# Plagiarism detection in game-playing software

Paolo Ciancarini
Dipartimento di Scienze dell'Informazione
University of Bologna, Italy
ciancarini@cs.unibo.it

Gian Piero Favini
Dipartimento di Scienze dell'Informazione
University of Bologna, Italy
favini@cs.unibo.it

## ABSTRACT

Plagiarism is a growing issue in the field of game-playing software. As new ideas and technologies are successfully implemented in free and commercial programs, they will be reused and revisited by later programs until they become standard, but on the other hand the same phenomenon can lead to accusations and claims of plagiarism, especially in competitive scenarios such as computer chess tournaments. Establishing whether a program is a "clone" or derivative of another can be a difficult and subjective task, left to the judgment of the individual expert and often resulting in a shade of gray rather than black and white verdicts. Tournaments judges and directors have to decide how similar is too similar on a case-by-case basis. This paper presents an objective framework under which similarities between game programs can be judged, using chess as a test case.

## Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; K.4.1 [**Computers and Society**]: Public Policy Issues—*ethics, intellectual property rights*; K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection

## 1. INTRODUCTION

According to Merriam-Webster, to plagiarize is "to steal and pass off (the ideas or words of another) as one's own", or "to use (another's production) without crediting the source." This is a particularly sensitive topic in the field of gaming software, in which ideas from previous programs are constantly worked upon and reused by the next generations and innovation is comparatively rarer. A unique situation arises with programs that implement artificial intelligences to play a well-known game, typically a board game like chess or go, or a card game like poker. These programs have a considerable market and by necessity share common traits since they all play by the same rules; plagiarism cases and accusations follow almost automatically from these premises. The situation can only worsen under a competitive scenario pitting these AIs against each other and allowing the general public to compare their moves. This paper stems from involvement in the examination of chess programs for the purpose of plagiarism detection and fitness for tournament participation under antiplagiarism rules. It provides a more objective framework for tournament judges and directors, as well as experts working for them, to make quantitative decisions on the originality of a game-playing program when compared to another, while trying to minimize the public controversy associated with plagiarism cases.

The paper is organized as follows: in Section 2, we motivate our research by citing past and current examples of plagiarism controversy in the field. In Section 3, we discuss the problem of originality in the context of computer game software, and how it is usually dealt with under tournament rules. Section 4 contains an overview of well-known plagiarism detection ideas and techniques. Section 5 presents a series of ideas and techniques, which can combined into a method for making plagiarism less of a concern in game tournaments. In Section 6, we test those ideas on a series of chess programs to give a practical example of how similarities between chess programs can be detected with as little effort and as much accuracy as possible. Finally, Section 7 contains conclusions and ideas for future work in the field.

## 2. PLAGIARISM IN COMPUTER GAMES

During the World Computer Chess Championships, which started in 1974, occasionally some program has been disqualified under accusation of plagiarism. For instance, in Graz 2003 and and in Turin 2006 this happened to some participants. Rybka by Vasik Rajlich is in 2008 the strongest chess program in the world, having won the last two editions of the World Computer Chess Championship (see [11]). When the source code for another program, Strelka, was released to the public, the author of Rybka accused it of being a reverse-engineering of an earlier version of his software, providing a list of similarities in the algorithms used by both as well as differences and improvements attempted by Strelka. Rajlich went as far as announcing that he would claim the Strelka source code as his own in [12], though he has not actually taken any steps in that direction as of yet. The computer chess community seems to have reached consensus on Strelka being a Rybka clone, despite the lack of public source code for the latter program, on the basis of similar behavior and playing style in a number of experiments. This is not the end of the story, however. Assuming that Strelka was indeed a derivative of Rybka, some programmers analyzing its source code also reported significant similarities


*ICFDG 2009* April 26–30, 2009, Orlando, FL, USA.

to Fruit [13], an earlier program by Fabien Letouzey. Originally open source, the author later changed this policy and did not release source code for the software after version 2.1, reportedly because he did not want to see any more clones of his program entering tournaments. Fruit's playing strength (at the time of its release) and the concise elegance of its source code are the main reasons behind the number of forks and derivatives it has had over the years. Obviously, seeing elements from Fruit in a supposed clone of Rybka immediately set transitivity into motion, and some people began a decompiling effort looking for signs of correlation between Fruit and Rybka. This strange reversal forced the author of Rybka out of his accusing position into having to defend the originality of his program and deny any ties to Fruit. The decompiling efforts are, as of the time of writing, substantially inconclusive and may not ever conclude at all. Reports of uncommon practices in non-AI code, similar terms and analogous policies in certain areas have been offered, but there is no smoking gun.

The issue of plagiarism and clones is not unique to these programs and their chess predecessors such as Crafty, which was among the first to have clones disqualified from computer tournaments [9]. While one might be led to think that the maturity of chess as a computer discipline is the primary reason why different people need to come up with similar solutions, the ancient game of go – a game at which programs are still considerably weaker than the best humans, and one that witnesses more innovation than chess – provided a very early example, about a decade before the Strelka controversy. The case of Professor Chen Zhixing's program Handtalk shook the computer go community in the late 90s when he accused two other programs, in particular the North Korean Silver Igo and its later derivatives, of being clones of his program. While Chen provided expert reports citing impressive similarities in assembly code sequences and noted very similar playing style in a number of situations, the location of Igo's developers as well as the novelty of the case made it impossible for tournament ruling bodies to reach any significant decisions regarding these programs.

Ever since these events, tournaments have started to place more and more emphasis on the requirements of originality for participating programs. Of course, there are also legal consequences to code and idea theft when it also constitutes copyright infringement, but the focus of this paper is on the issue of plagiarism in itself and in the context of a competitive scenario. Tournament directors have been taking steps to face the problem of plagiarism: the availability of source code for inspection at any time is the foremost requirement. Often, however, this results in subjective decisions because of time and money constraints on the organizers' part, and both parties are left unsatisfied.

## 3. WHAT MAKES ORIGINALITY

Most tournaments have anti-plagiarism rules and either require derivatives to be clearly listed as such in the submission details, or disallow them altogether. For example, item 2 in the ruleset for the 2008 edition of the ICGA International Computer Games Championship [15] (formerly known as Computer Olympiads), held in Beijing, reads as follows:

*Each program must be the original work of the entering developers. Programming teams whose code is derived from or including game-playing code written by others must name all other authors, or the source of such code, in their submission details. Programs which are discovered to be close derivatives of others (e.g., by playing nearly all moves the same), may be declared invalid by the Tournament Director after seeking expert advice. For this purpose a listing of all game-related code running on the system must be available on demand to the Tournament Director.*

The wording of this rule, which is discussed in greater detail in [10], is quite interesting and reflects the delicate issue at hand. The rule uses the words "derive" and "include", referring to indirect and direct plagiarism, respectively, and then mentions that playing a large number of identical moves is one possible criterion for invalidation after seeking expert advice on the source code. While close similarities in the output are strong indicators of a common origin, this particular field requires more evidence because most games played in these tournaments are two-player zero sum games of perfect information, for which there exist perfect strategies and all programs are after those strategies. In chess, it is often argued that there are several optimal strategies at any given time and even "perfect" programs would probably play different variations. However, very recent results in [3] show that *training* a chess program to play like another engine, for example by running genetic algorithms to tune its parameters, can be very effective and lead to a style of play that closely resembles the original (incidentally, this is how the author of Strelka justifies the similarities to Rybka). Moreover, while for most interesting games a full solution is very far away, now that checkers has been solved for most practical purposes, any program capable of drawing a game with the world champion program Chinook might be considered a clone based on output alone, unless it happens to play different but equivalent variations.

On the other hand, and maybe more importantly, false negatives are also possible, in that even a heavily plagiarized program is not guaranteed to play in a similar style to the original, especially if it only copies the structure and not the details. Moreover, since most programs rely on computational power and are allowed to run on any machine or cluster at the tournament, the same program can and will make different moves on different hardware. And finally, while this is not usually the case in chess, there are classes of programs, such as those based on Monte Carlo techniques, where randomization is the driving force and moves are like samples from a probability distribution.

This is, therefore, a delicate problem. Examining the source code might not suffice to establish originality or plagiarism, either. While trivial clones are easily discovered, more advanced modifications or obfuscation can fool an automatic checker or even a human, implementing the same ideas with different code. It does not help that quite a few features in these programs bring so much of an advantage that an AI would not be competitive without them - in chess, transposition tables, null moves, killer heuristics are but a few examples. Also, all programs need to conform to the same game-specific theory during evaluation: a chess evaluation function must contain such terms as material count, mobility and king safety because that is just the way chess works. The values for the pieces and other parameters are largely set in stone. These engines often do not include a graphical user interface, and instead communicate with an external one through a standard protocol such as XBoard/Winboard or UCI. Obviously, they all need to

implement the same commands.

It is logical to wonder what makes originality in a game-playing agent, or conversely what makes plagiarism in this kind of software. To this end, a brief overview of plagiarism detection in computer programs may be helpful.

# 4. PLAGIARISM AND DETECTING IT

The issue of plagiarism in computer programs has been the object of research for decades now. For an overview of the problem and approaches used to detect it, Clough [2] and Mozgovoy [6] offer a comprehensive listing; the former report was published in 2000, the latter in 2006, showing that the six years in between brought several innovations and evolutions into the field, but no true revolutions. Interestingly enough, the main practical driving force behind progress in plagiarism detection was not the necessity of forensic analysis in code theft cases; at least until more recent times it was, more simply, the necessity to identify cheating students in programming assignments. While this might appear strange at first, the problem of detecting student plagiarism is a complex one and has several traits in common with the analogous issue in computer game programs.

- Students can copy an assignment from other students taking the same test, look for previous solutions to the same test (assuming it was used more than once) or borrow portions of online code that solve specific parts of the problems. Likewise, a computer game plagiarist will resort to using existing open source software or parts thereof, and may sometimes compete in the same tournament with the source of their plagiarism. Here, computer games may actually have a small advantage for the time being, as the number of game-playing programs worth copying from is not very high.

- A class, in whole or in part, is given the same assignment. All computer agents of the same type play the same game. There has to be a degree of leniency towards similarities in the programs.

- Students vary considerably in their ability and willingness to paraphrase and conceal their plagiarism. At the very least, the same or better copying skills should be expected of a computer game plagiarist.

- In both fields, manual plagiarism checking for every pair of programs is unfeasible because of the time and resources this would take.

- In suspicious cases, a teacher or judge needs to make an official ruling as to whether a program is disqualified from the test or tournament.

Mozgovoy [6] shows that there are three main classes of plagiarism detection systems: they can be based on fingerprints, string matching or parse tree matching. Fingerprint detection relies on creating a fingerprint for each document, according to various metrics and criteria such as the number of attributes and variables, the number of operands, the number of code lines, and so on. If two documents have similar fingerprints, there may be some correlation between them. This system is generally considered to be the weakest as even small changes in the code can lead to significant changes in the fingerprint, and as such these methods

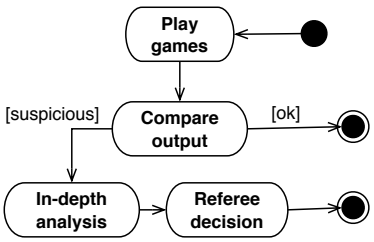


**Figure 1: UML activity diagram for standard plagiarism discovery in a tournament.**

are declining. String matching compares substrings of the two programs, possibly removing comments and taking the language's grammar into account. Some preliminary steps are usually taken, such as replacing identifiers with generic tokens in order to detect plagiarism when variable names have been changed. These algorithms are considered to be more accurate than fingerprint-based systems, though they are slower. Finally, parse trees matching compares language trees generated by the parser, like a compiler. In theory this can be very helpful, though the approach needs more testing.

Several tools exist for finding similarities between code in multiple groups of files. We recall Moss [8], now maintained by Stanford University, which works through a combination of advanced fingerprinting obtained by hashing and string matching. JPlag [7], which is an improvement over older software such as YAP, is a web service running at the University of Karlsruhe and implements an efficient string matching algorithm. Sherlock [16] is the merging of two utilities: one hashes text files into $n$-grams and the other compares the amount of common $n$-grams in a given pair of files. A newer version of Sherlock was developed for the Boss project [17]. We also mention a commercial software, CodeSuite by S.A.F.E, which is free to use for small enough tasks and implements a series of checks on several code features and provides a weighed average of their results. The last tool can also perform *online* analysis, that is, it can interface with search engines to find matching code on the internet, typically from open source projects.

None of the currently available tools are meant to work without a human operator, since false negatives and positives are still common. However, it is believed that these systems are sophisticated enough to detect plagiarism wherein the original code was lightly or moderately transformed. Each tool introduces its own measure and definition of plagiarism: for example, the presence of too many identical substrings or similar fingerprint parameters. However, none of these tools considers the *output* of a program, because none of them can understand the meaning of the code it is examining.

# 5. PLAGIARISM DETECTION FRAMEWORK

The main goal of this paper is to propose a methodology for maximizing the chance of detecting plagiarism among any number of game-playing programs in a competitive environment, while trying to minimize the amount of human resources involved in the tests. The usual workflow to detect plagiarism in a computer tournament is shown as the activ-

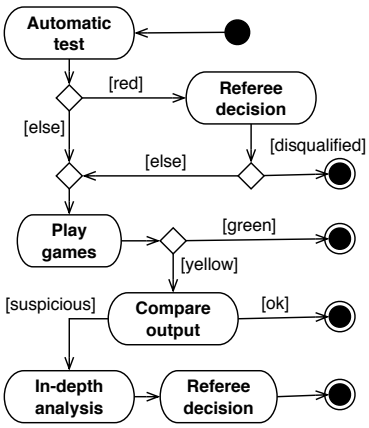


**Figure 2: UML activity diagram for an alternative procedure against plagiarism.**

ity diagram in Figure 1. Programs are normally only tested for plagiarism after the insurgence of suspicious behavior: their output in the games is compared with that of other game engines. Anyone can perform these tests, including other participants who can notify the judges upon finding something strange. As stated above, however, there is some risk of false positives, as well as a whole class of plagiarized programs that are unlikely or impossible to detect with this method.

It can be desirable to base a different method on the same philosophy and common sense behind plagiarism detection in academic courses, since a fair amount of research has been devoted to that topic. Teachers often stress that the most effective measure against plagiarism is not detection, but prevention. For this reason, they often warn their students that their work will run through a plagiarism detection system, and the warning alone has been shown to be a powerful deterrent, decreasing the instances of plagiarism [1]. Game programmers are probably more skilled than the average student, but the consequences of being accused at a tournament are severe on one's reputation and hence cheating is risky. All participants would have take an automatic *preliminary test* of their source code, as shown in Figure 2. We argue that the computer tournament community could only benefit from the academic approach, and that in order to discourage plagiarism a first barrier should exist for all programs, not just those that happen to trigger suspicion with their play. This tool can return one of three values for each program: "green" when no significant similarities are found, "red" when there is overwhelming evidence of similarity, and "yellow" representing moderate but still inconclusive evidence. Of course, yellow and red would also return a list of programs likely to have been copied. This tool should have the following properties:

- High speed. The program needs to analyze dozen of participants consisting of many thousands of lines of code against each other as well as popular open source engines, and needs to do so in a matter of minutes at most. This way, it could be run during the briefing that usually precedes a tournament.
- Little or no human effort to operate. Ideally, it should be as simple as placing each program's code directory in the same place, running a script, and collecting results on the screen.
- Entirely private. Since commercial code could be tested, there need to be guarantees that the files will not leave the computer performing the test. In particular, this may prevent the use of anti-plagiarism web services.
- The tool can act with some paranoia in its yellow zone, but the chance of a false positive upon returning red should be very low or negligible. Moreover, a human judge should be able to quickly identify plagiarism in a program marked as red.

The next section shows how such a tool can be built for the game of chess. There are several advantages from having this first test. First, as noted earlier in the paper, there is no *guaranteed* mapping between similarities in the sources and similarities in the output. Even verbatim plagiarism of a control interface will always go unnoticed, and so will every part of the program that is not directly involved in the generation of the AI's moves. A source test may identify those parts if plagiarized. Another reason is that the earlier plagiarism is found, the better. The discovery of plagiarism evidence building up throughout several tournament rounds may disrupt the serenity of the whole event and certainly cause more controversy and damage than if the offending program had been identified and removed at the start.

Of course, there is no certainty that the first test will identify a plagiarized program, however if a program is so close to another that a few games are enough for people to strongly suspect correlation between the two, the tool might also find correlation, if it exists, easier to detect.

Programs in the yellow zone are considered to be under investigation, though the judges may or may not inform their operators. Comparison of these programs' moves can proceed as usual, and it is possible to narrow the test to just the matching engines rather than every major engine. This can become more of a benefit in the future, when more and more program sources become available. Depending on the level of trust in the tool's accuracy, yellow zone programs may be examined by a human only if performing like another engine, such as in Figure 2, or regardless of its performance.

As it has been seen, however, plagiarism is a topic where computer software is able to make a good educated guess, but is unable to give a final answer on its own. While the purpose of the *computer phase* is to divide programs into broad categories and generate a report consisting of percentages and tokens, the *human phase* is meant to answer the question of whether a program is original enough. The question, as shown in Section 3, is not simple, as we expect yellow zone programs not to be trivial clones of another program but at the same time reminding of one. Currently, there is no system that we know of for making this particular judgment – it is simply up to the expert's opinion and tournament organizers. Ideally, we would like to offer a hard number expressing the seriousness of two programs' correlation, so that it can be compared to a threshold value: just like what the tool does, except with human accuracy. We propose the following scheme. In order to evaluate a program, one needs to define three things: a list of evidence collection *criteria* (how the source code is examined for evidence), a *judgment function* (how serious and meaningful is the evidence), and

a numeric *threshold* separating "pass" from "fail" (how serious the evidence needs to be.) These should be decided beforehand and enforced equally throughout a tournament.

Collection criteria include, for example: whether the human will be assisted by software and, if so, what kind of software; what metric will be used to calculate the size of matching code (for instance, lines of code or parser tokens); whether portions of the source will be excluded from the analysis and whether the code will be preprocessed. It is important to note these criteria as the evaluation may change with them. The judgment function, on the other hand, takes three parameters relative to a single code match: *severity*, *size* and *location* of the matching sequences. These three properties are defined as follows:

**Severity** is the degree of confidence that correlation exists, as defined by the agent performing the test. Some authors, such as Jones [5], try to list the techniques that a plagiarist can use to transform the code they have borrowed. These techniques include renaming identifiers, reordering statements, and replacing structures with equivalent ones, such as a different loop command. Some of the techniques listed by Jones are unlikely to apply to game programs, such as the addition of redundant statements: few programmers would damage their engine's performance in that way unless the statement was obvious enough for a compiler to remove. However, a plagiarist may very well add *non-redundant* statements, genuinely improving on the source. For the purposes of categorization, we propose a three-level scale of severity with some typical indicators that might motivate a human expert to express that opinion.

- **S1 (Suspicion)** The code sequences may do slightly different things and look slightly different, but the agent suspects the same origin. Typical source code indicators: similar control flow, presence of recognizeable equivalent code blocks.
- **S2 (Confidence)** The code sequences do the same thing, though with minor additions or replacements. The agent is confident that correlation exists between the two programs. Typical source code indicators: very similar local variables, conditional guards, statement ordering even when functionally irrelevant.
- **S3 (Certainty)** The code sequences do the same things and are exactly the same except for the transformations mentioned above. The agent is certain that correlation exists. Typical source code indicators: identical code short of trivial transformations such as variable name changes.

**Size** is the size of the matching code sequences, expressed in a metric of choice, such as lines of code or parser tokens.

**Location** is the logical part in the program's taxonomy where evidence was found. This is the only game-specific parameter, though most perfect information games can be decomposed into four parts: Rule handling, Search, Evaluation, and Input/Output. Engines for a game such as chess, which have a very consolidated structure, can be given a more complex taxonomy: we propose one in the next section. The reasoning behind this is that some parts of the program can be considered more important than others, and some might not be crucial enough to constitute plagiarism.

The method is quite straightforward, and consists of iteratively looking for evidence and running each piece of it through the judgment function. The total sum is then compared with the threshold. If the judgment function is linear in the size parameter, the tournament directors can define the function with a simple two-dimensional matrix such as the one in Table 1. The function can be revised over the years as, for example, features that used to be one program's signature become more and more standard.

| | S1 | S2 | S3 |
|---|---|---|---|
| Rule handling | 0.1 | 0.3 | 0.7 |
| Search | 0.3 | 0.5 | 1.0 |
| Evaluation | 0.3 | 0.5 | 1.0 |
| I/O | 0.0 | 0.2 | 0.5 |

**Table 1: Sample judgment function**

# 6. TEST CASE: CHESS

We simulate a hypothetical chess tournament situation by comparing a number of open source chess programs available on the internet. In a real tournament, participant programs would be grouped with pre-arranged source files for the main open source engines (such as those used in this test); in this way, they would be tested against each other as well as the open source engines. For the purpose of this test, source files were concatenated so that each program consisted of a single file. Headers were left out as their more or less standard structure may incorrectly lead to higher correlation ratios. Comments were stripped to avoid higher correlation among programs with comments in the same natural language. These preparation steps take almost no time at all with an appropriate shell script.

Of the fifteen programs in our test, ten are strong real-world players. In alphabetical order, they include the latest open source versions, as of November 2008, of: Arasan (abbreviated as A), Crafty (C), Fruit (F), Gambit Fruit (GF), Glaurung (G), GNU Chess (GN), Sjeng (S), Slow Chess (SC), Strelka (ST), and TogaII (T). These are several serious (non-educative) programs whose source is available on the net and are all written in C or C++. Only a few programs exist that are written in other languages, such as Delfi which is written in Pascal. Gambit Fruit and Toga are both well-known Fruit derivatives and as such we expect to be able to identify them easily. As mentioned, there have been several claims of Strelka containing Fruit-like code, and its programmer was quoted as saying that the inspiration for the program came from Fruit, not Rybka. If these similarities exist, the automatic test should also be able to find them.

There are five more programs obtained automatically from Fruit's source to test the anti-plagiarism tools: we introduce them in increasing order of difficulty. Inverted Fruit (IF) is Fruit with all non-keyword identifiers renamed, as well as all constant values. Identifiers are inverted so that 'queen' would become 'neeuq'; one-letter identifiers are given a prefix and a suffix to make them different from the original source. Even all constant numbers are different, with 0's becoming 9's, 1's becoming 8's and so on. Fruit 10% (F10) is Inverted Fruit with 10% of all code lines interleaved with a line of random code that does nothing. These lines are chosen from a set of about 20 patterns including assignments with and without arrays, conditional branches, cycles and

| | A | C | F | GF | G | GN | S | SC | ST | T |
|---|---|---|---|---|---|---|---|---|---|---|
| A | – | | | | | | | | | |
| C | 2 | – | | | | | | | | |
| F | 0 | 1 | – | | | | | | | |
| GF | 0 | 1 | 86 | – | | | | | | |
| G | 1 | 1 | 0 | 0 | – | | | | | |
| GN | 1 | 3 | 0 | 1 | 1 | – | | | | |
| S | 1 | 2 | 7 | 7 | 1 | 1 | – | | | |
| SC | 1 | 1 | 8 | 8 | 0 | 1 | 2 | – | | |
| ST | 0 | 1 | 11 | 10 | 0 | 1 | 1 | 3 | – | |
| T | 0 | 1 | 90 | 85 | 0 | 0 | 7 | 8 | 11 | – |

**Table 2: Trigram-based correlation.**

| | A | C | F | GF | G | GN | S | SC | ST | T |
|---|---|---|---|---|---|---|---|---|---|---|
| A | – | | | | | | | | | |
| C | 0.6 | – | | | | | | | | |
| F | 0.1 | 0.2 | – | | | | | | | |
| GF | 0.1 | 0.3 | 73 | – | | | | | | |
| G | 0.4 | 0.3 | 0.2 | 0.2 | – | | | | | |
| GN | 0.4 | 0.6 | 0.1 | 0.2 | 0.2 | – | | | | |
| S | 0.4 | 0.4 | 0.3 | 0.3 | 0.3 | 0.4 | – | | | |
| SC | 0.2 | 0.2 | 0.3 | 0.3 | 0.2 | 0.2 | 0.5 | – | | |
| ST | 0.1 | 0.3 | 1 | 0.9 | 0.2 | 0.2 | 0.3 | 0.5 | – | |
| T | 0.1 | 0.2 | 80 | 71 | 0.2 | 0.1 | 0.3 | 0.3 | 0.9 | – |

**Table 3: Trigram-based correlation with unique tokens.**

iterative commands. Identifier and constant values can appear in this code: they are replaced with random names and values. Fruit 50% (F50) is the same as F10 except that the amount of noise code is five times higher. Fruit 50% Replace (F50R) is more vicious because it does not just interleave Fruit's code with reduntant lines; the random lines actually replace Fruit's code, only half of which appears in the program. The average length of Fruit-originating snippets is then 2 lines. Finally, Fruit 90% Replace (F90R) is nine tenths random code and only one tenth, highly scattered Inverted Fruit.

## 6.1 Computer phase

Given the requirements for the automatic test tool outlined in the previous section, we selected two small programs for the test. The first is Sherlock by Rob Pike and Loki, a very simple tool - just a few hundreds of lines - that hashes files into $n$-grams and compares their hash values to estimate correlation. We kept the default values for the search, in particular choosing $n = 3$. Three-token sequences work best as this is the smallest amount for which there is a low likelyhood of the sequence being entirely made of C keywords. This version of Sherlock is not specifically programming-oriented as it can also detect plagiarism in natural language.

Table 2 summarizes the results of using Sherlock's default counting algorithm on the ten real chess programs. Unsurprisingly, Fruit, Gambit Fruit and Toga form an equivalence class and the program seems to pick up some correlation between these programs and Strelka – its score with all three is the highest after those of the obvious clones. However, this score is not very far from other pairs, such as Fruit and Slow Chess. We made minor modifications to the program so that it can offer some additional data: most importantly, the program now can display the shared sequences, so an operator can judge their importance. Moreover, in addition to providing a 0-100 correlation value calculated as the ratio of matching sequences to the total, it can now report a second value that only takes *unique* sequences into account. A quick analysis shows that Fruit and Slow Chess have 355 common sequences, but most of them are repeated: there are only 12 unique sequences, most of which are furthermore very generic and do not hint at correlation (like `<= 1 &&` or `> alpha &&`). On the other hand, out of the 420 common sequences between Fruit and Strelka, 31 are unique and those include some rather specific ones such as `delay = Infinite`, `entry->move_depth = depth` or `const int KingAttackWeight[16]`.

In view of this, we repeat the test only considering the amount of unique tokens in common divided by the amount of unique tokens in the two programs. This may prevent false positives as, for example, the coincidental use of the same flag variable is only counted once; moreover, it will now take fewer sequences to get suspicious since each is unique. The results are summarized in Table 3. Not only does the Strelka-Fruit pair stand out more, but other pairs are highlighted that were not before. GNU Chess and Crafty have 48 unique common sequences, some of which are meaningful: it turns out the two programs share the same 55-field random number table. These pseudorandom numbers, which in chess programs are usually stored in a table to avoid generation overhead, are a minor similarity, but still interesting to find. As for the other pairs around 0.5, these matches are largely due to the use of an EGTB (endgame tablebase), a standard source file that many programs use to interface with a position database that allows perfect play in some endgames. Since many plagiarism tools allow exception sets (files with text not to be considered in the test when found), a serious tournament tool would have to include this feature. In view of these results, one could set the yellow threshold for Sherlock at 10% in the global sequence count case and around 1% in the unique sequence count. Red threshold could be at 40% in the first case, and 10% in the second.

Unsurprisingly, the tools fails to report any of the five constructed Fruit derivatives. With global sequence count, correlation is 1% for the simpler two (IF and F10) and 0% for the rest. With unique sequence count, correlation never gets past 0.1%. So, Sherlock is ideal for cut-and-paste borrowing of even short code sections, but something more specific is needed to catch derived code.

The second tool is SIM by Dick Grune. It is mostly based on string matching and tokenization and has a solid performance as well as a simple command line interface. Unlike Sherlock, it is specificially meant for programming languages and supports some of the most popular ones. Like the first tool, it reports correlation as a number ranging from 0 to 100. The program has seen extensive use at the University of Amsterdam where it was created, and the authors state in [4] that in the computer lab assignments where SIM is used, they set their threshold to 60%. Above that value, a human compares the programs manually; values above 80% have always been correct. The authors also state that the threshold is quite high because their assignments are short, typically a few pages of code. Since we are now comparing programs that are dozens or hundreds of times larger, we expect our plagiarism threshold to be somewhat lower.

| | A | C | F | GF | G | GN | S | SC | ST | T |
|---|---|---|---|---|---|---|---|---|---|---|
| A | – | | | | | | | | | |
| C | 1 | – | | | | | | | | |
| F | 1 | 1 | – | | | | | | | |
| GF | 1 | 2 | 90 | – | | | | | | |
| G | 1 | 1 | 2 | 2 | – | | | | | |
| GN | 1 | 1 | 0 | 1 | 0 | – | | | | |
| S | 0 | 2 | 2 | 2 | 0 | 1 | – | | | |
| SC | 0 | 0 | 0 | 0 | 0 | 0 | 0 | – | | |
| ST | 0 | 1 | 3 | 3 | 0 | 0 | 2 | 0 | – | |
| T | 1 | 1 | 87 | 86 | 1 | 1 | 1 | 0 | 3 | – |

**Table 4: Tokenization-based correlation.**

The major problem with tools that use tokenization is paranoia. To these programs, `2+2` and `a+b` are the same thing, and as such they generate many false positives. Using the program's default values returns ratios that are, in many cases, much too high (over 30%) simply because chess programs all tend to have big arrays filled with constants. The main solution is to raise the length of the minimal detected sequence. The default value is 24 tokens; however, even raising it to 120 may not help in this case. The problem is that a 64 element array, which is very common in a chess program, occupies about 128 tokens, of which 64 are items and 64 are commas. This is one reason why we ignore header files in the analysis, but arrays are often declared in the body. If we increase the minimal sequence length to a value around 150, then we may have worked around the problem, but we will also have lost all but the most pedantic plagiarism, as 150 tokens may easily represent a dozen lines of code. The other way to overcome this problem is to run a preliminary script that literally empties all arrays, removing the content between the curly braces. This is the solution we chose, and it leads to much more realistic results, summarized in Table 4. SIM actually returns two values for each pair of files: one for the amount of file A appearing in file B, and vice-versa. Depending on file sizes, these values can differ: we select the smaller value to compensate for the paranoia effect. As expected, the programs' logic is mostly independent of each other except for the clones. Still, Strelka's 3% implies a slightly higher level of inspiration from Fruit than the other pairs: this also reflects in the length of the longest matching sequence, 156 tokens over 12 lines, higher than for any other pair except Fruit's derivatives. Since longer matches are more likely to be relevant, this can also be an interesting similarity metric. The power of this method is evidenced by the 51-token longest match between Fruit and Glaurung. Where Fruit reads:

```
  PosSEE = pos;
Code[pos++] = GEN_TRANS;
Code[pos++] = GEN_GOOD_CAPTURE;
Code[pos++] = GEN_KILLER;
Code[pos++] = GEN_QUIET;
Code[pos++] = GEN_BAD_CAPTURE;
Code[pos++] = GEN_END;
PosEvasionQS = pos;
```

the meaning of Glaurung's snippet is surprisingly similar:

```
  MainSearchPhaseIndex = i - 1;
PhaseTable[i++] = PH_TT_MOVE;
PhaseTable[i++] = PH_MATE_KILLER;
PhaseTable[i++] = PH_GOOD_CAPTURES;
PhaseTable[i++] = PH_NONCAPTURES;
PhaseTable[i++] = PH_BAD_CAPTURES;
PhaseTable[i++] = PH_STOP;
EvasionsPhaseIndex = i - 1;
```

| | IF | F10 | F50 | F50R | F90R |
|---|---|---|---|---|---|
| F | 88 | 69 | 24 | 16 | 1 |

**Table 5: SIM and Fruit derivatives**

| | Weight | S1 | S2 | S3 |
|---|---|---|---|---|
| **Rules** | | | | |
| Board representation | 0.8 | 0 | 0 | 0 |
| Move generation | 0.8 | 156 | 0 | 0 |
| **Search** | | | | |
| Main search | 1.0 | 0 | 0 | 93 |
| Tables | 1.0 | 27 | 83 | 30 |
| Multiprocessor | 1.0 | 0 | 0 | 0 |
| Other optimizations | 1.0 | 0 | 0 | 0 |
| **Evaluation** | | | | |
| Terms | 0.9 | 0 | 0 | 0 |
| Mating | 0.8 | 0 | 398 | 0 |
| Optimizations | 1.0 | 0 | 0 | 0 |
| **I/O** | | | | |
| Files and network | 0.5 | 0 | 0 | 76 |
| Interfaces | 0.5 | 0 | 0 | 203 |
| **Total** | | 183 | 481 | 402 |

**Table 6: Strelka vs. Fruit (in tokens)**

The last question to be answered is how well SIM fares against the five Fruit derivatives constructed for this test. This is summarized in Table 5. As can be seen, the tool detects all but the last program. In F50R, the average length of shared code portions is about 2 lines, but just over 1 line in F90R, which is rarely enough for the 24 tokens required by SIM for its minimal match. In fact, many matches contain at least some random code, and lowering the token threshold would only make this worse. In truth, even a human would have a very hard time finding correlation in F90R if the variables had meaningful names, so this is not too serious a concern. In view of the above findings, we argue that appropriate tournament values for SIM could be a 10% yellow threshold and a 40% red threshold. SIM can be used in combination with Sherlock or any other tool: all tools are run in sequence and then the strictest decision is applied.

## 6.2 Human phase

We define a common sense judgment function for chess programs as shown in the Weight column of Table 6. The values refer to severity S3: values for S1 and S2 are multiplied by 0.25 and 0.5, respectively. Supposing that Strelka entered yellow zone because of its similarities to Fruit, we run a software-assisted human search. In particular, SIM already provides us with a full list of matching code sequences together with their size, expressed in tokens. The human expert only has to examine each match in turn, discard any false positives and determine severity and location of the code snippets. The process is quite easy and fast for someone familiar with game engines: this comparison only took about 30 minutes. The results are summarized in Table 6. Since Strelka's size is estimated at 45,618 tokens and 1066

tokens are found to be correlating with Fruit, that amounts to 2.34% of its code base, with 0.9% having a strong correlation, and this is a conservative figure since it is only based on SIM's findings. However, if one were to use the judgment function in Table 6, which does not penalize file handling as severely and reduces the importance of S1 and S2 evidence, the adjusted result is only 501.15 tokens or 1.1%. At this point one needs only compare this value with a threshold.

# 7. CONCLUSIONS

We have proposed a method for evaluating and judging a game-playing program that is quite accurate in finding similarities between the tested programs. We have also defined a criterion for plagiarism disqualification that tries to be as formal and objective as possible. Of course, several open problems remain to be addressed, such as how easy it would be for a plagiarist to tune their code to every plagiarism detection tool or combination thereof, and whether the tuning would affect other program features or make it look more suspicious to a human expert. However, the judgment function applies regardless of how evidence is generated, so a human can still inspect the code unassisted. Moreover, the problem of whether a tool can be empowered by game-specific information, such as the ability to identify standard structures like boards and moves, is still to be investigated. Additionally, there is a class of more philosophical questions that stem from a judgment function. These are concerned with defining the essence of plagiarism. If to plagiarize is to borrow *any* amount of code without giving credit, then all programs with even a single snippet of S3 severity code would fall under this definition, no matter how trivial or public the code. If to plagiarize is to borrow any amount of *significant* code, then one has to define exactly what 'significant' code is. If to plagiarize is to borrow *too much* code, then one needs to decide exactly how much is too much.

# 8. REFERENCES


[1] B. Braumoeller and B. Gaines. Actions do speak louder than words: deterring plagiarism with the use of plagiarism-detection software. *PS: Political Science and Politics*, 34(04):835–839, 2002.

[2] P. Clough. Plagiarism in natural and programming languages: an overview of current tools and technologies, Research Memoranda: CS-00-05, Dept. of Computer Science, University of Sheffield, 2000.

[3] O. David-Tabibi, M. Koppel, and N. Netanyahu. Genetic algorithms for mentor-assisted evaluation function optimization. In *Procs 10th annual conference on Genetic and Evolutionary computation*, pages 1469–1475. ACM, 2008.

[4] D. Grune and M. Huntjens. Detecting copied submissions in computer science workshops, Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit, technical report, 1989.

[5] E. Jones. Metrics based plagarism monitoring. In *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 253–261. Consortium for Computing Sciences in Colleges, USA, 2001.

[6] M. Mozgovoy. Desktop tools for offline plagiarism detection in computer programs. *Informatics in Education*, 5(1):97–112, 2006.

[7] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[8] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Procs 2003 ACM SIGMOD Int. Conf. on Management of data*, pages 76–85. ACM, 2003.

[9] J. van der Herik. Augmented ideas. *ICGA Journal*, 20(1):1–2, 1997.

[10] J. van der Herik. The interpretation of rules. *ICGA Journal*, 29(2):53–54, 2006.

**Web References**

[11] Rybka wins WCCC 2008. `http://www.grappa.univ-lille3.fr/icga/game.php?id=1`.

[12] Rybka vs. Strelka. `http://rybkaforum.net/cgi-bin/rybkaforum/topic_show.pl?pid=39198#pid39198`.

[13] Rybka vs. Fruit. `http://www.talkchess.com/forum/viewtopic.php?topic_view=threads&p=222065&t=24047`.

[14] Plagiarism in computer go. `http://www.msoworld.com/mindzine/news/scandals/scandal0400.html`.

[15] Rules of WCCC 2008. `http://www.grappa.univ-lille3.fr/icga/event_info.php?id=20`.

[16] The Sherlock program. `http://www.cs.su.oz.au/~scilect/sherlock`.

[17] BOSS Submission system. `http://www.dcs.warwick.ac.uk/boss/history.php`.