Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments[☆]

Milena Vujošević-Janičić[a,*], Mladen Nikolić[a], Dušan Tošić[a], Viktor Kuncak[b]

[a] *Faculty of Mathematics, University of Belgrade, Studentski trg 16, 11000 Belgrade, Serbia*

[b] *School of Computer and Communication Sciences, EPFL, Station 14, CH-1015 Lausanne, Switzerland*

Abstract

Context: The number of students enrolled in universities at standard and on-line programming courses is rapidly increasing. This calls for automated evaluation of students assignments.

Objective: We aim to develop methods and tools for objective and reliable automated grading that can also provide substantial and comprehensible feedback. Our approach targets introductory programming courses, which have a number of specific features and goals. The benefits are twofold: reducing the workload for teachers, and providing helpful feedback to students in the process of learning.

Method: For sophisticated automated evaluation of students' programs, our grading framework combines results of three approaches (i) testing, (ii) software verification, and (iii) control flow graph similarity measurement. We present our tools for software verification and control flow graph similarity measurement, which are publicly available and open source. The tools are based on an intermediate code representation, so they could be applied to a number of programming languages.

[☆]This work was partially supported by the Serbian Ministry of Science grant 174021, by Swiss National Science Foundation grant SCOPES IZ73Z0_127979/1 and by COST Action IC0901 "Rich Model Toolkit — An Infrastructure for Reliable Computer Systems".

[*]Corresponding author. University of Belgrade, Faculty of Mathematics, Studentski trg 16, 11000 Belgrade, Serbia. Tel.: +381-11-2027801. Fax.: +381-11-2630151

Email addresses: milena@matf.bg.ac.rs (Milena Vujošević-Janičić), nikolic@matf.bg.ac.rs (Mladen Nikolić), dtosic@matf.bg.ac.rs (Dušan Tošić), viktor.kuncak@epfl.ch (Viktor Kuncak)

Results: Empirical evaluation of the proposed grading framework is performed on a corpus of programs written by university students in programming language C within an introductory programming course. Results of the evaluation show that the synergy of proposed approaches improves the quality and precision of automated grading and that automatically generated grades are highly correlated with instructor-assigned grades. Also, the results show that our approach can be trained to adapt to teacher's grading style.

Conclusions: In this paper we integrate several techniques for evaluation of student's assignments. The obtained results suggest that the presented tools can find real-world applications in automated grading.

Keywords: automated grading, software verification, graph similarity, computer supported education

---

1. Introduction

Automated evaluation of programs is beneficial for both teachers and students [1]. For teachers, automated evaluation is helpful in grading assignments and it leaves more time for other activities with students. For students, it provides immediate feedback which is very important in process of studying, especially in computer science where students take a challenge of making the computer follow their intentions [2]. Immediate feedback is particularly helpful in introductory programming courses, where students have little or no knowledge of basic algorithmic and programming issues and have frequent and deep misconceptions [3].

Benefits of automated evaluation of programs are even more significant in the context of online learning. A number of world's leading universities offer numerous online courses. The number of students taking such courses is measured in millions and is growing quickly [4]. In online courses, the teaching process is carried out on the computer, the contact with teacher is already minimal and, hence, the fast and substantial automatic feedback is especially desirable.

Most of the existing tools for automated evaluation of students' programs are based on automated testing [5]. In these tools testing is used for checking whether the student's program exhibits the desired behavior on selected inputs. There are also approaches for using testing for analyzing other properties of software [6]. Most interesting such properties in educational context

are efficiency (usually addressed by profiling on selected test cases) and the presence of bugs that could make memory violations or raise runtime errors. For example, in programming language C, some important bugs are buffer overflow, null pointer dereferencing and division by zero. Note that some of the errors are difficult to detect by tests, so they require static analysis (analysis of the code without executing it). There is a variety of software verification tools [7, 8, 9, 10, 11, 12, 13] that could enhance automated bug finding in students' programs.

Relevant aspects of program quality are also its design and modularity (an adequate decomposition of code to functions). These aspects are often addressed by checking similarity to teacher-provided solutions. In order to check similarity, among the aspects that can be analyzed are frequencies of keywords, number of lines of code, and number of variables. Recently, a sophisticated approach of grading students' programs by measuring the similarity of related graphs has been proposed [14, 15]. There are recent surveys of different approaches for automated evaluation of students' programs [16, 17].

In this paper, we propose a new grading framework for automated evaluation of students' programs aiming primarily at small sized problems from introductory programming courses, which have unique properties and which are the most critical since there are a lot of students enrolled in such courses. We do not propose a new submission system for automated tracking of student's assignments and projects, but a grading framework that could be a part of such system. The framework is based on merging information from three different evaluation methods:

1. Software verification (automated bug finding)
2. Control flow graph (CFG) similarity measurement
3. Automated testing

We also address the problem of choosing weights for these factors to tune automated grading to teacher's grading style. The synergy between automated testing, verification, and similarity measurement improves the quality and precision of automated grading by overcoming the individual weaknesses of these approaches. Our empirical evaluation shows that our framework can lead to a grading model that highly correlates to manual grading and therefore gives promises for real-world applicability in education.

We also review our tools for software verification [13] and CFG similarity [18], which we use for assignment evaluation. These tools, based on novel

methods, are publicly available and open source.[1] Both tools use the low-level intermediate code representation LLVM [19, 20]. Therefore, they could be applied to a number of programming languages and could be complemented with other existing LLVM based tools (e.g., tools for automated test generation). Also, the tools are enhanced with support for meaningful and comprehensible feedback to students, so they can be used both in the process of studying and in the process of grading assignments.

Overview of the paper. Section 2 presents necessary background information. Section 3 gives motivating examples for the synergy of the three proposed components. Section 4 describes the grading setting and the corpus used for evaluation. Section 5 discusses the role of the verification techniques in automated evaluation and Section 5 discusses the role of structural similarity measurement in automated evaluation. Section 7 presents an empirical evaluation of the proposed framework for automated grading. Section 8 contains information about related work. Section 9 gives conclusions and outlines possible directions for future work.

2. Background

This section provides an overview of intermediate languages, the LLVM tool, software verification, the LAV tool, control flow graphs and graph similarity measurement.

Intermediate languages and LLVM. An intermediate language separates concepts and semantics of a high level programming language from low level issues relevant for a specific machine. Examples of intermediate languages include the ones used in LLVM and .NET frameworks. LLVM is an open source, widely used, rich compiler framework, well suited for developing new mid-level language-independent analyses and optimizations [19, 20]. LLVM intermediate language is assembly-like language with simple RISC-like instructions. It supports easy construction of control flow graphs of program functions and of entire programs. There is a number of tools using LLVM for various purposes, including software verification [8, 9, 12, 21, 13]. LLVM has front-ends for C, C++, Ada and Fortran. Moreover, there are external projects for translating a number of other languages to LLVM intermediate

[1] `http://argo.matf.bg.ac.rs/?content=lav`

representation (e.g., Python [22], Ruby [23], Haskell [24], Java [25], D [26], Pure [27], Scala [28] and Lua [29]).

Software verification and LAV. Verification of software and automated bug finding are among the greatest challenges in computer science. Software bugs cost the world economy billions of dollars annually [30]. Software verification tools aim to automatically check functional correctness properties including the absence of bugs that could make memory violations or raise runtime errors. Different approaches to automated checking of software properties exist, such as symbolic execution [31], model checking [32] and abstract interpretation [33]. Software verification tools often use automated theorem provers as the underlying reasoning machinery.

LAV [13] is an open-source tool for statically verifying program assertions and locating bugs such as buffer overflows, pointer errors and division by zero. LAV uses the popular LLVM infrastructure. As a result, it supports several programming languages that compile into LLVM, and benefits from the robust LLVM front ends. LAV is primarily aimed at programs in the C programming language, in which the opportunities for errors are abundant. For each safety-critical command, LAV generates a first-order logic formula that represents its correctness condition. This formula is checked by one of the several SMT solvers [34] used by LAV. If a command cannot be proved safe, LAV translates a potential counterexample from the solver into a program trace that exhibits this error. LAV also extracts the values of relevant program variables along this trace.

Control flow graph. A control flow graph (CFG) is a graph-based representation of all paths that might be traversed through a program during its execution [35]. Each node of CFG represents one basic block, which is a sequence of commands without jumps, loops or conditional statements. The control flow graphs can be produced by various tools, including LLVM. A control flow graph clearly separates the structure of the program and its contents. Therefore, it is a suitable representation for structural comparison of programs.

Graph similarity and neighbor matching method. There are many similarity measures for graphs and their nodes [36, 37, 38, 18]. These measures have been successfully applied in several practical domains such as ranking of Internet query results, synonym extraction, database structure matching,

construction of phylogenetic trees, and analysis of social networks. A short overview of similarity measures for graphs can be found in [18].

A specific similarity measure for graph nodes called *neighbor matching* has properties relevant for our purpose that other similar measures lack [18]. It allows similarity measure for graphs to be defined based on similarity scores of their individual nodes. The notion of similarity of nodes is based on the intuition that *two nodes i and j of graphs A and B are considered to be similar if neighboring nodes of i can be matched to similar neighboring nodes of j*. More precise definition is the following.

In the neighbor matching method, if a graph contains an edge (i, j), the node i is called an *in-neighbor* of node j in the graph and the node j is called an *out-neighbor* of the node i in the graph. An *in-degree* $id(i)$ of the node i is the number of in-neighbors of i, and an *out-degree* $od(i)$ of the node i is the number of out-neighbors of i.

If A and B are two finite sets of arbitrary elements, a *matching* of elements of sets A and B is a set of pairs $M = \{(i, j)|i \in A, j \in B\}$ such that no element of one set is paired with more than one element of the other set. For the matching M, *enumeration functions* $f : \{1, 2, \ldots k\} \to A$ and $g : \{1, 2, \ldots k\} \to B$ are defined such that $M = \{(f(l), g(l)) \mid l = 1, 2, \ldots, k\}$ where $k = |M|$. If $w(a, b)$ is a function assigning weights to pairs of elements $a \in A$ and $b \in B$, the *weight of a matching* is the sum of weights assigned to the pairs of elements from the matching, i.e. $w(M) = \Sigma_{(i,j)\in M} w(i, j)$. The goal of the *assignment problem* is to find a matching of elements of A and B of the highest weight (if two sets are of different cardinalities, some elements of the larger set will not have corresponding elements in the smaller set). The assignment problem is usually solved by the well-known Hungarian algorithm of complexity $O(mn^2)$ where $m = \max(|A|, |B|)$ and $n = \min(|A|, |B|)$ [39], but there are also more efficient algorithms [40, 41].[2]

The calculation of similarity of nodes i and j, denoted x_{ij}, is based on iterative procedure given by the following equations:

$$x_{ij}^{k+1} \leftarrow \frac{s_{in}^{k+1}(i, j) + s_{out}^{k+1}(i, j)}{2}$$

[2]We are not aware of the available implementations of these more efficient algorithms. However, Hungarian algorithm performs very well in the problem we address (as can be seen from the runtimes given in Section 6.2).

where

$$s_{in}^{k+1}(i,j) \leftarrow \frac{1}{m_{in}} \sum_{l=1}^{n_{in}} x^k_{f_{ij}^{in}(l) g_{ij}^{in}(l)} \qquad s_{out}^{k+1}(i,j) \leftarrow \frac{1}{m_{out}} \sum_{l=1}^{n_{out}} x^k_{f_{ij}^{out}(l) g_{ij}^{out}(l)} \quad (1)$$

$$m_{in} = \max(id(i), id(j)) \qquad m_{out} = \max(od(i), od(j))$$
$$n_{in} = \min(id(i), id(j)) \qquad n_{out} = \min(od(i), od(j))$$

where functions f_{ij}^{in} and g_{ij}^{in} are the enumeration functions of the optimal matching of in-neighbors for nodes i and j with weight function $w(a,b) = x_{ab}^k$, and analogously for f_{ij}^{out} and g_{ij}^{out}. In the equations (1), $\frac{0}{0}$ is defined to be 1 (used in the case when $m_{in} = n_{in} = 0$ or $m_{out} = n_{out} = 0$). The initial similarity values x_{ij}^0 are set to 1 for each i and j. The termination condition is $\max_{ij} |x_{ij}^k - x_{ij}^{k-1}| < \varepsilon$ for some chosen precision ε and the iterative algorithm has been proved to converge [18].

The similarity matrix $[x_{ij}]$ reflects the similarities of nodes of two graphs A and B. The similarity of the graphs can be defined as the weight of the optimal matching of nodes from A and B divided by the number of matched nodes [18].

3. The Need for Synergy of Testing, Verification, and Similarity Measurement

In this section we elaborate on the need for synergy of testing, verification and similarity measurement and give motivating examples to illustrate shortcomings if some of these components is omitted.

Automated testing of programs plays an important role in the evaluation of students programs. However, the grading in this approach is directly influenced by the choice of test cases. Whether the test cases are automatically generated or manually designed, testing cannot guarantee functional correctness of a program or the absence of bugs.

For checking functional correctness, a combination of random testing with evaluator-supplied test cases is a common choice [42]. Randomly generated test cases can detect most shallow bugs very efficiently, but for bugs that are located in more convoluted paths, random tests may not succeed [43, 44]. It is not sufficient that test cases cover all important paths through the program. It is also important to carefully choose values of the variables for each path — for some values along the same path a bug can be detected, while for some other values the bug can stay undetected.

Manually generated test cases are designed according to the expected solutions, while the evaluator cannot predict all the important paths through a student's solution. Even running a test case that hits a certain bug (for example, a buffer overflow bug in a C program) does not necessarily lead to any visible undesired behavior if the running is done in a normal (or sandbox) environment. Finally, if one manages to trigger a bug by a test case, if the bug produces the *Segmentation fault* message, it is not a feedback that a student (especially novice in programming) can easily understand and use for debugging the program. In the context of automated grading, this feedback cannot be easily used since it may have different causes. In contrast to program testing, software verification tools like Pex [7], Klee [8], S2E [9], CBMC [10], ESBMC [11], LLBMC [12], and LAV [13] can give much better explanations (e.g., the kind of bug and the program trace that introduces an error).

```
0:  #define max_size 50
1:  void matrix_maximum(int a[][max_size], int rows, int columns, int b[])
2:  {
3:      int i, j, max=a[0][0];                int i, j, max;
4:      for(i=0; i<rows; i++)                 for(i=0; i<rows; i++)
5:      {                                     {
6:                                                max = a[i][0];
7:          for(j=0; j<columns; j++)              for(j=0; j<columns; j++)
8:              if(max < a[i][j])                     if(max < a[i][j])
9:                 max = a[i][j];                        max = a[i][j];
10:         b[i] = max;                           b[i] = max;
11:         max=a[i+1][0];
12:     }                                     }
13:     return;                               return;
14:  }
```

Figure 1: Buffer overflow in the code on left-hand side (which computes maximum values of each row in a matrix) cannot be discovered by simple testing or detected by code similarity. Functionally equivalent solution without a memory violation bug is given on right-hand side.

The example function shown at Figure 1 (left) is extracted from a student's code written on an exam. It calculates the maximum value of each row of a matrix and writes these values into an array. This function is used in a context where the memory for the matrix is statically allocated and numbers of rows and columns are less or equal to the allocated sizes of the matrix.

However, in the line 11, there is a possible buffer overflow bug, since $i+1$ can exceed the allocated number of rows for the matrix. It is possible that this kind of a bug does not affect the output of the program or destroy any data, but in only a slightly different context it can be harmful, so students should be warned and the points should not be awarded in such situations. Still, the corrected version of this function, given in Figure 1 (right) is very similar to the incorrect student's solution. Such bugs can be missed in testing, cannot be detected by code similarity, but are easily discovered by verification tools like LAV.

Functional correctness and absence of bugs are not the only important aspects of students' programs. Programs are often supposed to meet requirements concerning the structure, such as modularity (adequate decomposition of code to functions) or simplicity. Figure 2 shows fragments of two student solutions of different modularity and structure for two problems. Neither testing, nor software verification can be used to assess these aspects of the programs. This problem can be addressed by checking the similarity of student's solution with a teacher-provided solution, i.e., by analyzing the similarity of their control-flow graphs [14, 15, 18].[3]

Finally, using similarity only (like in [14, 15]) or even similarity with support of a bug finding tool, could fail to detect incorrectness of program's behavior. Figure 3 gives a simple example program that computes the maximum of a sequence and that is extracted from a student's solution. This program is very similar to the expected solution and has no memory violations or runtime errors. However, this program is not functionally correct and this can be easily discovered by testing.

Based on considerations and examples given above, we conclude that the synergy of these three approaches is needed for sophisticated evaluation of students' assignments.

4. Grading Setting

There may be different grading settings depending on aims of the course and goals of the teacher. The setting used at an introductory course of programming in C (at University of Belgrade) is rather standard: taking exams

[3]In Figure 2, the second example could also be distinguished by profiling for large inputs, because it is quadratic in one case and linear in the other. However, profiling cannot be used to assess structural properties in general.

Problem	First solution	Second solution

1.	`if(a<b) n = a;`
	`else n = b;`
	`if(c<d) m = c;`
	`else m = d;`
		`n = min(a, b);`
		
		`m = min(c, d);`

2.	`for(i=0; i<n; i++)`
	`  for(j=0; j<n; j++)`
	`  if(i==j)`
	`  m[i][j] = 1;`
		`for(i=0; i<n; i++)`
		`  m[i][i] = 1;`

Figure 2: Examples extracted from two student solutions of the same problem, illustrating structural differences that can be addressed by CFG similarity measurement.

```
max = 0;                        max = a[0];
for(i=0; i<n; i++)              for(i=1; i<n; i++)
   if(a[i] > max)                  if(a[i] > max)
       max = a[i];                     max = a[i];
```

Figure 3: Code extracted from a student's solution (left-hand side) and an expected solution (right-hand side). In the student's solution there are no verification bugs, it is very similar to the expected solution but it does not perform the desired behavior (in the case when all elements of the array `a` are negative integers). This defect can be easily discovered by testing.

on computers and expecting from students to write working programs. In order to help students achieve this goal, each assignment is provided with several test cases that illustrate the desired behavior of a solution. Students are provided with sufficient (but limited) time for developing and testing programs. If a student fails to provide a working program that gives correct results for the given test cases, his/her solution is not further examined. Otherwise, the program is tested by additional test cases (unknown to the students and generated by the instructors i.e. by the teachers or the teaching assistants) and a certain amount of points is given proportional to the test cases successfully passed. Only if all these test cases pass successfully, the program is further manually examined and additional points may be given with respect to other features of the program (efficiency, modularity,

simplicity, absence of memory violations, etc).

For empirical evaluations presented in the rest of this paper, we used a corpus of programs written by students on the exams, following the described grading setting. The corpus consists of 266 solutions to 15 different problems. These problems include numerical calculations, manipulations with arrays and matrices, manipulations with strings, and manipulations with data structures.[4] Only programs that passed all test cases were included in this corpus. These programs are the main target of our automated evaluation technique since the manual grading was applied only in this case and we want to explore potentials for completely eliminating manual grading. These programs obtained 80% of the maximal score (as they passed all test cases) and additional 20% were given after manual inspection. The grades are expressed at the scale from 0 to 10. The corpus, together with problem descriptions and the assigned grades, is publicly available.[5]

The automated grading approach we propose is flexible and can also be applied to different grading settings, i.e., to different distributions of grade weights that are awarded for different aspects of program quality (as discussed in Section 7).

5. Assignment Evaluation and Software Verification

In this section we discuss benefits of using software verification tool in assignment evaluation, e.g., for generating useful feedback for students and providing improved assignment evaluation for teachers.

5.1. Software Verification for Assignment Evaluation

No software verification tool can report all the bugs in a program without introducing *false positives* (due to the undecidability of the halting problem). False positives (i.e., reported "bugs" that are not real bugs) arise as a consequence of approximations that are necessary in modeling programs.

The most important approximation is concerned with dealing with loops. Different verification approaches use various techniques for dealing with loops. These techniques range from under-approximations of loops to over-approximations of loops and influence the efficiency of analysis. Under-approximation of loops, as in the bounded model checking techniques [32],

[4]Short descriptions of the problems are given in Appendix A.

[5]`http://argo.matf.bg.ac.rs/?content=lav`

uses a fixed number n for loop unwinding. In this case, if the code is verified successfully, it means that the original code has no bugs for n or less passes through the loop. However, it may happen that some bug remains undiscovered if the unwinding is performed an insufficient number of times. This technique does not introduce false positives, but also does not scale well on large programs or on programs where a big number of unwindings is necessary. Over-approximation of loops can be made by simulation of first n and last m passes through the loop [13] or by using abstract interpretation techniques [33]. If there are no bugs detected in the over-approximated code, then the original code has no bugs too. However, in this case, a false positive can appear after or inside a loop. These techniques scale well on larger programs but with a price of introducing false positives. On the other hand, a completely precise dealing with loops, like in the symbolic execution techniques, can be non terminating. Therefore, for educational purposes, an appropriate trade-off between efficiency and precision should be carefully chosen.

False positives are highly undesirable in software development, but still are not critical — the developer can fix the problem or confirm that the reported problem is not really a bug (and both of these are situations that the developer can expect and understand). However, false positives in assignment evaluation are rather critical and have to be eliminated. For teachers, there should be no false positives, because the evaluation process should be as automatic and reliable as possible. For students, especially for novice programmers, there should be no false positives because they would be confused if told that something is a bug when it is not. In order to eliminate false positives, a system may be non-terminating or may miss to report some real bugs. In assignment evaluation, the second choice is more reasonable — the tool has to be terminating, must not introduce false positives, even if the price is missing some real bugs. These requirements make applications of software verification in education rather specific, and special care has to be taken when these techniques are applied.

Despite the progress in software verification technology, verification tools can still take more time than it is adequate for a comfortable interactive work. Because of that, in real-world applications in education, time-outs have to be used. There could be different policies for time-outs. For instance, if the verification tool reached the time limit, no bug would be reported (in order to avoid reporting false positives) or a program can be checked using the same parameters, but with another underlying solver (if applicable for the

tool). Generally, there could be two time limits: a higher time limit for the teacher, when doing off-line grading, and a lower time limit for interactive work of students.

5.2. LAV for Assignment Evaluation

LAV is a general purpose verification tool and has a number of options that can adapt its behavior to the desired context. When running LAV in the assignment evaluation context, most of these options (e.g., the underlying SMT solver and the corresponding theory) can be fixed to default values.

The most important choice for the user is the choice of the way in which LAV deals with loops. LAV has support for both over-approximation of loops and for fixed number of unwinding of loops (under-approximation), two common techniques for dealing with loops (which are rarely together present in a same tool). Setting up the upper loop bound (if under-approximation is used), is problem dependent and should be done by the teacher for each assignment.

We use LAV in the following way. LAV is first invoked with its default parameters — over-approximation of loops. This technique is efficient, but it can introduce false positives. Therefore, if a potential bug is found after or inside a loop, the verification is invoked again but this time with fixed unwinding parameter. If the bug is still present, then it is reported. Otherwise, the previously detected potential bug is considered to be a false positive and it is not reported.

In industry, each bug detected by software verification is important and should be reported. However, some bugs can confuse novice programmers, like the one shown in Figure 4. In this code, at line `11`, there is a subtle possible buffer overflow. For instance, for $n = 0x80000001$ only 4 bytes will be allocated for the pointer `array`, because of an integer overflow. This is a verification error, that LAV will normally report, but a teacher may decide not to consider this kind of bugs. For this purpose, LAV can be invoked in the mode for students (so the bugs like this one, involving an integer overflow in memory allocation, are not reported). In the student mode, also, hints for discovered errors are always reported.

To a limited extent, LAV was already used on students' assignments at an introductory programming course [13]. The corpus consisted of 157 programs with the average number of lines 42 and it included both correct and incorrect solutions. LAV ran only with its default parameters (giving some false positives) and it discovered 423 genuine bugs in 121 programs. Possible

```
1:  unsigned i, n;
2:  unsigned *arr;
3:  scanf("%u", &n);
4:  array = malloc(n*sizeof(unsigned));
5:  if(array == NULL)
6:  {
7:      fprintf(stderr, "Unsuccessful allocation\n");
8:      exit(EXIT_FAILURE);
9:  }
10: for(i=0; i<n; i++)
11:    array[i] = i;
```

Figure 4: Buffer overflow in this code is a verification error, but the teacher may decide not to consider this kind of bugs.

buffer overflows were the most frequent bugs found in this corpus (240 bugs in 111 programs). The vast majority of bugs (90%), were made following wrong expectations — for instance, expectations that input parameters of the program will meet certain constraints (71%), that the program will always be invoked with appropriate number of command line arguments (10%), and that memory allocation will always succeed (8%). It was also noticed that a single oversight was often responsible for several bugs — in 73% of programs with bugs, omission of a necessary check produced two to ten bugs in the rest of the program. For example, omission of a check of a number of command line arguments introduced two to three buffer overflow errors per solution (at each place where command line arguments were used). Another example is omission of a check whether a memory allocation succeeded — this one oversight led to a possible null pointer dereferencing error at each point where the pointer was used and introduced four to ten reported bugs per solution. Therefore, the number of bugs, as reported by a verification tool, is not a reliable indicator of an overall program quality. This property should be taken into account in automated grading.

5.3. Empirical Evaluation

As discussed in Section 3, programs that successfully pass a testing phase can still contain bugs. To show that this problem is practically important, we used LAV to analyze programs from the corpus described in Section 4.

For each problem, LAV ran with its default parameters, and programs with potential bugs were checked with under-approximation of loops, as de-

scribed in Section 5.2.[6] The results are shown in Table 1. On average, on a system with Intel processor i7 with 8 GB of RAM memory, running Ubuntu, LAV spent 2.8s for analyzing one program.

LAV discovered bugs in 35 solutions that successfully passed manually designed test cases (following the grading setting described in Section 4). There was one false negative of manual inspection (the bug was detected by LAV) and one false negative of LAV (the bug was detected by manual inspection). The false negative of manual inspection was the bug described in Section 3 and given in Figure 1. The false negative of LAV was a consequence of the problem formulation which was too general to allow a precise unique upper loop unwinding parameter value for all possible solutions. There were just two false positives produced by LAV when the default parameters were used. These false positives were eliminated when the tool was invoked for the second time with a specified loop unwinding parameter, and hence there were no false positives in the final outputs. In summary, the presented results show that a verification tool like LAV can be used as a complement to automated testing that improves the evaluation process.

5.4. Feedback for Students and Teachers

LAV can be used to provide meaningful and comprehensible feedback to students while they develop their programs. Generated feedback follows directly from the detected counterexamples, which, further, follow directly from the way LAV operates. So, the cost of feedback generation is very low. Information such as the line number, the kind of the error, program trace that introduces the error, and values of variables along this trace can help the student improve the solution. This feedback can also remind the student to add an appropriate check that is missing. The example given in Figure 5, extracted from a student's code written on an exam, shows the error detected by LAV and the generated hint.

From a software verification tool, a teacher can obtain the information if the student's program contains a bug. The teacher can use this informa-

[6]When analyzing solutions of three problems (3, 5 and 8), only under-approximation of loops was used. This was the consequence of the formulation of the problems given to the students. Namely, the formulation of these problems contained some assumptions on input parameters. These assumptions implied that some potential bugs should not be considered (because these are not bugs when these additional assumptions are taken into account).

problem	solutions	average number of lines	programs with bugs by manual inspection	programs with bugs by LAV	bug-free programs by LAV		false positives by LAV	
					def.	custom	def.	custom
1.	44	29	0	0	44	-	0	-
2.	32	55	11	11	20	1	1	0
3.	7	30	2	2	-	5	-	0
4.	5	43	0	1	3	1	1	0
5.	12	39	3	2	-	10	-	0
6.	7	35	0	0	6	1	1	0
7.	33	14	0	0	33	-	0	-
8.	31	29	11	11	-	20	-	0
9.	10	83	6	6	4	0	0	0
10.	14	36	2	2	12	0	0	0
11.	31	13	0	0	31	-	0	-
12.	18	16	0	0	18	-	0	-
13.	3	20	0	0	3	-	0	-
14.	7	28	0	0	7	-	0	-
15.	12	21	0	0	12	-	0	-
total	266	30	35	35	193	38	2	0

Table 1: Summary of bugs in the corpus: the second column represents the number of students' solutions to the given problem; the third column represents the average number of lines per solution; the fourth and the fifth column represent the number of solutions with bugs detected by manual inspection and by LAV; the sixth column gives the number of programs shown to be bug-free by LAV using over-approximation of loops (default parameters) and, when necessary, using under-approximation of loops (custom parameters); the seventh column gives the number of false positives made by LAV invoked with default parameters and, if applicable, with custom parameters.

tion in grading assignments. Alternatively, this information can be taken into account within wider integrated framework for obtaining automatically proposed final grade discussed in Section 7.

5.5. Limitations of Random Testing Compared to Software Verification

Different sorts of test cases can be used in evaluation of students' assessments: manually designed test cases and test cases automatically generated by static or dynamic analysis, or some combination of these. As already said, manually designed test cases check if the code exhibits desired behavior on a range of selected inputs and are usually used in assessment of students' programs. Tools that generate test cases based on static analysis use software verification techniques that we also use. Dynamic analysis tools based on random testing are often used in educational context [42] and they target similar classes of programming defects as verification tools. Therefore, we

```
1:  #include<stdio.h>
2:  #include<stdlib.h>
3:  int get_digit(int n, int d);
4:  int main(int argc, char** argv)
5:  {
6:      int n, d;
7:      n = atoi(argv[1]);
8:      d = atoi(argv[2]);
9:      printf("%d\n", get_digit(n, d));
10:     return 0;
11: }
```

```
verification failed:
line 7: UNSAFE

function: main
error: buffer_overflow
in line 7: counterexample:
argc == 1, argv == 1

HINT:
A buffer overflow error occurs when
trying to read or write outside the
reserved memory for a buffer/array.
Check the boundaries of the array!
```

Figure 5: Listing extracted from student's code written on an exam (left-hand side) and LAV's output (right-hand side)

briefly discuss if verification tools can add new value to what random testing tools can provide in this context. For illustration, we confront the described verification tool LAV with one random testing tool, on the corpus described in Section 4.

Dynamic analysis of code based on random testing, also known as *fuzzing* or *fuzz testing*, is a black-box technique which attempts to discover security vulnerabilities by sending random inputs to a program [45]. It is used to detect vulnerabilities that can cause the program to crash, hang or lead to an exception (and cannot be used to detect other kinds of vulnerabilities). The main challenge for random testing tools is to achieve high code coverage and there are different strategies for accomplishing this goal. These tools are widely used [46], but they still have some weaknesses. For instance, they can generate many test cases that hit the same bug, but following different paths through the program. Although fuzzers execute program to be examined, they can still introduce false positives, for example, by concluding that the program is stuck, although it only waits for an input. There are some variations of fuzzing techniques and tools [47, 48, 49].

Bunny [50] is an open source, high-performance, general purpose protocol-blind black-box fuzzer for C programs. It injects instrumentation hooks into the traced program which allow it to receive real-time feedback on trace changes caused by variations on input data. This further allows getting high code coverage of the testing process.

We ran Bunny on the corpus described in Section 4 and then manually analyzed all the test cases it produced. LAV found 35 bugs in the corpus, while test cases generated by Bunny point to 12 of these bugs. Bunny did not discover 16 bugs (including one given in Figure 1) that result in buffer overflows (usually off-by-one errors). Since these bugs do not cause programs to crash, note that they cannot be discovered by other black-box fuzzers too.[7] The remaining 7 bugs are not shallow and Bunny did not manage to produce test cases that would trigger these bugs, which is a common problem with this sort of tools [44]. On the other hand, for all 49 solutions of the problems 3, 5 and 8, Bunny produced test cases which trigger bugs not relevant in this context due to assumptions given in the problems' formulations (but that could not be handled by Bunny, because it is a protocol-blind fuzzer). For another 31 programs from the corpus, Bunny generated false positives. For most of them (for 26 programs) Bunny reported they got stalled while they were only printing large amounts of correct data. Other false positives (for 5 programs) are not valid input data (for instance, the first input number determines the number of the input data that the program should read, while the test case does not supply enough input data). As for software verification tools, the number of test cases generated by a tool such as Bunny cannot be used for measuring the quality of the program examined because: (i) the numbers of generated test cases can drastically differ for programs representing solutions of the same problem and containing same bugs (for example, these numbers vary from 1 to 228 for solutions of one problem from the corpus); (ii) many test cases generated for one program may hit the same bug; (iii) some test cases can be false positives. Taking only the first generated test case (as we use only the first bug that LAV reports) is not a good solution neither since the first test case may be false positive. Finally, for analyzing the corpus, Bunny took significantly more time than LAV.

We believe that we would obtain similar results if some other black-box fuzzer was used instead of Bunny, since black-box fuzzers do not detect bugs that do not cause crashes and are not good in finding bugs that are not shallow. Therefore, we conclude that random testing can complement and

[7] In order to find bugs like these, it is necessary to use techniques that precisely track memory contents. For example, tools CRED [51], CCured [52] and Valgrind [53] do a detailed instrumentation of C code in order to detect buffer overflows. However, these tools introduce significant overhead on testing performance [54, 51] and do not generate test cases automatically.

improve manual testing in educational context, but it still has weaknesses (concerning missed bugs, false positives and time efficiency) compared to appropriate tools that use static analysis of code, including verification tools.

6. Assignment Evaluation and Structural Similarity of Programs

In this section we propose a similarity measure for programs based on their control flow graphs, perform its empirical evaluation, and point to ways it can be used to provide feedback for students and teachers.

6.1. Similarity of CFGs for Assignment Evaluation

To evaluate structural properties of programs, we take the approach of comparing students' programs to solutions provided by the teacher. A student's program is considered to be good if it is similar to some of the programs provided by the teacher [55, 14, 15]. This assumption cannot be made for large size students' projects where may be many different ways of solving a problem that cannot be predicted in advance. However, for programs written within introductory programming courses, there are no many sensible but substantially different solutions (or with substantially different structure). Innovative and good solutions are always possible, but in this context are rare. Therefore, the real risk is that students produce programs more complex than needed and our system aims at detecting this. This assumption is reasonable and is justified by good results of empirical evaluation given in Section 7.

In order to perform a comparison, a suitable program representation and a similarity measure are needed. As already discussed in Section 2, our system generates a control flow graph (CFG) corresponding to each program. The CFG reflects the structure of the program. Also, there is a linear code sequence attributed to each node of the CFG which we call the node content. We assume that the code is in the intermediate LLVM language [19, 20]. In order to measure similarity of programs, both the similarity of graph structures and the similarity of node contents should be considered. We take the approach of combining the similarity of node contents with topological similarity of graph nodes described in Section 2.

Similarity of node contents. The node content is a sequence of LLVM instructions. A simple way of measuring the similarity of two sequences of instructions s_1 and s_2 is using the edit distance between them $d(s_1, s_2)$ —

the minimal number of insertion, deletion and substitution operations over the elements of the sequence by which one sequence can be transformed into another [56]. In order for edit distance to be computed, the cost of each insertion, deletion and substitution operation has to be defined. We define the cost of insertion and deletion of an instruction to be 1. Next, we define the cost of substitution of instruction i_1 by instruction i_2. Let *opcode* be a function that maps an instruction to its opcode (a part of instruction that specifies the operation to be performed). Let $opcode(i_1)$ and $opcode(i_2)$ be function calls. Then, the cost of substitution is 1 if i_1 and i_2 call different functions, and 0 if they call the same function. If either $opcode(i_1)$ or $opcode(i_2)$ is not a function call, the cost of substitution is 1 if $opcode(i_1) \neq opcode(i_2)$, and 0 otherwise. Let $n_1 = |s_1|$, $n_2 = |s_2|$, and let M be the maximal edit distance over two sequences of length n_1 and n_2. Then, the similarity of sequences s_1 and s_2 is defined as $1 - d(s_1, s_2)/M$.

Although it could be argued that the proposed similarity measure is rough since it does not account for differences of instruction arguments, it is simple, easily implemented, and intuitive.

Full similarity of nodes and similarity of CFGs. The topological similarity of nodes can be computed by the method described in Section 2. However, purely topological similarity does not account for differences of the node content. Hence, we modify the computation of topological similarity to include the apriori similarity of nodes. The modified update rule is:

$$x_{ij}^{k+1} \leftarrow \sqrt{y_{ij} \cdot \frac{s_{in}^{k+1}(i,j) + s_{out}^{k+1}(i,j)}{2}}$$

where y_{ij} are the similarities of contents of nodes i and j and $s_{in}^{k+1}(i,j)$ and $s_{out}^{k+1}(i,j)$ are defined by Equations 1. Also, we set $x_{ij}^0 = y_{ij}$. This way, both content similarity and topological similarity of nodes are taken into account. The similarity of CFGs can be defined based on the node similarity matrix as described in Section 2. Note that both the similarity of nodes and the similarity of CFGs take values in the interval $[0, 1]$.

It should be noted that our approach provides both the similarity measure for CFGs and the similarity measure for their nodes (x_{ij}). In addition to evaluating similarity of programs, this approach enables matching of related parts of the programs by matching the most similar nodes of CFGs. This could serve as a basis of a method for suggesting which parts of the student's program could be further improved.

6.2. Empirical Evaluation

In order to show that the proposed program similarity measure corresponds to some intuitive notion of program similarity, we performed the following evaluation. For each program from the corpus described in Section 4, we found the most similar program from the rest of the corpus and counted how often these programs are the solutions for the same problem. That was the case for 90% of all programs. This shows that our similarity measure performs well, since with high probability, for each program, the program that is the most similar to it, corresponds to the same problem. The inspection suggests that in most cases, where the programs do not correspond to the same problem, student took an innovative approach for solving the problem.

The average size of CFGs of the programs from the corpus is 15 nodes. The average similarity computation time was 0.12s (on a system with Intel processor i7 with 8 GB of RAM memory, running Ubuntu).

6.3. Feedback for Students and Teachers

The students can benefit from program similarity evaluation while learning and exercising, assuming that the teacher provided a valid solution or a set of solutions to the evaluation system. In introductory programming courses, most often a student's solution can be considered as better if it is more similar to one of the teacher's solutions, as discussed in Section 6.1. In Section 7 we show that the similarity measure can be used for automatic calculation of a grade (a feedback that students easily understand). Moreover, we show that there is a significant linear dependence of the grade on the similarity value. Due to that linearity, the similarity value can be considered as an intuitive feedback, but also it can be translated into descriptive estimate. For example, the feedback could be that the solution is dissimilar (0-0.5), roughly similar (0.5-0.7), similar (0.7-0.9) or very similar (0.9-1) to one of the desired solutions.

Teachers can use the similarity information in automated grading, as discussed in Section 7.

7. Automated Grading

In this section we explore the potential of automated grading based on the synergy of the evaluation techniques that have been discussed so far. For this, relatively simple correlational study suffices. We train a prediction model based on a set of instructor graded solutions, and then check the

correlation of the model's predictions with instructor-provided grades on a separate set of solutions to different assignment problems. We also discuss the threats to validity of our research.

7.1. Predictive Model and Its Evaluation

We believe that automated grading can be performed by calculating a linear combination of different scores measured for the student's solution. We propose a linear model for prediction of the teacher-provided grade of the following form:

$$\hat{y} = \alpha_1 \cdot x_1 + \alpha_2 \cdot x_2 + \alpha_3 \cdot x_3$$

where

- $\hat{y}$ is the automatically predicted grade,
- x_1 is a result obtained by automated testing expressed in the interval $[0, 1]$,
- x_2 is 1 if the student's solution is reported to be correct by the software verification tool, and 0 otherwise,
- x_3 is the maximal value of similarity between the student's solution and each of the teacher provided solutions (its range is $[0, 1]$).

It should be noted that we do not use bug count as a parameter, as discussed in Section 5.2. Different choices for the coefficients α_i, for $i = 1, 2, 3$ could be proposed. In our case, one simple way could be $\alpha_1 = 8$, $\alpha_2 = 1$, and $\alpha_3 = 1$ since all programs in our training set won 80% of the full grade due to the success in the testing phase. However, it is not always clear how the teacher's intuitive grading criterion can be factored to automatically measurable quantities. Teachers need not have the intuitive feeling for all the variables involved in the grading. For instance, the behavior of any of the proposed similarity measures including ours [14, 15, 18] is not clear from their definitions only. So, it may be unclear how to choose weights for different variables when combining them in the final grade or if some of the variables should be nonlinearly transformed in order to be useful for grading. A natural solution is to try to tune the coefficients α_i, for $i = 1, 2, 3$, so that the behavior of the predictive model corresponds to the teacher's grading style. For that purpose, coefficients can be determined automatically using

least squares linear regression [57] if a manually graded corpus of students' programs is provided by the teacher.

In our evaluation, the corpus of programs was split into a training and a test set where the training set consisted of two thirds of the corpus and the test set consisted of one third of the corpus. The training set contained solutions of eight different problems and the test set contained solutions of remaining seven problems. Both the training and the test set were graded by one of the instructors.

Due to the nature of the corpus, for all the instances it holds $x_1 = 1$. Therefore, while it is clear that the percentage of test cases the program passed (x_1) is useful in automated grading, this variable cannot be analyzed based on this corpus.

The optimal values of coefficients α_i, $i = 1, 2, 3$, with respect to the training corpus, are determined using the least squares linear regression. The obtained equation is

$$\hat{y} = 6.058 \cdot x_1 + 1.014 \cdot x_2 + 2.919 \cdot x_3$$

The formula for $\hat{y}$ may seem counterintuitive. Since the minimal grade in the corpus is 8 and $x_1 = 1$ for all instances, one would expect that it holds $\alpha_1 \approx 8$. The discrepancy is due to the fact that for the solutions in the corpus, the minimal value for x_3 is 0.68 — since all the solutions are relatively good (they all passed the testing) there are no programs with low similarity value. Taking this into consideration, one can rewrite the formula for $\hat{y}$ as

$$\hat{y} = 8.043 \cdot x_1 + 1.014 \cdot x_2 + 0.934 \cdot x_3'$$

where $x_3' = \frac{x_3 - 0.68}{1 - 0.68}$, so the variable x_3' takes values from the interval $[0, 1]$. This means that when the range of variability of both x_2 and x_3 is scaled to the interval $[0, 1]$, their contribution to the mark is rather similar.

Since our goal is to confirm that the combination of evaluation techniques tuned (on a training set) to the instructor grading style is superior to the use of individual techniques or some reasonable first-guess combination with predetermined parameters, we compare the correlations between instructor-provided grades and the grades provided by each of these approaches. Table 2 shows the comparison between the model $\hat{y}$ and three other models. The model $\hat{y}_1 = 8 \cdot x_1 + x_2 + x_3$ has predetermined parameters, the model $\hat{y}_2$ is trained just with verification information x_2 (without similarity measure),

	r	$r^2 \cdot 100\%$	p-value	Rel. error
$\hat{y}$ (all/adaptable)	0.842	71%	<0.001	10.1%
$\hat{y}_1$ (all/predetermined)	0.730	53.3%	<0.001	12.8%
$\hat{y}_2$ (no similarity/adaptable)	0.620	38.4%	<0.001	16.7%
$\hat{y}_3$ (no verification/adaptable)	0.457	20.9%	<0.001	17.7%

Table 2: The performance of the predictive model on the training and test set. For each model we specify if it takes into account all proposed variables or not and if the coefficients were predetermined or adaptable. We provide correlation coefficient (r), the fraction of variance of y accounted by the model ($100 \cdot r^2$), p-value as an indicator of statistical significance, and relative error — average error divided by the length of the range in which the grades vary (which is 8 to 10 in the case of this particular corpus).

and the model $\hat{y}_3$ is trained only with similarity measure x_3 (without verification information). The results show that the performance of the model $\hat{y}$ on the test set (consisting of the problems not appearing in the training set) is excellent — the correlation is 0.842 and the model accounts for 71% of the variability of the instructor-provided grade. The statistical significance of our results is confirmed by statistical test against the null hypothesis that there is no correlation between predicted and teacher provided grades. It yielded very small p-values which shows that our results are statistically very significant. These results indicate a strong and reliable dependence between the instructor-provided grade and the variables x_i, meaning that a grade can be reliably predicted by $\hat{y}$. Also, $\hat{y}$ is much better than other models. This shows that the approach using both verification information and graph similarity information is superior to approaches using only one source of information, and also that automated tuning of coefficients of the model provides better prediction than giving them in advance.

7.2. Threats to Validity

Internal, external, and construct validity are usually used in the analysis of the experiments that confirm causal relationships between the considered variables [58]. Although we performed a correlational study, it is worth performing such analysis in our context, too.

Internal validity analysis is concerned with the extent to which the relationship between the independent variables and the dependent variable is established. For the predictive task we are addressing, the relationship that needs to be established is a correlational one. This relationship is clearly

established by the large correlation coefficient between the teacher provided grades and the predictions based on the proposed predictors.

Since our study is correlational and predictive, but not experimental, we cannot make claims about the factors that influence human grading and claims about causality between the variables we consider. While the explanation that the human program grading is to large extent led by the presence of bugs in the program and its similarity with the model solution may seem reasonable, we do not claim that our results corroborate that. It is possible that there are confounding variables that are highly correlated both with the independent variables and with the dependent variable. To analyze such hypothesis one would need proper experimental evaluation.

External validity analysis is concerned with the possibility of generalizing the relationship between the independent variables and the dependent variable to other situations. To avoid threats related to external validity, we restricted our approach to automated grading in introductory programming courses. Also, we performed testing on a different set of problems compared to the ones the training was performed on. Therefore, we expect that our approach generalizes well to other contexts, as long as the programs involved are not complex, as were the ones we used in the evaluation.

Construct validity analysis is concerned with the extent to which a measure correlates to the relevant theoretical concept. In our case, the relevant theoretical concept is the quality of students' solutions. Construct validity can be established by examining correlation of the measure with other measures related to the same concept. High correlation coefficient (0.842) between the automatically provided grades and the teacher provided grades testifies that these measures reflect tightly related concepts. A portion of variability (29%) of automated grades in our evaluation could not be related to the variability of teacher provided grades. Inspection of cases that yielded the biggest errors in the prediction suggests that the greatest source of discrepancy between automatically provided and teacher provided grades are the innovative solutions given by students and the solutions not predicted in advance by the teacher. Although such cases are not very frequent, they are still possible in our approach.

The construct validity could be questioned for teacher provided grades, too, especially since it is clear that different teachers can grade the same solutions differently. However, our approach is based on adapting to the grading style of a teacher and therefore depends on the quality of teacher's judgement.

8. Related Work

In this Section we discuss different approaches and tools for evaluation of student programs. We first briefly comment on manual grading, then give a short overview of tools based on automated testing (testing is used in the preliminary phase of our approach), and then of tools that assess design and employ verification techniques.

We did not use the tools described below for assessing solutions from our corpus and empirical comparison with our approach because: (i) some of the tools employ only automated testing, so comparison would not be fair, or even would not make much sense; (ii) most of the tools are not available, and even fewer are open-source. Overall, we are not aware of any tool that is publicly available, not based only on automated testing, and that can automatically produce grades, so it could be used for a reasonable empirical comparison with our approach.

8.1. Manual Grading

Over the decades of teaching programming languages, there have been a wide range of teaching and grading approaches. In the grading approaches, one of the key issues, often not formalized but followed only intuitively is assigning weights to certain aspects of a student solution (e.g. efficiency, design, correctness, style). For instance, Howatt proposes a grading system in which 25% of a maximal grade goes to the design of a program, 20% goes to program execution, and 20% goes to specification satisfactions [59]. These distributions are often individual or course dependent, so it would be very useful if a system can adopt grading style of a specific user. Our system allows using given weights for certain program aspects but also allows computing the user specific weights on the basis of a given manually graded examples.

8.2. Tools Based on Automated Testing

Automated testing is the most common way of evaluating students' programs [5]. In this context, test cases are usually supplied by a teacher and/or randomly generated [42]. A number of systems use this approach (for various programming languages), for example, PSGE [60], Automark [61] (fortran 77), Kassandra [62] (Maple or Matlab code in a scientific computing course), Schemerobo [63] (functional programming within Scheme), TRY [64] (Pascal), HoGG [65] (Java), BAGS [66], JEWL [67] (automated assessment of GUI-based programs in JAVA), and JUnit [68] (a unit testing framework for

Java). One of the drawbacks is that if a student's program does not produce the desired output in the expected format, a system may fail to give an appropriate mark. All of the above tools employ only automated testing and do not take into account the design of the program and the algorithm used (unlike our system that take these issues into account, implicitly, via similarity measures). Because of its limitations, for better performance, automated testing can be combined with manual inspection or other automated techniques.

Automated testing is used as a component of a number of web-based submission and evaluation systems. Some of them are Online Judge, a system for testing programs in programming contests [69], WebToTeach, a commercial tool which is designed to support a wide variety of programming exercises [70], Quiver, a server for building, maintaining, and administering programming quizzes [71], Praktomat, a system that allows students to read, review, and assess each others programs in order to improve quality and style [72], Web-CAT, a system that encourage students to write their own test cases [73] in order to experience test-first programming and understand the influence of testing on overall software quality [74], Marmoset, a project submission tool with support for automated testing and for collecting code snapshots [75]. In the new, open-source system Marmoset, the feedback that students receive is based on results obtained by testing, while final grading is done by the instructor (after the project's deadline). In contrast, our system aims at automated grading which would make the final grade immediately available to students (which is essential for interactive studying). Marmoset is a framework that deals with projects of different sizes and complexity, while our system focuses on small sized problems typical for introductory programming courses.

There are also course management tools that support instructor's grading by using automated testing, like Assyst [76], BOSS [77], CourseMarker [78] and GAME [79]. BOSS, CourseMarker, and GAME, within the grading mechanism, use efficiently calculated, general metrics that assess quality and style of the examined program. However, these metrics typically cannot assess the way the problem is solved, i.e. the design of the solution and the algorithm used [14].

Our current system is not an integrated framework for submission and testing or for course management. Instead, it should be useful component in evaluation parts of such systems.

8.3. Tools Assessing the Design of Student's Solution

To address the design of a solution, it is necessary to compare student's solution to some predefined solutions. The PASS tool [55], aimed for assessing C programs, evaluates the design of students' solutions by comparing it with a *solution plan* provided by an instructor. A solution plan of the student's solution is constructed using equivalent functions which are identified by automated testing. In contrast, our approach is based on control flow graph similarity to identify equivalent or similar parts of code and not only equivalent functions.

Wang et al. proposed an automated grading approach for assignments in C based only on program similarity [14]. It is based on dependence graphs [80] as program representation. They perform various code transformations in order to standardize the representation of the program. In this approach, the similarity is calculated based on comparison of structure, statement, and size which are weighted by some predetermined coefficients. Their approach is evaluated on 10 problems, 200 solutions each, and gave good results compared to manual grading. Manual grading was performed strictly according to the criterion that indicates how the scores are awarded for structure, statements used, and size. However, it is not quite obvious that human grading is always expressed strictly in terms of these three factors. An advantage of our approach compared to this one is automated tuning of weights corresponding to different variables used in grading, instead of using the predetermined ones. Since teachers do not need to have an intuitive feeling for different similarity measures, it may be unclear how the corresponding weights should be chosen. Also, we avoid language-dependent transformations by using LLVM, which makes our approach applicable to a large variety of programming languages. The approach by Wang et al. was extended to automated learning and examination system AutoLEP [81]. AutoLEP provides submission support, feedback about compiler errors, failed test cases, and the similarity of student's and teacher's solutions. Automated grading is also provided, but the way it is performed is not elaborated in the paper. AutoLEP is not publicly available. Very similar approach to the one of Wang et al. was presented by Li et al. [82].

Another approach to grading assignments based only on graph similarity measure is proposed by Naudé et al. [15]. They represent programs as dependence graphs and propose directed acyclic graph (DAG) similarity measure. In their approach, for each solution to be graded, several similar instructor-graded solutions are considered and the grade is formed by combining grades

of these solutions with respect to matched portions of the similar solutions. The approach was evaluated on one assignment problem and the correlation between human and machine provided grades is about the same as ours. For appropriate grading, they recommend at least 20 manually graded solutions of various qualities for each problem to be automatically graded. In the case of automatic grading of high-quality solutions (as is the case with our corpus), using 20 manually graded solutions, their approach achieves 16.7% relative error, while with 90 manually graded solutions it achieves around 10%. The adventage that our approach provides is reflected through several indicators. We used a heterogeneous corpus of 15 problems instead of one. Our approach uses 1 to 3 model solutions for each problem to be graded and a training set for weight estimation which does not need to contain the solutions for the program to be graded. So, after the initial training has been performed, for each new problem only few model solutions should be provided. With 1 to 3 model solutions, we achieve 10% relative error (see Table 2). Due to the use of the LLVM platform, we do not use language-dependent transformations, so our approach is applicable to a large number of programming languages. The similarity measure we use, called neighbor matching, is similar to the one of Naudé et al. but for our measure, important theoretical properties (e.g., convergence) are proven [18]. The neighbor matching method was already applied to several problems but in all these applications its use was limited to ordinary graphs with nodes without any internal structure. To apply it to CFGs, we modifed the method to include node content similarity which was independently defined as described in Section 6.1.

Program similarity measurement need not be based on graphs. Comparison of students programs with model solutions can be based on using different metrics, like in the tool ELP [83], WAGS [84] and the system proposed by Khirulnizam et al. [85]. In these approaches, teachers need to provide model programs for all the possible answer variations so these systems aim at small and "fill-in the gap" type programming exercises [81]. Aiken's tool MOSS [86] is based on clever substring matching between the programs. In contrast, our system uses graph similarity measures as they reflect the way parts of code are interconnected. So, for two programs, we take into account the similarity of connections between their parts, not only the similarity of the parts themselves.

We are not aware of other open source implementations of the graph similarity based approaches, so our system is unique in this respect.

8.4. Software Verification Techniques in Automated Grading

Surprisingly, software verification techniques are still not commonly used in automated evaluation of programs. There are limited experiences on using Java PathFinder model checker for automated test case generation in education [87]. The tool Ceasar [88] has integrated support for automated testing and verification, but is not aimed for educational purposes. For Java projects, Marmoset runs the FindBugs [89] tool — a static analysis tool that looks for Java coding defects. The system does not report all warnings that FindBugs generates, but can still have false positives. The tool LAV was already used, to a limited extent, for finding bugs in students' programs [13]. In that work, a different sort of corpus was used, as discussed in Section 5.2. Also, that application did not aim at automated grading, and instead was made in a wider context of design and development of LAV as a general-purpose SMT-based error finding platform.

9. Conclusions and Further Work

We presented two methods that can be used for improving automated evaluation of students' programs in introductory programming courses. The first one is based on software verification and the second one on CFG similarity measurement. Both techniques can be used for providing useful and helpful feedback to students and for improving automated grading for teachers. In our evaluation, against the instructor-provided grades, we show that synergy of these methods offers more information useful for automated grading than any of them independently. Also, we obtained good results in prediction of the grades for a new set of assignments. Our approach can be trained to adapt to teacher's grading style on several teacher graded problems and then be used on different problems using only few model solutions per problem. An important advantage of our approach is independence of specific programming language since LLVM platform (which we use to produce intermediate code) supports large number of programming languages. The presented methodology is implemented in our open source tools.

In our future work we plan to make an integrated web-based system with support for the mentioned techniques along with compiling, automated testing, profiling and detection of plagiarism of students' programs or to integrate our techniques into an existing system. We are planning to integrate LLVM-based open source tool KLEE [8] for automated test case generation along

with support for teacher supplied test cases. Also, we intend to improve feedback to students by indicating missing or redundant parts of code compared to the teacher's solution. This feature would rely on the fact that our similarity measure provides the similarity values for nodes of CFGs, and hence enables matching the parts of code between two solutions. If some parts of the solutions cannot be matched or are matched with very low similarity, this can be reported to the student. On the other hand, the similarity of the CFG with itself could reveal the repetitions of parts of the code and suggest that refactoring could be performed.

We also plan to explore the potential for using software verification tools for proving functional correctness of student programs. This task would pose new challenges. Testing, profiling, bug finding and similarity measurement are used on original students' programs, which makes the automation easy. For verification of functional correctness, the teacher would have to define correctness conditions (possibly in terms of implemented functions) and insert corresponding assertions in appropriate places in students' programs, which should be possible to automate in some cases, but it is not trivial in general. In addition, for some programs it is not easy to formulate correctness conditions (for example, for programs that are expected only to print some messages on standard output).

References

[1] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, J. Paterson, A Survey of Literature on the Teaching of Introductory Programming, in: Working group reports on ITiCSE on Innovation and technology in computer science education, ITiCSE-WGR '07, ACM, 2007, pp. 204–223.

[2] T. Nipkow, Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs, in: Verification, Model Checking, and Abstract Interpretation (VMCAI), pp. 24–38.

[3] M. Vujošević-Janičić, D. Tošić, The Role of Programming Paradigms in the First Programming Courses, The Teaching of Mathematics XI (2008) 63–83.

[4] I. E. Allen, J. Seaman, Learning on demand: Online education in the United States, 2009, Technical Report, The Sloan Consortium, 2010.

[5] C. Douce, D. Livingstone, J. Orwell, Automatic Test-based Assessment of Programming: A Review, Journal on Educational Resources in Computing 5 (2005).

[6] W. Afzal, R. Torkar, R. Feldt, A systematic review of search-based testing for non-functional system properties, Information and Software Technology 51 (2009) 957–976.

[7] N. Tillmann, J. Halleux, Pex White Box Test Generation for .NET , in: Proceedings of TAP 2008, the 2nd International Conference on Tests and Proofs, volume 4966 of *LNCS*, Springer, 2008, pp. 134–153.

[8] C. Cadar, D. Dunbar, D. Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, in: Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI), USENIX Association Berkeley, 2008, pp. 209–224.

[9] V. Chipounov, V. Kuznetsov, G. Candea, S2E: A Platform For In-vivo Multi-path Analysis of Software Systems, ACM SIGARCH Computer Architecture News 39 (2011) 265–278.

[10] E. Clarke, D. Kroening, F. Lerda, A Tool for Checking ANSI-C Programs, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, 2004, pp. 168–176.

[11] L. Cordeiro, B. Fischer, J. Marques-Silva, SMT-Based Bounded Model Checking for Embedded ANSI-C Software, International Conference on Automated Software Engineering (ASE) (2009) 137–148.

[12] F. Merz, S. Falke, C. Sinz, LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR, in: Verified Software: Theories, Tools and Experiments (VSTTE), LNCS, Springer, 2012, pp. 146–161.

[13] M. Vujošević-Janičić, V. Kuncak, Development and Evaluation of LAV: An SMT-Based Error Finding Platform, in: Verified Software: Theories, Tools and Experiments (VSTTE), LNCS, Springer, 2012, pp. 98–113.

[14] T. Wang, X. Su, Y. Wang, P. Ma, Semantic similarity-based grading of student programs, Information and Software Technology 49 (2007) 99–107.

[15] K. A. Naudé, J. H. Greyling, D. Vogts, Marking Student Programs Using Graph Similarity, Computers and Education 54 (2010) 545–561.

[16] K. M. Ala-Mutka, A Survey of Automated Assessment Approaches for Programming Assignments, Computer Science Education 15 (2005) 83–102.

[17] P. Ihantola, T. Ahoniemi, V. Karavirta, O. Seppälä, Review of Recent Systems for Automatic Assessment of Programming Assignments, in: Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling '10, ACM, 2010, pp. 86–93.

[18] M. Nikolić, Measuring Similarity of Graph Nodes by Neighbor Matching, Intelligent Data Analysis Accepted for publication (2013).

[19] C. Lattner, V. Adve, The LLVM Instruction Set and Compilation Strategy, 2002.

[20] C. Lattner, The LLVM Compiler Infrastructure, 2012. `http://llvm.org/`.

[21] D. Dhurjati, S. Kowshik, V. Adve, SAFECode: enforcing alias analysis for weakly typed languages, in: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and iImplementation (PLDI), ACM, New York, NY, USA, 2006, pp. 144–157.

[22] M. R., Llvm-py: Python Bindings for LLVM, 2012. `http://www.mdevan.org/llvm-py/`.

[23] T. Bagby, Llvm Ruby, 2012. `http://llvmruby.org/`.

[24] Haskell, Llvm, 2012. `http://www.haskell.org/haskellwiki/LLVM`.

[25] VMKit, A substrate for virtual machines, 2012. `http://vmkit.llvm.org/`.

[26] D, Llvm D Compiler, 2012. `http://www.ohloh.net/p/ldc`.

[27] Llvm Pure, The Pure Programming Language, 2012. `http://code.google.com/p/pure-lang/`.

[28] G. Reedy, Compiling Scala to Llvm, 2012. `http://greedy.github.com/scala-llvm/`.

[29] Lua, JIT/Static compiler for Lua using LLVM on the backend, 2012. `http://code.google.com/p/llvm-lua/`.

[30] G. Tassey, The Economic Impacts of Inadequate Infrastructure For Software Testing, Technical Report, National Institute of Standards and Technology, 2002.

[31] J. C. King, Symbolic Execution and Program Testing, Communications of the ACM 19 (1976) 385–394.

[32] E. M. Clarke, 25 Years of Model Checking — The Birth of Model Checking, LNCS, Springer, 2008.

[33] P. Cousot, R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: Symposium on Principles of Programming Languages (POPL), ACM Press, 1977, pp. 238–252.

[34] C. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, Satisfiability Modulo Theories, in: Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009, pp. 825–885.

[35] F. E. Allen, Control flow analysis, in: Proceedings of a symposium on Compiler optimization, ACM, New York, NY, USA, 1970, pp. 1–19.

[36] J. M. Kleinberg, Authoritative Sources in a Hyperlinked Environment, Journal of the ACM 46 (1999) 604 — 632.

[37] M. Heymans, A. Singh, Deriving Phylogenetic Trees from the Similarity Analysis of Metabolic Pathways, Bioinformatics 19 (2003) 138—146.

[38] V. D. Blondel, A. Gajardo, M. Heymans, P. Snellart, P. van Dooren, A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching, SIAM Review 46 (2004) 647—666.

[39] H. W. Kuhn, The Hungarian Method for The Assignment Problem, Naval Research Logistics Quarterly 2 (1955) 83–97.

[40] J. Edmonds, R. M. Karp, Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems, Journal of the ACM 19 (1972) 248–264.

[41] M. L. Fredman, R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Journal of the ACM 34 (1987) 596–615.

[42] A. K. Mandal, C. A. Mandal, C. Reade, A System for Automatic Evaluation of C Programs: Features and Interfaces, International Journal of Web-Based Learning and Teaching Technologies 2 (2007) 24–39.

[43] P. Loo, W. Tsai, Random testing revisited, Information and Software Technology 30 (1988) 402–417.

[44] P. Godefroid, M. Y. Levin, D. A. Molnar, SAGE: Whitebox Fuzzing for Security Testing, ACM Queue 10 (2012) 20.

[45] B. P. Miller, L. Fredriksen, B. So, An Empirical Study of the Reliability of UNIX Utilities, Communications of ACM 33 (1990) 32–44.

[46] J. W. Duran, S. C. Ntafos, An Evaluation of Random Testing, IEEE Transactions of Software Engineering 10 (1984) 438–444.

[47] P. Godefroid, N. Klarlund, K. Sen, DART: Directed Automated Random Testing, in: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, 2005, pp. 213–223.

[48] P. Godefroid, M. Y. Levin, D. Molnar, Sage: Whitebox fuzzing for security testing, Queue 10 (2012) 20:20–20:27.

[49] K. Sen, D. Marinov, G. Agha, CUTE: A Concolic Unit Testing Engine for C, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13, ACM, New York, NY, USA, 2005, pp. 263–272.

[50] M. Zalewski, Bunny the Fuzzer, 2008. `http://code.google.com/p/bunny-the-fuzzer/`.

[51] O. Ruwase, M. S. Lam, A Practical Dynamic Buffer Overflow Detector, in: Proceedings of the 11th Annual Network and Distributed System Security Symposium, pp. 159–169.

[52] J. Condit, M. Harren, S. McPeak, G. C. Necula, W. Weimer, Ccured in the real world, in: Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, 2003, pp. 232–244.

[53] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, SIGPLAN Not. 42 (2007) 89–100.

[54] S. H. Yong, S. Horwitz, Using static analysis to reduce dynamic analysis overhead, Form. Methods Syst. Des. 27 (2005) 313–334.

[55] D. G. Thorburn, G. W. A. Rowe, PASS: An Automated System for Program Assessment, Computers & Education 29 (1997) 195–206.

[56] V. I. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions, and Reversals, Soviet Physics Doklady 10 (1966) 707–710.

[57] J. Gross, Linear Regression, Springer, 2003.

[58] M. L. Mitchell, J. M. Jolley, Research Design Explained, Wadsworth Cengage Learning, 2012.

[59] J. W. Howatt, On criteria for grading student programs, SIGCSE Bull. 26 (1994) 3–7.

[60] J. B. Hext, J. W. Winings, An automatic grading scheme for simple programming exercises, Communications of ACM 12 (1969) 272–275.

[61] W. H. Fleming, K. A. Redish, W. F. Smyth, Comparison of manual and automated marking of student programs, Information and Software Technology 30 (1988) 547–552.

[62] U. V. Matt, Kassandra: The Automatic Grading System, SIGCUE Outlook 22 (1994) 22–26.

[63] R. Saikkonen, L. Malmi, A. Korhonen, Fully Automatic Assessment of Programming Exercises, ACM Sigcse Bulletin 33 (2001) 133–136.

[64] K. A. Reek, The TRY system -or- how to avoid testing student programs, SIGCSE Bull. 21 (1989) 112–116.

[65] D. S. Morris, Automatically Grading Java Programming Assignments Via Reflection, Inheritance, and Regular Expressions, Frontiers in Education Conference 1 (2002) T3G–22.

[66] D. Morris, Automatic Grading of Student's Programming Assignments: An Interactive Process and Suit of Programs, in: Proceedings of the Frontiers in Education Conference 3, volume 3, pp. 1–6.

[67] J. English, Automated Assessment of GUI Programs Using JEWL, SIGCSE Bull. 36 (2004) 137–141.

[68] M. Wick, D. Stevenson, P. Wagner, Using Testing and JUnit Across the Curriculum, SIGCSE Bull. 37 (2005) 236–240.

[69] B. Cheang, A. Kurnia, A. Lim, W.-C. Oon, On Automated Grading of Programming Assignments in an Academic Institution, Computers and Education 41 (2003) 121–131.

[70] D. Arnow, O. Barshay, WebToTeach: An Interactive Focused Programming Exercise System, Frontiers in Education, Annual 1 (1999) 12A9/39–12A9/44.

[71] C. C. Ellsworth, J. B. Fenwick, Jr., B. L. Kurtz, The Quiver System, in: Proceedings of the 35th SIGCSE technical symposium on Computer science education, SIGCSE '04, ACM, 2004, pp. 205–209.

[72] A. Zeller, Making Students Read and Review Code, in: ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education, ACM Press, 2000, pp. 89–92.

[73] S. H. Edwards, Rethinking Computer Science Education from a Test-First Perspective, in: Companion of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 148–155.

[74] L. Huang, M. Holcombe, Empirical investigation towards the effectiveness of Test First programming, Information and Software Technology 51 (2009) 182–194.

[75] J. Spacco, D. Hovemeyer, W. Pugh, J. Hollingsworth, N. Padua-Perez, F. Emad, Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses, in: Proceedings of the 11th annual conference on Innovation and technology in computer science education (ITiCSE), ACM Press, 2006, pp. 13–17.

[76] D. Jackson, M. Usher, Grading student programs using ASSYST, SIGCSE Bull. 29 (1997) 335–339.

[77] M. Joy, N. Griffiths, R. Boyatt, The BOSS online submission and assessment system, Journal of Educational Resources in Computing 5 (2005).

[78] C. A. Higgins, G. Gray, P. Symeonidis, A. Tsintsifas, Automated assessment and experiences of teaching programming, Journal on Educational Resources in Computing 5 (2005).

[79] M. Blumenstein, S. Green, S. Fogelman, A. Nguyen, V. Muthukkumarasamy, Performance analysis of GAME: A generic automated marking environment, Computers & Education 50 (2008) 1203–1216.

[80] S. Horwitz, T. Reps, The Use of Program Dependence Graphs in Software Engineering, in: Proceedings of the 14th international conference on Software engineering, ICSE '92, ACM, 1992, pp. 392–411.

[81] T. Wang, X. Su, P. Ma, Y. Wang, K. Wang, Ability-training-oriented automated assessment in introductory programming course, Computers and Education 56 (2011) 220–226.

[82] J. Li, W. Pan, R. Zhang, F. Chen, S. Nie, X. He, Design and implementation of semantic matching based automatic scoring system for C programming language, in: Proceedings of the Entertainment for education, and 5th international conference on E-learning and games, Springer, 2010, pp. 247–257.

[83] N. Truong, P. Roe, P. Bancroft, Automated feedback for "fill in the gap" programming exercises, in: Proceedings of the 7th Australasian conference on Computing education - Volume 42, ACE '05, Australian Computer Society, Inc., 2005, pp. 117–126.

[84] N. Zamin, E. E. Mustapha, S. K. Sugathan, M. Mehat, E. Anuar, Development of a Web-based Automated Grading System for Programming Assignments using Static Analysis Approach, 2006. International Conference on Technology and Operations Management (Institute Technology Bandung).

[85] A. R. Khirulnizam, A. Syarbaini, J. N. Md, The design of an automated C programming assessment using pseudo-code comparison technique, 2007. National Conference on Software Engineering and Computer Systems (Pahang, Malaysia).

[86] S. Schleimer, D. S. Wilkerson, A. Aiken, Winnowing: Local algorithms for document fingerprinting, in: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM, 2003, pp. 76–85.

[87] P. Ihantola, Creating and Visualizing Test Data From Programming Exercises, Informatics in education 6 (2007) 81–102.

[88] H. Garavel, OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 1384 of *LNCS*, Springer, 1998, pp. 68–84.

[89] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, Y. Zhou, Evaluating static analysis defect warnings on production software, in: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE '07, ACM, 2007, pp. 1–8.

Appendix A. Problem Descriptions

Short descriptions of 15 problems used in the empirical evaluation of the presented grading approach (problems with minor variations in formulations are listed as (a) and (b)):

1. (a) Write a program that checks whether the digits of a given four-digit number are in ascending order.
   (b) Write a program that computes the product of all even digits of a four-digit number.

2. (a) Write a function that computes the maximal value of a given array. Write a function that computes the mean value of a given array. Write a program that uses these two functions and that determines whether the maximal value is at least two times bigger than the mean value.
   (b) Write a function that computes an index of a minimal element of a given array. Write a function that computes an index of a maximal element of a given array. Write a program that uses these two functions and that computes whether the index of the maximal element is bigger than the index of the minimal element of a given array.
3. (a) Write a function that converts all lowercase letters that are on even positions in a given string into corresponding uppercase letters, and all uppercase letters that are on odd positions in the given string into corresponding lowercase letters. Write a program that uses this function. Input strings are not longer than 20 characters.
   (b) Write a function that converts all lowercase letters in a given string that are on positions that are divisible by three into corresponding uppercase letters, and all uppercase letters that are on positions which when divided by three give reminder one into corresponding lowercase letters. Write a program that uses this function. Input strings are not longer than 20 characters.
4. Write a function that calculates an array of maximal elements of rows of a given matrix. Write a program that uses this function.
5. (a) Write a function that deletes a character on a position k in a given string. Write a program that uses this function. Input strings are not longer than 20 characters.
   (b) Write a function that duplicates a character on a position k in a given string. Write a program that uses this function. Input strings are not longer than 20 characters.
6. (a) Write a function that calculates the sum of all elements that are above the secondary diagonal of a given matrix. Write a program that uses this function.
   (b) Write a function that calculates the sum of all elements that are below the secondary diagonal of a given matrix. Write a program that uses this function.
7. Write a program that calculates the maximum of two given real numbers.

8. Write a function int `strcspn(char* s, char* t)` that calculates a position of the first occurrence of a character from the string t in the string s. Write a program that uses this function. Input strings are not longer than 20 characters.
9. Define a data structure for fraction. Write a function for comparing two given fractions. Write a function that computes the minimal fraction in a given array. Write a program that uses these functions.
10. Write a program that prints a bow of a size n. For example, for $n = 5$ the output should be
```
xxxxx
.xxx.
..x..
.xxx.
xxxxx
```
11. Write a program that calculates the determinant of a given 2×2 matrix.
12. Write a program that calculates the maximal value of three given numbers.
13. Write a program that prints values of the cosine function in ten equidistant points from a given interval $[a, b]$.
14. Write a program that for a given time calculates the number of seconds until the next noon.
15. Write a program that for a number n prints the numbers from 1 to $n - 1$, then from 2 to $n - 2$, from 3 to $n - 3$ and so on.