Journal of Network and Computer Applications 35 (2012) 695–712

Contents lists available at SciVerse ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice

Francisco Jurado*, Miguel A. Redondo, Manuel Ortega

Department of Technologies and Information Systems, School of Computer Science, University of Castilla—La Mancha, Paseo de la Universidad 4, 13071 Ciudad Real, Spain

ARTICLE INFO

Article history:
Received 19 February 2011
Received in revised form
11 July 2011
Accepted 1 November 2011
Available online 18 November 2011

Keywords:
Intelligent tutoring systems
Programming and programming languages
Evaluation methodologies

ABSTRACT

Programming is an important competence in Computer Science and Engineering studies. Students have to pass the related programming subjects proving that they have acquired the necessary knowledge and skills. Such knowledge and skills will be essential for their academic training and future careers. However, students of programming subjects have to solve deep cognitive challenges. One of our objectives is to help them overcome these challenges.

To overcome the first difficulties the students may encounter while developing their assignments, we will present an approach that provides a first assessment of the students' solution. This will allow the system to estimate a mark on a student assignment; so it can provide some kind of adaption and guidance in the learning process. It will allow the students to ask the system what is wrong with the solution they are developing, without teacher intervention. Thus, our aim is to create a system that assists the students in understanding what they are doing and helps the teachers in their labour in the classroom.

Although the assessment technique we will show can be extrapolated to other programming areas, this paper focuses on a concrete one, namely Programming Algorithms. In this respect, throughout the paper we will show how to process and analyse algorithms written by students as solutions to programming assignments, something that is ongoing in those computer supported systems for learning to program. As a result, a proposal for assessing algorithms applying Fuzzy Logic to software metrics and test cases will be explained. In addition, the Computer Assisted Environment for Learning Algorithms (COALA) environment will be exposed in order to validate our proposals. Furthermore, we will explain the developed empirical study and also analyse and explain the results. These results seem to point out that the proposals are successful enough, but we must continue working on this research line to provide evaluations that are closer to those provided manually by a teacher.

1. Introduction

Nowadays, designing and coding computer programs is a competence that students of Computer Science have to acquire. It is probably one of the most important competences they will need in their academic training and future labour, because the knowledge and skills they will acquire are keys to both the rest of their academic training and their future professional lives (ACM/IEEE, 2008; ACM/IEEE, 2010).

It seems to be widely accepted that learning to program requires a "learning by doing" approach (Kumar, 2002; Schollmeyer, 1996), typically in the form of programming labs that provide active learning (McConnell, 1996). Nevertheless, these students have to solve some difficulties while learning the related knowledge and acquiring the corresponding skills (du Boulay, 1989; Brusilovsky et al., 1998; Gomes and Mendes, 2007). One objective of our research is to support these students in solving their difficulties in learning the program.

Thus, in this section, we will show the background and previous work on the systems that provide support in learning to program, focussing our attention on a system that provides students with assessment and advice. Then, we will expose the hypothesis and objectives of our research.

1.1. Background and previous work

Several systems and approaches have been proposed in order to support the acquisition of programming knowledge and skills (Kelleher and Pausch 2005; Garner, 2003). Among these approaches, we will centre our attention on those systems that assist the students

* Corresponding author. Tel.: +34 926295300.
E-mail addresses: Francisco.Jurado@uclm.es (F. Jurado), Miguel.Redondo@uclm.es (M.A. Redondo), Manuel.Ortega@uclm.es (M. Ortega).

1084-8045/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2011.11.002

with any kind of adaption and feedback allowing constructivist learning experiences (Ben-Ari, 2001).

Thus, in the case of learning to program, processing and analysing the algorithms implemented by the students will be essential so that they can be assessed and evaluated. So, from here on, we will distinguish between assessment and evaluation. Assessment will provide the mark for a concrete student's solution, while evaluation will provide a critical analysis of that solution, in order to explain why the assessment has been established in that way. This will allow a double adaption:

- The assessment will provide feedback to the system so that the instructional model can be adapted according to the previous marks that the student has gained.
- The evaluation will be the information or explanation provided to students about what is right and wrong in their solution. This will allow the students to better understand their faults and to try to correct them.

Therefore, providing an assessment is the basis for enabling the instructional adaption, that is, it will make it easier to know what the next learning activities for the students are, as the result of their previously delivered solutions. Furthermore, registering the marks during a teaching/learning process can contribute to the final evaluation of the student. These concepts match with the *follow-up and tutoring* model proposed by Andrade et al. (2008).

On the other hand, giving the students enough information about what they are doing right and wrong while solving the problems is essential for better knowledge acquisition (Gordijn and Nijhof, 2002) so that they can modify and improve their solutions (Bravo et al., 2009). These are the conclusions extracted from explained studies such as those provided by Kumar and Fernandes, who suggest, *"Tutors that provide feedback are clearly more effective in helping students learn than those that do not"* (Kumar, 2004). They go even further to point out: *"Since the tutor provides detailed feedback for its answers, it can be used as a supplement to classroom instruction"* (Fernandes and Kumar, 2005).

In this regard, Sanders and Hartman (1987) noticed that when the learners watched the evaluations of a program, it helped them to justify their choices of algorithms and data structures used when solving a problem. According to these authors, this makes them more effective programmers, since they get used to considering alternative solutions and evaluating the advantages and disadvantages of each possible choice.

To do so, we must use an Automated Programming Assessment System (APAS). A good survey about static and dynamic techniques used in this kind of system can be found in Ala-Mutka (2005) and Rahman and Nordin (2007). The use of the APAS has several advantages (Rahman and Nordin, 2007) that are the current focus of our research:

- assisting the teachers by relieving them from some evaluation tasks that can be automated, so that they can devote this time to the preparation of lectures, seminars, laboratories, etc.;
- providing instant information to students to improve the solution that had been reached;
- overcoming the innate impartiality of human nature in evaluating students' assignments and
- detecting possible plagiarism in the case where the APAS uses forensic techniques.

APAS is commonly used together with an Intelligent Tutoring System (ITS) to help students improve and get the skills and aptitudes they should acquire and develop, reducing the need for the slow trial and error process. ITSs allow adapting the learning process to each student. To do so, ITSs focus on determining what the student cognitive model is by identifying which parts of the domain he/she knows by means of evaluating the student's knowledge. Thus, ITSs can infer the next learning activity for each specific student. So, assessing and evaluating the assignments delivered by the students is an essential task.

As an example of an ITS that uses APAS for learning program, we can mention C-Tutor (Song et al., 1997). C-Tutor analyses the student's solutions, so that it gives the corresponding feedback and provides tutoring adaption. It has a program analyser, divided into two parts, which the authors call "reverse engineering system" and "didactic system". The reverse engineering system obtains a problem description starting from a teacher's algorithm. This problem description consists of a set of goals (common concepts of programming such as swapping, finding the maximum, etc.) that the students' solutions must implement. These goals can be achieved by means of following plans, which describe stereotypical action sequences in the algorithms. Then, the didactic system evaluates the students' solutions by matching the teacher's problem description with the student's solution. A knowledge-base in C-Tutor stores goals and plans. A goal can be related to more than one plan, so that the implementation variations can be considered. Thus, the system decides the next programming assignment, or to teach a new concept (goal) using this evaluation. The main problem of this approximation is that we create the knowledge-base that allows managing a wide range of programming concepts.

In order to show some examples of systems that use APAS, it is worth highlighting the student adaption provided by electronic books such as ELM-ART (Brusilovsky et al., 1996) for learning LISP and KBS-Hyperbook (Nejdl and Wolpers 1999) for learning Java. In general, these systems offer a guided navigation through the given didactic material, trying to adapt that navigation to each student's profile. In this regard, they are electronic books that offer adaption. The knowledge acquired by the students is assessed by questionnaires, quizzes or tests from algorithms developed by the students as solutions to assignments. In this way, ELM-ART shows examples to students, allowing them to modify, debug and execute the LISP programs in its evaluator web interface. This evaluator provides assessment and explanation for mistakes at runtime. As a consequence, there is no information about how the algorithm has been designed, that is, there is no information about the structure of the student's algorithms. Furthermore, the kind of programming environment provided through its web interface is limited, and it allows students to work only with simple code.

Another online problem-solving approach is the work presented in Kumar (2004) and Fernandes and Kumar (2005). The work presents *problets*, a set of tutors that provides visualisation and animation on several programming C++ topics such as expression evaluation, loops, encapsulation, pointers, parameters passing and scope concepts. Authors assert that *problets* provide detailed feedback to students; so they can be used as a supplement to classroom instruction. There is a *problet* for each topic, and each *problet* is a particular simulation environment, which makes it a specialised tool.

In order to allow students to write more complex code, the work presented in Pérez et al. (2006) provides a web interface for a Java development environment. The work is focused on the analysis of code written by the students. It allows detecting, removing and preventing mistakes using language-processing techniques. To do so, the system logs, stores and analyses the errors in the code. Thus, programmers can learn from their own mistakes and can avoid making the same mistakes in the future. However, this work is centred on improving the source code but not on evaluating and assessing it.

On the other hand, a development environment based application that allows students to write complex code is ProPAT

(de Barros et al., 2005). ProPAT uses Abstract Syntax Tree (AST) to process the students' algorithms. The teacher must set the programming assignment by specifying the name, the use, the syntax template, etc. Then, the syntax template will be matched with the AST of the students' code, giving them the corresponding explanation.

So far, the papers mentioned provide the corresponding feedback to the students as an explanation about what is right and wrong in their delivered code, but none of them provide a score for the assignments. In this sense, in Naudé et al. (2010) we can find a graph similarity approach to mark programming assignments. This approach analyses the similarities from the student-delivered solution to related programs by applying tree patterns transformation to the ASTs. However, although this approach provides automatic marking, it does not give advice to the students.

As can be seen, of the systems that provide feedback to the students who are learning to program, not all the proposals use environments that allow working with complex code. This is because analysing complex code is very difficult as it can present significant variability (Song et al., 1997). Furthermore, developing an APAS implies a difficult challenge that involves the use of several techniques and disciplines. In addition, most of them present drawbacks like lack of user friendliness, difficulty to install and that they cannot be used for multiple programming languages (Rahman and Nordin, 2007). This may be due to the complexity they imply. Thus, we will focus our research on a concrete area in learning to program, that is, programming of algorithms. So, in this article we will show an approach that allows analysing algorithms in an automatic way, and thus provides assessment and gives advice to the student. This approach can be included in systems that provide adaption and advice. Our aim is not to create a system that substitutes the teacher, but rather to implement an approximation that provides support in a traditional classroom (Schofield et al., 1994).

Therefore, our research starts by looking for how to represent the ideal algorithm that the teacher thinks most appropriate for solving a concrete assignment. Next, the algorithm developed by the student should be compared with the obtained ideal representation. This starting point leads us to think about similarity detection techniques: *the closer an algorithm written by the student to that written by the teacher, the closer the solution proposed by the student to the ideal representation*.

At this point, the similarity of code is introduced. This is the field of study for the Forensic Software (FS) or analysis of authorship (Gray et al., 1997; Sallis et al., 1996), a branch of Software Engineering. This discipline uses scientific methods to solve crimes of plagiarism, trying to determine the authorship of the source code by analysing the degree of similarity between two pieces of code. To do this, the aforementioned is based on software metrics, analysis of subjective elements such as style of writing (indentation, variable name, etc.) and so on. Therefore, if two pieces of code are similar enough, the authorship comes from the same programmer. In Sallis et al. (1996) it is stated, "*Software science metrics and those based on control-flow produce metric values that are clearly program-specific*". From this, it can be deduced that the similarity of these metrics on different codes is related to the existence of a similar functionality in both codes. This is the premise that we will use as a starting point on the work presented here. BOSS (Joy and Luck, 1999; Joy et al., 2005), Style++ (Ala-Mutka et al., 2004) and ELP (Truong et al., 2004) are some examples of APAS that perform an analysis based on software metrics.

The techniques employed in the field of FS are typically statistics (Gray et al., 1997; Sallis et al., 1996). However, Fuzzy Logic (Zadeh, 1965) has been used in the field of FS for managing the subjective elements of the styles of writing code (Kilgour et al., 1997) and other metrics analysis within Software Engineering, using fuzzy decision trees (Sahraoui et al., 2000).

As mentioned above, we have focused our attention on the teaching/learning of programming algorithms. The teaching of designing and programming algorithms implies not only that students must solve the assignments proposed by the teacher, but also that they must solve them by following a concrete coding style. Thus, the teacher can set a certain assignment to be solved in different ways. Let us assume the algorithms to order the elements of a vector. There are different algorithms with different complexities, performance, etc., but all of them solve the same problem: the ordering of data. If the teacher is teaching a particular ordering technique, any other technique raised by the student should not be valid even if it solves the problem. Furthermore, it should handle the vagueness in which both teacher and student express the solution. This is because many algorithms can be applied to solve the same problem.

This paper aims to design and validate a mechanism to cover some of the identified shortcomings in the analysed systems, that is, to support the automatic assessment of programming algorithms, processing their static and dynamic parts independently from the programming language, and to solve the drawbacks of user friendliness. Thus, we expect to obtain a proposal that provides feedback to the students who are learning to program and allows us to use it in environments that work with complex code. So, we will propose an approach which analyses the algorithms written by the students by using a representation of the ideal algorithm that the teacher thinks is most suitable to solve a given assignment. Then, the algorithm that the student has raised as a solution must be matched with the ideal representation that the teacher has pointed out.

1.2. Hypothesis, objectives and organisation

Our starting hypothesis can be established as follows: "it is possible to automatically assess the algorithms developed by the students and to give advice to them, applying Fuzzy Logic to the software metrics and test cases".

To corroborate the hypothesis, our main goal will be to design a technique that analyses algorithms by taking into account both their static part (structure and shape) and their dynamic part (correctness). So, in this article we will describe our proposal about how to use Fuzzy Logic techniques in software metrics and test cases to evaluate and give advice about the algorithms written by students, in an automatic way. Thus, to achieve our hypothesis, we will show how we have evaluated our proposal by means of a customised development environment built to do so.

This proposal is aimed at facilitating the creation of environments for learning to program. So, those environments will be able to process and assess the students' algorithms. In this regard, it will be possible to design a system that analyses the static and dynamic components of algorithms, so it can be used to build Intelligent Tutoring Systems for teaching how to design computer algorithms.

Thus, in the way we have defined our starting hypothesis, processing the static part of the algorithm by means of software metrics and the dynamic part by means of test cases will give us independence from the programming language. Consequently, we expect to provide a technique that is isolated from the programming language particularities. Furthermore, to introduce the use of Fuzzy Logic we will provide the mechanism for managing the imprecision and deviations. Therefore, we expect to solve the drawbacks of user friendliness and difficulties with the use of software metrics and test cases.

The rest of the article is structured as follows: first we will outline the elements required to carry out our approach (Section 2);

then, the process of assessing the algorithm (Section 3) and giving advice (Section 4) will be explained; next, a thorough example applying the proposal will be exposed (Section 5); after this, the validation we have followed for the assessment (Section 6) and for giving advice (Section 7) will be described in depth and analysed; and finally some concluding remarks and future work will be pointed out (Section 8).

2. Outline

As has been indicated in the previous section, among the systems for learning how to program computer algorithms, there are some proposals that analyse the algorithms designed and implemented by students as solutions to an assignment proposed by the teacher. However, as we have pointed out, to process complex algorithms and to obtain a solution independent from the programming language are both difficult challenges.

Our approach aims to use techniques to determine similarities and differences. The teaching of designing and programming algorithms implies not only that students must solve the problems proposed by the teacher, but also that they must solve them in a concrete way, which requires taking into account both the static part of the algorithms (their structure and shape) and their dynamic part (their correctness). Furthermore, several solutions can be valid to solve the same assignment but all of them must follow a unique abstract representation. Therefore, the proposal must handle all these concepts: structure, correctness and abstraction.

This work follows a model where the teacher (who plays the role of expert) is aware of how the ideal algorithm which solves the assignment, is implemented according to his own criterion. They know what features the solution should or should not have. Based on this ideal algorithm, the analysis of the algorithms delivered by students will be performed. Thus, the challenge is how to represent the ideal algorithm that is in the mind of the teacher. Furthermore, this representation must be manageable by a computer and also easy to specify by the expert. This work has raised the hypothesis that this is possible by applying Fuzzy Logic (Zadeh, 1965). Of course, we assume the absence of "ideal algorithms", but we believe that an algorithm can be considered ideal to solve a particular problem according to an expert, and this algorithm can be characterised by some key aspect that can be extracted and evaluated. Therefore, this will be the basis for the rest of our development.

The outline of the work is shown in Fig. 1. Here, the way that the teacher writes an algorithm that implements a solution to a concrete assignment is shown. Thus, we obtain an instance of the ideal approximated algorithm. Next, several software metrics that shape its structure will be calculated. These metrics provide information about static characteristics for the algorithm; that is, it is related with the style and form in the algorithm design, without taking into account whether the algorithm solves the problem. These software metrics will be the basis for obtaining the *fuzzy representation of the ideal algorithm.*

On the other hand, algorithms written by the students (on the right of Fig. 1) will be correct if they are instances of the *fuzzy representation of the ideal algorithm*. Knowing the degree of membership for each software metric obtained from the algorithm written by students in the corresponding fuzzy set for the fuzzy representation of the ideal algorithm will give us an idea of the quality of the algorithm developed by the students.

Furthermore, as we can see on the bottom of Fig. 1, the teacher writes some test cases that the students can run to test their algorithms. With these, we analyse the correctness of the algorithm. Thus, with the fuzzy evaluation of the metrics that allows

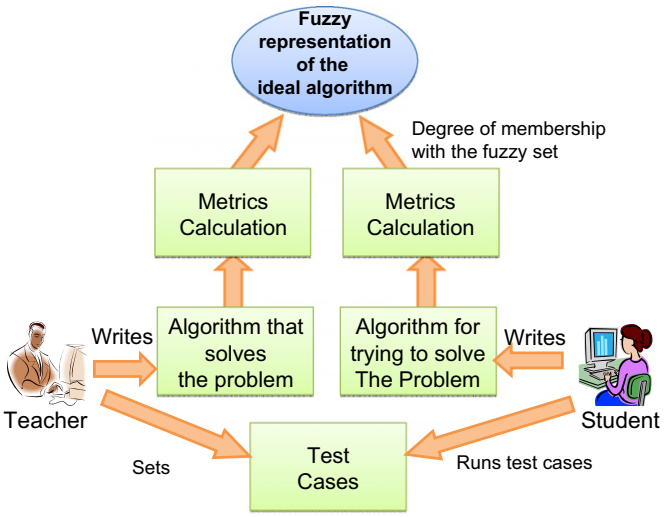

Fig. 1. Evaluating the student algorithm.

analysing the code structure, and the test cases that allow checking if the algorithm solves the problem, we have a complete mechanism that can help the students in the initial stage for solving a programming assignment. In the next subsections we are going to explain these two processes in depth.

2.1. Analysing the structure of the algorithm

Performing an analysis of the structure of the algorithm involves knowing the way in which the algorithm solves the assignment proposed by the teacher, that is, if it is coded according to the appropriate programming method. To do so, we propose the use of Fuzzy Logic applied to several software metrics calculated from algorithms written by the students.

2.1.1. Choosing the software metrics

As input to the analysis process, an initial selection of some standard software metrics from Software Engineering has been made. This subsection will explain in detail the adopted metrics.

The metrics for the analysis of the structure will allow us to know if the code is well implemented following the needs that the teacher has specified, that is, the metrics are those that help the teacher discern whether an algorithm is written correctly according to the appropriate programming method. Its calculation can be performed automatically by processing the source code with a syntactic parser. The software metrics we have chosen from Software Engineering are shown in Table 1.

As can be seen, this table reflects a wide range of issues. All the software metrics are representative from a structural point of view, that is, they will help establish the structure of the code, its skeleton. Thus, the teacher can consider that to solve a problem using a particular programming technique, the student should pass a certain number of parameters, do specific procedure calls, execute concrete recursive calls or use specific operational and control complexity.

Let us show an example that reflects the importance of what we have outlined so far. Let us propose a typically used assignment in programming classes to teach how to programme dynamic programming algorithms, the *knapsack* problem, whose statement established by the teacher could be read as follows: "*Given N items with weight w[i] and value v[i]. The maximum weight the knapsack can carry is "capacity". We want to fill the knapsack with objects, so as to maximise profit. Implementation must be carried out using dynamic programming in its iterative version*".

Table 1
Metrics definition for the analysis of the structure of the algorithm.

Metric	Meaning	Domain
Lines of code	Number of lines of code	$x \in [0..N]$
McCabe's Cyclomatic Complexity	Number of execution paths	$x \in [0..N]$
Parameterization	Number of parameters passed to the algorithm	$x \in [0..N]$
Operational complexity	Complexity of the operations used in the algorithm	$x \in R$
Control complexity	Complexity of the Boolean expressions used in the control statements	$x \in R$
Procedure/method calls	Number of procedures/method calls	$x \in [0..N]$
Recursivity	– Null: there is no recursivity – Lineal: there is only one recursive call – No lineal: there is more than one recursive call	$x \in [0..N]$
Blocks	Number of blocks	$x \in [0..N]$
Arrays	Number of declared arrays	$x \in [0..N]$
Variables	Number of declared variables	$x \in [0..N]$

```
public static int knapsack(int capacity, int[] v, int[] w){
    int[][] solution = new int[v.length+1][capacity+1];

    for (int i = 0; i < solution[0].length; i++) solution[0][i]=0;
    for (int i = 0; i < solution.length; i++) solution[i][0]=0;

    for (int i = 1; i < solution.length;i++ ){//for each object
      for (int j = 0; j < solution[0].length; j++) {//for each weight
        if (j < w[i-1])//there is no capacity for that object
          solution[i][j]=solution[i-1][j];
        else
          solution[i][j]=Math.max(
              solution[i-1][j],//we do not put the object into
              solution[i-1][j-w[i-1]]+v[i-1]);//we put the object into
      }
    }

    return solution[solution.length-1][solution[0].length-1];
  }
```

Fig. 2. Algorithm 1 to solve the *Knapsack* problem.

```
public static int knapsack(int capacity, int[] v, int[] w){
    int[][] solution = new int[v.length][capacity+1];

    for (int i = 0; i < solution.length;i++ ){//for each object
      for (int j = 0; j < solution[0].length; j++) {//for each weight
        if (i==0)//it is the first object
          if (j < w[i])//there is no capacity for that object
            solution[i][j] = 0;
          else
            solution[i][j]=v[i];
        else
          if (j < w[i])//there is no capacity for that object
            solution[i][j]=solution[i-1][j];
          else
            solution[i][j]=Math.max(solution[i-1][j],//we do not put the object into
                                    solution[i-1][j-w[i]]+v[i]);//we put the object into
      }//end of for
    }//end of for
    return solution[solution.length-1][solution[0].length-1];
  }
```

Fig. 3. Algorithm 2 to solve the *Knapsack* problem.

Two different algorithms implemented in Java to solve the problem can be found in Figs. 2 and 3. In Fig. 2, the developer has chosen to initialise part of the table to store intermediate results and then fills in the intermediate values calculated. Meanwhile, in Fig. 3, the developer has opted for an implementation in which the results are stored directly. The two solutions are different enough to show how software metrics provide representative values that depend on the functionality (Sallis et al., 1996).

Table 2 shows the metrics calculated for each of the previous algorithms. Here it is shown that except "lines of code", which varies in only three units, all other metrics are identical, despite differences in the implementations. In the cases shown for these two algorithms, the solutions have provided the same software metric. However, depending on how the algorithm is implemented, particularly due to inexperience and the students' mental models, they vary slightly.

Table 2
Metrics for each algorithm that solve the *Knapsack* problem.

	Algoritm 1	Algoritm 2
Lines of code	17	20
Control complexity	2.5	2.5
Number of arrays	1	1
Number of variables	0	0
Operational complexity	2.0	2.0
Number of value parameters	1	1
McCabe cyclomatic complexity	6	6
Number of reference parameters	2	2
Number of method calls	1	1
Number of recursive calls	0	0
Number of blocks	3	3

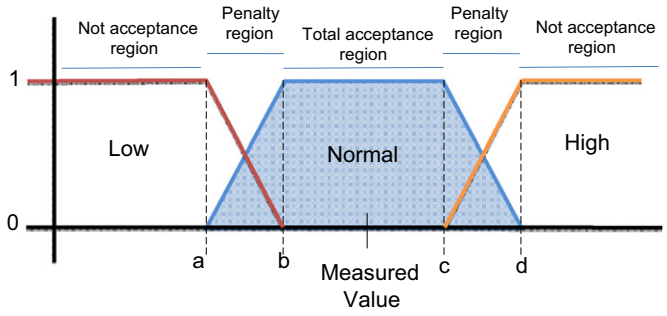

Fig. 4. Fuzzy sets for a software metric.

Therefore, a mechanism to specify and to take into account the possible deviations should be used, so that they are valued. That is, for each software metric, the system must be able to identify and assess the degree to which they comply with the requirements. To do so, in the next section we will show how to verify the degree to which the calculated software metrics for the students' algorithms meet the requirements determined by the teacher for a given programming assignment.

2.1.2. Obtaining the fuzzy representation for the ideal algorithm

For each software metric calculated from the expert's algorithm, we will obtain a situation like the one shown in Fig. 4. In this figure, we can see how the teacher builds fuzzy sets for each software metric, starting from a central calculated value, and in that way to specify labels as "normal", "low" or "high". To do so, the teacher has to specify:

- *The total acceptance region*: the values where the software metric is totally accepted as correct. This is the region where the membership value is 1 for the "normal" fuzzy set and 0 for the "low" and "high" fuzzy sets.
- *The penalty regions*: the values where the software metric is not totally correct but could be accepted in some way. These regions are the decreasing part of the "normal" fuzzy set and the increasing part of the "low" and "high" fuzzy sets where the membership value for each fuzzy set is between 0 and 1.
- *The non-acceptance region*: the values where the software metrics can never be accepted for the solution. These are the regions where the membership value is 1 for the "low" and "high" fuzzy sets and 0 for the "normal" fuzzy set.

Thus, the expert has the tool to determine when the metric calculated from a student's source code is located within the range he/she considers "normal" (correct) or, on the contrary, it is "low" or "high" (deviated from the correct solution).

To assist the teacher in determining the fuzzy sets, the teacher writes the *ideal algorithm* that solves the assignment and the corresponding software metrics are automatically calculated. These metrics can be used as a reference for the teacher ("measured value" in Fig. 4). Then, the teacher can *fuzzify* each software metric by setting *a*, *b*, *c* and *d* values for the trapezoidal function for the "normal" fuzzy set. Thus, the semantics for these values are

- *a*: The maximum value that the teacher considers low for the solution.
- *b*: The minimum value for correctness/acceptance.
- *c*: The maximum value for correctness/acceptance.
- *d*: The minimum value that the teacher considers high for the solution.

The "low" and "high" fuzzy sets are obtained as the complementary sets of the "normal" fuzzy set. Thus, several fuzzy sets are achieved for each software metric that characterise the algorithm, that is, the *fuzzy representation for the ideal algorithm* that solves the problem is obtained.

On the other hand, from the algorithms written by the students we will calculate the same software metrics. Then, we will analyse the membership degree of each fuzzy set of the *fuzzy representation for the ideal algorithm* written by the teacher. This will give us an idea about how similar the algorithm written by the student is to that written by the teacher. This will allow analysing the structure of the algorithm. In the following subsection we will describe the process for checking the correctness of the student's algorithm.

2.2. Analysing the correctness of the algorithm

Performing an analysis of the correctness of the algorithm involves knowing not only the way in which the algorithm solves the assignment proposed by the teacher, but also to what degree it is solved. That is, it helps determine those cases where an algorithm solves the problem. To do this, we propose the use of test cases on the algorithms written by the students. Test cases are used in Software Engineering for calculating the degree of fulfilment of the requirements for software to be developed. A goal is satisfied when some pre-conditions are given as input, certain post-conditions are obtained as output. A weighted aggregation for each goal achieved will give us an idea about how good a solution written by the students is.

The execution of test cases can be performed automatically using techniques and tools for automated testing software such as XUnit (JUnit for Java, CUnit for C, etc.). These tools allow setting the pre-conditions and post-conditions and executing the test cases.

Figure 5 shows an outline for a generic test case used on those tools. In this figure, we can see how it allows describing the test in terms of pre-conditions, post-conditions and the correct solution. The set of defined tests for a given algorithm will be evaluated automatically, prompting how many test cases have been successful and how many have been unsuccessful.

Thus far, we have explained how to perform the analysis for the structural and the dynamic parts of an algorithm. The following sections will show a proposal about how to obtain a fuzzy representation of an algorithm from data obtained from the analysis of the structure and the correctness, and finally make the assessment of what the student has developed as a solution through an analysis of the fuzzy algorithm.

3. Describing how to assess the algorithm

As mentioned above, if a system is able to provide an assessment for an assignment, this assessment can be used as feedback

```
PROCEDURE Test()
BEGIN
   Pre-conditions = Definition_of_pre-conditions();
   Correct = Definition_of_correct_output();
   Post-conditions = Definition_of_post-conditions();

   Output, Conditions = Invoking_algorithm(Pre-conditions);

   IF (Output=Correct) AND (Conditions=Post-conditions)
      THEN
         test satisfied;
      ELSE
         test not satisfied;
         Show_menssage('Message');
   END_IF
END;
```

Fig. 5. Generic code for test cases used in XUnit tools.

to the system, updating the students' cognitive model and leading them to the next appropriate learning activity.

In this section, we will explain how we perform the assessment of algorithms in our proposal. To do so, we have divided the evaluation process in two parts, as mentioned previously: the evaluation of the structure of the algorithm and the evaluation of the correctness of the algorithm. Thus, in the following subsection we will start describing the process for assessing algorithms, implemented by a student using the *fuzzy representation of the ideal algorithm*. Hence, we will explain our choice of aggregation operator used for calculating the similarity.

3.1. Describing how to assess the algorithm

As we have stated in previous sections, the structural assessment will be given by measuring the similarity degree between the algorithm written by the student and the *fuzzy representation of the ideal algorithm*. Calculating that similarity degree will give us an idea about the assessment that the system performs for the algorithm. In an evaluation, the characteristics that we must take into account will balance each other out. That is, the characteristics assessed as good will balance out those characteristics assessed as not so good, and vice versa. In this way, the assessment will be done fairly.

In Fuzzy Logic, using an aggregation operator type "AND", which is the case of *t*-norms, will mean that all the antecedents involved in an implication must be carried out. However, if one characteristic is not correct but all the other features are correct, then the evaluation must be consistent with it and be balanced to a good score.

On the other hand, to use an aggregation operator type "OR", which is the case of *t*-conorms, will mean that at least one of the antecedents involved in an implication must be carried out. But, if all antecedents are incorrect except one, the evaluation must be balanced to a bad score.

In this situation, we can apply the Ordered Weighted Averaging (OWA) aggregation operators (Yager, 1988). These kinds of operators are between the logic operators "AND" and "OR". According to Yager (1988), an OWA of n dimension is a mapping function $F:R^n \to R$ that has been associated to a weight vector $\omega=(\omega_1, \omega_2,\ldots, \omega_n)T$ where $\omega_i \in [0,1]$, $1 \le i \le n$, and $\omega_1+\ldots+\omega_n=1$. Furthermore $F(a_1,\ldots,a_n)=\omega_1 b_1+\ldots+\omega_n b_n$, where b_j with $j \in [1,n]$ is the jth element of the set $< a_1,\ldots a_n >$.

Thus, OWAs are aggregation operators that allow assigning a specific weight for each antecedent of an implication. This is the reason why we have chosen them for setting up the weight of the different characteristics involved in an evaluation.

3.2. Fuzzy assessment for algorithm structure

As we have mentioned above, the structure of an algorithm will determine if it is well coded following the concrete programming technique that the teacher considers interesting for the student. To calculate the degree to which the algorithm written by the student is similar to that specified *"ideal"* by the teacher will give us an idea about the correctness of its structure.

We start from the *fuzzy representation of the ideal algorithm* obtained in the way explained in Section 2.1.2. In that section, we have stated that the fuzzy sets must be normalised, that is, the membership value will be between 0 and 1. This means that a membership with value 1 will indicate that the metric coincides with the ideal algorithm. The other real values up to 0 will indicate the coincidence degree between the metric and the ideal algorithm. In this situation, a criterion we can apply is the arithmetic mean for the membership value of each software metric obtained from the source code with the corresponding fuzzy set. This will give us an idea about how close the algorithm is to the fuzzy ideal representation using as an aggregation operator an OWA where all the weights are the same for each antecedent in the implication. That is $\omega=(1/n, 1/n,\ldots, 1/n)T$.

If $\mu(x_i)$ is the membership degree for the software metric x_i calculated from the student's source code, the expression for calculating the similarity degree between that implementation and the fuzzy representation of the ideal algorithm will be given by

$$\sum \mu(x_i)/n \tag{1}$$

However, it is more common that the teacher gives more importance to some software metrics over others. In that case, the weight of the membership value will be more appropriate. So, taking into account the OWA definition, if a_i is the weight for the ith software metric, Eq. (1) will be transformed into the following expression:

$$\sum [a_i.\mu(x_i)]/\sum a_i \tag{2}$$

If it is that $a_1+\ldots+a_n=1$, then Eq. (2) can be expressed as it is shown in the following equation:

$$\sum [a_i.\mu(x_i)] \tag{3}$$

The weight for the membership value for each software metric can be achieved in two ways:

- *Exactly:* assigning a value from a range, for example between 0 and 1, between 0 and 5 or between 0 and 10. This explains the denominator for Eq. (2), that is, obtaining 1 as the sum of the weights, as the OWA operator definition requires.
- *Fuzzily:* assigning linguistics labels to concrete values from a scale, as is shown in Fig. 6. In this figure, we can see how to do the assignment of linguistics labels on a scale of $0-1$.

Thus far, we have explained how to do the evaluation for the structure of the algorithm. In the next subsection, we will show how to evaluate the correctness of that algorithm.

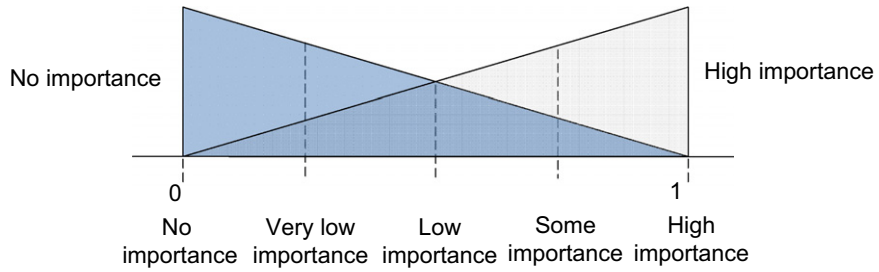

Fig. 6. Example of linguistics labels for weighing membership.

```
IF (a1 is x1) AND … AND (an is xn) THEN OUTPUT z=a1x1+    +anxn+k
```

Fig. 7. Takagi – Sugeno – Kang implication rule.

```
IF (x is A) AND (y is B) THEN z=K
```

Fig. 8. Takagi – Sugeno – Kang order 0 implication rule.

```
IF (control complexity is low AND cyclomatic complexity is high)
THEN
     show message ('The solution can be done easily by reviewing
                    the conditions and deleting some bifurcation')
```

Fig. 9. Example of rule for giving advice.

3.3. Method of checking algorithm correctness

To evaluate the correctness of the algorithm, we can apply a criterion based on a weighted average according to the relevance of each test case. As in previous cases, this weighting can be done exactly by directly assigning the weight, or fuzzily by applying linguistic labels in the same way as has been shown for the evaluation of the structure of the algorithm.

$$T(x_i) = \{1 \text{ if test is over; } 0 \text{ if test is not over}\} \quad (4)$$

Thus, if x_i is the ith test, $T(x_i)$ the function that determines if a test is over or not (as can be seen in Eq. (4)), and a_i the weight that the teacher considers right for the ith test, the function that represents the evaluation of the correctness can be expressed as shown in Eq. (5) in the case that the sum of the weights is other than 1, or the expression shown in Eq. (6) in the case that the sum of the weights is equal to 1:

$$\sum[a_i.T(x_i)]/\sum a_i \quad (5)$$

$$\sum[a_i.T(x_i)] \quad (6)$$

4. Method of providing advice

Up to now, we have shown a technique that allows providing an assessment that can be used as a feedback to the system, updating the cognitive student model and performing the corresponding adaption in the learning activity sequencing. Apart from this, it is possible to use the proposed technique as a tool for giving advice to the students to help them in the learning process. Thus, the system can define how good the algorithm written by the student is, explaining why and how it can be improved. That is, it is possible to implement a system for giving advice and suggestions to help the student understand what goes wrong and how to solve it.

From the point of view of the static analysis of the algorithm, this can be done by building a Fuzzy Logic rule based system. The rules will fire once a concrete situation is detected. In that way, if any calculated software metric differs from that specified in the ideal fuzzy representation, the system will advise the student on how to improve the algorithm. For example, if the solution written by the student is "quite similar" to the ideal solution in most of the metrics but one, for example conditional complexity, the system can suggest to the student how to review the conditions in bifurcations and loops because it looks more complex than necessary to solve the problem.

This kind of fuzzy rule follows the Takagi – Sugeno – Kang (Takagi and Sugeno, 1985) inference model. In these rules, the output in the implication is determined by a lineal function of the type $f(x_1,\ldots,x_n)=a_1x_1+\cdots+a_nx_n+k$, as shown in Fig. 7.

If $x_1=\ldots=x_n=0$ the output is a constant. This is the Takagi – Sugeno – Kang order 0 implication rule. Thus, if A and B are the antecedents fuzzy sets and k a non-fuzzy constant (*singleton* set), the rule of this type is of the form shown in Fig. 8.

With this, we have the appropriate mechanism for specifying rules that provide some explanation. In Fig. 9, we can see one of these rules specified by the teacher. The rule is fired when the control complexity is "low" and the cyclomatic complexity is "high", then the student will see the message: "*The solution can be done easily by reviewing the conditions and deleting some bifurcation*".

The main effort of this system would be implementing the knowledge base containing the fuzzy rules. In this sense, the teacher may suggest rules as shown in Fig. 9, or it is possible to apply some machine learning algorithm to detect certain situations, once we have enough cases to perform the machine learning process (Alcalá et al., 2000).

Furthermore, analysing correctness can also be used for giving advice. The nature of the test cases gives an idea about what goes wrong in the algorithm. As you can see in the *if – else* block in the test case of Fig. 5, if the output after invoking the algorithm is not correct or post-conditions not are not the expected, we can show a

message indicating the problem. This message can be representative both for the teacher and for the student, and it can indicate the way to solve the problem.

Hence, we have all we need to provide the students with some messages, allowing them to understand what goes right and wrong; so they can improve their algorithm. The proposal takes into account both the correctness and the structure of the algorithms.

5. Proposal application for algorithm assessment

In previous sections, we have exposed our proposal. Now, we are going to describe in detail the whole process, following an example with a subset of algorithms extracted from our early experimentations.

5.1. The teacher proposal for the ideal algorithm

The first thing the teacher will do is to write an approximate algorithm that he/she considers appropriate for solving the problem. Let us assume that the problem to solve is to return the *n*th element of the Fibonacci series using a recursive implementation.

The first thing we should do is to get an implementation by the teacher to solve the problem. Figure 10 shows the code written by the teacher in Java. We can see how it has an implementation where cyclomatic complexity is 2, since there are two execution paths in the code, namely one for the base case $n \geq 1$, and one for the recursive propagation case; the number of parameters passed by value is 1, the number of method calls is 2, coinciding with the number of recursive calls, the number of lines of code is 9, the operational complexity is 2.5, and the control complexity is 2.

5.2. Fuzzy representation of the ideal algorithm

Once the teacher has written the ideal algorithm and we have calculated the associated software metrics, we must adjust the values for the parameters *a*, *b*, *c* and *d* for each metric, as explained in Section 2.1.2. This will set the fuzzy abstraction of the ideal algorithm.

Table 3 shows the values set out by the teacher for each fuzzy set. On the other hand, Fig. 4 shows the graphs for the fuzzy sets labelled "normal", "low" and "high", obtained from the data in Table 3.

From this, we obtain the *fuzzy representation of the ideal algorithm* for each assignment. These fuzzy representations will be used in the automatic assessment. In this way, we obtain a collection of fuzzy sets that characterises the algorithm and allows us to know when a measured software metric extracted from an algorithm can be considered "normal", "low" or "high", and to what degree.

In Fig. 11, for each graphic, the *x* axis represents the values we can obtain for a software metric and the *y* axis shows the membership degree for the measure of the software metric for each fuzzy set. For instance, if the measured value for the McCabe Cyclomatic Complexity (at the top of Fig. 11) is 4, we obtain 1 for the membership value to the "normal" fuzzy set, and 0 to the "high" and "low" fuzzy sets. So, in that case, we can say that the value for the McCabe Cyclomatic Complexity is what the teacher considers "normal". As we increase the value for the metric, the membership value for the "normal" fuzzy set reduces and the membership value for the "high" fuzzy set increases. If we look at value 5 of the metric, we can see that the membership value for the "normal" and "high" fuzzy sets is equal to 0.5 in both cases, that is, the metric could be considered "normal" and "high" to the same degree, but not "low". In that case, we can say something like the measured metric is "a bit high". Moreover, if the value for the metric continues growing, it will come to a state where the membership value for the "normal" and "low" fuzzy sets will be 0, and 1 for the "high" fuzzy set. So we can say that the measured value is "high" according to the teacher.

As we can see, the fuzzy sets are normalised between 0 and 1. So, if a membership value is 1 then the metric is correct according to the ideal algorithm. The rest of the values up to 0 indicate the degree to which that metric is correct according to the ideal algorithm.

5.3. Test cases specification

Once we have the fuzzy representation of the ideal algorithm, the next thing we need to do is to specify the set of test cases that must pass the students' algorithms. Figure 12 shows the JUnit test cases that the teacher has specified for the Fibonacci assignment. In this figure, we can see three test cases with their corresponding

```
public static int FibRec(int n) {
    int retval=0;
    if (n <= 1) {
        retval=n;
    } else {
        retval = (FibRec(n - 1) + FibRec(n - 2));
    }
    return retval;
}
```

Fig. 10. Teacher's code to solve the Fibonacci assignment.

Table 3
Example of Parameters to build the Fuzzy abstraction of the Ideal algorithm.

	Cyclomatic complexity	Value parameters	Method calls	Recursive calls	Reference parameters	Lines of code	Operational complexity	Control complexity
A	1	0	0	0	0	3	0	2
B	2	1	0	0.5	1	4	0	3
C	2	3	2	1.5	1	6	1	3
D	3	4	3	2	2	7	2	4

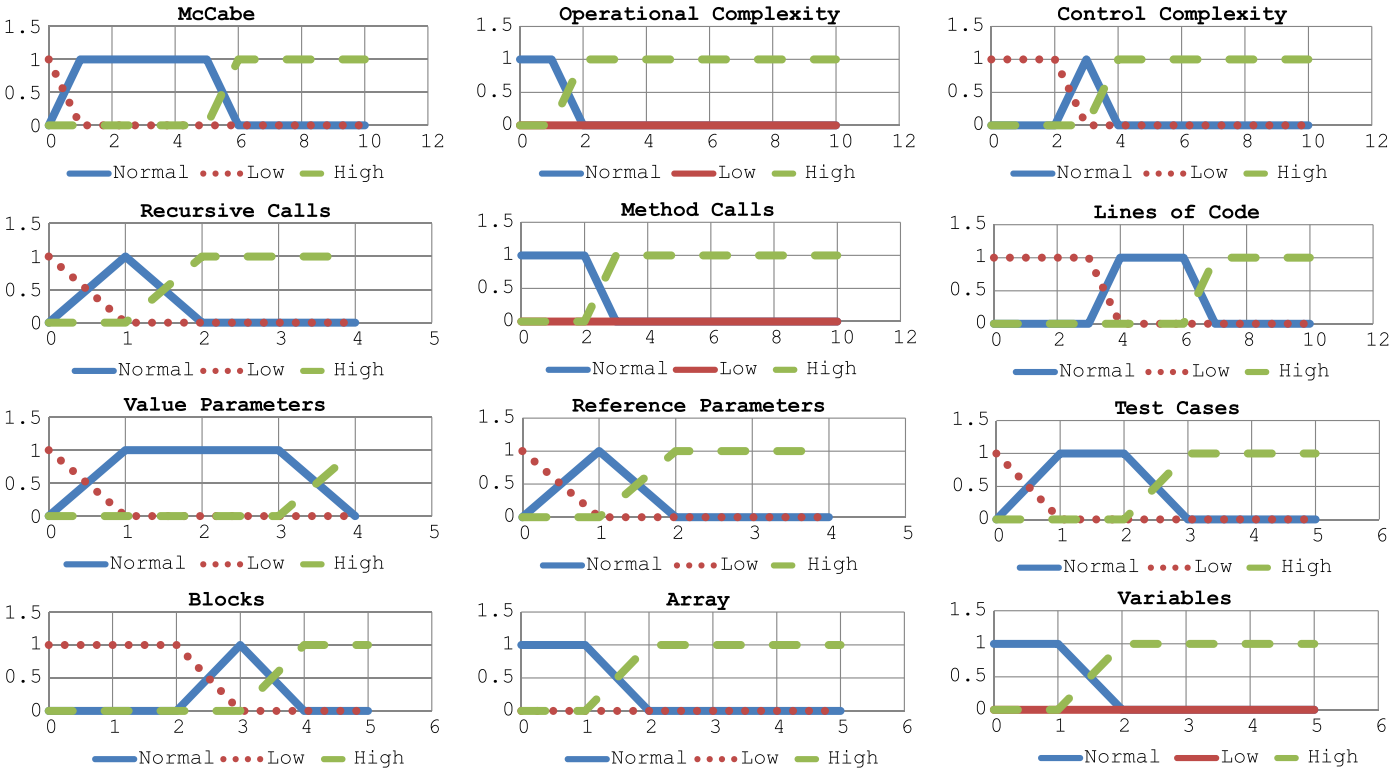

Fig. 11. Fuzzy sets for some calculated software metrics.

warning messages in case of failure, these being for the base case where $n=0$, for the base case where $n=1$ and for the recursive case.

5.4. Assessing the algorithms

Once all the settings have been loaded in the assessment module, we must deliver the assignments to the students. After finishing, the students upload the algorithms they have developed as solutions for the assignment and the assessment process starts.

Assessing the structural component of an algorithm means assigning a degree of similarity between the algorithm written by the student and the *fuzzy representation of the ideal algorithm*. The degree of membership of the metrics calculated from the algorithms written by the students to the fuzzy sets will give us an idea about how similar those algorithms are to the algorithm written by the teacher. Thus, in the next subsection we will describe in detail the whole process, following an example with a small data set.

5.4.1. Obtaining the software metrics from the students' algorithms

Table 4 shows the metrics obtained from the teacher's algorithm (Fig. 10) and from four students' algorithms, which implement, in Java language, a recursive algorithm that returns the *n*th element for the Fibonacci series. It can be seen how for each algorithm (rows) all the necessary software metrics (columns) are calculated.

5.4.2. Calculating the membership for each software metric to the normal fuzzy set in the fuzzy representation of the ideal algorithm

Table 5 shows the membership value for each software metric from Table 4 to the corresponding normal fuzzy set. As we have mentioned above, the membership value is between 0 and 1. Analysing Tables 4 and 5 together, we can see how the system assigns a membership value lower than 1 for those metrics in the students' algorithms that are different from the teacher's metrics in the terms the teacher has set. Notice, for example, the cyclomatic complexity (execution paths in the code). If we see Table 4, the students whose cyclomatic complexity is equal to the teacher's have obtained a membership value of 1 in Table 5 (*student* 4). However, students whose cyclomatic complexity is different to the teacher's have obtained a membership value lower than 1. Thus, students 1 and 2 have obtained a membership value of 0.5 due to having 3 in their cyclomatic complexity. Student 4 has a membership value of 0 due to his/her algorithm having 4 as its cyclomatic complexity, and that result is "high" as the teacher had specified.

5.4.3. Aggregating membership values: calculating the assessment for the structural part of the algorithm

Table 5 has a column with the global assessment for the structural analysis. This evaluation is normalised between 0 and 1. We can see how the system gives a 1 to the teacher's algorithm and how students obtain assessments depending on how good their algorithm is. The assessment criterion we have followed in this table is the arithmetic average for all the membership values (see Section 3). Following this, if $\mu(x_i)$ is the membership value for the metric x_i calculated from the code and n the number of metrics, the expression used for calculating the similarity between the student's algorithm and the fuzzy ideal representation will be given by Eq. (1).

However, it is usual that the teacher considers some metrics more relevant than others. In that case, we must weigh the membership values. Thus, if a_i is the weight for the membership of the metric ith, the applied expression will be Eq. (2). By applying weight average as an aggregation operator we obtain the data shown in Table 6. This table shows the weight between 0 and 10 that the teacher specified for each software metric (first row) and the $a_i\mu(xi)$ values, that is, the membership values after

```
import junit.framework.TestCase;

public class FibonacciTest extends TestCase{
  public void testFibonacci_BaseCase0() {
    int expected = 0, output = 0;

    output = Fibonacci.fibonacci(0);

    if(output != expected)
      fail("Error on base case for 0");
  }
  public void testFibonacci_BaseCase1() {
    int expected = 1, output = 0;

    output = Fibonacci.fibonacci(1);

    if(output != expected)
      fail("Error on base case for 1");
  }
  public void testFibonacci_Recursive() {
    int expected = 75025, output = 0;

    output = Fibonacci.fibonacci(25);

    if(output != expected)
      fail("Error on recursive case");
  }
}
```

Fig. 12. Test cases for the Fibonacci assignment.

Table 4
Example of metrics calculated from the teacher's algorithm and four students' algorithms.

	Cyclomatic complexity	Value parameters	Method calls	Recursive calls	Reference parameters	Lines of code	Operational complexity	Control complexity
Teacher	2	1	2	2	0	9	2.5	2
Student1	3	1	2	2	0	13	2	3
Student2	3	1	2	2	0	13	2	3
Student3	4	1	2	2	0	11	2	4.5
Student4	2	1	2	2	0	8	1.5	1.5

Table 5
Example of membership values for each software metric to their corresponding "normal" fuzzy set.

	Cyclomatic complexity	Value parameters	Method calls	Recursive calls	Reference parameters	Lines of code	Operational complexity	Control complexity	Global assessment
Teacher	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Student1	0.50	1.00	1.00	1.00	1.00	0.20	0.80	0.60	0.76
Student2	0.50	1.00	1.00	1.00	1.00	0.20	0.80	0.60	0.76
Student3	0.00	1.00	1.00	1.00	1.00	0.60	0.80	0.00	0.68
Student4	1.00	1.00	1.00	1.00	1.00	0.80	0.60	0.80	0.90

weighting. The last column shows the global assessment using Eq. (2). We can see how the assessment has been changed as regards the data obtained in Table 5. This is a direct consequence of applying weights, that is, it is due to considering some metrics more relevant than others.

5.5. Giving advice

Up to now we have analysed the proposal for assessing algorithms, so that they can be marked automatically by the system. However, as we noted previously, the proposed technique

Table 6
Example of How to Use Weights in the Assessment Process.

	Cyclomatic complexity	Value parameters	Method calls	Recursive calls	Reference parameters	Lines of code	Operational complexity	Control complexity	**Global assessment**
Weigh	5	5	10	10	5	1	2	2	
Teacher	5.00	5.00	10.00	10.00	5.00	1.00	2.00	2.00	**1.00**
Student1	2.50	5.00	10.00	10.00	5.00	0.20	1.60	1.20	**0.89**
Student2	2.50	5.00	10.00	10.00	5.00	0.20	1.60	1.20	**0.89**
Student3	0.00	5.00	10.00	10.00	5.00	0.60	1.60	0.00	**0.81**
Student4	5.00	5.00	10.00	10.00	5.00	0.80	1.20	1.60	**0.97**

```
IF cyclomatic complexity is "high"

THEN
     Show_message ('The method is more complex than expected, because
                    there are more paths and bifurcations than needed.
                    Suggestion: Review the conditions.');
```

Fig. 13. Example of fuzzy rule for giving advice.

```
public static int FibRec(int n) {
    int resul;
    if (n == 1) {
        resul = 0;
    }
    else {
        if (n == 2) {
            resul = 1;
        }
        else {
            resul = FibRec (n - 1) + FibRec (n - 2);
        }
    }
    return resul;
}
```

Fig. 14. Example code as input to the giving advice system.

can be used to give advice to students so that they know what they are doing right and wrong, and to improve their solution. This section attempts to validate the proposal for creating tools that provide explanation and suggestions for improving the learning process. This will help the students to understand what they have done wrong and how they could solve it, thereby being able to correct it and thus achieve significant learning.

To do so, we have defined a minimum set of fuzzy rules that have the form shown in Fig. 13. In this figure, a rule can be seen, which checks the membership degree of the cyclomatic complexity metric to the fuzzy set labelled as "high", as we defined in the fuzzy abstraction ideal algorithm.

Despite having a reduced set of rules, the experience allows us to see some consistent results. Thus, following the example of the Fibonacci assignment, applying the rules to the metrics calculated for student 1, whose code is shown in Fig. 14, the system will return the following output as an explanation and suggestion:

- *The method is complex and it establishes more branching paths than necessary. You should look for a simpler solution.*
- *The number of lines of code is high. The additional functionality could be separated in more methods.*
- *There are too many conditions or conditional statements are too complex. Conditions should be examined.*

Taking into account that both the example and the rules are simple, the system provides an explanation consistent with what any teacher can see in the code. This leads us to believe that the development of a knowledge base with a well-defined set of rules will provide an explanation and suggestion system useful in the student's learning process.

On the other hand, the test cases specified in Section 5.3 show that the number of executions is 3, the number of errors is 1 and the number of failures is 2. The difference between error and failure is that the errors are caused by bad programme implementation, while the failure indicates that the code has been executed, but its output was not correct.

After running the JUnit test cases, we can see the output message shown in Fig. 15. We can see that there has been a stack overflow while trying to run the test for the base case of 0. If we look at the code of Fig. 14, we see that the base case for 0 is not implemented. This is the cause of error.

Now, let us consider the failures. JUnit will show the following two messages:

- *"Error on base case for 1"*
- *"Error on recursive case"*

This will indicate to the students that something is wrong in the base case for $n=1$; so they must analyse it. Furthermore, the

```
FibonacciTest (3)
FibonacciTest
testFibonacci_BaseCase0(FibonacciTest)
java.lang.StackOverflowError
        at Fibonacci.fibonacci(Fibonacci.java:26)
        at Fibonacci.fibonacci(Fibonacci.java:26)
        at Fibonacci.fibonacci(Fibonacci.java:26)
...
```

Fig. 15. Output from JUnit.

recursive case has also failed, which may or may not be triggered by the earlier failure, but anyway it must be checked. Looking at the student's code again (Fig. 14), it shows that the student had not actually properly managed the case for $n=1$. Thus, this message is really useful to correct the mistake. On the other hand, the student's code returns 1 when $n=2$; so it leads to failure in the test case for the recursive call.

As we can see, the messages provided by the static analysis by means of fuzzy rules and those provided by the dynamic analysis tool are complementary. This allows the student to receive the relevant information to understand what is wrong and to correct it.

6. Proposal validation

As we have explained our proposal for evaluating and assessing programming assignments in the previous sections, we will now turn our attention to the validation process followed to corroborate the starting hypothesis. To do so, throughout this section we will show the environment we have developed for this purpose and based on the methods and techniques previously described; then we will describe the steps followed in an experiment for determining the validity of the proposal in an empirical way, and finally we will analyse the results.

6.1. The working environment

To develop our approach we have selected the widely available Eclipse platform (Eclipse, 2011). Eclipse is an integrated development environment, which allows creating extensions using its own API. Such extensions are implemented as *plug-ins* that can be optionally loaded by users. Eclipse is a fully fledged development environment that works with Java, C/C++ and other programming languages.

So, by means of *plug-ins*, we have customised Eclipse and created Computer Assisted Environment for Learning Algorithms (COALA) (Jurado et al., 2009, 2011) to allow (a) creating all the settings for the automatic assessment by including the implementation of a Fuzzy Logic module; (b) processing the algorithms and calculating the software metrics using the abstract syntax tree provided by the internal syntactic parser of Eclipse; thus we have a solution independent from the programming language; (c) providing the distribution and delivery mechanism for the assignments using a distributed architecture; (d) monitoring the whole process by analysing the corresponding data.

Figure 16 shows a screenshot of the developed *plug-in*, once integrated into Eclipse. The figure shows the appearance of the user interface that supports student activities. It displays the code written by the students, the test cases they can execute, the evaluation and explanation the system can give about the algorithm they have written as a solution to an assignment, the fuzzy representation that the teacher has specified, and the monitoring the teacher can perform.

6.2. Validating the proposal for assessing

In this subsection, we are going to expose the steps we have followed to perform the validation of the proposals using the COALA system. Then, we will analyse and discuss the results obtained.

6.2.1. Method

To validate the proposal we have developed an experiment with seven first-year students at the Faculty of Computer Science of the University of Castilla-La Mancha. These students have done several programming assignments in Java language that the system has assessed automatically. Then, we asked a teacher to evaluate the algorithms delivered by the students. With this, the method used to do the experimentation is as follows:

1. **Make the students' assignments.** As mentioned above, to make the assessment of the proposal we have selected a group of seven first-year students at the Faculty of Computer Science of the University of Castilla-La Mancha. These students have done five programming assignments to be solved by implementing an algorithm written in Java language. Thus, we deal with thirty-five cases to test our proposal.
   We have selected the assignments corresponding with the academic level of the students. Thus, the students are given the following assignments to develop:
   - Design and implement in Java language an algorithm that orders the elements of the vector passed as argument using the bubble method.
   - Design and implement in Java language a recursive algorithm that returns the factorial of an integer number passed as argument.
   - Design and implement in Java language a recursive algorithm that returns the *n*th element for the Fibonacci series.
   - Design and implement in Java language an algorithm that calculates the maximum element of an array passed as argument using the divide and conquer technique.
   - Design and implement in Java language a recursive algorithm that determines whether the given string passed as argument is palindrome.
2. **Do the manual assessment.** To prove the proposal, we asked a teacher of the programming subject in the Faculty of Computer Science of the University of Castilla-La Mancha to assess the algorithms delivered by students in two different ways:
   2.1 Subjectively, by assigning subjective global punctuation: For each algorithm the teacher provides a subjective assessment between 0 and 5. Thus, we will have the necessary data to contrast the automatic assessment against the teacher's subjective assessment. This will give us an idea about the need to include more criteria in our proposal.
   2.2 Manually, according to the adopted software metrics: For each software metric the teacher will provide punctuation

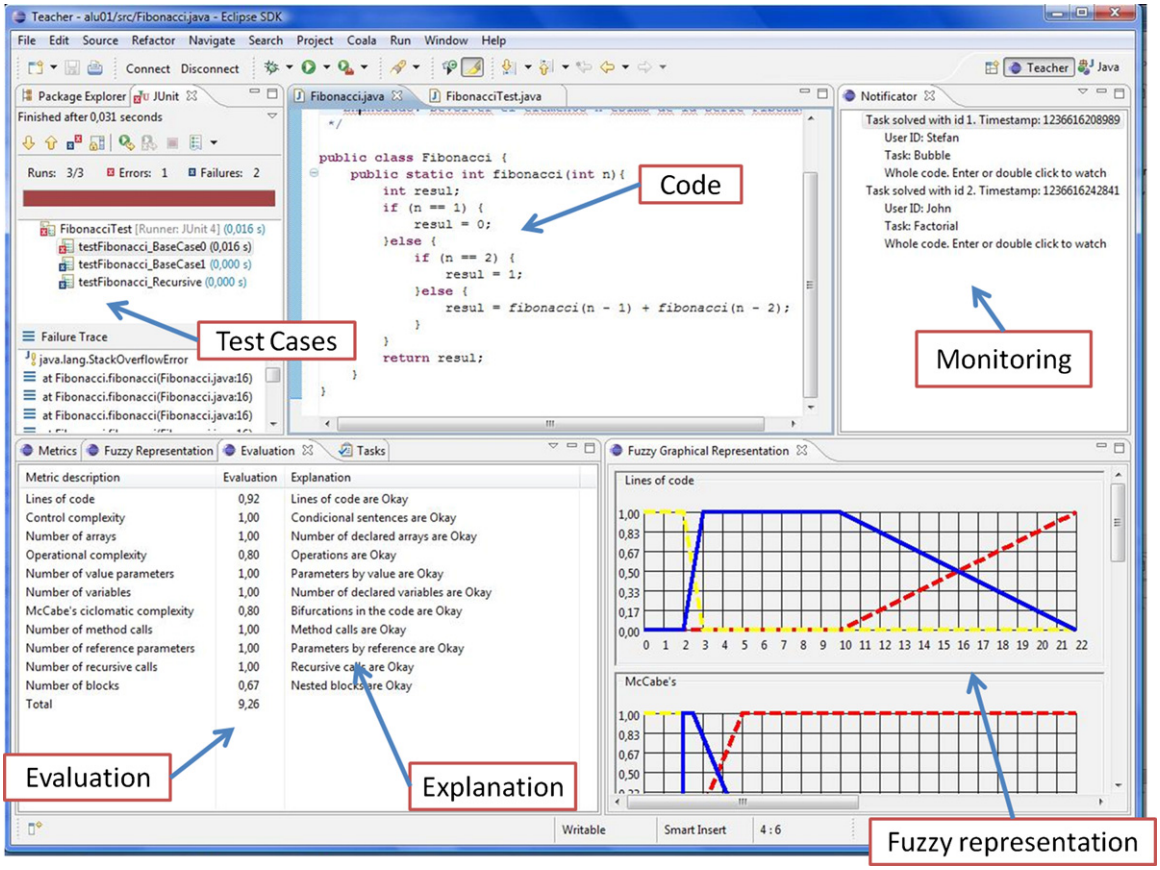

Fig. 16. Customised Eclipse environment.

between 0 and 5, where 0 is the worst assessment and 5 the best. Then, the OWA operator will be applied (see Section 3.1). This will provide us with the necessary data to contrast the automatic assessment against the teacher's manual assessment. As a consequence, it will give us an idea about how close the assessments are using the software metrics as criteria.

3. **Obtain the necessary information to perform the automatic assessment.** We ask the teacher for the necessary data to create the fuzzy abstraction, that is
   - 3.1 the code that implements the ideal algorithms that solve the assignments;
   - 3.2 the values he considers to be the minimum and maximum, which are absolutely correct, as well as the maximum low and minimum high for the solution (see section 2.3) and
   - 3.3 the weight for each software metric we have adopted in our proposal. With this, we have all the necessary data to create the fuzzy representation for each algorithm.
4. **Get the *fuzzy ideal algorithm representation***. In our implemented system, we loaded algorithms written by the teacher and set up all the parameters obtained from the previous step and so we obtain the fuzzy representation ideal algorithm. In this way, we get a collection of fuzzy sets that characterise the algorithms, and allows us to know when a software metric extracted from a student's algorithm may be considered "normal", "low" or "high" and to what degree.
5. **Assess the algorithms automatically.** Once all the necessary data are in the assessment module, the system is used for the distribution of tasks to students. After finishing, students deliver their solutions to the system.
6. **Contrast the assessments.** The final step is to compare the marks provided by the teacher with those calculated by the system. In cases where there was disagreement, the teacher was asked for an explanation about why he had assigned a different mark.

To discuss the results obtained in the experiment and analyse the similarity degree between the automatic marks and the manual marks is the purpose of the next section.

6.2.2. *Contrast of results*

The final step to acknowledge the correctness of our proposal is to contrast the teacher's assessments and the automatic assessment provided by the system. As we have indicated in previous sections, there are two *a priori* assessments done by the teacher (see Section 5.2.2). On the one hand, he assigned an assessment between 0 and 5 for each algorithm and then the OWA operator was applied (manual assessment); on the other hand, he provided a subjective global assessment for each algorithm (subjective assessment). Now we want to contrast the teacher's assessments against the automatic assessment. Contrasting the automatic assessment against the teacher's assessment will give us an idea about how close the assessments are using the software metrics as criteria. On the other hand, contrasting the automatic assessment against the teacher's subjective assessment will give us an idea about the need to include more criteria in our proposal.

Because the system provides a $0-10$ scoring system for the algorithms, we have normalised the teacher's assessments. Thus, we have obtained the results shown in Fig. 17. In this figure, the x axis represents the algorithms delivered by the students, enumerated from 1 to 35 (7 students with 5 assignments for each student), and the y axis represents the assessment for each algorithm (between 0 and 10). On the left side of the figure we can see the automatic assessment provided by the system; on the

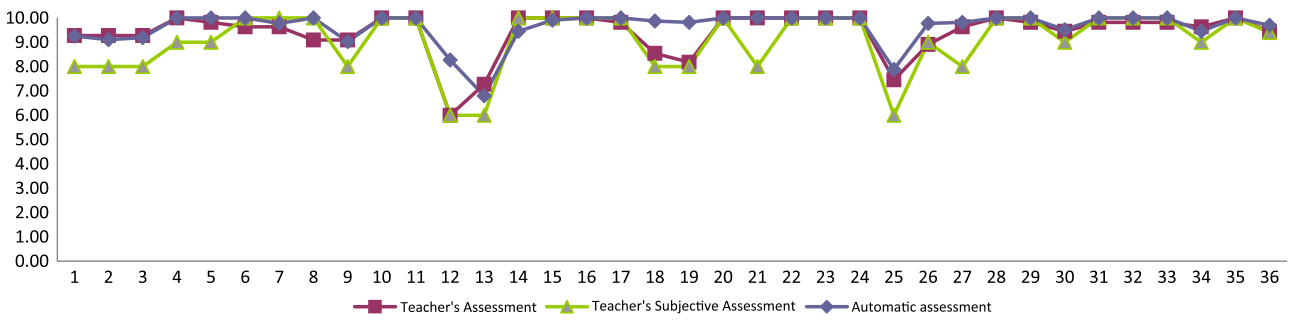

Fig. 17. Contrasting the obtained results.

right side we can see the teacher's assessment; at the bottom the subjective assessment provided by the teacher is shown.

6.2.2.1. Automatic assessment against teacher's assessment. In Fig. 17, we can see how *most of the assessments are quite similar*. However, it is interesting to point out cases 12, 18 and 19. On asking the teacher about those cases, we obtained the following answer:

- Case 12: *"The student has divided the functionality of the algorithm into several methods, so the software metric does not provide the right measure"*.
- Case 18: *"The student has used the unary operator "?" and has coded most of the functionality with few and complex lines. Instead of this, the student must use if-then bifurcations and write a clear implementation"*.
- Case 19: *"The student makes use of standard API methods, and then he gives some of the functionality he must implement on hands of the invoked method"*.

From the previews analysis we can extract:

- If there is a subdivision of the functionality of the algorithm in several methods, then software metrics calculation must process it.
- If the student writes complex and incomprehensible code, the system must be aware of it although the calculated software metrics could be the same.
- If the student makes use of API that helps him/her in the implementation, then the evaluation must be lower.

6.2.2.2. Automatic assessment against teacher's subjective assessment. As we can extract from Fig. 17, most of the teacher's assessments that take into account the software metrics are quite similar to the teacher's subjective global assessment. However, it is interesting to point out cases from 1 to 5, 21, 25 and 27. On asking the teacher about those cases we obtained this answer:

- Cases 1 – 5: *"The return value is not the property. It takes into account the correct bifurcations, operations, etc. but values returned are not correct"*.
- Cases 21 and 27: *"It is not a generalised solution because the lengths of the arrays are defined as constants used in loops"*.
- Case 25: *"The student uses Vector class from the standard API instead of an array. Furthermore he/she creates unnecessary additional methods and uses unnecessary type conversions"*.

From this we can extract:

- The first and second points will be easily detected by test cases on the correctness evaluation. So, it can be fixed by adding the test cases execution in the evaluation.
- The third point can be fixed as described above.

Table 7
Percentile range for the differences in the assessments.

Difference	Percentile range automatic vs. manual (%)	Percentile range automatic vs. subjective (%)
0	0.00	0.00
0.1	43.42	47.25
0.2	72.24	50.90
0.3	75.60	56.25
0.4	78.77	60.15
0.5	83.92	65.71
0.6	86.20	69.79
0.7	87.10	70.28
0.8	88.00	74.07
0.9	90.85	77.30
1	92.04	76.78
1.1	92.73	76.22
1.2	93.41	79.96
1.3	94.09	79.56
1.4	94.95	79.04
1.5	95.88	78.47
1.6	96.80	77.84
1.7	97.42	77.14
1.8	97.87	82.25
1.9	98.32	94.71

6.2.2.3. How close are the assessments. To check how close the assessments are, we have calculated the percentile range for the differences in the scoring system. So, Table 7 shows in the first column the differences between the marks and the second and third columns show the percentile range for the automatic assessment against the teacher's manual assessment and the automatic assessment against the teacher's subjective assessment. That is, the second and third columns will show the percentage of the cases where the difference between assessments is lower than the value indicated in the first column.

Thus, it is significant to see how we have obtained that 43.42% and 47.25% of the cases have a difference lower than 0.1 for the automatic assessment against the teacher's manual assessment and the automatic assessment against the teacher's subjective assessment, respectively. Furthermore, for 92.04% and 76.78% of the cases we have obtained a difference lower than 1.

Graphically, this data is shown in Fig. 18. This figure shows the percentile range for the differences between automatic assessment and teacher's manual assessment with a solid line, and the differences between automatic assessment and teacher's subjective assessment with a dashed line. According to what we have discussed above, in this figure we can see how the automatic assessment is closer to the teacher's manual assessment than to the teacher's subjective assessment. These results can be considered satisfactory if we compare them to the achievements outlined by other proposals (Naudé et al., 2010) where 65% of submissions received automatic assessment scores within 1 point of the teacher's assigned scores.

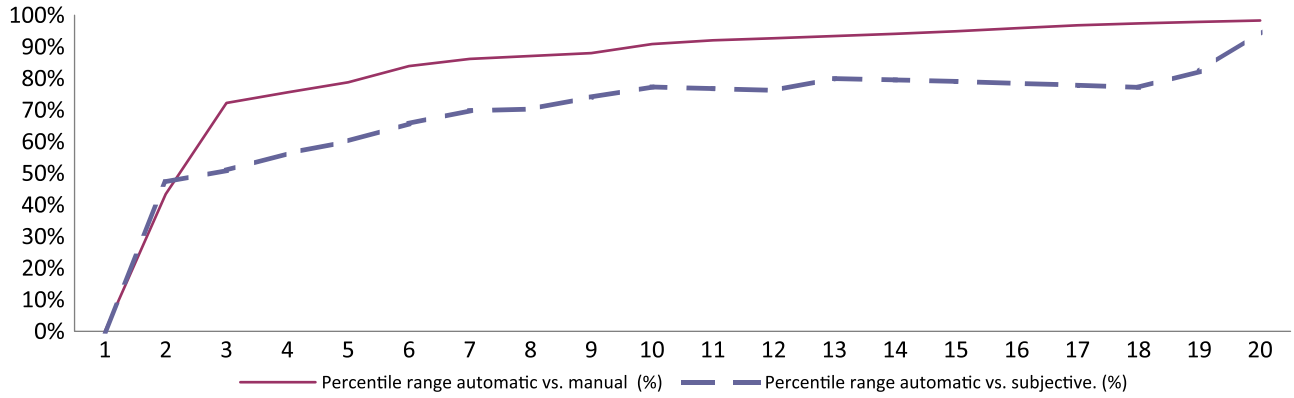

Fig. 18. Percentile range for the differences in the assessments.

```
IF cyclomatic complexity is "high"
THEN
     Show-message("Cyclomatic complexity is high,
                  because there are more path and bifurcations than needed.
                  You should implement a simpler solution");
```

Fig. 19. Example rule for the explanation and advice system.

```
public static int FibRec(int n) {
    int result;
    if (n == 1) {
        result = 0;
    }
    else {
        if (n == 2) {
            result = 1;
        }
        else {
            result = FibRec (n - 1) + FibRec (n - 2);
        }
    }
    return result;
}
```

Fig. 20. Input source for the explanation and advice system.

6.3. Validating the proposal for giving advice

In previous sections, we have analysed the proposal for its application in assessing algorithms. However, as we have pointed out in Section 4 that the proposed technique can be used not only to evaluate algorithms, but also to give advice. In this section, we try to validate the applicability of the proposal for building tools for explanation and suggestions, and in this way to improve the student's learning process. The aim is to build systems to help the learner to understand what goes wrong and how they could fix it, and in that way to be able to correct it and achieve significant learning.

To do this, we have designed a minimum set of fuzzy rules as shown in Fig. 19. In that figure, we can see a rule that checks the membership degree for the "high" fuzzy set of the cyclomatic complexity of the fuzzy ideal algorithm.

Despite the fact that the rule is quite simple, the experience yielded the following results: following the example of the Fibonacci algorithm, after applying the fuzzy rules on the calculated metrics for student 1, whose code is shown in Fig. 20, the system returned the following output as explanation and advice:

- *"Cyclomatic complexity is high, because there are more paths and bifurcations than needed. You should implement a simpler solution."*
- *"The number of lines of code is high. Maybe you could divide the functionality by adding methods."*
- *"Conditional complexity is high, because there are too many conditions or sentences are too complex. Conditions should be examined."*

In Truong (2007) 63% of the students thought that the analysis was useful in providing them with feedback about how their solutions compare to the model solution. In our approach, improving the knowledge base with a set of well-defined rules could provide an explanation and suggestions, which are useful in the learning process. However, a study must be performed in order to analyse it.

6.4. Discussion

From the point of view of the proposal for assessing the algorithms, differences between the teacher's assessment and those performed automatically by the system are because the teacher had taken into account issues such as coding style and readability, division of functionality, use of API, returned values, etc. That is, the teacher had taken into account a number of restrictions and requirements that the system still does not take into account. These elements help the teacher find out how good the algorithm written by a student is. As a solution, we can point out that by taking into account the test cases in the assessment, it will be possible to close the gap between the teacher's assessment

and the automatic assessment. Nevertheless, the results suggest that the proposal is consistent, but we must continue working on it to consider other aspects and parameters.

On the other hand, from the point of view of the proposal for giving advice, taking into account the fact that both the example and the rules are simple, the system provides an explanation that is consistent with that can be seen in the code. This leads us to think that the development of a knowledge base with a set of well-defined rules, and the application of the technique described here, could provide an explanation and suggestions useful in the learning process. However, this is already beyond the scope of work described in this article.

7. Concluding remarks and future work

Throughout this article, after having introduced the motivation for this research, we have focused our work on those systems for learning programme algorithms, showing the state of the art in the analysis of algorithms in different programming disciplines, identifying the techniques and tools used and setting up the working hypothesis and objectives to be achieved.

In the remaining sections, we have described how we have worked in achieving the goals to prove our starting hypothesis. Thus, we have detailed a proposal about how to assess the structural part of an algorithm by means of obtaining a fuzzy ideal representation. The fuzzy representation is obtained by making the fuzzy representation of several software metrics, calculated from an ideal algorithm written by a teacher. Furthermore, how to assess the correctness of the algorithm has been pointed out, taking as a basis the use of automated techniques for testing software.

To perform the corresponding experimentation, we have developed the system called COALA. This system implements the necessary components to use the proposal in the context of an Eclipse based development environment.

Once the proposal is exposed, its validation has been shown. Thus, the procedure we have followed for validating the proposal has been explained in detail, as well as an analysis of the obtained results. These results suggest that the proposal is consistent, but still needs to be extended to consider other aspects and parameters.

Taking all this into consideration, it can be concluded that we have contributed with the goal of designing a technique that analyses algorithms, taking into account both their static part (structure and shape), and their dynamic part (correctness). So, the proposal can be used in the implementation of Intelligent Tutoring Systems for teaching/learning algorithm programming.

Thus, we have proved that is possible to use test cases and Fuzzy Logic for software metrics to assess the algorithms written by students and to give advice. From the results obtained in our early experiments, it seems that it is possible to use Fuzzy Logic applied to software metrics to evaluate the algorithms written by students, obtaining results close to those provided by a teacher. Hence, we could say that: ***"it is possible to automatically assess the algorithms developed by the students and to give advice to them, applying Fuzzy Logic to the software metrics and test cases"***.

Thus, it can be concluded that we have managed an analysis technique that takes into account its static and dynamic parts, that is, its structure and correctness. So, the proposal may be used in Intelligent Tutoring Systems for the teaching/learning process for the programming competence. Furthermore, the validation shown in the article leads to some further steps in our research to improve the proposal:

- To study the behaviour of the system with a larger number of cases. In this sense, the system must be used by a large amount of students and several teachers must assess each algorithm delivered by the students. This will provide us with interesting information about how our proposal fits the evaluation process.
- To take into account more features in our analysis, that is, the definition of a set of restrictions and requirements that the teacher can consider.
- To apply machine learning techniques to teacher's evaluations. It would be interesting to use the system implemented to extract and generalise rules and to create a knowledge base for evaluating algorithms. That is, build a set of fuzzy rules that help determine whether an algorithm is correct based on the fuzzy representations of algorithms stored in the system.
- To perform a comparative study with other related works in order to check the accuracy of the proposal.

With all this in mind, we expect to have a complete system of automatic evaluation of computer programming algorithms. This will allow creating tools for Intelligent Tutoring Systems that facilitate the implementation of modules for giving advice to students and provide a feedback mechanism that allows the system to modify the students' profile and provides a pedagogical adaption.

References

ACM/IEEE. Computer science curriculum 2008: an interim revision of CS 2001. Report from the Interim Review Task Force; 2008.

ACM/IEEE. Curriculum guidelines for undergraduate degree programs in information systems. Report from the Interim Review Task Force; 2010.

Ala-Mutka K, Uimonen T, Järvinen HM. Supporting students in C++ programming courses with automatic program style assessment. Journal of Information Technology Education 2004;3:245–62.

Ala-Mutka K. A survey of automated assessment approaches for programming assignments. Computer Science Education 2005;15:83–102.

Alcalá R, Casillas J, Cordón O, Herrera F, Zwir I, Leondes CT, editors. Knowledge-based systems: techniques and applications. Chapter learning and tuning fuzzy rule-based systems for linguistic modelling. Academic Press; EE.UU.; 2000. p. 889–941.

Andrade J, Ares J, García R, Rodriguez S, Seoane M, Suárez S. Guidelines for the development of e-learning systems by means of proactive questions. Computers & Education 2008;51:1510–22.

Ben-Ari M. Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching 2001;20:45–73.

du Boulay B. Some difficulties of learning to program. In Soloway E, Sprohrer J. Studying the Novice Programmer 1989:283–300. Lawrence Erlbaum Associates.

Bravo C, van Joolingen WR, de Jong T. Using co-lab to build system dynamics models: students' actions and on-line tutorial advice. Computers & Educaction 2009;53:243–51.

Brusilovsky P, Schwarz EW, Weber G. ELM-ART (1996). An intelligent tutoring system on world wide web, ITS '96. In: Proceedings of the third international conference on intelligent tutoring systems; 1086, p. 261–9.

Brusilovsky P, Calabrese E, Hvorecky J, Kouchnirenko A, Miller P. Mini-languages: a way to learn programming principles. Education and Information Technologies 1998;2:65–83.

de Barros LN, dos Santos Mota AP, Delgado KV, Matsumoto PM. A tool for programming learning with pedagogical patterns. In: eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology exchange. New York, NY, USA: ACM; 2005. p. 125–9.

Eclipse, online 〈http://www.eclipse.org〉, last visited July 2011.

Fernandes E, Kumar A. A tutor on subprogram implementation. Journal of Computing in Small Colleges 2005;20:36–46.

Garner S. Learning resources and tools to aid novices learn programming. In: Proceedings of the informing science & information technology education joint conference (INSITE); 2003. p. 213–22.

Gomes A, Mendes AJ. Learning to program—difficulties and solutions. In: Proceedings of the international conference on engineering education—ICEE; 2007.

Gordijn J, Nijhof WJ. Effects of complex feedback on computer-assisted modular instruction. Computers & Education 2002;39(2):183–200.

Gray A, Sallis P, MackDonell S. Software forensics: extending authorship analysis techniques to computer programs. In: Proceedings of the 3rd biannual conference of the international association of forensic linguists (IAFL). International Association of Forensic Linguists (IAFL); 1997.

Joy M, Griffiths N, Boyatt R. 'The boss online submission and assessment system'. Journal on Educational Resources in Computing (JERIC) 2005;5(3):2.

Joy M, Luck M. Plagiarism in programming assignments. IEEE Transactions on Education 1999;42(1):129–33.

Jurado F, Molina AI, Redondo MA, Ortega M, Giemza A, Bollen L, Hoppe HU. Learning to program with COALA, a distributed computer assisted environment. Journal of Universal Computer Science 2009;15:1472–85.

Jurado F, Molina AI, Redondo MA, Ortega M. A practical case of agents and services integration in e-learning environments by means of tuple spaces. In: Demetrios G, Sampson Antonio Sarasa-Cabezuelo Ignacio Aedo-Cuevas K, Sierra-Rodríguez J-L, editors. CETIS UPGRADE, Vol. 2011; 2011. p. 51–8.

Kelleher C, Pausch R. Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. ACM Computing Surveys 2005;37:83–137.

Kilgour R, Gray A, Sallis, P, MacDonell S. A fuzzy logic approach to computer software source code authorship analysis. In: Proceedings of the 1997 international conference on neural information processing and intelligent information systems; 1997. p. 865–8.

Kumar AN. Learning programming by solving problems. Informatics Curricula and Teaching Methods 2002:29–39.

Kumar A. Using online tutors for learning—what do students think? In: Proceedings of frontiers in education conference (FIE 2004); 2004).

McConnell JJ. Active learning and its use in computer science. SIGCUE Outlook 1996;24:52–4.

Naudé KA, Greyling JH, Vogts D. Marking student programs using graph similarity, Computers &. Education, Vol. 54. Elsevier Science Ltd.; 2010. p. 545–61.

Nejdl W, Wolpers M. KBS hyperbook—a data-driven information system on the Web. In: Proceedings of the WWW8 Conference; 1999.

Pérez JRP, del Puerto Paule Ruiz M, Lovelle JMC. SICODE: a collaborative tool for learning of software development. In: Proceedings of the IV international conference on multimedia and information & communication technologies in education (m-ICTE2006); 2006.

Rahman, KA, Nordin MJ. A review on the static analysis approach in the automated programming assessment systems. In: Proceedings of the national conference on programming 07; 2007.

Sahraoui HA, Boukadoum MA, Lounis H. Using fuzzy threshold values for predicting class libraries interface evolution. In: Proceedings of the 4th international ECOOP workshop on quantitative approaches in object-oriented software engineering. 2000.

Sallis P, Aakjaer A, MacDonell S. Software forensics: old methods for a new science. In: Proceedings of the 1996 international conference on software engineering: education and practice (SE: E&P'96); 1996. p. 481–5.

Sanders D, Hartman J. Assessing the quality of programs: a topic for the CS2 course. In: SIGCSE '87: Proceedings of the eighteenth SIGCSE technical symposium on Computer science education. New York, NY, USA: ACM; 1987. p. 92–6.

Schofield JW, Eurich-Fulcer R, Britt CL. Teachers, computer tutors, and teaching: the artificially intelligent tutor as an agent for classroom change. American Educational Research Journal 1994;31:579–607.

Schollmeyer M. Computer programming in high school vs. college. In: SIGCSE '96: Proceedings of the 27th SIGCSE technical symposium on computer science education; 1996. p. 378–82.

Song J, Hahn S, Tak KY, Kim JH. An intelligent tutoring system for introductory C language course. Computers & Education 1997;28(2):93–201.

Takagi T, Sugeno M. Fuzzy identification of systems and its applications to modeling and control. IEEE Transactions on System, Man, Cybernetics 1985;15:116–32.

Truong N, Roe P, Bancroft P. Static analysis of students' Java programs. In: Raymond Lister, Alison Young, editors. Proceedings of the Sixth Australasian computing education conference (ACE 2004), Dunedin, New Zealand, January 18–22, 2004', Australian Computer Society; 2004. p. 317–25.

Truong N. A web-based programming environment for novice programmers. PhD thesis. Queensland University of Technology, Australia; 2007.

Yager R. On ordered weighted averaging aggregation operators in multi-criteria decision making. IEEE Transactions on Systems, Man and Cybernetics 1988;18:183–90.

Zadeh L. Fuzzy sets. Information and Control 1965;8:338–58.