Execution Traces as a Powerful Data Representation for Intelligent Tutoring Systems for Programming

Benjamin Paaßen
CITEC center of excellence
Inspiration 1
33619 Bielefeld, Germany
bpaassen@techfak.uni-bielefeld.de

Joris Jensen
CITEC center of excellence
Inspiration 1
33619 Bielefeld, Germany
jjensen@techfak.uni-bielefeld.de

Barbara Hammer
CITEC center of excellence
Inspiration 1
33619 Bielefeld, Germany
bhammer@techfak.uni-bielefeld.de

ABSTRACT

The first intelligent tutoring systems for computer programming have been proposed more than 30 years ago, mostly focusing on well defined programming tasks e.g. in the context of logic programming. Recent systems also teach complex programs, where explicit modelling of every possible program and mistake is no longer possible. Such systems are based on data-driven approaches, which focus on the syntax of a program or consider the output for example cases. However, the system's understanding of student programs could be enriched by a deeper focus on the actual execution of a program. This requires a suitable data representation which encodes information of programming style as well as its functionality in a suitable way, thus offering entry points for automated feedback generation.

In this contribution we propose a representation of computer programs via execution traces for example input and demonstrate the power of this representation in three key challenges for intelligent tutoring systems: identifying the underlying solution strategy, identifying erroneous solutions and locating the errors in erroneous programs for feedback display.

Keywords

execution traces, data-driven tutoring systems, computer science teaching, sequence alignment, sorting programs

1. INTRODUCTION

Teaching computer programming has been a long-standing goal of intelligent tutoring systems research. The earliest example, the LISP tutor, has been released in 1985 [1] and since then many different approaches have evolved, such as learning by examining and manipulating examples, by simulation and debugging, by dialogue with the system, by collaboration with peers or by feedback [7]. Most of these approaches rely on extensive domain knowledge about program structure, typical mistakes (so-called *buggy rules*) and syntactic concepts, which is expensive to obtain and difficult to encode [5, 10]. In particular, such approaches get infeasible if the space of possible programs (and mistakes) gets too large, and if the goal of the computer program is ill-defined [8]. To push the boundaries of intelligent tutoring systems towards such scenarios, data-driven approaches have been developed which provide feedback to students based on example programs handed in by other students, e.g. by highlighting the difference of the student solution and a similar, correct program [2, 16]. However, such approaches focus strongly on the *syntax* of programs, which is problematic because the relation between a programs functionality and its syntax is highly non-linear.

As an example, consider the Java code shown in Figure 1. The programs on the left and on the middle are both (correct) sorting programs, which have a very similar syntactic structure. Both sort the array via two nested loops, compare the current element to its successor and swap them if the order is incorrect. However, the programs implement *different* algorithms, namely *BubbleSort* (left) and *InsertionSort* (middle). Thus, minor syntactic changes correspond to major changes in terms of function [14]. If an intelligent tutoring system provides feedback based on a functionally dissimilar example (e.g. a different underlying algorithm) the system might recommend changes to the student's program which lead the learner away from her intended strategy. Such feedback might be detrimental to the student's learning success.

This poses a challenge to educational datamining research. How do we estimate the similarity between programs on a functional level, without exceeding effort in knowledge engineering? We propose to represent computer programs by their execution traces, to compare such traces using sequence alignment and to define the similarity between programs based on the alignment distance. An execution trace is a sequence of variable states for each step of the program's execution for some input. They are a usual representation of computer programs for debugging purposes and can provide insight into the dynamic behaviour of programs [6]. In particular, traces and alignments of traces have been successfully applied in educational programming environments to offer students an alternative view on their own program for self-reflection [17, 18]. We build upon this research by utilizing the trace representation for educational datamining,

```java
public static int[] bubblesort(int[] A) {
  final int l = 0;
  final int r = A.length - 1;
  for (int i = r; i > l; i--) {
    for (int j = l; j < i; j++) {
      if (A[j] > A[j + 1]) {
        final int tmp = A[j];
        A[j] = A[j + 1];
        A[j + 1] = tmp;
      }
    }
  }
  return A;
}
```

```java
public static int[] insertionSort(int[] A) {
  final int l = 0;
  final int r = A.length - 1;
  for (int i = l; i < r; i++) {
    for (int j = i - 1; j >= l; j--) {
      if (A[j] > A[j + 1]) {
        final int tmp = A[j];
        A[j] = A[j + 1];
        A[j + 1] = tmp;
      }
    }
  }
  return A;
}
```

```java
public static int[] insertionSort(int[] A) {
  final int l = 0;
  final int r = A.length - 1;
  insertionSort(A, l, r);
  return A;
}
private static void insertionSort(int[] A, int l, int r) {
  if (l < r) {
    insertionSort(A, l, r - 1);
    insert(A, l, r);
  }
}
private static void insert(int[] A, int l, int r) {
  if (l < r) {
    if (A[r - 1] > A[r]) {
      final int tmp = A[r - 1];
      A[r - 1] = A[r];
      A[r] = tmp;
    }
    insert(A, l, r - 1);
  }
}
```

Figure 1: Three correct sorting programs in Java code. Important syntactic constructs and variable initializations are highlighted. The corresponding code parts between all three programs are visualized via background highlighting. Left: An iterative *BubbleSort* implementation. Middle: An iterative *InsertionSort* implementation. Right: A recursive *InsertionSort* implementation.

Bubble	*Insertion*	recursive
$[4,7,2,1]$	$[4,7,2,1]$	$[4,7,2,1]$
$[4,2,7,1]$	$[4,2,7,1]$	$[4,2,7,1]$
$[4,2,1,7]$	$[2,4,7,1]$	$[2,4,7,1]$
$[2,4,1,7]$	$[2,4,1,7]$	$[2,4,1,7]$
$[2,1,4,7]$	$[2,1,4,7]$	$[2,1,4,7]$
$[1,2,4,7]$	$[1,2,4,7]$	$[1,2,4,7]$

Table 1: The execution traces for the three programs from Figure 1 for the input array $A = [4,7,2,1]$. Only the values for the variable A are shown and intermediate steps that do not manipulate A have been omitted.

that is, for automated classification and analysis of student's computer programs in order to provide helpful, automated feedback.

As an example, consider the programs from Figure 1 again. Their execution traces for the input array $A = [4,7,2,1]$ are shown in Table 1. Despite the apparent syntactic similarity, the implementations of *BubbleSort* and *InsertionSort* do indeed map to different traces, while the iterative and recursive implementation of *InsertionSort* map to the same trace. This indicates that traces have a more direct relationship to the semantics of the underlying program, making them a promising representation for intelligent tutoring systems.

The main contributions of our work are as follows: First, we introduce execution traces with the purpose to capture syntactic as well as semantic aspects of the underlying program (Section 3). Second, we provide an efficient methodology for automatically comparing such traces via edit distances and inferring a measure of similarity for further datamining applications (Section 4). Finally, we evaluate our approach in comparison with the state of the art in syntactic representation in three key challenges for educational data mining: 1.) identifying the student's underlying algorithmic approach (Section 5.2), 2.) identifying erroneous implementations (Section 5.3), and 3.) detecting the location of errors for feedback (Section 5.4). To our knowledge, no data-driven approach exists to date which tackles all three challenges. Syntax-based representations have been successful in identifying the programming strategy [11, 13] but fail in identifying erroneous solutions as well as error locations (as we will show later). On the other hand, test case-based evaluations are very successful in identifying erroneous solutions but treat programs as a black box and thus can make no claims regarding the implemented strategy or the location of the error [17].

2. BACKGROUND AND RELATED WORK

2.1 Tutoring Systems for Computer Programming

In a review of AI-supported tutoring approaches for computer programming, Le and colleagues found six categories of approaches, namely: 1.) displaying examples of programs in order to learn to construct programs of a similar type or modify examples; 2.) simulating the execution of a program in a micro-world and visualizing it to the user; 3.) providing a dialogue environment in order to complete a programming task in an interactive dialogue with the system; 4.) presenting buggy example code in order to learn via program analysis and debugging; 5.) providing feedback to students during development of their program in order to guide them towards a correct solution and detect errors; and finally 6.) providing a collaborative work environment in which students can help each other in developing a program, guided by the system's group model [7]. We note that Le and colleagues do not yet consider recent data-driven approaches, which are mostly feedback-based systems, such as the FIT Java Tutor [2], BOTS [4] and ITAP [16]. Our own approach is targeted mainly at such feedback-based systems working on examples. We analyze the execution trace of a student's program in order to find similar programs for feedback purposes and we intend to locate errors in the student's program to help her correct them. However, our approach also bears similarity to simulation-based approaches as we consider the execution of the program's statements as the main characteristic of a program.

2.2 Representations of Computer Programs for Data-Driven Systems

Most existing data-driven systems for computer programming represent programs as abstract syntax trees, which are subjected to some form of canonalization in order to abstract from mere stylistic differences [15]. Recently, Piech and colleagues have criticized this approach and judged syntax trees not sufficiently discriminative to capture the strong functional consequences of small syntactic changes [14]. Instead, they propose a neural network-based approach to infer a vectorial representation of programs, such that standard machine learning methods can be applied in the resulting Euclidean space. Similar to our approach, Piech and colleagues intend to represent a programs function (or semantics) in opposition to its syntax. However, they focus on a direct mapping between input and output of program segments, while the trace representation provides more procedural (or dynamic) insight into the programs function.

2.3 Edit Distances on Computer Programs

Computing similarities and dissimilarities between computer programs is a crucial step towards data-driven intelligent tutoring system [9]. Edit distances have been particularly prominent in this regard. For example, Rivers and Koedinger used tree edit distances to compute similarities between syntax trees of Python programs to identify adjacent states [16]. Gross and colleagues similarly applied edit distances on syntax trees to infer clusters of computer programs and select the most similar sample solution for feedback [2, 3]. Finally, Paaßen, Mokbel and Hammer have identified the underlying algorithm of sorting programs using machine learning techniques based on alignment distances and adapted the parameters of those alignment distances to yield better classification results [11, 13]. Note that all these approaches rely on alignment distances on program *syntax*, not on execution traces. Striewe and Goedicke applied sequence alignment on execution traces, but did not apply the alignment distances for further datamining purposes [18].

2.4 Classification of Computer Programs

Recently, the value of classification methods for feedback provision in intelligent tutoring systems for computer programming has been recognized. Such machine learning methods enable tutoring systems to infer e.g. the underlying programming strategy of a learner with explicit human labelling only for a small example set [13]. Piech and colleagues report multiplication factors of up to 214, that is, a human tutors annotation for one program permits inference of said annotation for up to 214 other programs [14]. Of course, such approaches rely on a representation of computer programs in a format that can be fed into machine learning methods, such as pairwise similarities and dissimilarities [9, 13] or an explicit vectorial embedding [14]. In this contribution, we employ a classification paradigm to distinguish between different algorithmic approaches, as well as between erroneous and correct solutions.

3. REPRESENTING COMPUTER PROGRAMS VIA EXECUTION TRACES

In general, execution trace recordings can be defined as the "*detection* and *storage* of relevant events during run-time, for later off-line analysis" [6]. More specifically, we consider executions of statements in the program as relevant events, which we characterize by the value of variables of interest after the statement has been executed. This is equivalent to a step-wise execution of the program in a debugger, where we record the state of an interesting variables in each step [17]. As an example, consider traces in Table 1 for the programs in Figure 1.

Only modest technical requirements have to be fulfilled to apply a trace representation. 1.) The programming language has to offer a debugging environment which permits monitoring of a program's execution; 2.) a valid and non-trivial example input for the task has to be available; and 3.) the student's program has to compile and execute without errors on the example input [17]. Thus, the trace representation is more demanding compared to the very flexible syntactic representation of computer programs, but has less prerequisites compared to extensive knowledge engineering. In that sense, the trace representation can be seen as a "middle road" between entirely data-driven approaches and systems based on expert knowledge.

4. COMPARING EXECUTION TRACES

If a student's program is analyzed via test cases, the output is compared with the pre-defined reference value via a simple equality test. However, such a strict equality test is not a viable option for the comparison of execution traces. For example, the traces on the left and the middle in Table 1 are not equal. But they are more similar to each other than to an erroneous program that does not sort the input array at all. Therefore, we require a more flexible measure of similarity or dissimilarity between traces [9].

Similarities and dissimilarities on sequential data can be obtained via *alignment distances* or *edit distances*. The overarching scheme is to extend both input sequences such that there length becomes equal and similar elements of both sequences become aligned. The alignment distance is then defined as the summed cost over all aligned elements [13]. The choice of alignment algorithm depends on the extensions of input sequences that should be permitted. In case of execution traces we intend to abstract from sequence elements that leave the relevant variables unchanged. As an example, consider lines two and three of the program in Figure 1 (left). These two lines could be removed from the program without changing its function, if all expressions of r and l are replaced by their value in the rest of the program. A classic edit distance scheme would punish this with a higher dissimilarity between the shorter and the longer version of the program. Instead, we propose that the same state of the relevant variables may be copied without cost. This corresponds to the *dynamic time warping* dissimilarity D_{DTW} for speech processing, first introduced by Vintsyuk [20]. Given two traces $\bar{x} = (x_1, \ldots, x_M)$ and $\bar{y} = (y_1, \ldots, y_N)$ as well as a dissimilarity measure $d(x_i, y_j)$ between the variable states

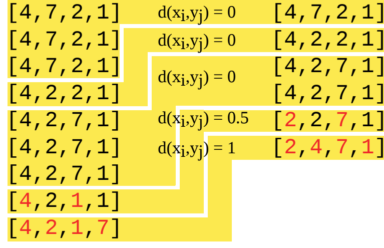

Figure 2: An illustration of the dynamic time warping distance between two traces. Aligned array states are connected by yellow background. Mismatching parts of the aligned variable states are highlighted in red. The dissimilarity between aligned array states is shown in the middle.

x_i and y_j, it is defined recursively as:

$$D_{\text{DTW}}\Big((x_1,\ldots,x_i),(y_1,\ldots,y_j)\Big) := d(x_i,y_j) + \min\Big\{ \quad (1)$$

$$D_{\text{DTW}}\Big((x_1,\ldots,x_{i-1}),(y_1,\ldots,y_{j-1})\Big),$$

$$D_{\text{DTW}}\Big((x_1,\ldots,x_{i-1}),(y_1,\ldots,y_j)\Big),$$

$$D_{\text{DTW}}\Big((x_1,\ldots,x_i),(y_1,\ldots,y_{j-1})\Big)\Big\}$$

$$D_{\text{DTW}}\Big((x_1),(y_1)\Big) := d(x_1,y_1) \quad (2)$$

This can be calculated efficiently in $O(M \cdot N)$ via dynamic programming (D_{DTW} is tabulated for all prefixes of $\bar{x}$ and $\bar{y}$).

An illustration of the dynamic time warping dissimilarity between two example traces is shown in Figure 2. The first three array states of the left trace are just repetitions and thus are aligned with the first array state of the right trace. This occurs again for the fourth to sixth array state of the left trace. Only afterwards the array states differ and lead to a non-zero dissimilarity between both traces. Note that the explicit alignment of array states between two compared traces in dynamic time warping can be retrieved efficiently via backtracing in linear time.

As other edit distances, the dynamic time warping algorithm crucially relies on a dissimilarity measure between variable states. If prior knowledge regarding the interesting variables is available, defining such a measure becomes fairly straightforward (e.g. a Hamming-distance on arrays, just counting the number of unequal entries). In absence of such prior knowledge, defining a dissimilarity on variable states becomes a challenge in itself. One has to infer a semantic matching between the variables in both programs, determine their relevance (as some variables might be less central to the semantic function than others) and then compute the relevance-weighted distance between all matched variables. As a first step in this direction, we propose a simple summary scheme. We build a histogram H_{x_i} in each state x_i that counts the number of variables of each type $t \in \mathcal{T}$, and compare these histograms with a normalized L1 distance:

$$d(x_i, y_j) := \frac{1}{|\mathcal{T}|} \cdot \sum_{t \in \mathcal{T}}^{R} \frac{|H_{x_i}(t) - H_{y_j}(t)|}{|H_{x_i}(t)| + |H_{y_j}(t)|} \quad (3)$$

Note that we consider only types t which occur in both programs at least once.

5. EXPERIMENTS

Our experimental evaluation concerns three key challenges for data-driven intelligent tutoring systems: 1.) Identifying the underlying algorithmic approach, 2.) identifying erroneous programs, and 3.) detecting the location of an error, once a program is identified as erroneous. We compare the performance on these tasks between the trace representation (with dynamic time warping as dissimilarity measure) and the state-of-the-art in terms of syntax representation: syntax-trees with learned edit distance parameters via machine learning techniques [13]. As implementation of the alignment techniques we applied the *TCS Alignment Toolbox* [12].

5.1 Datasets

For our evaluation, we use two benchmark datasets. The *palindrome* data set consists of 48 (correct) programs deciding whether all words in an input sentence are palindromic, using one of eight different programming strategies [9]. We used the histogram-approach to define a dissimilarity between variable states and generated traces using the input sentence “OTTO ANNA MOPS”. As this data set does not contain erroneous programs, we only used it for the first experiment.

The second dataset is an extended version of the *sorting* dataset from [11]. It consists of 126 (correct) sorting programs collected from various web sources, each implementing one of six sorting algorithms (35 *BubbleSorts*, 29 *InsertionSorts*, 15 *MergeSorts*, 17 *QuickSorts*, 20 *SelectionSorts* and 10 *ShellSorts*). For each of the programs we created an erroneous counterpart, with one or more *semantic* errors, that is, errors that are neither detected by the compiler nor do they lead to a program crash (e.g. due to an index being out of bounds). Thereby, we focused on errors that are non-trivial to detect for technical systems. As a dissimilarity between variable states we employed a Hamming distance on the array to be sorted. As input we generated a uniform random array of 10 integers in the range $[0, 99]$.

Both datasets are available online at `http://doi.org/10.4119/unibi/2900666` and `http://doi.org/10.4119/unibi/2900684` respectively.

5.2 Classifying Programming Strategies

Our first experiment concerns the identification of the underlying sorting algorithm. We assume that a human expert has already labelled some example programs and want to infer the correct label for some new, unlabelled program. We evaluate the classification accuracy of an 1-nearest neighbor classifier for the syntactic as well as the trace-based representation in a crossvalidation with 6 folds (for the *palindrome* dataset) and 10 folds (for the *sorting* dataset) respectively.

The results are shown in Table 2. For the *palindrome* dataset, the accuracy for the trace representation is more than 10% higher compared to the syntactic representation. Yet, likely due to the small sample size, this difference is not significant (Wilcoxon rank-sum test). In case of the *sorting* data set,

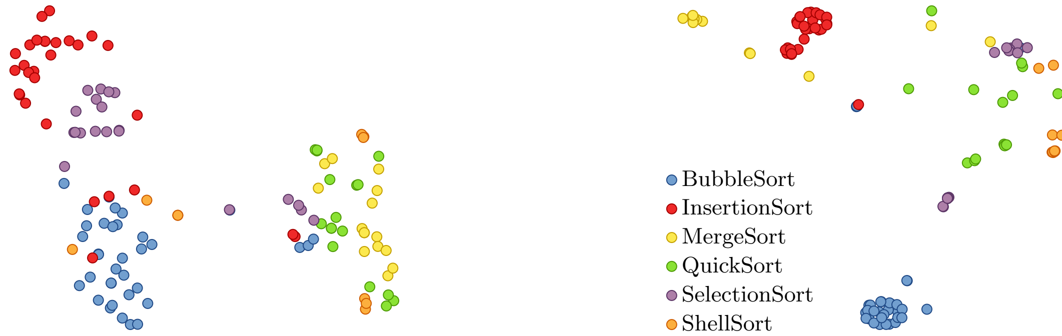

Figure 3: The *sorting* dataset embedded in 2 dimensions via t-stochastic neighborhood embedding (t-SNE) [19]. The sorting algorithms are indicated by color. On the left side, the embedding is shown for adapted, syntactic edit distances [13]. On the right side, we show the embedding for dynamic time warping dissimilarities on traces.

	palindromes		sorting	
method	acc.	std. dev.	acc.	std. dev.
syntax	0.875	0.158	0.812	0.068
traces	0.979	0.051	0.954	0.040

Table 2: The mean classification accuracy and its standard deviation of a 1-nearest neighbor classifier distinguishing six different sorting algorithms. Mean and standard deviation are calculated across 6 (for palindromes) and 10 (for sorting) crossvalidation trials.

we gain an increase in accuracy of more than 14%, which is highly significant ($p < 0.01$, Wilcoxon rank-sum test). This is also reflected in the corresponding dissimilarities. In Figure 3 we show 2-dimensional embeddings of the *sorting* dataset according to syntax-based (left) and trace-based (right) dissimilarities. The trace representation yields more compact clusters corresponding to the correct class label, thereby making classification easier. Interestingly, closer inspection of the misclassified data points for the trace representation revealed that the 1-nearest neighbor classifier correctly identified a *BubbleSort* implementation the programmers had wrongly labelled as an *InsertionSort*.

In order to apply a classification algorithm in praxis, labelled data is required. To reduce human work, one would like to reduce the amount of labelled data necessary. We tested the required amount of labelled data experimentally, by reducing the number of labelled data points (and increasing the number of unlabelled points). The results are displayed in Figure 4. For the *palindrome* data set, only two data points per class are sufficient to achieve good performance. For the *sorting* data set, about 40 labelled programs suffice to achieve a classification accuracy of 90% using the trace representation, while the classification accuracy for the syntactic representation saturates at 80% for about 60 programs.

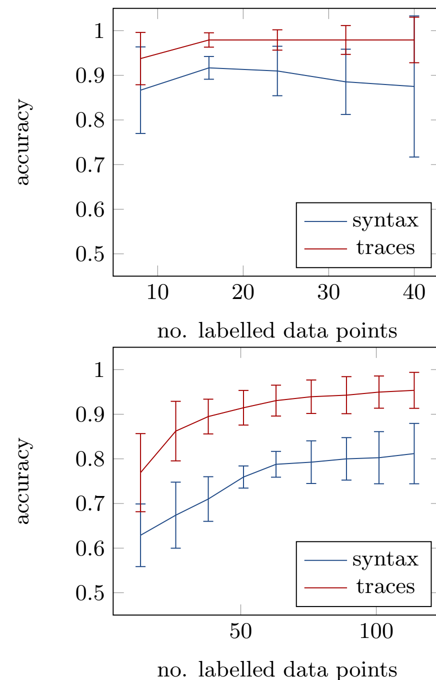

Figure 4: The classification accuracy on the strategy classification task using the syntactic as well as the trace-based data representation if the number of available labelled data points is reduced and the number of unlabelled points is increased. The upper plot displays the result for the *palindromes* dataset, the lower plot for the *sorting* dataset. The errorbars mark the standard deviation across 6 and 10 crossvalidation trials respectively.

5.3 Classifying Erroneous Programs

We phrase the identification of erroneous problems as a classification task as well: We assume that a human expert

method	Accuracy	std. dev.
syntax	0.211	0.107
traces	0.861	0.086

Table 3: The mean classification accuracy and its standard deviation of a 1-nearest neighbor classifier distinguishing erroneous from correct sorting programs. Mean and standard deviation are calculated across 20 crossvalidation trials.

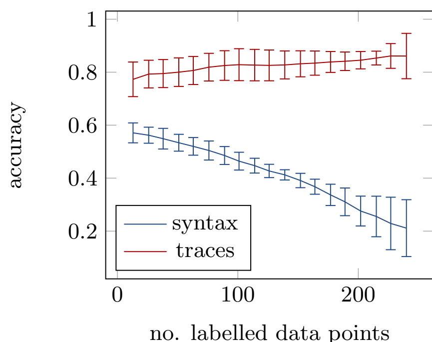

Figure 6: The classification accuracy on the error classification task using the syntactic as well as the trace-based data representation if the number of available labelled data points is reduced and the number of unlabelled points is increased. The errorbars mark the standard deviation across 20 crossvalidation trials.

has labelled a few example programs as correct and erroneous respectively. Then, we want to infer the label for new programs. We evaluate the classification accuracy of an 1-nearest neighbor classifier in a 20-fold crossvalidation.

The results are shown in Table 3. As expected, the syntactic information is not at all sufficient to judge the correctness of a program. The trace-based representation, on the other hand, identifies correct and false solutions in most cases (about 86% accuracy). Again, we can observe the difference between both representation in 2-dimensional embeddings. Figure 5 shows embeddings for the syntactic-based (left) as well as the trace-based dissimilarities (right). While erroneous and correct solutions are almost indistinguishible for the former representation, we observe a much clearer separation of the classes for the latter representation.

We also tested the classification performance if less labelled data is available (see Figure 6). Interestingly, the classification accuracy of the syntactic representation decreases if more labelled data is available. This is likely due to the fact that we created the erroneous programs based on the correct ones, such that the nearest neighbor from a syntactic point of view often was the respective counterpart solution, such that errors get more prevalent if more of such neighbors are available for classification (also refer to Figure 5). Conversely, the trace representation steadily increases in performance and reaches 80% accuracy at about 50 labelled data points.

5.4 Detecting Error Locations

As a final challenge, we try to locate the errors within the erroneous programs. More precisely, the challenge is to identify a set of lines of code in an erroneous program, such that all errors are included, but few other lines are included. Such a set of lines can then be utilized in an intelligent tutoring system. The identified lines can be highlighted such that the student is able to find the error in her program. We apply two strategies based on alignment algorithms, one on the syntactic representation and one on the trace representation.

Syntax-Based Error Detection. We select the nearest correct neighbor and retrieve a syntactic alignment of the erroneous program and the correct program via backtracing. Thereby we obtain the contribution of each line of code in the erroneous program to the overall alignment distance. In order to identify contributing neighbors as well, we apply Gaussian blur to this distribution and then select the line of code with the highest contribution as well as its neighbors, if their contribution is sufficiently high (at least half as high compared to the maximum).

Trace-Based Error Detection. Our trace-based strategy is similar to the syntax-based one. We again select the nearest correct neighbor and retrieve a trace alignment of the erroneous program and the correct program via backtracing. However, we can apply additional domain knowledge. We assume that an erroneous program has the wrong output given the input. The output of the program includes the value of the relevant variables at the end of the trace. Therefore, we can start from the end of the trace alignment and work back until the state of the relevant variables is equal to the state in the correct program. This is the point where the error in the program influences the programs execution negatively. However, it is not sufficient to highlight this particular line of code, because the actual error might be earlier in the code (e.g. a wrongly set index). Therefore, we select not only this line, but the most frequently executed five lines of code until the last change of the relevant variables.

Further, we included three trivial baseline strategies for comparison: 1.) Selecting a line of code at random, 2.) selecting a line of code at random according to its distribution in the trace, and 3.) selecting *all* lines in the program that occured in the trace.

We evaluated all five strategies in a 20 fold crossvalidation. For each erroneous program, we excluded the correct counterpart from the available neighbors in order to make the scenario more realistic.

The results are shown in Table 4. We report the classic pattern recognition measures precision (how many of the selected lines of code contain an error?), recall (how many of the erroneous lines of code have been selected?) and F1-score (harmonic mean of precision and recall). In terms of F1-score, the trace-based error detection method clearly outperforms the syntax-based one ($p < 10^{-4}$, Wilcoxon rank-sum test). Further, as expected, both random baseline meth-

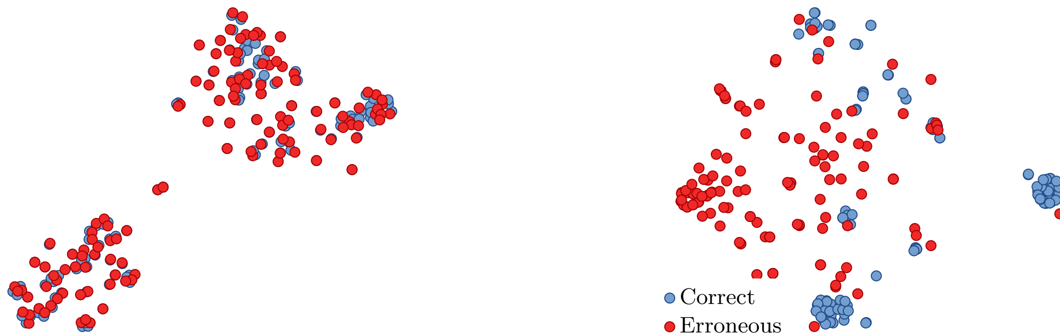

Figure 5: The *sorting* dataset including erroneous solutions embedded in 2 dimensions via t-stochastic neighborhood embedding (t-SNE) [19]. The correctness of each program is indicated by color. On the left side, the embedding is shown for adapted, syntactic edit distances [13]. On the right side, we show the embedding for dynamic time warping dissimilarities on traces.

method	precision	std. dev.	recall	std. dev.	F1 score	std. dev.
traces	0.183	0.071	0.520	0.211	0.269	0.104
syntax	0.103	0.086	0.134	0.100	0.115	0.091
traces_random	0.157	0.122	0.119	0.098	0.134	0.107
syntax_random	0.121	0.116	0.095	0.095	0.105	0.103
traces_all	0.103	0.022	0.976	0.050	0.186	0.037

Table 4: The mean classification accuracy and its standard deviation of a 1-nearest neighbor classifier distinguishing erroneous from correct sorting programs. Mean and standard deviation are calculated across 20 crossvalidation trials.

ods seldomly select an erroneous line, thereby limiting the recall. However, selecting all lines of code occuring in a trace provides a strong baseline to compete with ($F1 = 0.186$). Still, the trace-based error location method performs significantly better ($p < 0.01$, Wilcoxon rank-sum test).

6. DISCUSSION

In this contribution we introduced an alternative representation of computer programs for classification and error detection in intelligent tutoring systems (ITSs), namely execution traces. On two example data sets we have demonstrated that this representation can improve upon state-of-the-art syntax-based representation in terms of strategy classification, error classification and error detection. In a full-blown ITS for computer programming, the trace representation can thus be applied to help students in solving programming tasks. As soon as a student has managed to reach a working state (without syntax errors and program crashes) we can generate a trace and compare it with the traces of different programs. The resulting (dis-)similarity measure can be used to identify possible partners for peer-review and peer-tutoring by matching students that apply the same approach in their solution attempt. Further, the trace representation can be applied to identify erroneous programs, enabling an ITS to detect whether a student has finished a task or still needs to continue. Further, as not only the end result is checked but the whole execution, the trace representation can be utilized for detecting unusual or deceptive solutions that are geared towards the test cases without actually implementing the desired function. Finally, if an error is still present in a student's program but the error is not obvious, the trace representation may help to identify and highlight the location of the error in the program code, thereby providing scaffolding to students that get stuck in searching for their error.

Overall, the trace representation appears to be highly useful for data-driven ITSs on computer programming. However, important challenges remain. If no a priori knowledge regarding the relevant variables in the program is available, computing a dissimilarity on variable states is not trivial. We have suggested a first attempt using a histogram of variable types. This representation, however, disregards the content of variables and thus is likely not sufficiently powerful in many applications where differences in variable values are important markers of program semantics. A solution might be to match variables probabilistically according to the alignment distance a certain matching produces. This is an interesting direction to pursue in further research.

Finally, we note that the trace representation does not have to be the sole source of information for an ITS. A syntactic representation is necessary when a program does not yet compile or crashes and wherever the high level of abstraction applied by a program trace is not helpful (e.g. when teaching certain syntactic constructs). Fusing the strengths of both representations is likely to lead to the best learning outcomes for students.

7. ACKNOWLEDGMENTS

Funding by the DFG under grant number HA 2719/6-2 and the CITEC center of excellence (EXC 277) is gratefully acknowledged.

8. REFERENCES

[1] J. R. Anderson and E. Skwarecki. The automated tutoring of introductory computer programming. *Commun. ACM*, 29(9):842–849, Sept. 1986.

[2] S. Gross, B. Mokbel, B. Paaßen, B. Hammer, and N. Pinkwart. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology*, 9(3):248–280, Nov. 2014.

[3] S. Gross and N. Pinkwart. How do learners behave in help-seeking when given a choice? In C. Conati, N. Heffernan, A. Mitrovic, and M. F. Verdejo, editors, *Artificial Intelligence in Education*, volume 9112 of *Lecture Notes in Computer Science*, pages 600–603. Springer International Publishing, 2015.

[4] A. Hicks, Y. Dong, R. Zhi, V. Catete, and T. Barnes. BOTS: selecting next-steps from player traces in a puzzle game. In *Workshops Proceedings of EDM 2015 8th International Conference on Educational Data Mining, EDM 2015, Madrid, Spain, June 26-29, 2015.*, 2015.

[5] K. R. Koedinger, E. Brunskill, R. S. Baker, E. A. McLaughlin, and J. Stamper. New potentials for data-driven intelligent tutoring system development and optimization. *AI Magazine*, 34(3):27–41, 2013.

[6] J. Kraft, A. Wall, and H. Kienle. Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, pages 315–329. Springer Berlin Heidelberg, 2010.

[7] N. T. Le, S. Strickroth, S. Gross, and N. Pinkwart. A review of ai-supported tutoring approaches for learning programming. In N. T. Nguyen, T. Do, and H. A. Thi, editors, *Advanced Computational Methods for Knowledge Engineering - Proceedings of the 1st International Conference on Computer Science, Applied Mathematics and Applications (ICCSAMA)*, number 479 in Studies in Computational Intelligence, pages 267–279, Berlin, Germany, 2013. Springer Verlag.

[8] C. Lynch, K. D. Ashley, N. Pinkwart, and V. Aleven. Concepts, structures, and goals: Redefining ill-definedness. *International Journal of Artificial Intelligence in Education*, 19(3):253–266, 2009.

[9] B. Mokbel, S. Gross, B. Paaßen, N. Pinkwart, and B. Hammer. Domain-Independent Proximity Measures in Intelligent Tutoring Systems. In S. K. D'Mello, R. A. Calvo, and A. Olney, editors, *Proceedings of the 6th International Conference on Educational Data Mining (EDM)*, 2013.

[10] T. Murray, S. Blessing, and S. Ainsworth. *Authoring tools for advanced technology learning environments: Toward cost-effective adaptive, interactive and intelligent educational software*. Springer, 2003.

[11] B. Paaßen, B. Mokbel, and B. Hammer. Adaptive structure metrics for automated feedback provision in java programming. In M. Verleysen, editor, *23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*, pages 307–312, 2015.

[12] B. Paaßen, B. Mokbel, and B. Hammer. A toolbox for adaptive sequence dissimilarity measures for intelligent tutoring systems. Proceedings of the 8th International Conference on Educational Data Mining, pages 632–632. International Educational Datamining Society, 2015.

[13] B. Paaßen, B. Mokbel, and B. Hammer. Adaptive structure metrics for automated feedback provision in intelligent tutoring systems. *Neurocomputing*, 192, 2016.

[14] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning*, International Conference on Machine Learning, pages 1093–1102, 2015.

[15] K. Rivers and K. R. Koedinger. A canonicalizing model for building programming tutors. In S. A. Cerri, W. J. Clancey, G. Papadourakis, and K. Panourgia, editors, *Intelligent Tutoring Systems: 11th International Conference, ITS 2012, Chania, Crete, Greece, June 14-18, 2012. Proceedings*, pages 591–593. Springer Berlin Heidelberg, 2012.

[16] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, pages 1–28, 2015.

[17] M. Striewe and M. Goedicke. Using run time traces in automated programming tutoring. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 303–307, New York, NY, USA, 2011. ACM.

[18] M. Striewe and M. Goedicke. Trace alignment for automated tutoring. In *Computer Assisted Assessment Conference*, 2013.

[19] L. van der Maaten and G. E. Hinton. Visualizing high-dimensional data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[20] T. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968.