then the algorithm iterates the two phases until no match longer than $MinimumMatchLength$ was found and the algorithm terminates with the resulting set $tiles$. The matches contained in this set satisfy the rules given at the beginning of this section. Every match can now be traced back to corresponding ranges of lines in the source code.

Intuitively, the similarity measure should reflect the fraction of tokens from the original programs that are covered by matches. There are two sensible choices: If we want a similarity of 100% to mean that the two token strings are equivalent, we must consider both token strings in the computation. If, on the other hand, we prefer that each program that has been copied completely (and then perhaps extended) will result in a similarity of 100%, we must consider the shorter token string only. We choose the first variant here, which results in the following similarity measure $sim$:

$$\begin{aligned} sim(A,B) &= \frac{2 \cdot coverage(tiles)}{|A|+|B|} \\ coverage(tiles) &= \sum_{match(a,b,length) \in tiles} length \end{aligned}$$

## 3.3 Run time complexity of the basic algorithm

In each iteration of the algorithm, the first phase is more expensive than the second one. The reason is that there are up to $(|A| - MML) \cdot (|B| - MML)$ matches ($MML := MinimumMatchLength$) that can be found. For the second phase, in the worst case all the tokens have to be marked, which can be done in time linear to the length of the shorter token string.

**Worst case.** To make things easier, it is assumed from now on that both strings have the same length: $|A| = |B| = n$. In the worst case, all three nested loops are executed to their fullest extent. This means that in each iteration the algorithm finds only one maximal match, this match only one token shorter than the one found in the previous round, the last one found has only length 1 (for $MML = 1$), and the match is always located at the very end of the remaining string. In this case, $k$ iterations cover a string of length $n = \frac{k(k+1)}{2}$ and there are roughly $\sqrt{2n}$ different matches in a string of length $n$.

Now we count the number of steps. In the last iteration there is only one match of length 1 left and only one possible place in both strings where it can be found. To check this, exactly one comparison has to be made, whereas the match in the step before (length: 2) could be found in two places within the space of 3 unmarked tokens. To decide the placement of the match, in the worst case $2 \cdot 2$ token comparisons are required for each location which makes $2 \cdot 2^2$ comparisons in total. There are 4 possible locations for matches of length 3 within the unmarked 6 tokens at that time. To make the correct choice, $3 \cdot 4^2$ comparisons are required.

Using this pattern, the following formula for calculating the number of comparisons required for a match of length $k$ can be found by induction:

$$k \cdot (\sum_{i=1}^{k} i - k + 1)^2 = \frac{k}{4}(k^2 - k + 2)^2$$

which has an upper bound of $k \cdot n^2$, since the length $k$ of a substring is always less than $k \leq \sqrt{2n}$ as argued above.

Summing up all the comparisons of $\sqrt{2n}$ iterations, the upper bound for the worst case total number of comparisons is:

$$\sum_{k=1}^{\sqrt{2n}} (kn^2) = n^3 + \sqrt{1/2} \quad n^{\frac{5}{2}}$$

It follows that the complexity of the algorithm is $O(n^3)$.

**Best case.** Even if two completely different strings are compared (i.e. not even one token from string $A$ can be found in $B$), the algorithm needs to compare at least each token from $A$ with all the tokens in $B$. To do that, $|A| \cdot |B|$ comparisons are needed, which results in a run time of $O(n^2), n = max(|A|, |B|)$.

## 3.4 Run time optimizations

Although the worst case complexity can not be reduced, it is possible to bring down the average complexity for practical cases to about $O(n)$.

This impressive improvement is achieved by an idea from the Karp-Rabin algorithm [8]. This algorithm tries to find all occurrences of a short string (the "pattern" $P$) in a longer string (the "text" $T$) by using a hash function. To do that, the hash values of all substrings with length $|P|$ in $T$ are calculated. This can be done in linear time by using a hash function $h$ that is able to compute the value of $h(T_t T_{t+1}..T_{t+|P|-1})$ from the values of $h(T_{t-1}T_t..T_{t+|P|-2})$, $T_{t-1}$ and $T_{t+|P|-1}$. All the hash values are then compared with the value of $P$. If two values are the same, a character-wise comparison takes place to verify that an occurrence of $P$ in $T$ has been found. The complexity of this algorithm in practice is almost linear.

We use the idea of comparing whole substrings by their hash values and make the following modifications of the matching algorithm:

1. The hash values are computed for all substrings of length $MML$ in $A$ and $B$. As said before, this can be done in time $O(|A| + |B|)$ and allows for a very quick comparison of two substrings later.
2. Each hash value from string $A$ is then compared with each one from $B$. If two values are the same, a possible match of two substrings beginning at this token has been found. This possible match is verified by comparing the substrings token by token. At the same time the algorithm tries to extend the match as far as possible beyond the range that is covered by the hash function.
3. Still, a quadratic number of pairs of hash values needs to be compared. To overcome this problem, a hash table is used for locating the substrings from $B$ that have the same hash value as a given substring from $A$. This reduces the effort to a linear number of steps.

With these modifications, the worst case complexity is still $O(n^3)$, since all the substrings have to be compared token by token, but in practice a complexity of less than $O(n^2)$ is usually observed.

Two additional tricks are also used to further reduce the run time:

- The token-by-token comparison does not start at the beginning of the substring, but rather backwards after $maxmatch$ tokens. This is the position where the new match should at least end to be interesting for the current iteration. This is an improvement, since the substrings represent programs (which contain recurring phrases) and hence differences are more likely to occur farther away from the beginning. After verifying the substring on the first $maxmatch$ tokens, the algorithm attempts to extend it further.
- When comparing two strings $A$ and $B$, $A$ is always chosen to be shorter than $B$ if possible, because string $A$ is *always* traversed completely whereas $B$ is only accessed through the hash table.

# Chapter 4

# Empirical evaluation of JPlag

From the design of JPlag, several questions naturally arise:

- What fraction of plagiarisms will JPlag detect?
- How many “innocent” program pairs will be flagged as plagiarisms?
- How do these properties change for different program length?
- How do these properties change for different program structure?
- How robust is the JPlag algorithm against changes of its parameters?

These questions will be answered in the current chapter.

We have performed an empirical study based on several sets of real student programs plus a number of explicitly made plagiarisms. We will first describe the setup of the study, including the program sets, and then various aspects of the study results.

## 4.1 The setup of our study

This section describes the sets of programs, the sets of free parameters considered in the evaluation, and the criteria used for quantifying the results.

### 4.1.1 The original program sets used: i12orig, i27orig, i51orig, j5orig

We used four different kinds of programs as the benchmarks in our study (code-named i12, i27, i51, and j5). Three were programming exercises from a second-semester informatics course, one was from a graduate advanced programming course that introduced Java and the AWT to experienced students. The original sets of programs obtained from these courses will be called i12orig, i27orig, i51orig, and j5orig. Other name suffixes than ‘orig’ will be used for identifying other program sets related to the same exercises; see Section 4.1.2.

Table 4.2 gives a short overview of the four original program sets. Figure 4.1 shows the distribution of program lengths in each program set. The following paragraphs shortly describe the programs of each program set, in particular any salient features.

**i12, i12orig: maximize flow in a network.** An algorithm for computing the maximum flow through a directed graph with capacity-weighted edges. The program is based on a reusable GraphSearch class not included in the source investigated here.
This program set shows rather large variability in program length and structure. It contains 2 programs that are plagiarisms of others (that is, 4 programs forming 2 plagiarism pairs).

**i27, i27orig: multiply permutations.** Multiply two permutations represented as permutation matrices represented by an array of integers indicating the position of the 1 for each row.
This is a very simple program with a rather fixed structure. We may expect that even programs written independently will look very similar. The expectation is corroborated by the rather small variability in program length as shown in Figure 4.1. Therefore, this program is a very hard test for JPlag. The program set contains 6 programs that are plagiarisms of others (that is, 12 programs forming 6 plagiarism pairs).

**i51, i51orig: k-means procedure.** The k-means clustering procedure for the one-dimensional case, using absolute distance as the distance function and initializing with equidistant means over the range of the data.
This program set does not contain any plagiarisms at all.

**j5, j5orig: Jumpbox.** A simple graphical game where the player has to move the mouse into a square jumping around on the screen. The mouse must enter the square from a particular side indicated by an ever-changing color code.
This is the largest program on average and also a large program set (with 59 programs). The problem allows for fairly large variation in some aspects of the program design. The program set contains 4 programs that are plagiarisms of others (that is, 8 programs forming 4 plagiarism pairs). It also contains two other pairs that have a lot of similarity, but that we do not consider to be actual plagiarisms. For one of these, the two programmers have apparently worked together in an early phase, but then finished their programs independently. For the other, the two programs share a common fraction used as a base and taken from an earler AWT programming exercise.

We determined the plagiarism pairs within these programs by a careful manual comparison of each pair. Note that not all programs in all program sets are necessarily operational.

### 4.1.2 Artificial program sets: name suffixes plags, all, small, triple

In order to investigate the behavior of JPLag more closely, the amount of actual plagiarisms in our program sets is insufficient. Therefore we collected further plagiarisms by publicly posting a “Call for plagiarisms” on

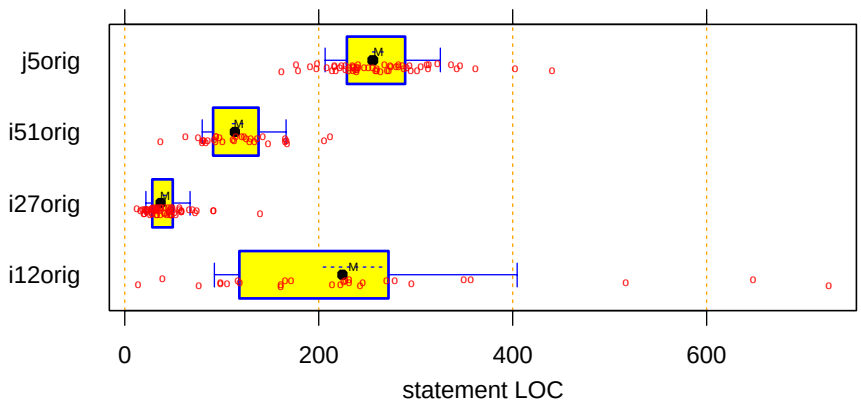



Figure 4.1: Distribution of program lengths in the original program sets, measured in non-comment lines of code. Each small circle represents one program (the vertical jittering is for legibility only). The Box indicates the 25% to 75% quantiles, the whiskers indicate the 10% and 90% quantiles, the fat dot is the median, the M and dashed line indicate the mean and plus/minus one standard error of the mean.

Table 4.2: The original program sets: program set code name, purpose of the programs, number $n$ of programs, mean length of programs in non-comment lines of code, number of plagiarism program pairs within the program set.

| set | purpose | n | Ø LOC | plag pairs |
|---|---|---|---|---|
| i12orig | maximize flow | 28 | 236 | 2 |
| i27orig | multiply permutations | 60 | 43 | 6 |
| i51orig | k-means procedure | 32 | 118 | 0 |
| j5orig | Jumpbox | 59 | 263 | 4 |

a web site and collecting submissions via email. Persons who answered our call downloaded a source program from our web page, modified it, and sent it back. They were given the following directions:

> Assume you are participating in a programming course but don't want to prepare the current exercise yourself (for whatever reason: You are so busy writing programs for money that you don't have time; you are incompetent, because you never prepared any exercise yourself in your life; you are bored, because the exercise is too simple; you are in love. Choose one.). Fortunately, you have a friend who has already produced a solution and is willing to give his/her source code to you. However, you do not want to submit an exact copy of somebody else's solution, so you invest a little, but just a little, time to change his file so it will look different.
>
> [...]
>
> Take a short period of time to massage the source code in the manner that you would use if you wanted to plagiarize the program but cover up the similarities. The goal is to obtain a good camouflage but invest only a small amount of work. [...] You may do anything you want to the source file: from 'nothing at all' up to 'rewriting it all from scratch' or anything in between. However, [...] remember that the goal is not to use toooo much time. The latter point in particular means that you normally should not write new code of your own but only massage the original.

We collected additional plagiarisms in this manner for two of our program sets: i27orig and j5orig. For each of these, we posted six of the original program versions, one of which was to be chosen at will.

Note that having more than one plagiarism for one original leads to a much higher number of plagiarism pairs overall, because for an original program P and three plagiarisms A, B, C derived from it, not only (P,A), (P,B), and (P,C) are plagiarism pairs, but also (A,B), (A,C) and (B,C). The number of pairs grows quadratically with the number of plagiarisms.

Table 4.3 shows the number of new plagiarisms collected for each of the original programs we posted and the resulting number of new plagiarism pairs we obtained.

Based on these additional plagiarisms, we formed additional program sets for the evaluation of JPlag:

**i27plags and j5plags** contain only all programs for which a plagiarism exists as well (original or collected). In these program sets, a large fraction of all program pairs is a plagiarism pair.

**i27all and j5all** are the union of i27orig and i27plags (or j5orig and j5plags, respectively).

**i27small and j5small** each contain only ten programs, representing five plagiarized pairs.

**i27tripl** contains 26 programs: 4 triples of plagiarisms and all 7 other pairs of plagiarisms.

**j5triple** contains 21 programs: 6 triples of plagiarisms and 3 non-plagiarized other programs.

As far as possible, i27small and i27tripl are disjoint and j5small and j5triple are disjoint.

Table 4.4 gives an overview of the name, size, and plagiarism content of all program sets used in the study.

Table 4.3: Number $n'$ of additional plagiarisms collected for each of the six original programs posted (for each of the two program sets). The last column indicates the number of new plagiarism pairs resulting from these $n'$ new collected plagiarisms. This number is $(n'+1) \cdot n'/2$, because all program pairs involving any of the $n'$ or the original program forms a plagiarism pair.

| set | prog | $n'$ | plag pairs |
|---|---|---|---|
| i27 | 1 | 7 | 28 |
| i27 | 2 | 2 | 3 |
| i27 | 3 | 8 | 36 |
| i27 | 4 | 1 | 1 |
| i27 | 5 | 7 | 28 |
| i27 | 6 | 0 | 0 |
| j5 | 1 | 5 | 15 |
| j5 | 2 | 7 | 28 |
| j5 | 3 | 14 | 105 |
| j5 | 4 | 5 | 15 |
| j5 | 5 | 4 | 10 |
| j5 | 6 | 5 | 15 |

Table 4.4: Overview of all program sets: Name, size in number of programs, size in number of program pairs, number of pairs that are plagiarism pairs, fraction of plagiarism pairs.

| set | $n$ | $n \cdot (n-1)/2$ | plag pairs | % plag pairs |
|---|---|---|---|---|
| i12orig | 28 | 378 | 2 | 0.5 |
| i27orig | 60 | 1770 | 6 | 0.3 |
| i27plags | 42 | 861 | 102 | 11.8 |
| i27all | 85 | 3570 | 102 | 2.9 |
| i27small | 10 | 45 | 5 | 11.1 |
| i27tripl | 26 | 325 | 19 | 5.8 |
| i51orig | 32 | 496 | 0 | 0.0 |
| j5orig | 59 | 1711 | 4 | 0.2 |
| j5plags | 54 | 1431 | 192 | 13.4 |
| j5all | 99 | 4851 | 192 | 4.0 |
| j5small | 10 | 45 | 5 | 11.1 |
| j5triple | 21 | 210 | 18 | 8.6 |

### 4.1.3 Other parameters varied in the study

As described in Chapter 3, there are two free parameters in the JPlag algorithm: the mimimum match length and the token set used. Both of these were varied in our study as well.

Besides the default token set, (called “normal”) we also used a fairly minimal token set (called “struc”) containing only tokens related to control flow (if, while, switch, return, throw, etc.) and program block structure (begin/end of class, method, interface etc.). Further, we used a maximal token set (called “full”) containing all possible tokens. Two intermediate token sets “op” and “opmore” that are supersets of “struc” and subsets of “full” were also used, but the results are not very interesting and will not be reported separately.

We used minimum match lengths of 3, 4, 5, 7, 9, 11, 14, 17, 20, 25, 30, and 40.

### 4.1.4 Evaluation criteria, Definitions

The evaluation will mostly be based on the measures "precision" ($P$) and "recall" ($R$), defined as follows. Assume we have a set of $n$ programs. This set allows to form $p = n \cdot (n-1)/2$ pairs. Assume further that $g$ of these pairs are plagiarism pairs, i.e., one program was plagiarized from the other or both were (directly or indirectly) plagiarized from some common ancestor that is also part of the program set.

Now assume that we let our fully automatic plagiarism detector run and it returns $f$ pairs of programs flagged as plagiarism pairs. If $t$ of these pairs are really true plagiarism pairs and the other $f - t$ are not, then we define precision and recall as

$$P := 100 \cdot t/f \qquad R := 100 \cdot t/g$$

that is, precision is the percentage of flagged pairs that are actual plagiarism pairs and recall is the percentage of all plagiarism pairs that are actually flagged.

Furthermore, we define $100 \cdot g/p$ as the *plagiarism content*, i.e., the fraction of all pairs that are plagiarism pairs (see column "% plag pairs" in Table 4.4).

## 4.2 Cutoff criteria

In a normal interactive mode of operation, one will usually look at the most similar pairs of programs found by JPlag and decide for each pair individually whether it is a plagiarism pair or not. One will progress in this manner towards lower similarity values until one is satisfied that all plagiarisms were found.

In some cases, though, it would be better to have a fully automatic decision as to whether something should (preliminarily?) be considered a plagiarism. To do that, one needs a criterion that computes a similarity threshold value: pairs with this or higher similarity will be considered plagiarisms, while pairs with lower similarity will be considered independent. We call such a criterion a cutoff criterion. A cutoff criterion receives as input a vector $s$ of similarity values and it computes a cutoff threshold $T$ as described above.

For evaluating JPlag in such a fully automatic mode, we have used a number of different such cutoff criteria. Some of them are fixed, but most are adaptive to the similarity distribution of the program set under investigation.

**thresh.** The threshT family of cutoff criteria uses the simplest possible method: it does not look at $s$ at all, but rather applies a fixed cutoff threshold to make the decision. We have used various thresholds $T$ from 30 to 95 percent, resulting in the criteria thresh30, thresh40, thresh50, thresh60, thresh70, thresh80, thresh90, and thresh95.

**mplus.** The mplusD family of cutoff criteria is somewhat adaptive towards systematically higher or lower similarity values in $s$. It returns the median (50% quantile, $q_{50}$) of the similarity values in the vector plus $D$ percent of the distance from the median to 100: $T = q_{50}(s) + D/100 * (100 - q_{50}(s))$. In contrast to fixed thresholds, these criteria can somewhat adapt to different "base similarities"; they assume that the median similarity represents a typical non-plagiarism pair, because much less than half of all pairs will be plagiarism pairs. We have used mplus25, mplus50, and mplus75.

**qplus.** The qplusD family is equivalent to the mplusD family, except that the starting point of the offset is the third quartile ($q_{75}(s)$), rather than the median: $T = q_{75}(s) + D/100 * (100 - q_{75}(s))$. The idea is that $q_{75}$ may represent a larger case of accidental similarity, so that even small values of $D$ should not result in false positives. We have used qplus25, qplus50, and qplus75.

**kmeans.** The kmeans cutoff criterion uses one-dimensional k-means clustering to partition the vector into two classes. The class with the higher similarity values will be considered the plagiarism pairs.

**avginf.** The avginfP family of cutoff criteria considers the information content of pairs with about the same similarity value. The idea here is that plagiarisms should be rare and hence the range of similarity values that indicate plagiarisms must have high information content (in the information-theoretical sense of the word). Therefore, we select the threshold $T$ as the minimum threshold for which the average information content $\overline{C_{v \geq T}}$ of pairs with this or higher similarity is at least $P$ percent above the overall average $\overline{C}$. To do this, the avginf criteria group similarity values into overlapping classes of width 5 percent: Given the vector $s$ of similarity values for all pairs, let $S_v$ be the set of similarity values from $s$ that have values $v \ldots v + 5$. Then the information content of each such pair is $C_v := -\log_2(|S_v|/|s|)$ and empty classes are defined to have no information content, i.e., $C_v := 0$ if $S_v = \emptyset$. Based on these values $C_v$, the threshold can be determined. We have used avginf050, avginf100, avginf200, and avginf400.

## 4.3 Distribution of similarity values; precision/recall tradeoff

Now we will review the distributions of similarity values among those pairs of a program set that are plagiarism pairs in comparison to those pairs that are not, i.e., that consist of independent programs. For perfect discrimination we need that the lowest similarity value among the plagiarism pairs is higher than the highest among the non-plagiarism pairs.

### 4.3.1 i12: The "maximize flow" program

Unless something else is stated explicitly, all subsequent examples use the default token set and a minimum match length of 9 Our first example is the program set i12orig. The results are shown in Figure 4.5. Again, the box and whiskers indicate the 10%, 25%, 75%, and 90% quantiles etc. (see caption of Figure 4.1). The curved line is a kernel estimation of the probability density function. This program set leads to a total of 378 program pairs, only 2 of which are actual plagiarism pairs. The top part of the figure shows the similarity value distribution of the 376 non-plagiarism pairs, the bottom part of the 2 plagiarism pairs.

As we see, JPlag will perfectly discriminate the plagiarisms from the other programs for a fairly wide range of cutoff thresholds. The left part of Figure 4.6 shows how recall changes when we gradually increase the cutoff threshold from 0 to 100 percent similarity: only for rather high cutoff thresholds will we miss any plagiarisms.

The resulting tradeoff between precision and recall is shown in the right part of the figure. In this case, it is trivial, as we already know from the discussion of the similarity value distributions above: we will always have at least either perfect precision or perfect recall and for appropriately chosen cutoff thresholds we even get both at the same time, i.e., the curve reaches the top right corner (100/100) of the plot.

Summing up, we can say that JPlag's behavior is perfect for this program set.

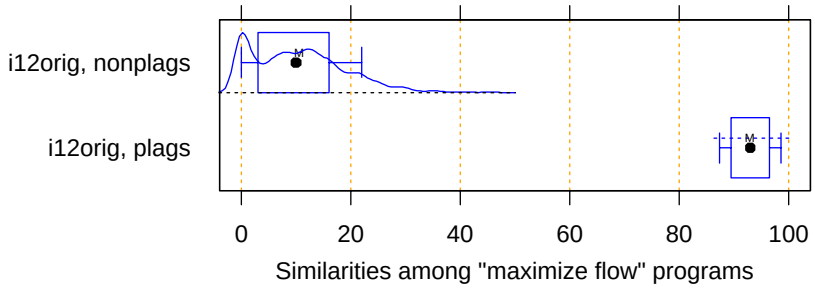



Figure 4.5: Distribution of similarity values found among plagiarism pairs and among non-plagiarism pairs of the i12orig program set. JPlag will achieve perfect separation of plagiarisms and non-plagiarisms with any cutoff threshold between 47 and 85, i.e. both precision and recall are 100 in this case.

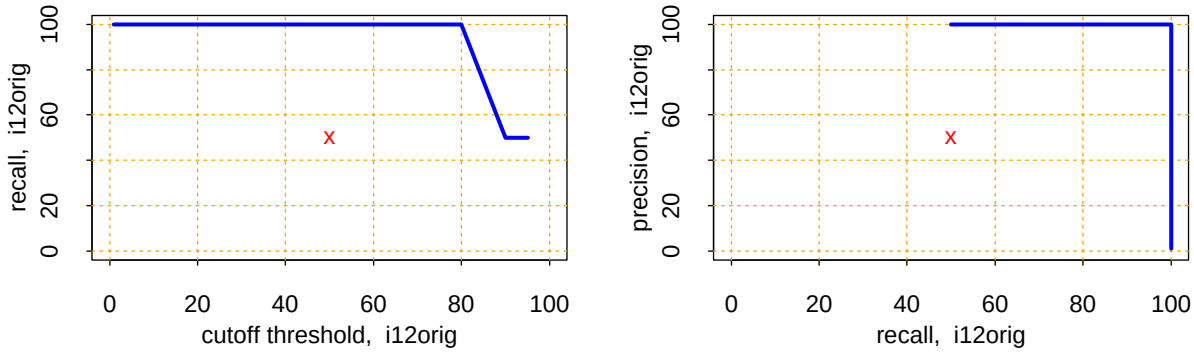



Figure 4.6: Left: recall/threshold tradeoff for i12orig. This dataset can stand very high cutoff thresholds before recall begins to decline. Right: precision/recall tradeoff for i12orig. The behavior is ideal.

### 4.3.2 i27: The "multiply permutations" program

Remember that the i27orig programs are all very short and that the simplicity of the algorithm suggests a somewhat canonical program structure. We can therefore expect that this program will be an extremely difficult test for JPlag.

Figure 4.7 indeed indicates that the behavior for the i27 program sets is less ideal: There is quite a bit of overlap of the similarity value distributions. Let us consider i27orig first and note that in absolute numbers, the plagiarism pair distribution is almost negligible because the plagiarism content is only 0.3%.

Despite the difficulty, the distribution of similarity values for the non-plagiarisms is almost the same as for i12orig.

The plagiarism pairs, on the other hand, with one exception show only moderate similarity in this program set, even for a human observer. Looking at the 12 source programs, we got the impression that the students worked at most partially together, but in any case probably finished their programs independently. However, given the small size of the programs, it is impossible to be sure. So one could just as well say these are not plagiarisms at all.

But to get a sort of worst case analysis, let us assume, these 6 pairs are indeed all real plagiarisms. Then

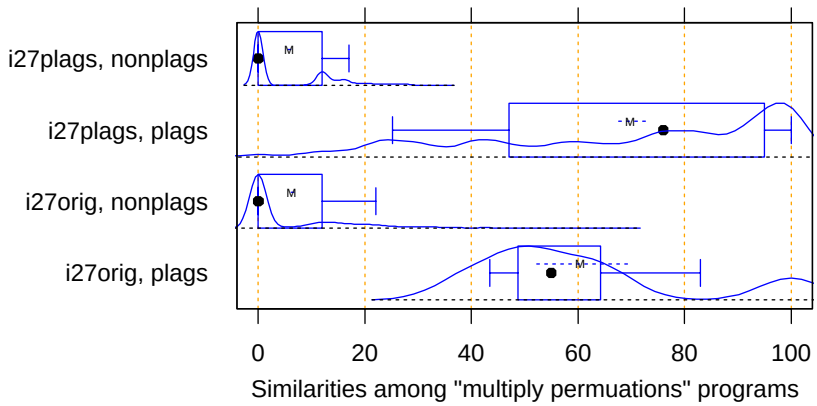



Figure 4.7: Distribution of similarity values found among plagiarism pairs and among non-plagiarism pairs of "multiply permutations" programs, both for the i27plags (top) and i27orig (bottom). For the original programs, there is only little overlap of the similarity ranges. The best tradeoffs for the cutoff threshold are in the range 39 to 68 percent. At 39 percent cutoff, all 6 plagiarisms will be detected, but 28 out of the 1764 non-plagiarism pairs (1.6%) will falsely be considered plagiarism pairs (that is, $P = 18$, $R = 100$). At 68 percent cutoff, no false positives will be found, but 5 of the 6 plagiarism pairs (83%) will be missed as well (that is, $P = 100$, $R = 17$). Other cutoff thresholds result in a behavior that is in between, e.g. with a cutoff threshold of 50 percent we have $P = 36$, $R = 66$.

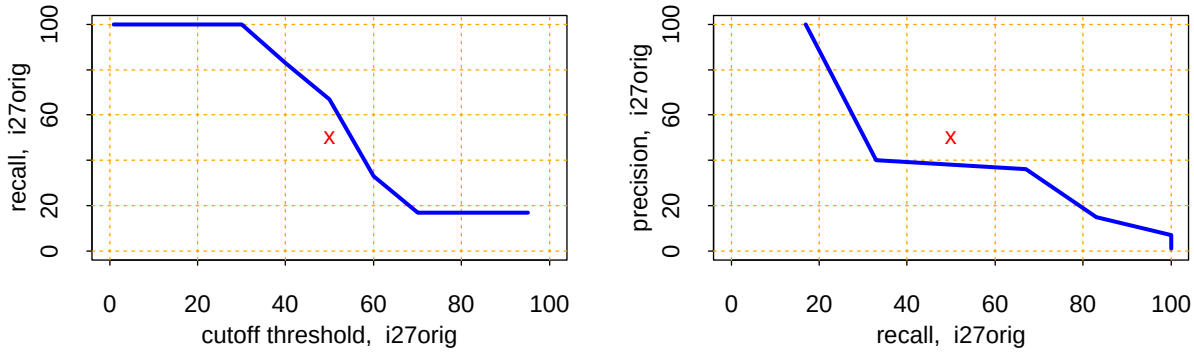



Figure 4.8: Left: recall/threshold tradeoff for i27orig. Recall rapidly declines with large thresholds, but at threshold 50 we get a reasonable $R = 66$. Right: precision/recall tradeoff for i27orig. With high recall values, only somewhat unsatisfying precision can be achieved.

the precision/recall tradeoff looks far from ideal (see Figure 4.8), but medium cutoff thresholds still lead to a reasonable compromise with a recall of for instance 66.

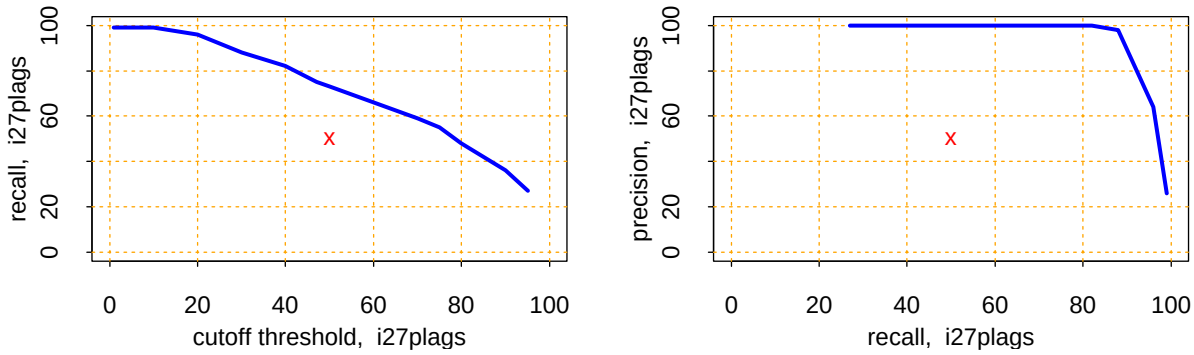



Figure 4.9: Left: recall/threshold tradeoff for i27plags. Recall declines only slowly and steadily for larger cutoff thresholds. Right: precision/recall tradeoff for i27plags. $R > 80$ can be realized with near-perfect precision; a rather good behavior.

For the plagiarisms-only program set, the plagiarism pair similarity distribution becomes even wider, as we see in Figure 4.7, but its median is at a promising 76% similarity. And indeed, the recall curve and the precision/recall tradeoff show a rather benign and satisfying behavior (Figure 4.9): if the threshold is chosen too low, precision drops sharply, but a recall of 70 to 80 can easily be realized with perfect precision.

Summing up, JPlag shows good performance (possibly even very good or perfect performance, we don't know) even for this extremely difficult benchmark.

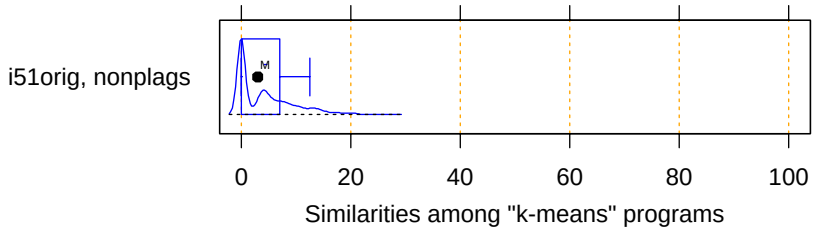



Figure 4.10: Distribution of similarity values found among all pairs of "k-means" programs. This program set does not contain any plagiarisms at all and all program pairs have rather low similarity values. The maximum is 27.

### 4.3.3 i51: The “k-means” program

In this benchmark, there is nothing to find, since the program set does not contain any plagiarisms. Hence, the only goal is to produce all low similarity values in order to avoid false positives. As we see, the similarity values found by JPlag are indeed all so small that we will obtain perfect performance with almost any cutoff criterion.

### 4.3.4 j5: The “Jumpbox” program

As can be seen in Figure 4.11, the separation is very good for the large Jumpbox benchmark as well. For the original program set, perfect performance can be achieved (Figure 4.12). For the collected plagiarisms, only a single well-camouflaged program spoils the otherwise perfect result (Figure 4.13). The deterioration behavior of the recall curve is fairly good in either case.

## 4.4 Influence of token set and match length

All of the data presented so far used JPlag’s default parameters: The default token set and the standard minimum match length of 9. However, we are also interested how robust JPlag is against changes in these parameters, and how such changes interact with the program set to be analyzed. Therefore, we will now introduce a single performance measure and then investigate how performance changes for different minimum match lengths, cutoff thresholds, token sets, and program sets.

### 4.4.1 Performance measure: $P + 3R$

We measure the total plagiarism discrimination performance of JPlag by a weighted sum of precision and recall. We choose a relative weight of 3 for recall (versus precision) since it makes sense to penalize false negatives (non-detected plagiarisms) far more than false positives, which merely introduce more work for the final judgement by the human user. Hence, the performance measure becomes $P + 3R$. The value 3 is not important, the results with a weight of 2 or 4 would be similar.

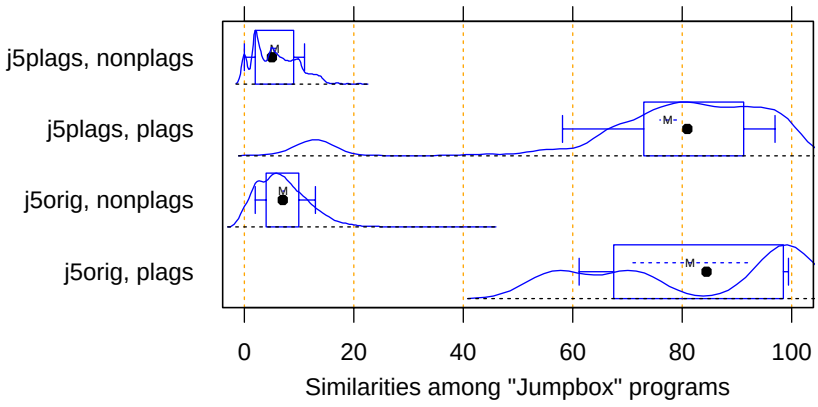



Figure 4.11: Distribution of similarity values found among plagiarism pairs and among non-plagiarism pairs of “Jumpbox” programs, both for the original set of programs (j5orig, bottom half of figure) and the plagiarisms-only set (j5plags, top half of figure). For the original programs, JPlag will achieve perfect separation of plagiarisms and non-plagiarisms with any cutoff threshold between 44 and 56. For the plagiarized programs, there is a single program that is very dissimilar to all others in its plagiarism group, resulting in 14 similarity values in the range 8 to 16, but all others can be separated perfectly by cutoff thresholds between 22 and 43.

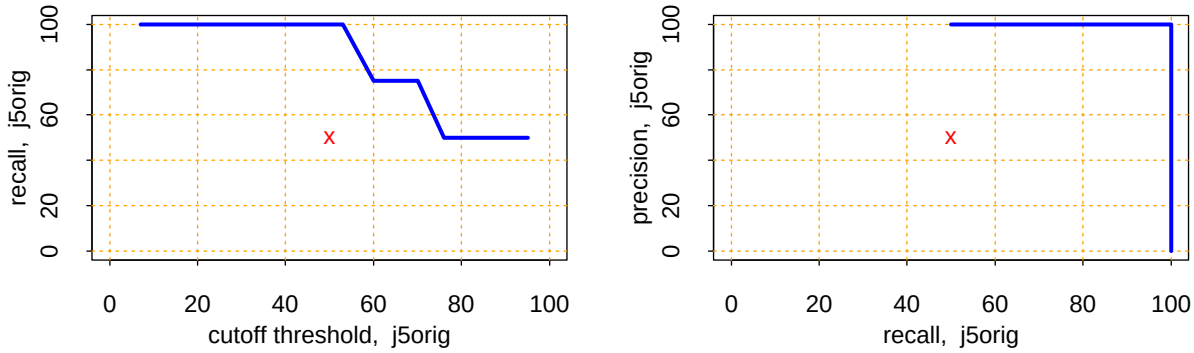



Figure 4.12: Left: recall/threshold tradeoff for j5orig. Right: precision/recall tradeoff for j5orig.

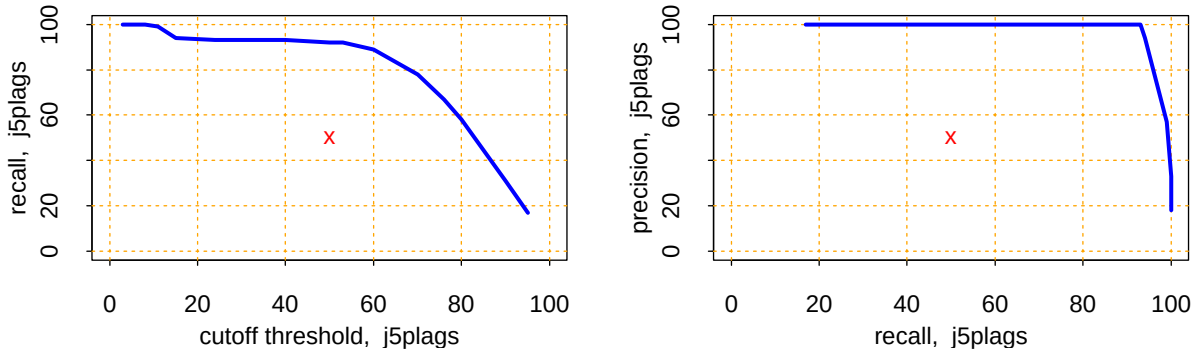



Figure 4.13: Left: recall/threshold tradeoff for j5plags. Right: precision/recall tradeoff for j5plags.

### 4.4.2 Original program sets

Let us look at the default token set first. These results are shown in Figure 4.14 below. We can make the following observations:

1. The best match length may depend on the cutoff threshold chosen. This is not surprising, because shorter match lengths result in generally higher similarity values.
2. As a result, the general performance trend for increasing match lengths may be upwards, downwards, or hill-shaped. (Downwards trends tend to be most common, because our performance measure emphasizes recall and small match lengths produce higher similarity and hence higher recall.)
3. Therefore, any fixed minimum match length must be considered a compromise.
4. However, unless the value chosen is very far away from the optimal one, the loss of performance is only small. JPlag is robust against modestly non-optimal choice of minimum match length.

For the small “struc” token set, we find the following (see Figure 4.15 on page 26):

1. Due to the shorter token streams, larger minimum match lengths are less advisable.
2. Otherwise, the results are remarkably similar to that for the default token set.

Finally, the largest possible token set, “full”, shows the following behavior (see Figure 4.16 on page 27):

1. Due to the longer token streams, larger minimum match lengths can more often be tolerated, but modest lengths still tend to be superior.
2. Otherwise, the results are again quite similar to both the default and the reduced token set.

We conclude that JPlag is highly robust against different choices of token set. This is good news, because it suggests that JPlag may work similarly well for many other programming languages, too.

### 4.4.3 Artificial program sets

Now let us fix the cutoff threshold at 50 and look at the behavior for the artificial program sets with their higher plagiarism content. These results are shown in Figure 4.17 on page 28. We conclude the following:

1. For these program sets with higher plagiarism content, high minimum match lengths produce consistently bad performance (due to low recall).

2. However, in many cases, very small minimum match lengths hurt performance as well (due to low precision).

3. Again, the dependence of performance on the token set is modest.

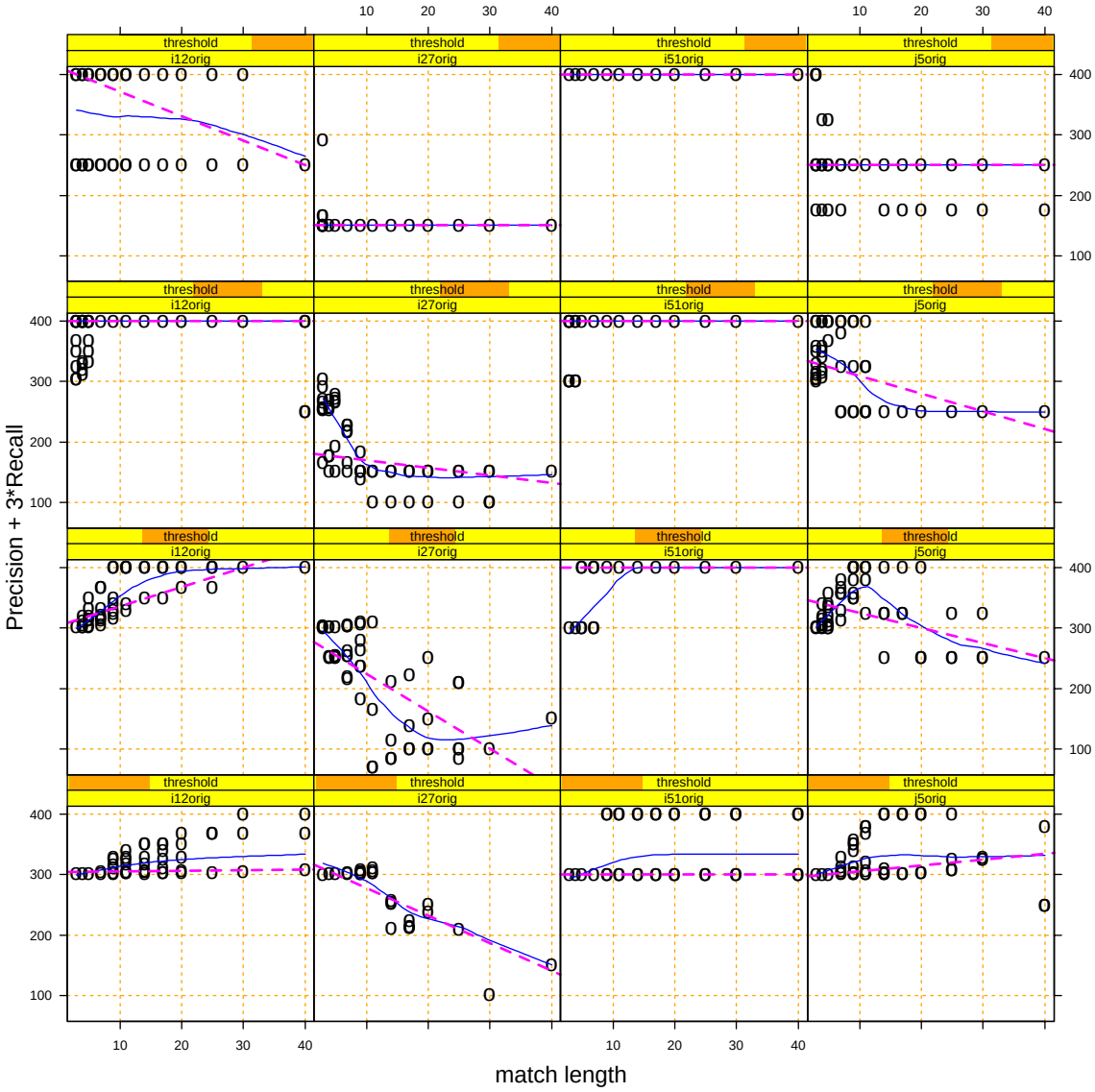



Figure 4.14: Influence of minimum match length and cutoff threshold on P+3R measure for default token set ("normal") on the various original program sets. Each point represents one JPlag run. Each subplot groups the runs for one program set with different minimum match lengths, but similar cutoff thresholds. The straight line is a robust (least absolute distance) regression line. The curve is a smoother line (Loess local linear regression with span 0.666). The subplots are arranged left-to-right by different program sets and top-to-bottom by decreasing overlapping cutoff threshold ranges (75 to 100; 51 to 79; 30 to 57; 0 to 33).

4. The most robust values for minimum match length appear to be in the range 7 to 11.

We conclude that the default token set and the default minimum match length of 9 are reasonable and robust parameter choices for JPlag for a wide variety of program sets.

## 4.5 Influence of cutoff criteria

Now let us have a look at the cutoff criteria. From the above discussion of similarity distributions we already know, that cutoff thresholds in the range 30 to 60 will usually yield the best results.

However, it is not clear whether any fixed threshold exists that will almost always be optimal. An adaptive criterion that takes the current similarity distribution into account might be more successful.

Let us first look at the variability of the cutoff threshold that the different criteria (as introduced in Section 4.2) produce when applied to our program sets (as introduced in Sections 4.1.1 and 4.1.2). This is shown in Figure 4.18. One might argue that a good cutoff criterion should be able to adapt to different optimal cutoff

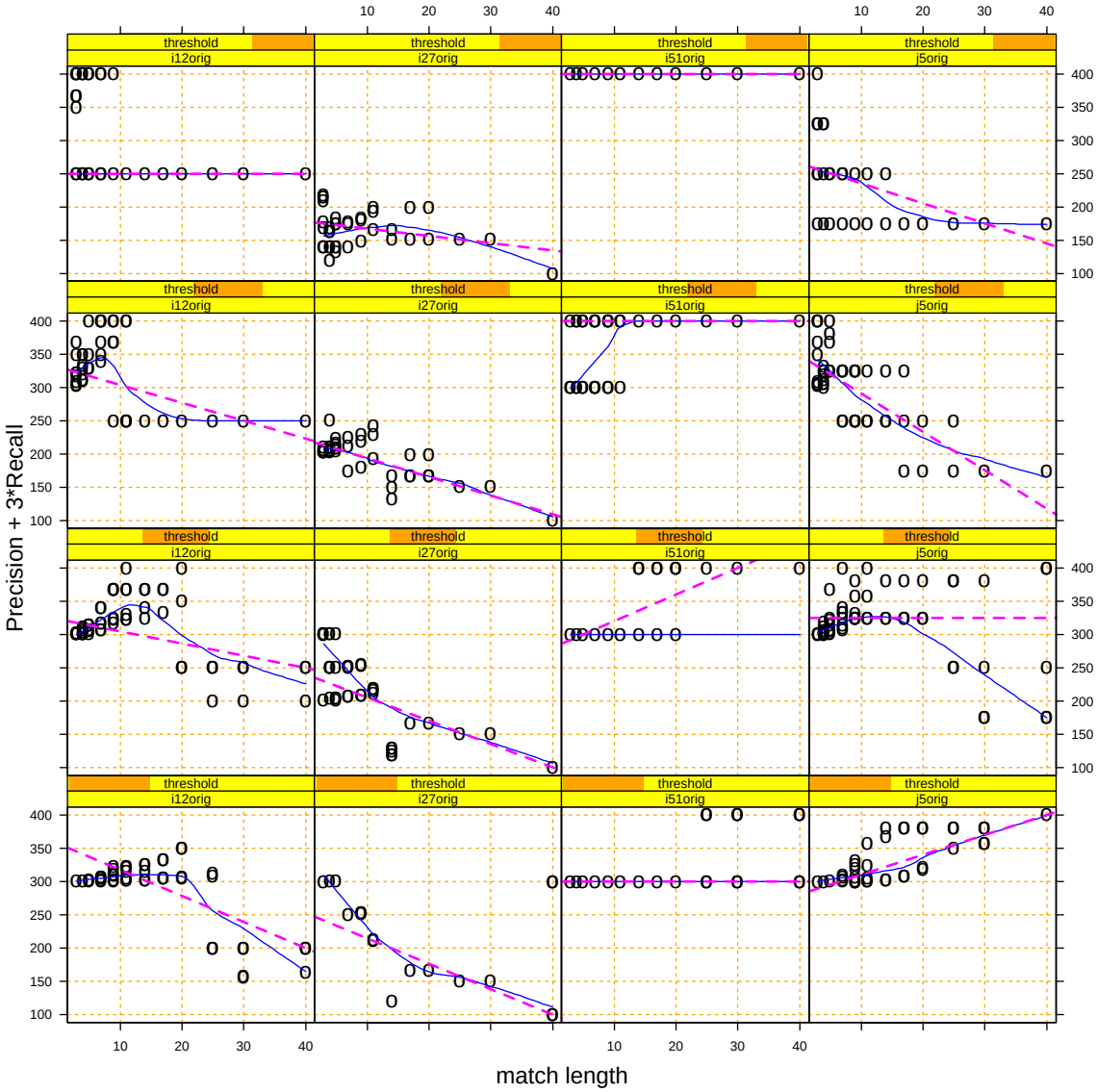



Figure 4.15: Influence of minimum match length and cutoff threshold on P+3R measure for minimal token set ("struc") on the various original program sets. For explanation see Figure 4.14 above.

thresholds, depending on the given program set, and should hence produce some amount of variation when applied to different program sets. For now, this is only an assumption, we will investigate the actual performance results further below.

As we see, the variability introduced by the mplus and qplus criteria is only small. kmeans is somewhat more variable for the artificial program sets, but produces rather low cutoffs for the original sets — presumably because the fraction of plagiarism pairs is smaller there than what k-means is likely to return as a cluster of its own. The largest threshold variation is introduced by the avginf criteria. However, the avginfo400, which looks most promising on the original program sets (right), produces mostly exaggerated thresholds of 100 for the artificial program sets (left).

Figure 4.19 compares the performance distribution of the cutoff criteria, based on the performance measure $P + 3R$ introduced in Section 4.4.1.

For the original program sets, the best criteria are thresh50, thresh60, qplus50, and mplus50. They all have a median of 400 (i.e. at least half of all results are perfect), a mean of about 330 and a first quartile around 280.

For the artificial program sets, no criterion manages to be as good as that. The best ones are thresh40, qplus25,

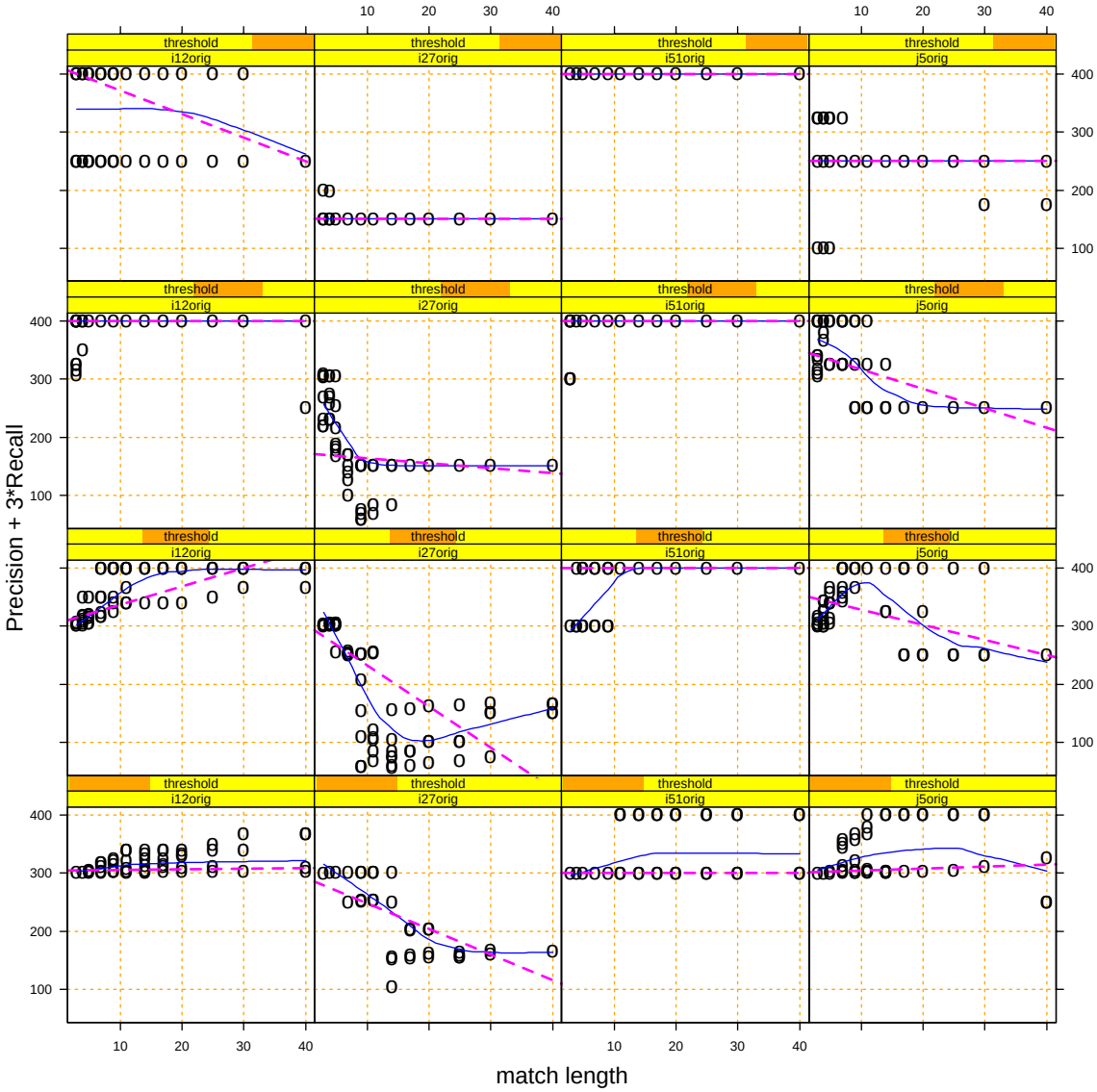



Figure 4.16: Influence of minimum match length and cutoff threshold on P+3R measure for maximum token set (“full”) on the various original program sets. For explanation see Figure 4.14 on page 25.

and mplus25.

This is good news: for practical purposes (where the amount of plagiarisms is usually small), a rule as simple as thresh50 appears to yield about the best possible results.

## 4.6 Runtime efficiency

The runtime of JPlag increases quadratically with the number of programs in the program set, and slightly superlinearly with the size of the programs.

However, the resulting runtimes are usually small. The largest of our program sets, j5all, contains 99 programs averaging about 250 lines. The total run time of JPlag for reading and parsing these programs and performing all of the pairwise comparisons, is under 12 seconds wall clock time on our Sun Ultra II (300 MHz) workstation using JDK 1.2.1 on Solaris 7 (SunOS 5.7).

For i51orig (32 programs, 118 lines average), the equivalent wall clock time is 3 seconds.

## 4.7 Successful and non-successful plagiarizing attacks

This section analyzes the disguising techniques and the types of attacks we have seen in the programs used for this study (i12orig, i27all, i51orig, j5all). In addition to a discussion of those disguising techniques that are futile because of JPlag's token-based approach, we focus on those attacks that result in a local confusion

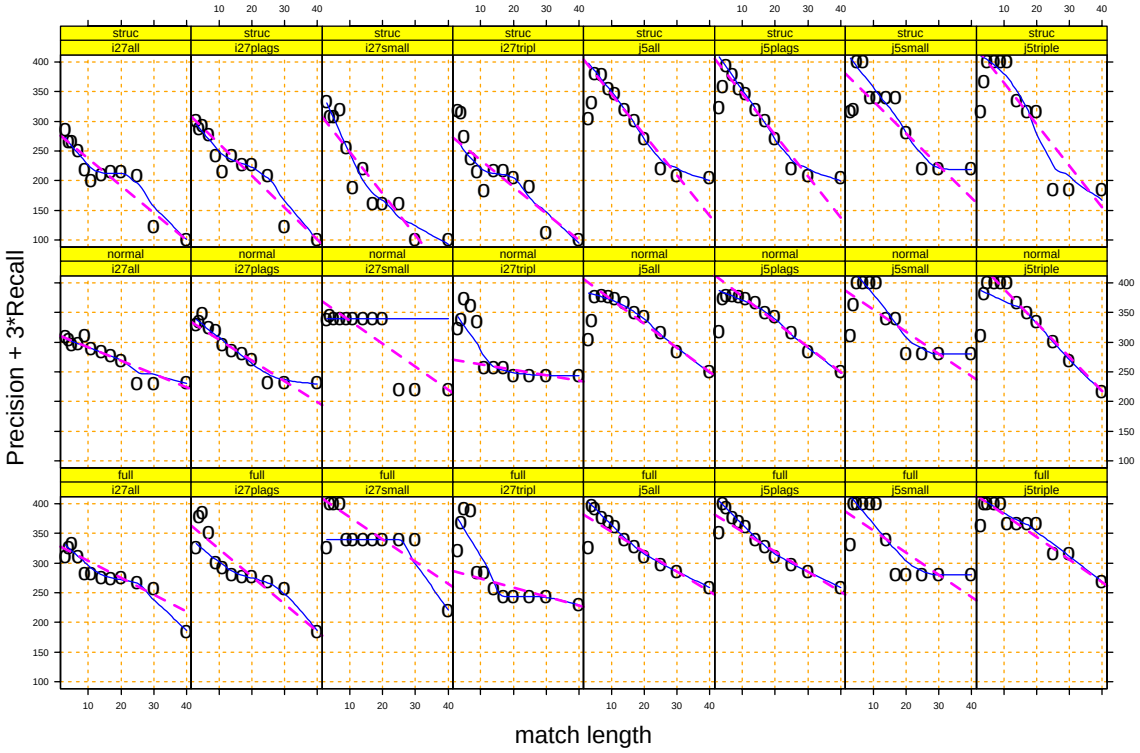



Figure 4.17: Influence of minimum match length and token set on P+3R measure for the various artificial program sets at fixed cutoff threshold 50. Each point represents one JPlag run. Each subplot groups the runs for one program set with different minimum match lengths. The straight line is a robust (least absolute distance) regression line. The curve is a smoother line (Loess local linear regression with span 0.666). The subplots are arranged left-to-right by different program sets and top-to-bottom by increasingly larger token sets.

of JPlag. For the definition of local confusion we consider the shortest segment of the original code that is necessary to apply a disguising technique. If JPlag does not determine any similarity between the original code segment and the result of the attack, we say that JPlag is locally confused. Local confusion critically depends on both the token set used and the minimal match length.

For example, a local confusion can be caused if a single line of code (or a single token) is inserted into a code segment of minimal match length. After the insertion, JPlag will (in general) no longer find code segments of enough tokens to match. For another example, again consider the code segment of minimal match length. If this segment is split into two parts that are then swapped, JPlag will not signal any similarity (unless of course both parts are represented by identical token lists).

A local confusion of JPlag is a necessary condition of a successful attack. However, only for very small programs (at most twice the minimal match length) a single local confusion is sufficient for a successful attack. For longer programs, the number of local confusions needed for a successful attack depends on the cutoff criterion used. A perfect attack that is to be successful with any cutoff criterion will need to achieve local confusion in every single segment of the original code with minimal match length. Thus, JPlag's success in finding plagiarisms critically depends on the plagiarists' lazyness: it is a lot of work to apply *many* of the individual disguising techniques *frequently* all over a given program. Few plagiarists were creative enough to find enough disguising techniques that could be applied all over the given program (most techniques are only applicable in certain situations). Even if plagiarists found enough techniques, few were eager enough to apply

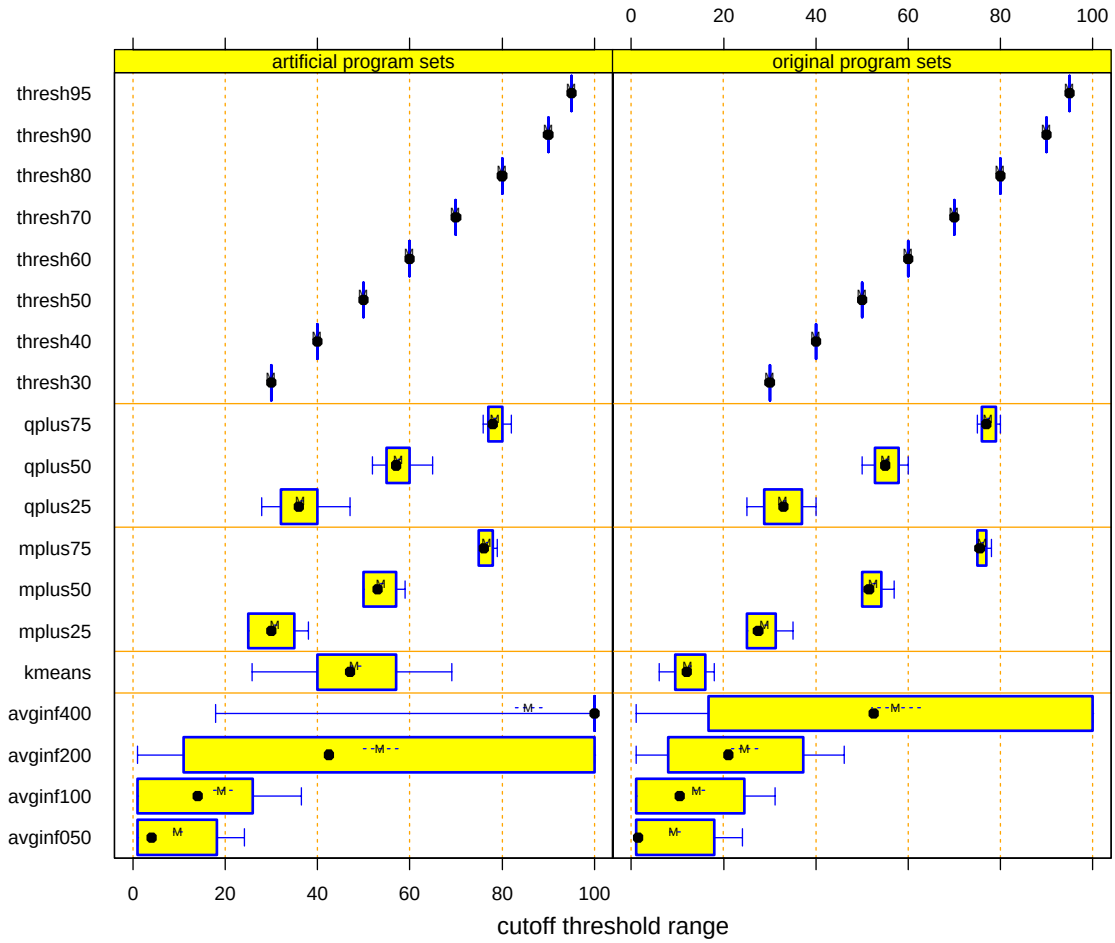



Figure 4.18: Variability of the cutoff thresholds chosen by the various cutoff criteria for the artificial program sets (i27plags, i27all, i27small, i27tripl, j5plags, j5all, j5small, j5triple; left hand part) or the original program sets (i12orig, i27orig, i51orig, j5orig; right hand part), using minimum match lengths of 7, 9, or 11, and any of the token sets.

them sufficiently often.

### 4.7.1 Futile attacks

The attacks discussed in this section will not cause any modification in the list of tokens that are generated and considered by JPlag. Hence, the attacks are futile – they do not work at all.

It is interesting to note, that almost every plagiarist has used at least some of these futile disguising techniques. In total we have seen 174 applications of futile attacks. A possible explanation is that the plagiarists expected JPlag to apply only very simplistic text-based types of analysis. But even those plagiarists that have applied very sophisticated disguising techniques (see subsequent sections) because they probably suspected a more elaborate analysis still have used some of the techniques mentioned here. A possible explanation is that those plagiarists probably hoped to calm down a suspecting human person that might look carefully into the code if JPlag would signal a potential attack.

On the right hand side of the following headlines we show two numbers. The first of which gives the total number of programs that have used this type of attack (at least once). After the slash, the second number gives the number of programs where this attack resulted in at least one local confusion of JPlag.

- **Modification of code formatting** **48/0**

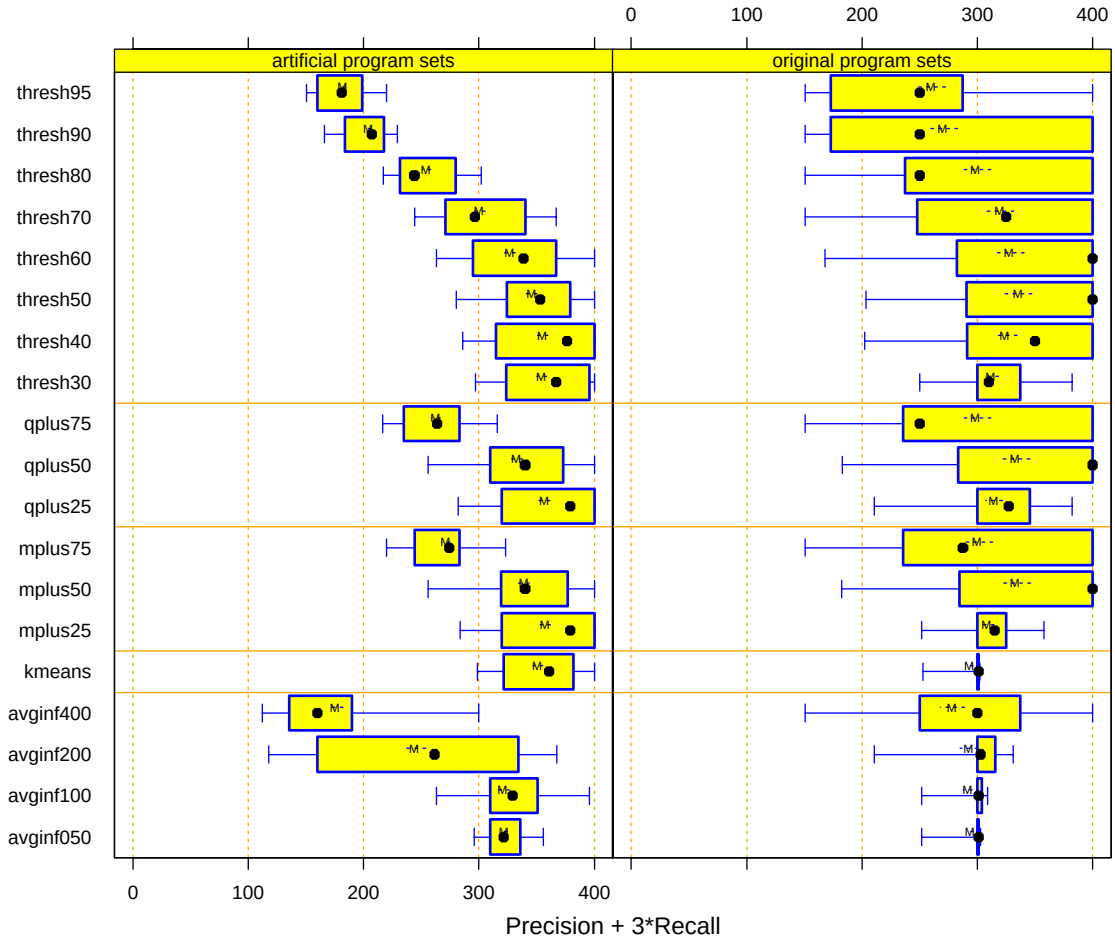



Figure 4.19: Performance of various cutoff criteria for the artificial program sets (left) or the original program sets (right), using minimum match lengths of 7, 9, or 11, and any of the token sets. The quality criterion is the sum of precision and recall, with recall being weighted 3-fold.

In this category of attacks, there are changes of the indentation depth, modified positions of { and }, insertion or deletion of empty lines, insertion or deletion of extra line breaks, etc.

We have seen modified code formatting in 48 programs, none of which has confused JPlag locally, since JPlag's parser removes all those formatting differences.

- **Insertion, modification, or deletion of comments** **30/0**

  Many plagiarists thought that it would be a great idea to modify comments.

  We have seen this attack in every imaginable way in a total of 30 programs, none of which has confused JPlag locally, since JPlag's parser removes all comments while parsing.

- **Translation from English to German or vice versa** **19/0**

  The programs used in this study interacted with their users either in English or in German. Several plagiarists chose to alter the language used for interaction with the users and translated the text of the I/O-statements.

  We have seen this attack in 19 programs, none of which has confused JPlag locally. The reason for JPlag's resistance is that this attack only affects string constants which are mapped to a single token, independent of their concrete values. Hence, after parsing, I/O messages in different languages are no longer visible.

- **Modification of program output or of its formatting** **14/2**

  All original programming problems had a clear specification of what is considered to be a correct result. The concrete formatting of result output with respect to indentation, spaces, number format etc. was left open for the programmer. Hence, it was a natural target for attacks. Plagiarists added additional spaces, added colons, changed the number of decimal digits, added the output of structuring characters, etc.

  In 12 programs, these modification only affected string constants, which is irrelevant for JPlag (see above). Only in 2 programs, the modified output format resulted in extra string concatenation or other method invocation which affected the list of tokens generated and considered by JPlag. With the default token set, JPlag has been locally confused by this type of attack twice.

- **Change names of variables, methods, or classes** **44/0**

  It has been very popular to change names of variables and methods, which is easy to do by means of the editor's global find and replace feature.

  Plagiarists appended letters to existing names, they modified the spelling, they changed letters of names from upper case to lower case or vice versa, they translated names from German to English or vice versa, they prepended "do" to method names, etc.

  We have seen name changes in 44 programs, none of which has confused JPlag locally, since JPlag's parser removes all those names.

- **Split/merge of variable declarations** **6/0**

  Java offers a special construct to declare several variables of the same type in a single declaration. For example, all temporary `int` variables could be declared (and initialized) in a single declaration.

  In 6 programs we have seen an attack where these combined declarations have been split into separate declarations (or alternatively, some individual declarations have been merged into a combined declaration).

  Since JPlag's parser represents a combined declaration as a list of individual declarations, JPlag's parser implements resistance against this type of attack.

- **Insertion, modification, or deletion of modifiers** **6/0**

  In Java, variables and methods can be attributed with additional modifiers that affect accessibility, e.g. `private`, `protected`, or `public`. It does not affect the semantics of an existing program when modifiers are changed to allow a more general accessibility. In some of the original programs it was even possible to restrict accessibility without breaking the program; plagiarists could easily check whether such modifications were legal by recompiling the modified code.

  Another modifier, `final`, can be removed without affecting the semantics of an existing program because `final` will only affect performance and reuse in later-to-be-added sub-classes.

  Since JPlag's default token set ignores modifiers completely, this attack which we have seen in 6 programs, did not have any affect at all.

- **Modification of constant values** **3/0**

  For some of the applications, slight modifications of some constants did not really matter, e.g., the allowed number of microseconds, the absolute size of a GUI-component in pixels etc. In three programs, plagiarists tried to attack JPlag by modifying such constants. But since JPlag ignores constant values, JPlag did not get confused by this approach.

- **No change at all** **4/0**

  The most trivial attack was to submit an unchanged program. We have seen that 4 times. Obviously, JPlag is resistant against this (primitive) attack.

### 4.7.2 Granularity-sensitive attacks

Roughly spoken, Java classes consist of declarations of methods and variables. The order of declaration is irrelevant, i.e., a variable can be used textually before it is declared. Similarly, methods can be called textually before their declaration.

Therefore, a promising line of attack is to reorder the declarations in the implementation. If the reordered code segments are longer than the minimal match length, the reordering will not result in a local confusion. JPlag will signal block moves instead of being locally confused. Thus, the success of such attacks critically depends on the granularity of their application.

It is interesting to note that this type of reordering has been used quite frequently by the plagiarists of this study (55 times). However, only about 15% of the plagiarists confused JPlag locally once.

- **Reordering within blocks of variable declarations** **25/6**

  With the standard token set, every declaration is mapped to a single token, completely ignoring the type of the variable. Hence, on first sight JPlag should be resistant against any form of reordering within blocks of variable declarations. Unfortunately, it is not, because initializations that can accompany variable declarations are represented by tokens as well. For example, the original programs have initializers that are constructor calls, constant values, array initializations etc., all of which result in different token lists. Therefore, reordering of variable declarations that have differently structured (or missing) initialization expressions, will result in different token lists.

  In our programs, we have seen this type of attack 25 times, but only in 6 cases JPlag got locally confused since the reordering has been done on the level of one or very few declarations with different initialization expressions.

- **Global reordering of variable and method declarations** **30/3**

  Reordering of class level declarations can only be a successful attack if the reordered items are very small. Large methods are represented by more tokens than the minimal match length. Hence their reordering will be signaled as block moves. Only small methods and individual variable declarations are below the limit of the minimal match length so that their reordering can cause local confusion.

  In our programs, we have seen this type of attack 30 times; only in three cases a list of very small methods (intermixed with declarations of some variables) has been reordered and succeeded in confusing JPlag locally.

### 4.7.3 Locally confusing attacks

In this section we discuss disguising techniques that are most likely to affect the list of tokens generated and considered by JPlag. With typical token sets, JPlag will be locally confused. Only with very coarse token sets, the following types of attacks might leave the list of tokens unchanged.

The techniques have been used 134 times, only in 12 cases JPlag has *not* been confused locally. Hence, these techniques appear quite successful at first glance, however, only very few plagiarists achieved so much local confusion all over a given program as to escape detection. Only one plagiarist succeeded with a single technique (redundancy, see below).

The last three types of attacks in the following list are the really clever ones, which are hard to detect even for a human reader.

- **Modification of control structures** **35/35**

  To retain the intended semantics with modified control structures, new expressions or new statements have to be inserted into (or removed from) the original code. Moreover, the transformation typically moves small segments of the code around, or even into and out of block structures. All these effects modify the token list in such a way that almost no matching blocks will be found by JPlag (except for control structures with large bodies).

  The following types of attacks have been done to loop statements.

  – In 8 successful attacks we have seen a `for`-loop being replaced by a `while`-loop or the other way round. This modification has certain effects: usually, additional temporary variables need to be declared or existing temporary variables are declared in different positions of the code. Moreover, the loop's condition has to be re-evaluated somewhere in the body of the loop where it has not been evaluated in the original loop.
  – In three successful attacks, plagiarists have eliminated auxiliary index variables by expressing them in terms of the main iteration variable.
  – In one case the plagiarist has replaced a regular loop by an endless loop with explicit `break`-statements.

  In principle, JPlag could be made more resistant against these types of attacks. The general idea of such an improvement is to try to normalize token lists for control structures of the language on a level of semantics instead of syntax. For loop constructs, two ideas are obvious. First, the loop structure and its body needs to be analyzed more carefully – the loop condition needs to be identified wherever it is located in the loop. The intention is to generate a standard token representation for all types of loops. Second, JPlag should eliminate redundant index expressions and hoist loop invariant expressions out of the loop.

  The following attacks have been done to `switch`-statements.

- In 6 successful local attacks, a `switch`-statement has been replaced by a sequence of `if`-statements.
- One plagiarist has confused JPlag locally by adding redundant `break`-statements to every case of the `switch`-statement.
- One plagiarist has confused JPlag locally by reordering the cases of the `switch`-statement.
- In four successful local attacks, plagiarists have moved the `default` case out of the `switch`-statement.

Again we can envision normalizations of the generated token sequence that would reduce the amount of disguising that can get by unnoticed.

The following attacks have been done to `if`-statements.

- In two successful attacks, programmers have reordered a cascading `if`-statement.
- Three plagiarists have confused JPlag locally by negating the condition of an `if`-statement and switching the `then`- and `else`-clauses.
- Four plagiarists have confused JPlag locally by replacing a `?`-operator by an explicit `if`-statement or the other way round.
- One clever plagiarist added a redundant `if`-statement – one with an identical `then`- and `else`-clause.
- One plagiarist moved the code that followed an `if ... return` into a newly added `else`-clause.

Again we can envision normalizations of the generated token sequence that would reduce the amount of disguising that can get by unnoticed. The idea is to construct a hierarchical control flow tree that represents the potential flow of control. This tree is independent of the particular elements of the language used to express that control flow. By enforcing an order when generating tokens from such a control tree some of the attacks mentioned above can be identified.

- **Temporary variables and subexpressions** **28/28**

Another line of attack targeted subexpressions. With any but the coarsest token set, JPlag will generate and consider tokens for at least certain subexpressions. Hence, moving around subexpressions may reorder the list of tokens and hence can confuse JPlag locally.

- In 16 programs, plagiarists have moved subexpressions (including constants) into the declaration of additional temporary variables. In the original expressions, the new temporary variables are used instead of the hoisted subexpression. In 7 programs we have seen the same idea the other way round: instead of using explicit temporary variables to store the result of expressions or to store constants, some expressions have been inserted where their results are consumed.
- In two programs, plagiarists have replaced array initializers that initialize all elements of an array at once by a list of explicit assignment statement – one for every single array element. Quite similar, two plagiarists separated the declaration of a variable from its initialization by means of an extra assignment statement. Both types of attack could be prevented by means of a slightly different parsing in JPlag.
- In Java, local variables have a default initialization value. Extra tokens have been inserted into JPlag's list of tokens by additional assignments that set this default value explicitly. We have seen this type of attack in one program.

- **Inlining and refactoring** **20/16**

  In 5 programs, plagiarists have confused JPlag locally by inlining of small methods. Two plagiarists have inlined methods that were longer than the minimal match length so that JPlag signaled block moves.

  The orthogonal approach is to extract functionality and refactor it into new methods. In 11 programs, plagiarists have confused JPlag by this type of attack (3 of which have added additional wrapper methods). Again, two plagiarists have extracted methods that are longer than the minimal match length so that JPlag signaled block moves.

- **Modification of scope** **9/9**

  We have seen three types of attacks that affect the scope and hence the order of tokens generated by JPlag.

  – Three plagiarists have pushed plausibility tests or `try`-statements that catch exceptions towards the outer perimeter of methods instead of dealing with wrong argument values and exceptions as soon as possible.
  – Three plagiarists have extended the scope of temporary variables by declaring them in surrounding blocks. One plagiarist added redundant declarations of temporary variables in inner blocks that shadow temporary variables with the same names declared before.
  – Two plagiarists replaced class variables by instance variables in a singleton situation. This influences the token sequence at the invocation of the constructor. Furthermore, it might enable the "`this`-trick", see below.

- **Statement reordering in absence of data dependences** **8/6**

  Statements form a basic block of code if the flow of control will either execute all of them or none. Although programmers have ordered the statements within a basic block to write them down, it might be possible to reorder those statements if there are no data dependences that require a certain order.

  We have seen 8 programs where plagiarists have reordered statements within basic blocks without changing the semantics of the code. In two cases, this approach did not confuse JPlag since the reordered token lists remained constant.

- **Mathematical identities** **5/2**

  Five plagiarists tried to use mathematical identities to hide their attacks, only two of them have been locally successful. They focused on library methods: one of them removed a block of statements and instead called a library routine that provides the replaced functionality. Another plagiarist exploited the fact that some of Java's libraries offer different ways to achieve the same effect.

  The unsuccessful attacks used pre-increment (`++i`) instead of post-increment (`i++`), replaced increment operators (`i++`) by explicit assignments (`i=i+1`), or modified the boundary checks of loops, e.g. used `<n` instead of `<=n-1`.

- **Introduction of bugs on purpose** **5/3**

  Five plagiarists inserted bugs on purpose. In three locally confusing cases the plagiarists removed code or added additional statements. The other two attempts were made with modified constants that had no effect on the list of tokens generated and considered by JPlag.

- **Modification of data structures** **6/5**

  One plagiarist has used a character array instead of a string. This confused JPlag locally, since in addition to the changed declaration, all accesses to the original string have been changed to work on the character array instead.

One plagiarist has replaced a two-element array of `int` values (`int[2]`) by two individual `int` variables. In this case even method signatures have been changed: instead of the array, two primitive type values have been passed into the methods. Three plagiarists have used a similar idea, but the other way round. Their idea was to replace a group of temporary variables of the same type by an array of that type. In addition, all occurrences of the temporary variables have been replaced by array expressions. One of these plagiarists could afterwards even replace a list of statements that worked on each of the original temporaries by a loop working on the array.

One unsuccessful attempt promoted an `int` variable to `long` – this does not have any effect on the list of tokens generated and considered by JPlag.

- **Redundancy** **15/15**

  Six plagiarists have confused JPlag locally by adding methods that are never called or variables that are never used. Analogously, some dead code has been removed from given programs.

  Two plagiarists have confused JPlag locally by working on the package names. Either a class has been moved into another package, or class names have been used with fully qualified complete package names.

  The remaining redundancy attacks we have seen only once.

  – Almost perfect confusion can be achieved by inserting invocations of dummy methods all over a given program. Every method invocation inserts a token and hence breaks token sequences that would match without the extra token. Of course this attempt is instantly recognized by a human inspector.
  – It confuses JPlag locally if additional packages and classes are imported into the name space.
  – Quite often `Thread.yield()` can be inserted without doing any harm.
  – At the end of `void` methods, additional `return` statements can be inserted without any effect on the semantics.
  – One plagiarist used another clever attack: he simply duplicated (or repeated) assignment statements if their right hand side did not have side effects. Obviously, it does not change the semantics if the same value is assigned to a variable several times. However, it does modify the list of tokens generated and considered by JPlag.
  – A final clever attack in this category is the "`this`"-trick. Inside of a class implementation, an object can access its instance variables simply by their names. Alternatively, the same variable can be accessed in a fully qualified way, i.e., with a prepended `this.` Since this causes modified token lists, JPlag can be confused locally. As a counter measure, JPlag's default token set ignores `this.`

- **Structural redesign of code** **3/3**

  In this category, we have seen two different types of attacks that both have proven quite successful.

  Consider a method that works on the state of its object. If this method is moved to a newly introduced helper class, the method's signature has to be changed because the affected object now needs to be passed to the new method. Moreover, any access to instance variables of the affected object has to be changed. Either the instance variable of the affected object can be accessed with a fully qualified name or (even worse from the perspective of JPlag), additional set- and get-methods can be used for this purpose. Because of its dramatic effect on the appearance of the code such an attack is confusing both to JPlag and to a human inspector.

  The other type of attack focussed on a class with state changing methods as well. Two plagiarists have replaced the class by an alternative implementation that had value semantics, i.e., instead of changing the internal state stored in the instance variables of an object, a new object was created for every state change and returned to the caller. Of course the signature of all methods changed correspondingly. The

code containing the invocations changed as well because the returned object needs to be handled in the new implementation. Although for the application the modified class was just a helper class, this change affected a lot of the core implementation.

# Chapter 5

# Summary and conclusions

Our empirical evaluation of JPlag using 4 real sets of Java programs and 8 further Java program sets containing additional plagiarisms can be summarized as follows:

- For clearly plagiarized programs, i.e. programs taken completely and then modified to hide the origin, JPlag's results are almost perfect — often even if the programs are less than 100 lines long.
- Even for only partially plagiarized programs, as in our i27 program set, JPlag will pinpoint the similarities and can often discriminate them fairly well from accidental similarities.
- The camouflage approaches (if any) chosen by the plagiarists from the real program sets were utterly useless against JPLag.
- Even the attacks chosen by informed plagiarists were successful in less than 10 percent of all cases. These persons knew they had to fool a program and had no other goal (except using only a modest amount of time to do it).
- Successful attacks against detection by JPlag require a lot of work and will produce a program structure that looks ridiculous to any human inspector.
- JPlag is quite robust against non-optimal choice of its two free parameters, token set and minimum match length.
- Given the similarity values computed by JPlag, a fixed cutoff threshold is sufficient as a discrimination criterion that separates plagiarized program pairs from non-plagiarized ones with near-optimal recall and nevertheless good precision.

We do not know to what degree these results transfer to other situations. They might apply to a lesser degree for C and C++, because these languages are currently not parsed but just scanned by JPlag. They might also be weaker for differently structured (or much larger) programs if those make some of the attacks more effective or for different plagiators if those use still different attacks.

However, on the whole we would be very surprised if the effectiveness of JPlag was ever much lower than demonstrated in this study. It appears that the token-based string-similarity approach is a highly effective one for finding plagiarisms, at least if the token strings ignore much detail. JPlag is an easy-to-use implementation of this approach for programs written in Java, C, C++, and Scheme.

# Appendix A

# Individual results for submitted plagiarisms

When we collected the additional plagiarisms, we promised to each "author" that we would publish his or her individual result and sent a submission identification number by email. This section provides all the individual results. Program identifiers starting with the digit 1 were created by the external volunteers, all others are from the respective 'orig' program set. The results are arranged into a ranking of all submissions within each program set.

The j5 plagiarisms were collected between January 1998 and June 1999, the i27 plagiarisms were collected between October 1998 and April 1999. (Yes, we were not *really* fast in preparing this report...)

## A.1 j5: The “Jumpbox” program

`rnk` is the rank of the plagiarism and `sim` is the similarity to the corresponding original, in percent. Lower similarity values are “better” plagiarisms and hence have a higher rank. Pairs of original plagiarisms are not counted in the ranking. `orig` and `plag` are the identifiers of the original and plagiarized program version, respectively.

Only a single plagiarism pair escapes detection if a 50% cutoff threshold is used.

| rnk | sim | orig | plag |
|---|---|---|---|
| 1 | 13 | 827052 | 132226 |
| - | 57 | 862531 | 878135 |
| 2 | 68 | 826606 | 132213 |
| 3 | 69 | 827052 | 132224 |
| - | 71 | 942261 | 943151 |
| 4 | 74 | 827052 | 132229 |
| 5 | 76 | 786143 | 132230 |
| 6 | 78 | 792145 | 132232 |
| 6 | 78 | 827052 | 132204 |
| 7 | 80 | 826606 | 132217 |
| 8 | 81 | 862224 | 132210 |
| 9 | 81 | 862224 | 132234 |
| 9 | 81 | 862224 | 132235 |
| 10 | 82 | 786143 | 132205 |
| 11 | 83 | 786143 | 132216 |
| 11 | 83 | 786143 | 132238 |
| 12 | 84 | 792145 | 132214 |
| 13 | 85 | 827052 | 132228 |
| 14 | 86 | 827052 | 132206 |
| 15 | 88 | 769531 | 132223 |
| 15 | 88 | 769531 | 132233 |
| 16 | 90 | 792145 | 132237 |
| 17 | 91 | 792145 | 132209 |
| 18 | 92 | 827052 | 132218 |
| 18 | 92 | 827052 | 132231 |
| 19 | 93 | 792145 | 132225 |
| 19 | 93 | 827052 | 132202 |
| 20 | 94 | 786143 | 132211 |
| 21 | 96 | 792145 | 132207 |
| 21 | 96 | 826606 | 132219 |
| 21 | 96 | 827052 | 132201 |
| 21 | 96 | 827052 | 132220 |
| 22 | 97 | 827052 | 132221 |
| 22 | 97 | 862224 | 132203 |
| 23 | 98 | 769531 | 132222 |
| - | 98 | 861005 | 861641 |
| 24 | 99 | 792145 | 132212 |
| 25 | 100 | 769531 | 132236 |
| 25 | 100 | 769531 | 132326 |
| - | 100 | 826366 | 826764 |
| 25 | 100 | 826606 | 132227 |
| 25 | 100 | 827052 | 132208 |
| 25 | 100 | 827052 | 132215 |
| 25 | 100 | 862224 | 132200 |

## A.2 i27: The “multiply permutations” program

For the description of the table structure see the previous section.

Only 5 plagiarism pairs escape detection if a 50% cutoff threshold is used.

| rnk | sim | orig | plag |
|---|---|---|---|
| 1 | 23 | winfb063_27 | 132312 |
| 2 | 31 | winfb122_27 | 132321 |
| - | 39 | winfb075_27 | winfb172_27 |
| 3 | 41 | winfb122_27 | 132303 |
| - | 48 | winfb071_27 | winfb121_27 |
| - | 51 | winfb189_27 | winfb224_27 |
| 4 | 58 | winfb122_27 | 132309 |
| - | 59 | winfb099_27 | winfb100_27 |
| - | 66 | winfb195_27 | winfb207_27 |
| 5 | 67 | winfb206_27 | 132314 |
| 6 | 74 | winfb122_27 | 132323 |
| 7 | 76 | winfb063_27 | 132301 |
| 8 | 82 | winfb206_27 | 132311 |
| 9 | 83 | winfb206_27 | 132315 |
| 10 | 86 | winfb087_27 | 132320 |
| 10 | 86 | winfb087_27 | 132322 |
| 11 | 91 | winfb122_27 | 132324 |
| 12 | 93 | winfb063_27 | 132313 |
| 13 | 95 | winfb122_27 | 132308 |
| 14 | 96 | winfb206_27 | 132307 |
| 15 | 98 | winfb206_27 | 132302 |
| - | 100 | winfb035_27 | winfb055_27 |
| 16 | 100 | winfb063_27 | 132310 |
| 16 | 100 | winfb063_27 | 132317 |
| 16 | 100 | winfb063_27 | 132318 |
| 16 | 100 | winfb063_27 | 132325 |
| 16 | 100 | winfb122_27 | 132305 |
| 16 | 100 | winfb122_27 | 132316 |
| 16 | 100 | winfb158_27 | 132306 |
| 16 | 100 | winfb206_27 | 132304 |
| 16 | 100 | winfb206_27 | 132319 |

# Appendix B

# Token sets

Table B.1 lists the tokens in JPLag's **default token set "normal"** for Java.

The maximal token set **"full"** contains additional tokens of the following kinds:

- declarator keywords (ABSTRACT, FINAL, PUBLIC, STATIC, PROTECTED, PRIVATE, NATIVE, TRANSIENT, VOLATILE, SYNCHRONIZED, THROWS, EXTENDS),
- basic type keywords (BOOLEAN_TYPE, CHAR_TYPE, BYTE_TYPE, SHORT_TYPE, INT_TYPE, LONG_TYPE, FLOAT_TYPE, DOUBLE_TYPE),
- operators (ASSIGNOP, ASSIGNBITOP, COND_OR, COND_AND, COND_IOR, COND_XOR, EQUALITY, INSTANCEOF, AND, SHIFT, RELATIONAL, ADD, MULT),
- literals (INT, FLOAT, CHAR, STRING, BOOLEAN),
- special constructs (ARRAY_INIT, CAST, LABEL),
- special keywords (THIS, SUPER, NULL).

The minimal token set **"struc"** contains only tokens related to control flow (if-else, while, for, switch, return, throw, etc.), program block structure (begin/end of class, method, interface, class initializer block), and two hard-to-change declarators (abstract, throws).

The two unused token sets (**"op"**, **"opmore"**) mentioned in Section 4.1.3 represent two intermediate levels of token detail, the first representing a few groups of operators in addition to the "struc" tokens, the second also adding a few declarators and individual other tokens.

The **scanner-based C/C++ token set** has tokens with less semantic content. For instance, the various beginX/endX discriminations shown in the table are all replaced by just OPEN_BRACE, CLOSE_BRACE and complex tokens such as APPLY (for function calls) do not occur; parenthesis tokens are used instead.

Table B.1: The tokens of JPlag’s Java default token set and their relative frequency across hundreds of small programs

| | token | example | freq % |
|---|---|---|---|
| 0 | PACKAGE | `package pizza.jplag;` | 0.000 |
| 1 | IMPORT | `import java.io.*;` | 1.404 |
| 2 | BEGINCLASS | `public class Class {` | 0.633 |
| 3 | ENDCLASS | `} // end of class` | 0.633 |
| 4 | BEGINMETHOD | `public void test() {` | 4.049 |
| 5 | ENDMETHOD | `} // end of method` | 4.049 |
| 6 | VARDEF | `int i;  String text;` | 16.591 |
| 7 | BEGINSYNC | `synchronized(obj) {` | 0.273 |
| 8 | ENDSYNC | `} // end of ‘synchronized’` | 0.273 |
| 9 | BEGINDO | `do {` | 0.093 |
| 10 | ENDDO | `} while(condition);` | 0.093 |
| 11 | BEGINWHILE | `while (condition) {` | 0.423 |
| 12 | ENDWHILE | `} // end of ‘while’` | 0.423 |
| 13 | BEGINFOR | `for (i=0; i<n; i++) {` | 1.391 |
| 14 | ENDFOR | `} // end of ‘for’` | 1.391 |
| 15 | BEGINSWITCH | `switch (expr) {` | 0.595 |
| 16 | ENDSWITCH | `} // end of ‘switch’` | 0.595 |
| 17 | CASE | `case 5:, default:` | 2.398 |
| 18 | BEGINTRY | `try {` | 1.029 |
| 19 | BEGINCATCH | `catch(IOException e) {` | 1.052 |
| 20 | ENDCATCH | `} // end of ‘catch’` | 1.052 |
| 21 | BEGINIF | `if (test) {` | 4.679 |
| 22 | ELSE | `} else {` | 1.836 |
| 23 | ENDIF | `} // end of ‘if’` | 4.679 |
| 24 | BEGINCOND | `(test ? 0 :  1` | 0.086 |
| 25 | ENDCOND | `) // end of conditional` | 0.086 |
| 26 | BREAK | `break;` | 1.565 |
| 27 | CONTINUE | `continue;` | 0.216 |
| 28 | RETURN | `return x;` | 2.486 |
| 29 | GOTO | `goto` | 0.000 |
| 30 | THROW | `throw(exception);` | 0.022 |
| 31 | BEGININNER | `test = new A() {` | 0.000 |
| 32 | ENDINNER | `}; // end of inner class` | 0.000 |
| 33 | APPLY | `System.out.print(’!!’);` | 21.969 |
| 34 | NEWCLASS | `test = new A();` | 3.864 |
| 35 | NEWARRAY | `testarr = new int[i];` | 0.503 |
| 36 | ASSIGN | `i = i+5;  i++;  i+= 5;` | 19.013 |
| 37 | BEGININTERFACE | `public interface if {` | 0.004 |
| 38 | ENDINTERFACE | `} // end of interface` | 0.004 |
| 39 | ENDFILE | (artificial token for end of file) | 0.549 |

# Bibliography


[1] Alex Aiken. MOSS (Measure Of Software Similarity) plagiarism detection system. http://www.cs.berkeley.edu/~moss/ (as of April 2000) and personal communication, 1998. University of Berkeley, CA.

[2] H. L. Berghel and D. L. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, 19(8):65–76, August 1984.

[3] John L. Donaldson, Ann-Marie Lancaster, and Paul H. Sposato. A plagiarism detection system. *ACM SIGSCE Bulletin (Proc. of 12th SIGSCE Technical Symp.)*, 13(1):21–25, February 1981.

[4] J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity within a university programming environment. *Computers and Education*, 11(1):11–19, 1987.

[5] Sam Grier. A tool that detects plagiarism in Pascal programs. *ACM SIGSCE Bulletin (Proc. of 12th SIGSCE Technical Symp.)*, 13(1):15–20, February 1981.

[6] Maurice Howard Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier North-Holland, New York, 1977.

[7] H. T. Jankowitz. Detecting plagiarism in student Pascal programs. *The Computer Journal*, 31(1):1–8, 1988.

[8] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, 31(2):249–260, March 1987.

[9] Karl J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGSCE Bulletin*, 8(4):30–41, 1976.

[10] K. K. Verco and M. J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In John Rosenberg, editor, *Proc. of 1st Australian Conference on Computer Science Education*, Sydney, July 1996. ACM.

[11] G. Whale. Identification of program similarity in large populations. *The Computer Journal*, 33(2):140–146, 1990.

[12] Michael J. Wise. Detection of similarities in student programs: YAP'ing may be preferable to Plague'ing. *ACM SIGSCE Bulletin (Proc. of 23rd SIGCSE Technical Symp.)*, 24(1):268–271, March 1992.

[13] Michael J. Wise. String similarity via greedy string tiling and running Karp-Rabin matching. Dept. of CS, University of Sydney, ftp://ftp.cs.su.oz.au/michaelw/doc/RKR_GST.ps, December 1993.