Trace Alignment for Automated Tutoring

Michael Striewe, Michael Goedicke
University of Duisburg-Essen

michael.striewe@paluno.uni-due.de

Abstract

In automated tutoring systems for programming exercises, feedback can be given by comparing a student solution to a sample solution. Running test cases on both of them can give hints on flaws, but cannot look inside methods or functions, as long as they are handled as black boxes. This paper suggests to compare full traces instead of just output in order to provide more detailed feedback. A simple heuristic approach for trace alignment is described and evaluated in this paper.

Automated analysis and grading of programming exercises is a common feature in modern e-learning scenarios in computer science education. Several techniques known from automated testing are in use in automated tutoring systems. Running test cases against a student's solution is one of the easiest ways to generate feedback on the solution. However, this approach is limited to a comparison between actual output and sample output as long as black-box-tests are used. Moreover, first-year students may be inexperienced in understanding complete program behavior if they are only informed about the input and output of a particular test case. In an earlier publication (Striewe and Goedicke (2011)), it was proposed to create full traces of program behavior automatically while running test cases to make the program behavior visible to students. This approach has been used in practice since then and was accepted as helpful by the students. However, it was left to the students to read the traces and draw the right conclusions. Hence this paper continues with this idea and suggests to compare traces automatically in order to generate more direct feedback to students or teachers.
This idea is stimulated by the fact that a program trace is basically a sequence of steps of program behavior, where each step is characterized by the values of the scoped variables. Thus heuristic techniques known from sequence alignment can be applied to match two traces with each other, finding out whether these two traces represent a rather similar or rather different. Basically, a human reader of a trace would do the same thing, comparing the actual values in the trace with the conceptual ideas on what should have happened.

Besides spotting errors, the comparison between a student's solution and a sample solution can also be used to spot unnecessary code. This is code that does not lead to an error, but performs operations which do not contribute to the purpose of the program. For example, these might be operations on variables not needed or extra loops on a set of elements that do not change anything. These extra lines of code are identified during the alignment by being mapped to gaps in the trace of the sample solution. Consequently, lines of code that create these steps in a trace can be marked as potentially superfluous, being reported to the students as automated feedback as well.

Related Work

Traces as such have been studied in depth for decades (Diekert and Rozenberg (1995)), including verification of program or model behavior. Also automated generation and analysis of traces for programs is a commonly used technique in several fields of software engineering, e.g. model recovery (Moaz (2009)), adaptive systems (Ulam et al. (2004)), run time model checking (Visser et al. (2003)) and of course debugging. It is also used in some e-assessment systems for counting execution steps to gather performance measurements (e.g. Jackson and Usher (1997)), but not for automated feedback on particular errors. Algorithms for comparing strings and sequences are well known from various fields, including application areas like computational biology (Gusfield (1997)). To the best of the authors' knowledge, no e-assessment system exists that generates automated feedback based on the comparison of program traces.

Run time analysis without explicit tracing but based on symbolic execution, invariant generation and theorem proving is used in industrial approaches for infinite loop detection (Burnim et al. (2009), Gupta et al. (2008)). Symbolic execution is also used in some game-like programming tools (Tillmann et al. (2011)), where new test cases are automatically derived from the results from previous ones. However, neither traces nor hints referring to these traces directly are presented to the programmer.

Generating Traces

Means for generating run time traces are usually provided by the execution environment in which the program is executed. The approach presented in this paper is based on Java (version 6), thus using the Java Debugging Interface (JDI) for retrieving tracing information. However, the approach is not limited to Java, but can be applied to other programming languages.

An important aspect in tracing is the choice of granularity. For Java, there is a choice between fine-grained byte code steps and more coarse-grained line based stepping through the source code. Both cases have individual weaknesses: Line based stepping produces different results for the same program if the source code layout is changed by splitting statements over several lines or merging several lines into one. Byte code based stepping may produce different results for the same program if different compilers have been used, applying different byte code optimizations. The results presented in this paper have been produced using line based stepping because this way of trace generation is
reproducible by students using a debugger. However, the general approach is independent from this choice and can also be used with byte code based trace generation.

Trace recording using the JDI happens by suspending the Java Virtual Machine (JVM) after each single step execution, reading any available variable values from the stack frame and the object heap, and resuming the JVM for the next step execution. See Figure 1 for a part of the source code of a programming exercise. Figure 2 shows the trace generated for this method using the array \{(1,2,1),(4,3,2),(2,2,7)\} as input parameter.

```
219 public static float[] arithmetic_average(int[] [] mat) {
220   float [] vec = new float [mat.length];
221   float temp=0;
222   for (int i=0;i<mat.length-1;i++){
223     for (int j=0;j<mat[0].length-1;j++){
224       temp+=mat[i][j];
225     }
226     temp=temp/mat[0].length;
227     vec[i]=temp;
228     temp=0;
229   }
230   return vec;
231 }
```

**Figure 1.** Piece of source code cut out from a programming exercise. The method is supposed to return the arithmetic average of each row of a quadratic array. See Figure 2 for a trace created by this solution.

**Figure 2.** Trace generated for the source code shown in Figure 1, when invoked for the array \{(1,2,1),(4,3,2),(2,2,7)\}. If a cell is empty, the respective variable is not scoped by the current step.
Trace Alignment

Once traces for different solutions have been created using the same test case, they can be compared in order to spot similarities and differences.

Preliminary Considerations

Considering programming exercises, not each and every comparison of two traces is meaningful. If for example the task is to sort elements of an array, students may be free to implement different sorting algorithms. Comparing a student's trace based on Quick Sort with a sample trace based on Bucket Sort can be expected to be largely useless, because there will be many deviations. Consequently, it is considered useful to perform a static analysis of the source code first, trying to find significant bits of source code revealing the strategy used in the program.

If traces are compared, two main questions have to be answered: (1) Which variable in the first trace corresponds to which variable in the second trace, and (2) which step in the first trace corresponds to which step in the second trace. The approach thus consists of three stages:

1. Split each trace into columns for each variable used within that trace and find the best match for each column.
2. Find the best match for each step in the traces using only columns matched in stage 1.
3. Give a rating to each step in the traces based on the matching found in stage 2.

The stages are discussed in more detail in the subsections below. Afterwards, evaluations will be given based on actual data from an existing e-assessment system.

Identifying Variables

In general, it cannot be assumed that traces from two different solutions use the same variable names in identical situations. In addition, even the type of two variables need not to be the same although they are used for the same purpose. Thus variables have to be identified based on the change of their values over time. For this purpose, each trace is split into columns for each variable. Each column from the first trace is then aligned with each column of the second trace using a modified version of the algorithm by Ukkonen (Ukkonen (1985)). This algorithm allows for alignment of sequences of characters (where each character is a variable value in our case) based on a scoring function for matches allowing alignments with gaps. The latter is important, because two traces may contain the same steps with one trace having some extra steps in between.

The scoring function adds or subtracts the following scores on matching two variable values: If the values are equal (ignoring the type of the variable), 1 is added to the score. If the variables are not in the scope of the current step or are null-pointers, the score remains unchanged. Otherwise, the score is reduced by 1. Since types of variables have been ignored so far, an extra check on variable types is made in addition: If the types match exactly, 1 is added to the score. If types differ just in precision (as double and float do), 1 is added to the score, too. In all other cases, the score is reduced by 1. The gap penalty used in the algorithm is -1. Consequently, a match on value and type gains a score of 2, a match on either value or type gains a score of 0, a mismatch of both value and type gains a -2 and a gap gains a -1. The algorithm thus prefers partial maps over gaps as well as gaps over complete mismatches.
Based on the individual scores for each alignment, variables are identified by maximizing the sum of all scores for the matched columns. If the number of variables in the two traces differs, only the smaller set of variables is considered. The remaining columns of the larger trace are then discarded and not used throughout the remaining process. In addition, a threshold can be defined to discard variables with bad scores in general.

For an intermediate evaluation, each solution in a set of 51 unique solutions to a programming exercise has been aligned with the trace of a sample solution. The sample solution (see Figure 1) contains five relevant variables: One 2-dimensional array of integers (which was not supposed to change during the execution), two integer variables for line and row indices on the array being used in loops, one 1-dimensional array of type double, and one auxiliary variable of type double. The latter is not absolutely necessary, so some of the student's solutions contained less variables. Others contained additional variables or the same number of variables but with different meanings. Finally, some of the solutions defined the same set of variables as the sample solution.

Every variable matching has been inspected manually to find out whether it is correct. A variable matching is considered correct, if the algorithm mapped two variables which indeed are used for the same purpose in the programs. 45 out of 51 variable matchings had been considered correct, while in 6 out of 51 cases at least one variable was matched that should not have been matched. The threshold turned out to be largely ineffective, since it ruled out a variable mapping just in one case. Raising the threshold provided better results for the mapping, but worse results for the subsequent stages.

Identifying Steps

In the second stage of the overall process, an alignment for the two traces as a whole (rather than for individual columns as in stage 1) is calculated. If the length of the input traces differs by more than the factor 10, no matching is calculated, because results would be largely random in these cases. Again a modified version of Ukkonen's algorithms is used for the calculation, now using the complete set of variable values in one step as a single character in the sequence. The scoring function computes the score for a matching on steps by iteration over all variable values contained in this step: For each variable that matches by value, the score is increased by 2. For each variable that is not in the scope of the current step in both traces, the score is increased by 1. For each variable that is not in the scope of the current step in only one of the traces, the score is decreased by 1. In all other cases, the score is decreased by 2. Gaps do not receive a penalty, but a score which is the number of variables divided by 3 in an integer division. Thus the algorithm tries to maximize the overall score for the traces by aligning steps that contain as much similar variable values as possible, but prefers gaps in favor of alignments with poor matching.

See Figure 4 for a student's trace and Figure 2 for the sample trace. The first step (matching the variables) in this example decided to ignore variables rows and cols. Consequently, the first line gets a score of 6, because it has one perfect match for variable mat and four none-scoped variables. The next two lines are mapped to gaps, so that the fourth line of the student's solution can be mapped to the second line of the sample solution, receiving a score of 7. Next lines are mapped as well, getting a score of just 3, while line six of the student's solution mapped to line four of the sample solutions receives a score of 9.

In contrast to stage 1, no post-processing is necessary, but the alignment is directly used as input for the next stage. No manual inspection for evaluation was done, because it cannot clearly be said how to consider an alignment correct. It can only be said that an alignment of two traces is sufficient, if it allows to identify significant deviations between the traces. Since this is the goal of the next stage, evaluation is deferred to this stage.
Trace Alignment for Automated Tutoring

```java
public static float[] arithmetic_average(int[][] mat) {
    int rows = mat.length;
    int cols = mat[0].length;
    float[] v = new float[mat.length];

    for (int i = 0; i < rows; i++) {
        int temp = 0;
        for (int j = 0; j < cols; j++) {
            temp += mat[i][j];
        }
        v[i] = (float)(temp / 3);
    }
    return v;
}
```

**Figure 3.** Piece of source code cut out from a student’s solution for the example exercise. See Figure 4 for a trace created by this solution.

**Figure 4.** Trace generated for the source code shown in Figure 3, when invoked for the array `{ (1, 2, 1), (4, 3, 2), (2, 2, 7) }`. The first encounter to source code line 173 is marked as candidate step and indeed contains the relevant error by applying the type cast to float at the wrong position in the integer division.

### Rating

The last stage of the overall process is to rate each step in a trace based on the alignment calculated in the previous stage. The goal is to identify steps that cause a significant deviation between the two traces. Candidates are steps where significant differences between the scores before this step and the scores after this step can be observed.

A simple heuristic approach for this task works as follows: For step \( n \), the number of variables is counted (\( \text{count}_n \)) and also the number of variables that got a positive score during stage 2 (\( \text{match}_n \)). The numbers for step \( n \) are then compared to the numbers for the next step \( n+1 \) (thus \( \text{count}_{n+1} \) and \( \text{match}_{n+1} \)). If \( \text{match}_{n+1} - \text{match}_n < 0 \) is true, then step \( n \) is rated as a candidate step.

If a correct sample solution and a faulty student’s solution are compared, candidate steps found in the student’s solution are likely to stem from lines in the program that contain...
errors, if they are not neutralized by a subsequent step in which \((\text{match}_{n+1} - \text{match}_n) - (\text{count}_{n+1} - \text{count}_n) > 0\) is true. If such a neutralizing step is found, then the deviation between student's solution and sample solution is only temporarily and does not hint to an error. Thus the candidate step starting this temporarily deviation can be ignored. Note lines four to six in the case discussed above as an example.

**Evaluation**

The method for evaluating the overall approach described above follows straightforward from the results obtained: All traces and their remaining candidate steps can be inspected manually. If the lines in the program code corresponding to these candidate steps do contain errors indeed, the rating is considered correct (true positive). If some line in the program code corresponding to a candidate step does not contain an actual error, the rating is considered incorrect (false positive). If the code contains at least one error, but none of them is marked as candidate step, the rating is also not correct (false negative). The case of true negatives is irrelevant here, since for correct solutions no traces need to be analyzed at all. See Figure 3 and Figure 5 with their traces given in Figure 4 and Figure 6 for two examples of faulty student's solution. The sample solution they are compared to is the one shown in Figure 1 and Figure 2.

Quality of the approach can be measured by calculating precision and recall for a given set of traces. Two things can be observed in advance: First, there may be cases in which some line is marked as candidate step, which does not contain an error, while other lines containing an error are not marked. These traces consequently count as false positive and false negative results at the same time. Second, a remarkable number of false negatives can be expected due to the fact that traces are not compared if their lengths differ too much. Such a difference may hint to the fact that the two programs are using different strategies, so it is indeed desirable to give no fine grained hints in these cases. These kinds of false negatives are not counted in the results discussed below. If they are added, recall will decrease. It has to be noted that more decisions of these kind can be taken, lowering or raising the expectations towards the number of candidate steps that should be produced by the algorithm.

```
137   public static float[] arithmetic_average(int[][] mat){
138       float arith=0; float n = (float) mat.length;
139       float [] vektor =new float [mat.length];
140       for (int i=0; i<=mat.length; i++){
141           for (int j=0; j<=mat[0].length; j++){
142               arith+= (float) mat[i][j];
143           }
144           vektor [i]= arith/n;
145           arith=0;
146         }
147       return vektor;
148   }
```

**Figure 5.** Another piece of source code cut out from a student's solution for the example exercise. See Figure 6 for a trace created by this solution.
Figure 6. Trace generated for the source code shown in Figure 5, when invoked for the array \{ (1, 2, 1), (4, 3, 2), (2, 2, 7) \}. Execution stops because of an exception in the last step, since 3 is no valid index for array access. However, already the second but last line is rated as a candidate step by the simple heuristic approach. Indeed this line contains the relevant error by using <= instead of <.

Table 1. Evaluation results for the simple heuristic approach applied to five test cases from two different exercises. Test case 1 from exercise 2 is the one used as running example throughout the paper.

<table>
<thead>
<tr>
<th></th>
<th>Exercise 1</th>
<th>Exercise 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Test 1</td>
<td>Test 2</td>
</tr>
<tr>
<td>True positive</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>False positive</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>False negative</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>Precision</td>
<td>0.54</td>
<td>0.84</td>
</tr>
<tr>
<td>Recall</td>
<td>0.48</td>
<td>0.74</td>
</tr>
</tbody>
</table>

The evaluation has been carried out by analyzing traces from five test cases from two different exercises. The results are shown in Table 1. Results of the evaluation show a fair performance of the simple heuristic approach: Precision varies from 54% (which is a rather poor value) to 84% (which is a rather good value). Recall varies from 48% (which is a poor value again) to 74% (which is a rather good value as well). These figures prove the simple heuristic approach to be effective, but they also show that there is a lot of space for improvement. While a recall of about 50% can be considered acceptable, a precision of little over 50% is clearly not yet satisfying for an automated tutoring system.

It can be noticed in the results that the test cases used to create the traces may have an impact on the performance of the approach. This seems to be obvious because different test cases may reveal different errors, where some of these are easier to spot by trace comparison than others. However, no possibilities to enhance trace alignment with knowledge about the errors to be spotted by a particular test case have been explored so far.

Conclusions and Future Work

In this paper, a simple heuristic approach for trace alignment has been described and discussed. The performance of the approach has been evaluated using actual data from an existing e-assessment system. The results proved that the approach is useful, but should be improved to achieve a better performance before it can be considered mature enough for full productive use. Ideas for improving the approach both on the stage of
variable identification and the stage of rating have been discussed and will be implemented and evaluated as future work.

It should be noted that full traces of programs can of course be analyzed in different ways and not only by heuristic comparison to other traces. Another way also suitable for automated tutoring systems is specifying desired characteristics of traces using modal logic. It has to be studied whether this approach provides better precision or recall than the simple heuristic approach presented in this paper.

References


