# Online detection of source-code plagiarism in undergraduate programming courses

**D. Pawelczak**
Faculty of Electrical Engineering and Computer Science
University of Bundeswehr Munich (UniBw M), Neubiberg, Germany

**Abstract -** *Plagiarism in programming courses has increased in recent years. Therefore, courses need to be adapted in concept and organization. A relentless pursuit of plagiarizing or the ineffective attempt to completely suppress all communication in the classroom are certainly as inadequate approaches as the installation of an offline plagiarism detection tool without a proper feedback mechanism for the students. Ideally, teamwork is allowed, but each student submits her/ his own solution. A "real-time" plagiarism detection tool gives immediate feedback on a submitted solution during class time. Based on typical indicators, which course instructors use to manually detect plagiarism, a new software system is developed. This system allows submissions of programs, which are automatically tested for proper functionality and checked against other submissions. An evaluation of the system with data sets from an undergraduate C-programming course reveals a high rate of plagiarisms while false detection (coincident similarity) rarely occurred.*

**Keywords:** source-code plagiarism, coincident similarity

## 1 Introduction

Plagiarism is not only found in doctoral theses, but also a typical problem in undergraduate hands-on programming courses. A former survey by Sraka and Kaucic among students revealed, that 72.5% of the students admitted plagiarizing and even 75.5% of all students estimated, that students would plagiarize at least once during their study [1]. A random sample of 31 source files taken from the undergraduate C-programming course at our *Faculty of Electrical Engineering and Computer Science* (*ETTI/ UniBw M*, Munich) checked with JPlag [2] exposed 9 pairs of obvious plagiarism and even one pair with a 100% match [3]. Due to the small number of samples and according to the impression of the course instructors, the actual percentage of plagiarism is probably much higher. This selective testing raised the demand for establishing a tool for plagiarism detection in programming courses as well as further investigations on plagiarism at our university.

Today's technologies provide many simple means of sharing solutions among course participants. The technical means are certainly only one aspect that leads to plagiarism: a large list of reasons for plagiarism in computer science courses is found in [4]. Another reason especially for undergraduate courses is found in the wide range of different knowledge among the first-year students: some students have not yet seen a development environment while other students already earn money as programmers in addition to their studies. The challenge for a lecturer in undergraduate programming courses therefore is to simplify the learning contents for beginners on the one hand, and on the other hand, provide enough substance and diversification not to bore experienced students. Still the different knowledge of the students leads to totally diverging course preparation times. As a concrete example, the C-programming course at *ETTI/ UniBw M* requires according to the curriculum an average course preparation time of 10 hours per week, while students state in a survey their individual preparation time between 1 and 16 hours. The big effort for preparation arouses the desire among some students to shorten this time by taking a copy from a fellow student instead of preparing their own solution. This behavior has increased in recent years and leads to lower grades and a higher failure rate in the final examination.

This paper discusses different approaches at our faculty to minimize plagiarism (chapter 2-3), it presents an online detection tool, that checks for plagiarism during the class time (chapter 4-5) and summarizes the results of an evaluation of this tool with data from a C-programming course (chapter 6).

## 2 Prevent plagiarizing

At our faculty we utilize two different approaches (summarized in 2.1 and 2.2) to prevent plagiarizing [3]. A third approach (2.3) shall be established.

### 2.1 Prevent sharing

One typical but certainly not satisfactory approach is to prevent any import or sharing of files in the classroom. This typically involves disabling of Internet access, file-sharing access and ban on USB devices, mobile phones and so on. A further restriction is not to reveal the assignment before class time and not to reuse assignments. With all these precautions, plagiarizing is limited to class time. If a course needs preparation, the students are requested to bring in a printout or handwritten sheet of their prepared solution. The printout is typed in during class time and each student provides her/ his own solution (at least regarding the individual typing). One could say, that students might have a benefit from typing in a prepared program. In case the student types his/ her own solution, it might help as a review. But certainly a beginner typing in the solution of an experienced program-

mer does rarely give a benefit, because these solutions often require programming skills, which are not covered by the accompanying lectures yet. Cutting off the Internet during a hands-on course does not reflect the state of the art, as many professional programmers query the Internet for online tutorials, programming standards, etc. As cited in [5]: "(…) code copied from a website that assists in a specific task is potentially good practice." Last but not least, a cat-and-mouse game begins, as students might try to find leaks in the communication restrictions of the classroom.

### 2.2 Teamwork with an additional colloquium

A second approach is to ignore technical restrictions. This will certainly invite students to share solutions, so an additional colloquium is required to grade the individual accomplishment. Often a single question is enough to find out whether a student prepared the assignment on his own. The colloquium has the advantage that students can work in teams as long as each team member passes the colloquium. Allowing teamwork does not only provide the benefit of learning from each other, it also reduces the need for clandestine plagiarism. Ideally this approach requests a colloquium for each student after each lesson. Only the regular basis allows detecting asymmetric teams and enables the possibility to react quickly on cheating. Students have the most benefit of this approach because of the individual discussion with the lecturer. Unfortunately, this approach requires additional resources to perform the required colloquia.

### 2.3 Teamwork with individual solutions

The third approach takes some advantages of the second: ignoring technical restrictions and allowing teamwork. Instead of a colloquium, an individual solution is demanded from each student. This solution can be checked manually, tool-based afterwards or tool-based during the class time. Checking the solutions outside the class time requires a feedback mechanism while checking during class time enables an immediate feedback. Although it is often not difficult to detect plagiarism manually (see chapter 4.2), tracing the plagiarism is laborious and discussing the offense with the students is time-consuming. A tool-based approach allows passing the buck to the software tool, which claims the plagiarism.

## 3 Former and future course set-up

As up to now, the instructors manually checked the source codes of the students. It was often an individual decision of the instructor how to deal with cheating. One possibility was to request additional tasks or changes in the assignment, another was to completely reject a student's solution. The decision was often based on the student's level and hindsight. A similar problem arose with the completeness of an assignment. While an instructor might accept the solution of a skilled student although it does not catch every exception, the instructor might reject a solution of another student, which is more exhaustive. This situation was certainly non-satisfying neither for the instructors nor the students, but found its origin as so often in too less time and staff. So both, checking sources against plagiarism and testing the proper functionality of a student's solution should be tool-based in order to perform a fair-minded and uniform grading.

For our new didactical concept students prepare as before parts of their programming assignment at home and use the class time for electronic submission of their solutions. We will have the same class time and the same number of instructors, but due to the electronic submission, the instructors are relieved from the laborious and time-consuming task of checking sources manually for plagiarism. The instructors therefore should have more time for discussing different solutions or for answering questions regarding the assignment.

## 4 Plagiarism detection

### 4.1 Defining plagiarism

We define plagiarism as discussed in [5] as reproduction/ copying source-code without making any adaptions or just making moderate alternations. Comparing sources is restricted to sources submitted during class time. Checking against Internet sources or other databases is not performed. The level of alternations is represented by a threshold, which defines the maximum allowed similarity between two source submissions. The threshold shall be variable as it depends on the assignments: e.g. coding a function for calculating the factorial *n!* does not result in too many different solutions, compared to an implementation of a Sudoku-solver-algorithm.

### 4.2 Detecting Plagiarism

Many different methods how to detect plagiarism are implemented in existing tools. A detailed overview of different approaches is found in [7]. Apart from an implementation point of view, let us first focus on how instructors manually detect plagiarism, without actually comparing source files in detail. Typical indicators are:

- *Output*: Identical or similar output during run, which does not fully reflect the output as expected by the assignment, sometimes even with typos
- *Behavior*: Identical behavior when processing invalid input, e.g. entering letters instead of numbers
- *Tests*: Identical tests or test input provided by the students
- *Structuring*: Identical or similar structuring of the source code, like: defines before/ after includes, functions declarations at the beginning/ in header files/ no declarations at all, main as the first / last function, and so on
- *Indention*: Lack of indention (as typically the indention gets lost when sending source code per email)
- *Comments*: Numerous identical explaining comments, out-commented instructions reflecting a different version

- *Coding-Styles*: The source code represents different coding styles
- *Flipping*: The source code completely changes between submissions in a very short time
- *Questions*: Asking a question like: "Why does your program crash on that specific input?" often reveals, that a student is actually not the author

From a technical point of view, it is much easier to implement a token-based comparison on files as to focus on these indicators. Still every plagiarism detection tool has to deal with coincident similarity, i.e. a false detection. The challenge is to reduce on the one hand the information in order to be able to deal with simple alternations like renaming identifiers or changing string literals, and on the other hand prevent that the reduction leads to a false detection.

## 4.3 Course-specific requirements

As all course participants work on the same given assignment, sources will reflect a high degree of similarity and the risk of coincident similarity (false detection of plagiarism) is high [6]. Often programming assignments provide a project template, a code basis or skeleton to be completed in the assignment or have strict requirements regarding input or output of the program. These additional requirements further increase the similarity. Plagiarism detection therefore shall take code templates into account to distinguish between code fragments provided by the instructor and the actual implementation by the course participants. In case of false-detection (coincident similarity) students shall be given the chance to submit their solution (based upon the decision of the instructor) without plagiarism detection enabled. From where and how students obtained their sources in case of plagiarism is out of the scope of the detection system except with respect to the reporting: the system shall report, which participant has already submitted a similar source. Further investigations are up to the instructors.

Actually designing a tool for plagiarism detections raises the question: "What does the system expect from the student." It would certainly give a drawback, if students modify a copy of an efficient and well-programmed code towards a worse or erroneous code just to circumvent the plagiarism detection. The evaluation as discussed in chapter 6 shows, that plagiarizing rises with the difficulty of the assignment. If we resign to the fact, that some students are not able to cope with the assignment (these students probably will not pass the final examination, but they have to attend the course), we need to define, what qualifications (with respect to the course contents) students get from collaboration with other students. If a student starts working with a copy from a fellow student, but is capable of doing syntactical and semantical changes, fixing of errors (e.g. regarding invalid input), the student accomplishes certain skills, he might not have accomplished without using the initial copy. On the other hand there is certainly no learning effect by just performing editorial (lexical) changes.

# 5 System design

## 5.1 Overall system

With respect to the indicators listed in chapter 4.2, it is obvious, that a single tool would become too complex to accomplish all tasks. Neither is it possible to implement all features at once. Therefore the functionality is distributed among different sub-systems, which allows to further extent the system in future. These are:

- *PlagC2* (core plagiarism detection tool)
- *CGuide* (coding style test)
- *MOPVM* (a virtual machine for functionality testing)
- *IDE* (an integrated development environment for editing and debugging purposes), that includes the *TaskController* (an extension to the IDE, which controls the submission of an assignment)
- *Compiler & Linker*

Tab. 1 shows the responsible sub-systems for the different indicators and reflects the current development state.

Table 1. Responsibilities of sub-systems

| Indicator | Responsible sub-systems |
|---|---|
| Output | *PlagC2* (checking string literals)<br>*MOPVM* (checking for the expected output) |
| Behavior | *MOPVM* (checking for the correct I/O and exception handling) |
| Tests | *PlagC2* (checking string literals, token streams of test functions, overall source file structure)<br>*MOPVM* (if requested by the assignment, check, if proper tests have been performed) |
| Structuring | *PlagC2* (checking the overall source file structure) |
| Indention | *CGuide* (checking the source code for proper indention: sources without indention can not be submitted) |
| Comments | *PlagC2* (currently only based on the file structure, a future extension is planned) |
| Coding-Styles | *CGuide* will warn on changing coding-styles. This indicator is not yet used for plagiarism detection. |
| Flipping | *TaskController* & *PlagC2* (flipping is not yet supported, as each submitted file is treated as a new without respect to previous submissions), see chapter 7. |
| Questions | *TaskController* & *MOPVM* (asking questions is not yet supported. In future, some prepared questions can be asked with the solution derived from the source code of the student, e.g. "is it call-by-value or call-by-reference?" |
| Token-based comparison | *PlagC2* (checking the token stream) |

Fig. 1 shows the structure of the overall system and the interaction of the sub-systems. We use as front-end for the students the *Virtual-C IDE*, as it already implements a virtual machine (*MOPVM*), which is suitable for automated testing of C-source files [8]. All transactions between the webserver and the IDE are PIN and TAN based.

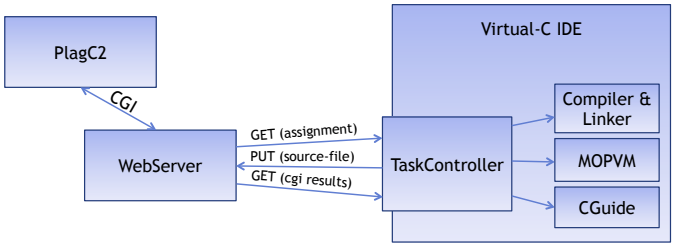

Figure 1. Structure of the plagiarism detection system

## 5.2 The core plagiarism detection tool

For a high flexibility and reusability the plagiarism detection tool is developed as a CGI script for usage within a webserver. The *TaskController* submits source files to the webserver and invokes the *PlagC2* tool via a CGI request (compare Fig. 1). A precondition for running *PlagC2* is, that the source file meets a minimum of requirements regarding its code style, compiles, links and fulfills most of the functional tests. This simplifies the implementation of *PlagC2*, as it gets well-formed source files.

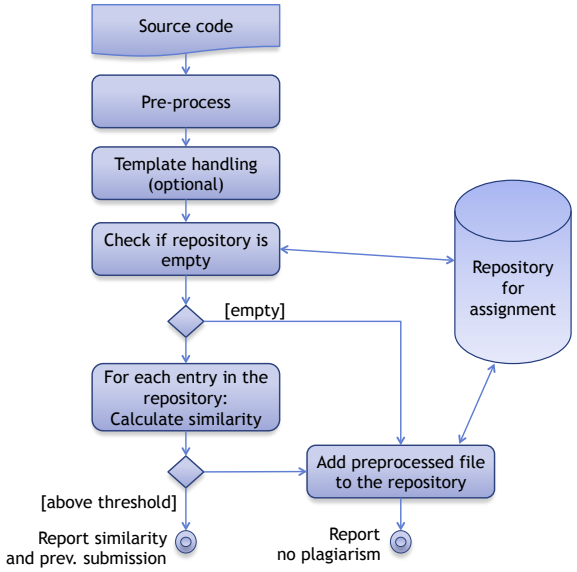

Figure 2. Schematic workflow of the core plagiarism detection tool

Fig. 2 shows the schematic workflow of the plagiarism detection tool. The source code is pre-processed and template code is optionally removed. This reduces size of the source code for a better storage within the repository as well as for a faster comparison with other submissions. In case the repository is empty, the pre-processed file is added. Otherwise, each entry in the repository is compared (see chapter 5.2.2). If a similarity is above the threshold, plagiarism will be reported with the similarity value and the TAN of the corresponding submission.

### 5.2.1 Preprocessing

The pre-processing of the source file reduces the C-source to four strings as shown in Fig. 3:

- *File structure*: A line-oriented representation of the code (each line is represented by a single character)
- *Identifiers*: A list of all identifiers using hash-codes
- *Literals*: A list of all string literals
- *Tokens*: A sequence of all tokens (each token is represented by a single character).

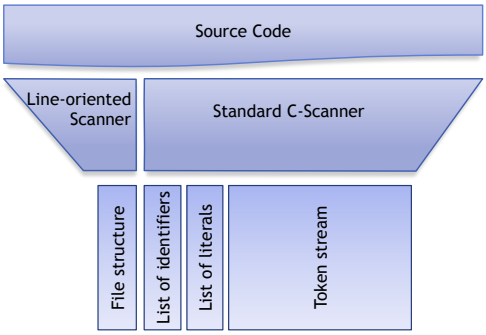

Figure 3. Pre-processing the source files

The pre-processor consists of two scanners. The first scanner is line-oriented and the second scanner is a standard C-scanner that removes whitespaces and comments, but does not perform includes. The *DLex*-Scanner generator is used to create table driven finite state automata from sets of regular expressions.

```c
/* This is my solution */
#include <stdio.h>

long Fibonacci(int n);

int main(void) {
   int n=1;
   scanf("%d", &n);
   printf("Fibonacci(%d) is %ld\n", n,
     Fibonacci(n));
   return 0;
}
```
Figure 4. Example source file

Fig. 4 represents a short example file, which is used to demonstrate the four string representations in short:

*File structure*: The line-oriented scanner divides the source code into conceptual lines, which are represented by a single character, such as e.g.: I: includes, K: comments, L: empty lines, P: prototypes, R: return-statement, V: variable definitions, a: variable initialization, m: definition of function main(), p: usage of a printf-function, s: usage of a scanf-function, {: opening brace and }: closing brace and ?: for expressions, initialization lists, etc. In case a line contains more than one statement, only the first statement is detected, as a typical regular expression catches the whole line, like: "[\s\t]*while[^\n]*\n". The resulting string representing the structure of the example source is "KILPLmasp?R}".

*Identifiers*: The second string, which is derived from the source code, is holding hash-values of all identifier names

(alphabetically sorted). While the source-file is tokenized, the C-scanner stores all identifiers in a symbol table. 32-Bit hash codes are derived from the identifiers in the symbol table and assembled as a hexadecimal representation in a string, as for example: "42e76a5a42fd9d16431337c1435069c145a21b63".

*Literals*: The third string holds trimmed versions of all string literals, i.e. all whitespaces (space-, tabulator-, carriage-return- and line-feed-characters) are removed. The result is derived from the symbol table, which keeps all string literals during the tokenizing phase. For our example the literal string results to: "Fibonacci(%d)is%ld%d".

*Tokens*: The forth string represents the tokens of the source file. A single character per token is used to reduce the original source file into a single string representation. Here the string is: "I<x>5x(4x);4x(#){4x;x(",&x);x(",x,x(x));R0;}". Most operators and punctuation marks are directly taken, while keywords, symbols and operators with multiple characters are reduced to a single character, as for example: 0: octal literal, 1: decimal literal, 2: char, 3: short, 4: int, 5: long, I: #include, R: return, x: identifiers, ": string literal, #: void.

### 5.2.2 Comparing

The four string representations are compared to the corresponding string representations stored in the repository. Except for the list of identifiers, the *Greedy-String-Tiling* algorithm is used [9]. The algorithm splits the strings into character-sequences of a minimum match length and tries to match these sequences. On a successful match the consecutive characters are checked until a different character is found. The algorithm returns the percentage of matching characters to the overall characters in the string. The advantage of this comparison is, that sequences equal or above the minimum match length can be rearranged in an arbitrary order, but the algorithm still returns a similarity of 100%. The identifiers are alphabetically ordered and provide a fixed structure, so a fast linear string comparison can be used instead of the *Greedy-String-Tiling* algorithm.

Comparing the file structure shall indicate a plagiarism as discussed in chapter 4.2. We use a minimum match length of 3 for the comparison. It is obvious, that inserting additional empty lines or putting all in one line will break the structure of the file and would reduce the similarity of two files although they are identical regarding the semantic. Detecting a high similarity in the files' structure can serve as an indication of plagiarizing but not vice versa. The same applies for identifier names. Changing the names of identifiers (e.g. using "search & replace") is regarded as an editorial change and represents plagiarism according to our definition (see chapter 4.1). It is different with string literals. A typical indicator is the output of a program that varies from the expected output. While the expected output usually is identical throughout an assignment, a match on a variation strongly indicates plagiarism. The same applies for test code, which contains identical string literals. If we can ignore string literals containing the expected output, we can put more weight on all other string literals.

The token stream is apparently the strongest indicator for plagiarism and is used in many existing solutions, e.g. such as JPlag[2] or YAP3[11]. A former approach for plagiarism detection at our faculty was also primarily based on token comparison [3]. Yet this method does not take the indicators as described in chapter 4.2 into account. For *PlagC2* we test the similarity of the token stream first. If the similarity is above the threshold, the remaining three strings are compared and the overall similarity is calculated as:

$$S = \frac{S_f + S_i + 10 \cdot S_t}{12} \quad (1)$$

$$S = \frac{S_f + S_i + 2 \cdot S_l + 10 \cdot S_t}{14} \quad (2)$$

where $S$ represents the total similarity as a weighted average of the four similarities: $S_f$ (file structure), $S_i$ (identifier), $S_l$ (literals) and $S_t$ (tokens). Equation (1) is applied in case $S_l$ is zero. This is typically the case for assignments with a graphical output. The equations implicitly define a theoretical threshold for a plagiarized source: In case of a 100% match on the token, but a complete different structure, different identifier names and string literals, $S$ results for (1) in: 83.3% and for (2) in: 71.4 % respectively.

### 5.2.3 Template handling

A programming assignment often requests a specific output of the program or provides a file, which students complete. A template can be provided for *PlagC2*. This file represents the file as given to the students plus string literals of the requested output. The *PlagC2* tool performs the same pre-processing on the template as described in chapter 4.2.1. A variation of the *Greedy-String-Tiling* algorithm removes all sequences that match the template from the four string representations of the source code.

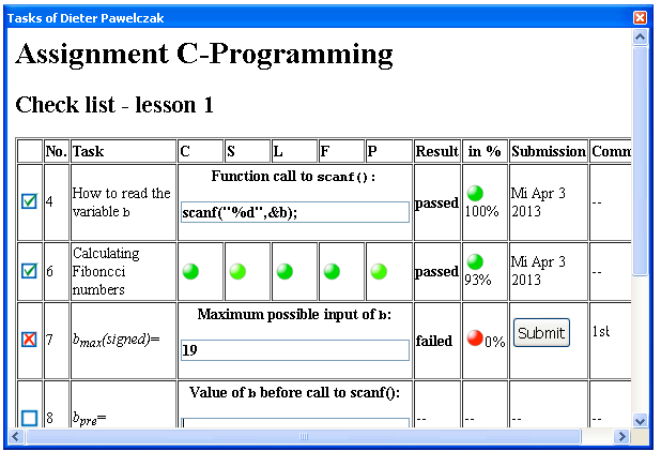


Figure 5. Screenshot of the *TaskController* view.

### 5.2.4 The user interface

The *TaskController* extends the IDE by an additional web view, which allows to answer questions regarding the current assignment and to submit the source code for code and plagiarism checking. Fig. 5 shows the user interface. The results are presented in a traffic light scheme for the individual checks: Green refers to success (no errors, no warnings), yellow (no errors) and red (errors). The abbreviations for the checks are: C (source compiles), S (coding style), L (object file links), F (functional tests) and P (plagiarism detection).

# 6 Evaluation of the System

For the development and evaluation of the system, students were asked to provide in an anonymous form their assignment results of the undergraduate course *C-Programming* (spring 2012), which have not yet been checked against plagiarism. The course consists of 8 different assignments (see Tab. 2). 50 students submitted about 30 (different) source files per assignment. Instead of a submission during class time, the source files have been consecutively submitted with a random PIN to the webserver as a simulation of the real system.

Table 2. Plagiarism Results
(Programming in C, Spring-Course 2012)

| **Threshold for plagiarism: 80%** | **Assignments 1-8** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ***Integer*** | ***Floating Point*** | ***Strings I*** | ***Structures*** | ***Strings II*** | ***Arrays*** | ***Lists*** | ***Files*** |
| | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** | ***8*** |
| Detected plagiarism (Similarity above threshold) | 25% | 32% | 37 % | 52% | 37% | 57% | 58% | 65% |
| Average similarity | 64% | 74% | 77% | 75% | 71% | 78% | 77% | 79% |
| Average similarity above threshold | 87% | 89% | 93% | 89% | 85% | 89% | 89% | 90% |
| Linear gradient[a] | 0.55% | 0.51% | 0.28% | 0.06% | 0.34% | 0.16% | -0.11 % | 0.04% |
| Linear gradient below threshold[b] | 0.42% | 0.56% | 0.46% | 0.24% | -0.02% | -0.40% | -0.07 % | -0.17% |
| Number of submissions | 32 | 28 | 30 | 31 | 30 | 30 | 26 | 29 |

[a]gradient of a linear trend line from submission 2 to the last submission
[b]gradient without samples above the threshold of 80%

The results in Tab. 2 show, that the similarity of source files increases with the complexity of assignments. Although the number of tokens is much smaller in the first assignments, these diverge much stronger than the results of the later assignments (see also Fig. 7). The source lines of code are about 40 for the first and 220 for the last assignment. The highest similarity of assignment 8 can be explained, because this assignment had been reused from the previous year.

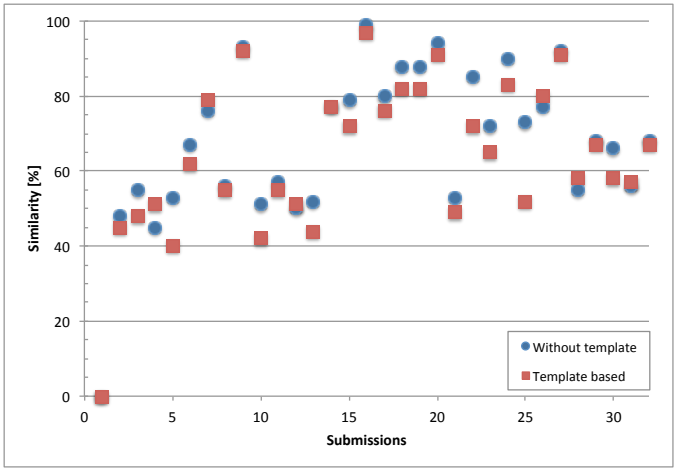


Figure 6. Evaluation of assignment 1 (Integer)

Fig. 6 shows the similarity results of the 1st assignment with and without applying a template file. The first submission always provides zero similarity, because the repository is still empty. The usage of the template file reduces as expected the calculated similarity. In Fig. 6 the average reduction is about 5%. The similarity increases with the submissions as the data in the repository grow. Nevertheless, as can be seen in Fig. 6, students can still provide individual solutions up to the last submission. This is an important outcome of the evaluation as a system-inherent drawback is, that the conditions for submissions are not equal for all students: with each submission, the repository might get a new data set for comparison. It obviously depends on the assignment, if the condition to pass the plagiarism detection gets tougher: Tab. 2 lists the linear gradient for the calculated similarity versus the number of submissions. This is high for the first (about 24 submissions let the similarity increase by 10%) and irrelevant for the later assignments.

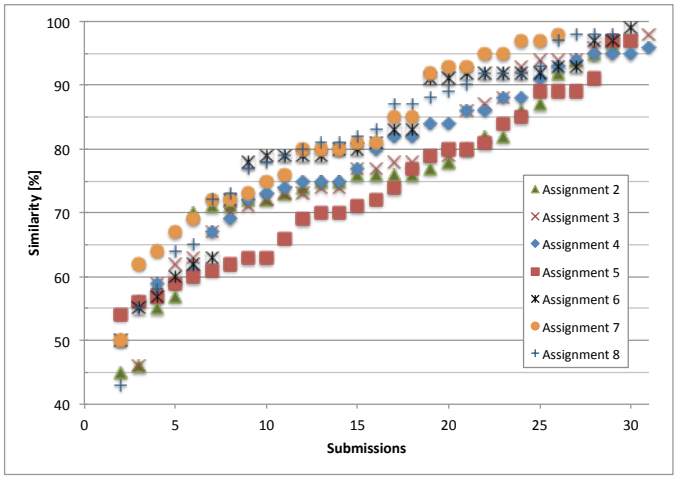


Figure 7. Similarity of assignments 2-8 (sorted)

The results in Fig. 7 reflect the increase of plagiarisms for later assignments. For a better representation the similarity is viewed in ascending order.

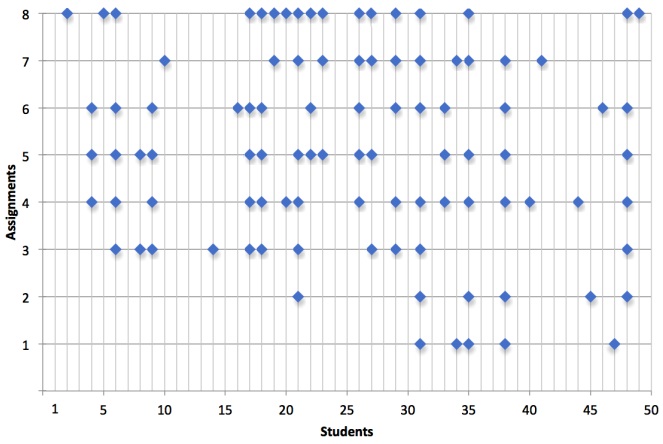


Figure 8. Distribution of plagiarizing (students versus assignments)

Unfortunately, the submissions to the assignments are incomplete. There are only a few students, who provided their source code for all assignments. Therefore Fig. 8 lets us expect, that some students (e.g. no. 31, 35, 38, 48) utilized unpermitted collaboration through out the course, while other students started taking advantage of collaboration in the later assignments (e.g. no. 17, 18, 26). The assignments 4, 5 and 6 fell in the time of re-examinations; this explains the accumulation here (e.g. no. 4, 33). Apart from assignment 8 the number of isolated cases is small (e.g. no 10, 14, 16, 40, 41). Checking these cases manually revealed, that except for two cases, the source files are nearly identical to previous submissions. The first case is obviously an earlier version of another submission, as it contains code fragments, which have been removed or out-commented. The second case (no. 40, assignment 4) is a false detection (coincident similarity) with a weighted similarity of 80%: the manual comparison revealed an individual work. As assignment 4 is a graphical application and does not provide text output, equation (1) has been applied. A high similarity on identifier names, structure and token streams obviously led to this false detection.

# 7 Conclusion and Future Work

Cheating in programming courses has widely spread. On the other hand prohibition of collaboration does not reflect the daily work of a software engineer. A compromise needs to be found between collaboration and individual accomplishment. A tool based plagiarism detection, which reports the offense during class time to the student, provides the advantage that students can discuss the offense with the instructors immediately. Furthermore we need to avert the danger, that students modify a well-programmed code towards an inefficient code just to circumvent the plagiarism detection. Therefore code-checking and detecting plagiarism should always be performed in combination.

Apart from the source code and token comparison several indicators have been defined, which call course instructors' attention to plagiarizing. Based on these indicators a software system is build to check source code automatically for proper functionality and against other submissions. The evaluation of the system shows that almost all cases of reported plagiarism actually represent copies with none or minor (mostly lexical) modifications or simple alternations. Due to the sound evaluation results, the system will go on stream for the upcoming C-programming course (spring 2013).

Some indicators for plagiarisms, which are discussed in chapter 4.2, have not been fully integrated into the system. An interesting approach as described in [6] is to rather report on a change in the coding style of a student, than to report a suspicious similarity. This approach can be integrated into the system in future and would further avoid coincident similarity. It would also reduce the so-called "flipping" we observed during the evaluation phase of the electronic submission system: A student tried at the end of class time hastily to submit different versions (received via e-mail) into the system, but all versions were rejected. Such a behavior is expected with the system in operation and should be prevented by the system in future.

# 8 References

[1] D. Sraka, B. Kaucic, "Source Code Plagiarism" Proceedings of the ITI 2009, 31st Int. Conf. on Information Technology Interfaces, June 22-25, 2009, Cavtat, Croatia

[2] L. Prechlet, G. Malpohl, M. Philippsen, "JPlag: Finding plagiarisms among a set of programs." Technical Report 2000-1. Karlsruhe: Fakultät für Informatik, Universität Karlsruhe; 2000.

[3] C. Engelhardt, "Implementation and evaluation of a software for plagiarism checking of program source code", Bachelor thesis, UniBw M (in German: "Realisierung und Evaluation einer Software zur Plagiatsprüfung von Programmquelltexten", 2012.

[4] E. Roberts, "STRATEGIES FOR PROMOTING ACADEMIC INTEGRITY IN CS COURSES," 32nd ASEE/IEEE Frontiers in Eductation Conference, November 6-9, 2002, Boston MA

[5] G. Cosma and M. Joy, "Towards a Definition of Source-Code Plagiarism," IEEE Trans. Education, vol. 51, no. 2, pp. 195-200, May 2008.

[6] A. Ohno, H. Murao, "Work in Progress - A Novel Methodology to Reduce Instructors' and Student's Psychological Burdens in Source Code Plagiarism Detection", 40th ASEE/IEEE Frontiers in Eductation Conference, October 27-30, 2010, Washington DC

[7] G. Cosma and M. Joy, "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis", IEEE Trans. Computers, vol. 61, no. 3, pp. 379-394, March 2012.

[8] D. Pawelczak, "Virtual-C IDE - USER MANUAL", available online at: https://sites.google.com/site/virtualcide/ [2012-04-06]

[9] M. Wise, "String similarity via greedy string tiling and running Karp-Rabin matching", Department of Computer Science, University of Sydney, Technical Report Number 463, March 1993

[10] A. Aiken, "Moss: A System for Detecting Software Plagiarism," Univ. of California-Berkeley, available online at: http://theory.stanford.edu/~aiken/moss/ [2012-04-06]

[11] M.J. Wise, "YAP3: Improved Detection of Similarities in Computer Program and Other Texts," Proc. 27th SIGCSE Technical Symp., 1996, Philadelphia, Pennsylvania, USA, February 15-17, pp. 130-134