The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces

Benjamin Paaßen
CITEC Center of Excellence
bpaassen@techfak.uni-bielefeld.de

Barbara Hammer
CITEC Center of Excellence
bhammer@techfak.uni-bielefeld.de

Thomas W. Price
North Carolina State University
twprice@ncsu.edu

Tiffany Barnes
North Carolina State University
tmbarnes@ncsu.edu

Sebastian Gross
Humboldt-Universität zu Berlin
sebastian.gross@informatik.hu-berlin.de

Niels Pinkwart
Humboldt-Universität zu Berlin
niels.pinkwart@hu-berlin.de

Intelligent tutoring systems can support students in solving multi-step tasks by providing hints regarding what to do next. However, engineering such next-step hints manually or via an expert model becomes infeasible if the space of possible states is too large. Therefore, several approaches have emerged to infer next-step hints automatically, relying on past students' data. In particular, the Hint Factory (Barnes and Stamper, 2008) recommends edits that are most likely to guide students from their current state towards a correct solution, based on what successful students in the past have done in the same situation. Still, the Hint Factory relies on student data being available for any state a student might visit while solving the task, which is not the case for some learning tasks, such as open-ended programming tasks. In this contribution we provide a mathematical framework for edit-based hint policies and, based on this theory, propose a novel hint policy to provide edit hints in vast and sparsely populated state spaces. In particular, we extend the Hint Factory by considering data of past students in all states which are similar to the student's current state and creating hints approximating the weighted average of all these reference states. Because the space of possible weighted averages is continuous, we call this approach the Continuous Hint Factory. In our experimental evaluation, we demonstrate that the Continuous Hint Factory can predict more accurately what capable students would do compared to existing prediction schemes on two learning tasks, especially in an open-ended programming task, and that the Continuous Hint Factory is comparable to existing hint policies at reproducing tutor hints on a simple UML diagram task.

Keywords: next-step hints, Hint Factory, edit distances, computer science education, Gaussian Processes

1. Introduction

In many educational domains, learning tasks require more than a single step to solve. For example, programming tasks require a student to iteratively write, test, and refine code that accom-

plishes a given objective (Gross et al., 2014; Price et al., 2017; Rivers and Koedinger, 2015). When working on such multi-step-tasks, students start with an initial state and then change their state by applying an action (such as inserting or deleting a piece of code). At some point, a student may not know how to proceed or may be unable to find an error in her current state, in which case external help is required. In particular, such a student may benefit from a next-step hint, guiding her toward a more complete and/or more correct version and allowing her to continue working on her own (Aleven et al., 2016). Many intelligent tutoring systems (ITSs) attempt to create such next-step hints automatically, and adjust such hints to the student's current state as well as her underlying strategy (Van Lehn, 2006). Typically, hints are created using an expert-crafted model. However, designing such an expert model becomes infeasible if the space of possible states is to variable to cover with expert rules (Murray et al., 2003; Koedinger et al., 2013; Rivers and Koedinger, 2015). This is the case for most computer programming tasks because the space of possible programs grows exponentially with the program length and the set of programs which perform the same function is infinite (Piech et al., 2015). Other examples are so-called ill-defined domains where explicit domain knowledge is not available or at least very hard to formalize (Lynch et al., 2009).

Several approaches have emerged which provide next-step hints without an expert model. Typically, these approaches provide hints in the form of *edits*, that is, actions which can be applied to the student's current state to change it into a more correct and/or more complete state, based on the edits that successful students in the past have applied (Gross and Pinkwart, 2015; Price et al., 2016; Rivers and Koedinger, 2015; Zimmerman and Rupakheti, 2015, for example). Such edit-based next-step hints constitute an elegant and simple approach to feedback for complex learning tasks. The most basic version of the approach requires only two ingredients: a function, which is able to compute the shortest sequence of edits to transform one partial solution to the task into another one, and a correct solution for the task. If a student issues a help request, the system can simply compute the edits from the student's state to the solution and use one of these edits as a hint (Rivers and Koedinger, 2015; Zimmerman and Rupakheti, 2015). Even though this approach is fairly simple, it achieves *personalized* feedback, in the sense that the hint depends on the student's personal state and may thus be fitting to her specific strategy and style (Le and Pinkwart, 2014). Further, this approach needs very little task-specific work on the side of domain experts because they only need to construct example solutions for the task and can apply a general-purpose edit function which is applicable across tasks or even across domains (Mokbel et al., 2013).

A key challenge to such an edit-based hint approach is that it attempts to follow a single reference solution and tries to adjust the student's individual solution in all aspects to the reference solution. In contrast, it may be more desirable to provide hints which correspond to what capable students would do in any given situation *in general* but avoid emulating specific style choices by individual solutions. To identify such generic solution steps, most existing approaches rely on *frequency* information, that is, how often a certain edit or state has occurred in past student's data (Barnes and Stamper, 2008; Lazar and Bratko, 2014; Rivers and Koedinger, 2014; Piech et al., 2015). Unfortunately, for many programming tasks, the space of possible programs is so large that hardly any state is visited more than once, even if aggressive pre-processing methods are applied to canonicalize program representations (Price and Barnes, 2015).

Therefore, a novel approach is needed which can select generic edits even in cases where frequency information is not available. We base this approach on the Hint Factory, which generates hints that have led past students in the same situation to a correct solution (Barnes and

Stamper, 2008; Stamper et al., 2012). To transfer this approach to vast and sparsely populated spaces, we consider not only the data of past students who have visited the same state, but also *similar* states, and we represent the reference state to which a student should move as a weighted average of past students' states, where the weights are chosen in a probabilistically optimal sense. Because the space of possible weighted averages is continuous, our reference states exist in an implicit, continuous state, which is why we call our approach the *Continuous Hint Factory* (CHF). By performing a weighted average, we avoid any individual specificities and focus on generic steps toward a correct solution.

More precisely, the key contributions of our paper are as follows: First, we provide precise definitions of key concepts in the field of edit-based hint policies and integrate them into a mathematical framework. Second, we extend this framework by introducing the notion of an *edit distance space*, a continuous space in which each state corresponds to one vector and the Euclidean distance between vectors corresponds to the edit distance between two states. Finally, we utilize this space for the CHF by identifying the most likely hint as a vector in this space and translating this vector back into a human-readable edit. As such, the CHF constitutes a novel hint technique which is applicable whenever an edit distance and some, possibly few, data samples of past students are available, making it an interesting option for vast and sparsely populated state spaces. For example, such spaces occur in programming tasks where a solution involves many lines of codes and solutions are highly variable in terms of strategy and style.

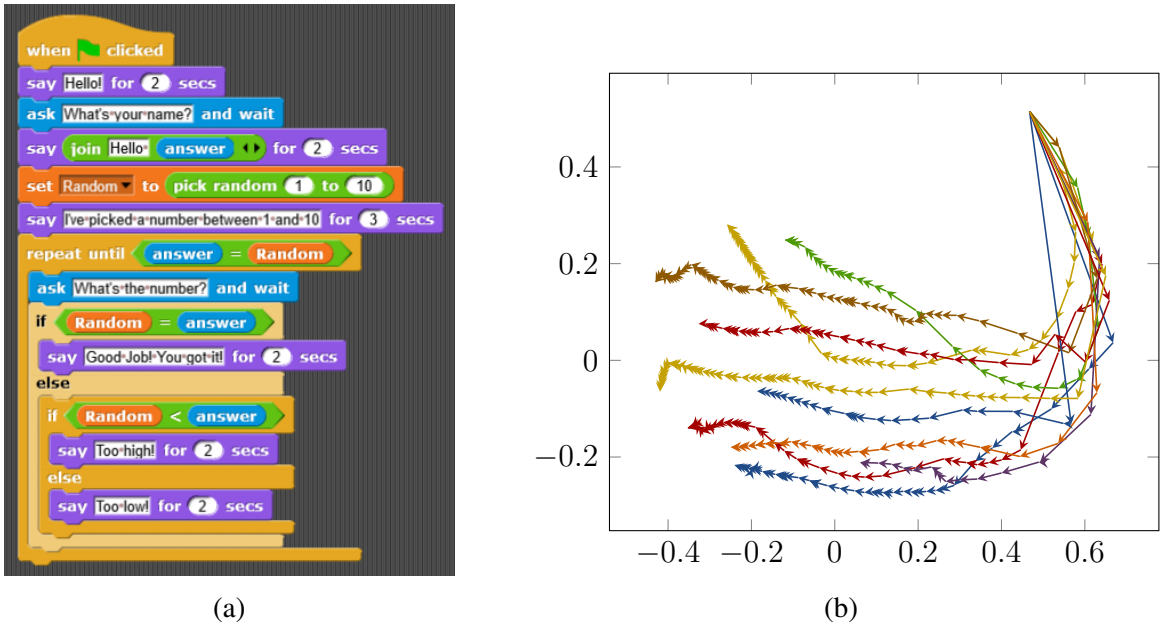
In experiments on two datasets we provide evidence that the CHF is able to predict what capable students would do in solving a learning task, that the CHF is able to disambiguate between many possible edits, and that the hints provided by the CHF match the hints of human tutors at least as well as other established hint techniques.

We begin our work by introducing precise definitions of key concepts of edit-based hint policies and review existing approaches within this novel framework. In Section 3 we introduce the Continuous Hint Factory based on this framework and in Section 4 we report on our experimental evaluation of the Continuous Hint Factory.

2. An integrated view of edit-based hint policies

In this section, we review existing approaches to edit-based hint policies. Alongside this review, we develop a mathematical theory of edit distances and edit-based hint policies, which helps us to contextualize past approaches and will form a firm basis for our own approach in the next section. First, let us start with an example of a learning task, for which an edit-based hint policy may be helpful. Consider the task of programming a guessing game, which should ask the player for their name, generate a random number between 1 and 10, and then let the player guess the number, providing feedback to the player regarding whether the number was too low, too high, or correct. A correct solution for this task in the *Snap* programming language[1] is shown in Figure 1a. In a tutoring system involving this task, a student would start off with an empty program and then would add blocks to the program, delete them, or replace them with others until the student obtains a correct solution or gets stuck. In the latter case, the student may hit a "help" button in the system which would, in turn, provide a hint in the form of an edit which leads the student closer to a correct solution (e.g., to add an "ask" block to ask for the player's name in the beginning).

[1] http://snap.berkeley.edu



(a) (b)

Figure 1: (a) A screenshot from the Snap programming environment. (b) A 2D embedding of ten example traces in the Snap dataset The 2D embedding was obtained via non-metric multi-dimensional scaling (Sammon, 1969) using the pairwise edit distances as input, that is, if two states have an edit distance of 0, they are mapped to the same point in 2D, and if they have a higher edit distance, they are mapped to points which are further apart. Colors are used to distinguish between different traces. States within one trace are connected by arrows.

From a pedagogical point of view, it may be suboptimal to immediately tell the student which edit to apply. In doing so, we deprive students of the possibility of finding the correct next step themselves and do not require the students to reflect on underlying concepts, as suggested by Fleming and Levie (1993) as well as Le (2016). Indeed, Aleven et al. (2016) suggest displaying such hints, which reveal the next step, only as a last resort after exhausting options for more principle-based hints. This begs the question why the focus of our work lies on such bottom-out hints.

First, edit hints are different from other bottom-out hints in that they display only a very small part of the solution (a single edit), allowing the student to finish most of the problem themselves. Second, bottom-out hints may lead to learning gains if students reflect on the hint and engage in sense-making behavior (Aleven et al., 2016; Shih et al., 2008). Conversely, if students aim to abuse the system, this is not hindered by principle-based hints: students simply skip through such hints to reach the bottom-out hint (Aleven et al., 2016; Shih et al., 2008). Third, we point to a study by Price et al. (2017b) which indicates that edit hints are judged as relevant and interpretable by human tutors. Finally, and most importantly, we argue that more elaborate hint strategies are simply not available in many important learning tasks because they require expert-crafted hint messages which are difficult to apply at scale (Le and Pinkwart, 2014; Murray et al., 2003; Rivers and Koedinger, 2015).

In particular, there have been some approaches to make expert-crafted hints available in larger state spaces, for example, authoring tools for tutoring systems, which aim at reducing the expert work that needs to be put in to design feedback for individual tasks. A prime example are the Cognitive Tutor Authoring Tools (CTAT), which support the construction of cognitive tutors (Aleven et al., 2006). Cognitive tutors can be seen as a gold standard of intelligent tutoring systems because their effectiveness has been established in classroom studies, and they have been successfully applied in classrooms across the US (Koedinger et al., 2013; Pane et al., 2014). However, even with authoring tools, covering all possible variations in a suficiently variable state space with many viable solutions may be infeasible (Le and Pinkwart, 2014; Murray et al., 2003; Rivers and Koedinger, 2015). For example, in our programming dataset (see Figure 1a), we consider more than 40 different solution strategies, each of which involves more than 40 steps.

Another approach is "force multiplication," which assumes that a relatively small number of expert-crafted hint messages are available, which are then applied to new situations automatically, thereby "multiplying the force" of expert work (Piech et al., 2015). Examples include the work of Choudhury et al. (2016), Head et al. (2017), as well as Yin et al. (2015) who apply clustering methods to aggregate many different states and then provide the same hint to all states in the same cluster. Another example is the work of Piech et al. (2015) who let experts develop hints which are annotated with example states for which the respective hint makes sense and example states for which the respective hint does *not* make sense. Then, they train a classifier function for each hint via machine learning which decides for any new state whether the hint should be displayed or not. Finally, Marin et al. (2017) annotate expert-crafted hints with small snippets of Java code for which the given hint makes sense and then display the hint whenever the respective snippet is discovered in a student's state. Note that these approaches are limited by the number of hints that are provided by the teaching experts. If for some situation no hint is contained in the database, the system is not able to provide fitting feedback.

Due to these scaling limitations, we focus on edit-based bottom-out hints, which are easy to individualize and generate automatically, as ample work in the past has demonstrated (Gross

and Pinkwart, 2015; Lazar and Bratko, 2014; Price et al., 2016; Rivers and Koedinger, 2015; Zimmerman and Rupakheti, 2015).

In the remainder of this section, we will analyze edit-based next-step hint approaches in more detail. We will highlight key concepts, provide precise definitions and shed some light on the theory behind edit-based hint approaches. We start our investigation by defining edits, legal move graphs, and edit distances in a rigorous fashion. Second, we discuss techniques to change our data representation in order to support meaningful hints. Third, we incorporate student data in the form of traces and interaction networks. Finally, we provide an overview of the different approaches that have emerged in the literature and compare them in light of our mathematical framework.

2.1. Edit Distances and Legal Move Graphs

Recall that we wish to support students in solving a multi-step learning task by providing on-demand edit hints. More precisely, we assume the following scenario. A student starts in some initial state provided by the system, and then successively edits this initial state until she finishes the task or gets stuck and asks the system for help. An edit hint for such a case should be an edit which gets the student closer to a desirable next state. To formalize this notion, we first define what kind of edits are possible at all (the *edit set*). Then, we define how to combine such edits to transform some student state into another (the *legal move graph*). Finally, we can define a notion of *distance* between states based on how many edits are necessary at least to transform one state into another (the *edit distance*).

The notion of an edit set should cover all actions which a student can perform to change their current partial solution to a different state. Recall our example of the guessing game programming task in Figure 1a. In this scenario, the set of possible states is the set of possible Snap programs. The possible edits are to add a block at any point in the program, replacing a block with another one, or deleting a block. For example, we may delete the "say 'Hello!' for 2 secs" block in Figure 1a or replace it with a "say 'Hello!' for 1 sec"-block. Note that this edit set is *symmetric*, in the sense that we can reverse every edit we have applied by deleting an inserted block, re-inserting a deleted block, or replacing a replaced block with its prior version. This is a desirable property for edit sets because it ensures that any action of a student can be reversed and thus any state that can be reached by edits from the initial state is reachable from each other state. Formally, we define edit sets as follows.

Definition 1 (Edit Set)**.** Let X be the set of all possible states for a learning task. Then we call X the *state space* of the learning task. An *edit* on X is a function $\delta : X \to X$. A set Δ of edits on X is called an *edit set*. We call an edit set *symmetric* if for all states $x \in X$ and all edits $\delta \in \Delta$ there exists an edit $\delta^{-1} \in \Delta$ such that $\delta^{-1}(\delta(x)) = x$. We call δ^{-1} an *inverse edit* for δ on x.

Formally, our goal is to devise a function which can, for any state students may find themselves in, return an edit they should apply. Inspired by Piech et al. (2015) we call such a function a *hint policy*.[2]

[2]Note that Piech et al. (2015) define a hint policy differently, namely as a function π' mapping a state to a state the student should proceed to next. Our definition presented here is a proper generalization of Piech et al.'s definition because every policy π according to our definition we can be converted into a Piech-style hint policy π' by setting $\pi'(x) = \delta(x)$ where $\delta = \pi(x)$.

Definition 2 (Hint Policy)**.** Let X be a set and Δ be an edit set on X. A *hint policy* is a function $\pi : X \to \Delta$.

A helpful hint policy should return a hint that gets students closer to a desirable next state. For example, the hint policy of Zimmerman and Rupakheti (2015) recommends the first edit in the shortest sequence of edits which transforms the student's current state to a correct solution for the task. To develop such a hint policy, we need to properly define what *closeness* between two states means and what the shortest sequence of edits is. To provide an intuitive meaning to closeness and shortest sequences, we can rely on graph theory. In particular, we can regard the state space as nodes of a directed graph and the edits as edges in that graph. More precisely, we draw an edge from a state x to another state y if and only if there is an edit in the edit sets Δ which transforms x into y. This results in the notion of a *legal move graph* (Piech et al., 2015).

Definition 3 (Legal Move Graph)**.** Let X be a set and Δ be an edit set on X. Then, the legal move graph according to X and Δ is defined as the directed graph $G_{X,\Delta} = (X, E)$ where $E = \{(x, y) | \exists \delta \in \Delta : \delta(x) = y\}$.

For our programming example in Figure 1a, the legal move graph is too large to list here. Instead, consider the set of strings $X = \{\text{a}, \text{aa}, \text{aac}, \text{ab}, \text{abc}, \text{b}, \text{bb}, \text{bbc}\}$ and as edit set Δ consider deletions, replacements, and insertions of single characters in these strings. More precisely, we have

$$\Delta = \left\{\text{del}_n, \text{ins}_{n,u}, \text{rep}_{n,u} | n \in \mathbb{N}, u \in \{\text{a}, \text{b}, \text{c}\}\right\} \quad \text{where} \tag{1}$$

$$\text{del}_n(v_1, \ldots, v_N) = v_1, \ldots, v_{n-1}, v_{n+1}, \ldots, v_N \tag{2}$$

$$\text{ins}_{n,u}(v_1, \ldots, v_N) = v_1, \ldots, v_n, u, v_{n+1}, \ldots, v_N \tag{3}$$

$$\text{rep}_{n,u}(v_1, \ldots, v_N) = v_1, \ldots, v_{n-1}, u, v_{n+1}, \ldots, v_N \tag{4}$$

An excerpt of the legal move graph for this example is shown in Figure 2a. In particular, "ab" is connected to "a", "aa", "b", "bb", and "abc" because we can delete b, replace b with a, delete a, replace a with b, and insert c to transform "ab" to the respective other strings. Note that all arrows in this legal move graph are bi-directional, indicating the symmetry of the edit set.

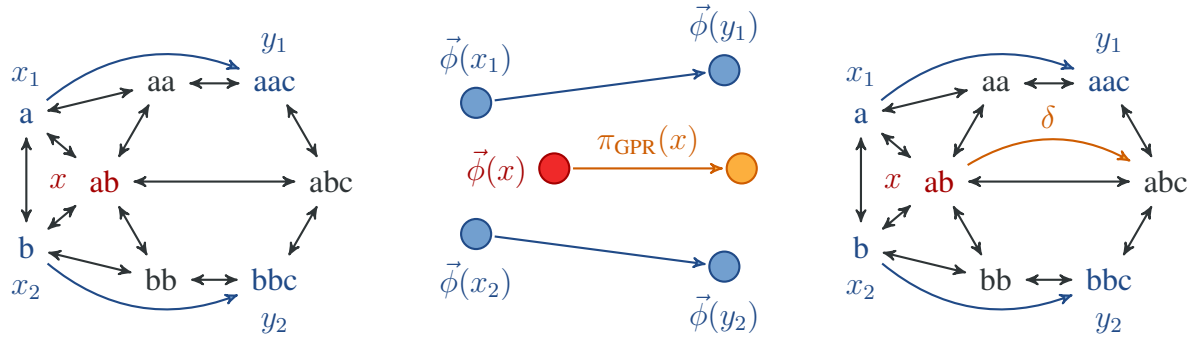
The notion of a shortest sequence of edits transforming a student's state x to a solution y now has a proper graph-theoretical interpretation. It is the shortest path from x to y in the legal move graph. The length of this shortest path is a *distance* d between x and y. As an example, consider once more the legal move graph in Figure 2a (right). Here we find, for example, $d(\text{ab}, \text{ab}) = 0$, $d(\text{ab}, \text{abc}) = 1$ and $d(\text{ab}, \text{bbc}) = 2$.

We implicitly assumed that all edges in a legal move graph have the length 1. We can generalize this notion by specifying the length of each edge via an *edit cost function*, which leads us to the concept of an *edit distance*.

Definition 4 (Edit Cost Function and Edit Distances)**.** Let X be a set and Δ be an edit set on X. A function $C : \Delta \times X \to \mathbb{R}^+$ is called an *edit cost function* on Δ. We call $C(\delta, x)$ the *cost* of applying edit δ to the state x.

We call an edit cost function *symmetric* if $C(\delta, x) = C(\delta^{-1}, \delta(x))$ for all states $x \in X$, all edits $\delta \in \Delta$, and all inverse edits δ^{-1} for δ on x.

Let $G_{X,\Delta}$ be the legal move graph according to X and Δ and let C be an edit cost function on Δ. The *edit distance* $d_{\Delta,C}$ according to Δ and C is defined as the shortest path distance in



(a) The legal move graph using the edit set of the string edit distance on the state space $X = \{\text{a}, \text{aa}, \text{aac}, \text{ab}, \text{abc}, \text{b}, \text{bb}, \text{bbc}\}$. $x = \text{ab}$ is the current student state (red). Further, two traces are given with the states $x_1 = \text{a}, y_1 = \text{aac}$, and $x_2 = \text{b}, y_2 = \text{bbc}$ respectively (blue).

(b) The embedding of the trace states (blue) and the student state (red) from the left into the edit distance space via the embedding $\vec{\phi}$. The recommendation of the Gaussian Process Regression (GPR) policy $\pi_{\text{GPR}}(x)$ for the current student state x is shown in orange.

(c) The legal move graph from the left figure, including the edit δ (orange) which corresponds to the recommended edit of GPR from the center figure.

Figure 2: An illustration of the Continuous Hint Factory on a simple dataset of strings. First, we compute pairwise edit distances between the student's current state (red) and trace data (blue). These edit distances correspond to the shortest paths in the legal move graph (left). The edit distances correspond to a continuous embedding, which we call the edit distance space (center). In this space, we can infer an optimal edit (orange) using machine learning techniques, such as Gaussian Process regression (GPR). Finally, we infer the corresponding hint in the original legal move graph (right), which can then be displayed to the student.

the legal move graph $G_{X,\Delta}$ with the edge weights $\beta(x, y) = \min_{\delta \in \Delta}\{C(\delta, x) | \delta(x) = y\}$. So the formula for the edit distance is:

$$d_{\Delta,C}(x, y) := \min_{\substack{\delta_1,\ldots,\delta_T \in \Delta \\ x_1,\ldots,x_T \in X}} \left\{ \sum_{t=1}^{T} C(\delta_t, x_t) \middle| x_1 = x, \delta_t(x_t) = x_{t+1}, \delta_T(x_T) = y \right\} \tag{5}$$

If no path between x and y exists, we define $d_{\Delta,C}(x, y) = \infty$.

Edit distances enable us to specify hint policies formally. For example, the policy of Zimmerman and Rupakheti selects for the input state x the closest correct solution y according to a given edit distance and returns the first edit on the shortest path between x and y.

Unfortunately, not all edit distances are applicable in practice. Consider the Snap example from Figure 1a. In this domain, the order of many blocks in the program is insignificant to the function of the program. Therefore, one may wish to apply an edit distance which works on unordered trees. However, edit distances on such unordered trees are NP-hard (Zhang et al., 1992), making them infeasible in practice. The subset of efficiently computable edit distances includes the following. First, the string edit distance (Levenshtein, 1965) from our example in Figure 2a. For this example, we can define the cost function as $C(\mathrm{del}_n, x) = C(\mathrm{ins}_{n,u}, x) = C(\mathrm{rep}_{n,u}, x) = 1$ for all $n \in \mathbb{N}, x \in X, u \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$. Note that C is symmetric. In this case, the overall edit distance can be computed in $\mathcal{O}(N^2)$ using a dynamic programming algorithm (Levenshtein, 1965). This dynamic programming scheme can be extended to a broad class of edit distances on strings, including skip operations and arbitrary (metric) cost functions (Giegerich et al., 2004; Paaßen et al., 2016). Such string edit distances have also been successfully applied to computer programs by representing them as sequences of code statements or as execution traces (Paaßen et al., 2016; Paaßen et al., 2016).

Second, we note the tree edit distance of Zhang and Shasha (1989) which permits deletions, insertions, and replacements of single nodes in trees. The tree edit distance has been particularly popular in learning environments for computer programming tasks as computer programs are oftentimes represented by *abstract syntax trees* (Choudhury et al., 2016; Freeman et al., 2016; Nguyen et al., 2014; Rivers and Koedinger, 2015). For example, the program shown in Figure 1a would correspond to the abstract syntax tree shown in Figure 3. The added complexity of tree data structures compared to strings is reflected in the higher computational complexity of the tree edit distance of $\mathcal{O}(N^4)$ (Zhang and Shasha, 1989). More flexibility is offered by a two-stage approach where some special subtrees, such as functions in a program, may be arbitrarily re-ordered and a string or tree edit distance is used to compare the single functions (Mokbel et al., 2013; Price et al., 2017a). A computationally cheaper alternative has been suggested by Zimmerman and Rupakheti (2015) who compute an edit distance on pq-grams in trees, meaning small subtrees, which results in a considerably faster runtime of $\mathcal{O}(N \cdot \log(N))$ (Augsten et al., 2008).

Beyond computational complexity, a key challenge to edit distances is that they do not necessarily correspond to the *semantic* distance between states. Consider again the Snap example in Figure 1a. Here, we could replace any of the strings in "say" or "ask" blocks with a slightly different version without changing the basic computed function of the program. We could also exchange the "too high" and "too low" statements in the program if we also replace the "random < answer" comparison with a "random > answer" comparison. In theory, we can apply arbitrarily many edits to a given program without changing the computed function. Conversely,

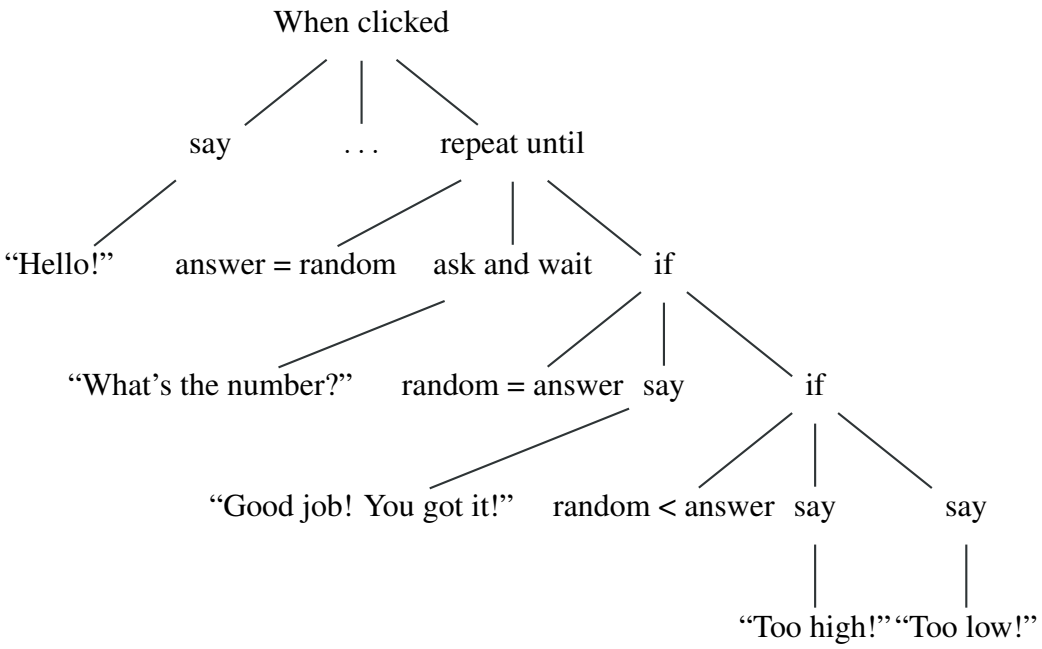


Figure 3: An abstract syntax tree, simplified for clarity, corresponding to the Snap program shown in Figure 1a.

we could also remove the “repeat until” block in the program, changing only a single block, and severely change the behavior of the program. This mismatch between edit distance and semantic distance can negatively impact the utility of generated hints. In particular, edits may be recommended which get the student syntactically closer to a correct solution but may be semantically irrelevant or even confusing.

One approach to address this issue is *canonicalization*, which essentially transforms the raw states in a state space X to a canonic form, such that semantically equivalent states have the same canonic form. The edit distance is then defined between canonic forms instead of raw states, leading to a smaller legal move graph and edits which put stronger emphasis on semantically relevant changes. Canonicalization is particularly common for computer programs, where the order of binary relations (such as $<$) or variable names can be normalized or unreachable code can be removed (Rivers and Koedinger, 2012). Note that canonicalization is a quite general and powerful concept. In particular, canonicalization enables us to apply an edit distance to any kind of data, as long as we can provide a canonicalization f which converts this data to editable canonic forms. For example, Paaßen et al. (2016) canonicalize computer programs by representing them in terms of their execution trace, to which simple string edit distance measures can be applied. As such, a canonicalization f can also be seen as an extension of an edit distance d to a new domain by defining the edit distance $\tilde{d}$ on the new domain as $\tilde{d}(x, y) = d(f(x), f(y))$.

A challenge in canonicalization lies in the fact that edits on the canonic form may not be directly applicable or interpretable for students. For example, students cannot easily adapt their program to directly influence the program’s execution in the way indicated by an edit on the execution trace. To address this problem, Rivers and Koedinger (2015) suggest aligning the edits on the canonic form with the student’s original state in a process called *state reification*. Another

challenge lies in the fact that too drastic canonicalization may remove features of the original state for which feedback would be desirable. For example, tutoring systems for computer programming often not only intend to teach functionally correct programming but also programming style, such that important stylistic differences, even though semantically irrelevant, need to be preserved in the canonic form (Piech et al., 2015; Choudhury et al., 2016).

Another approach to adapting an edit distance is offered by *metric learning*. Instead of mapping two semantically equivalent states x and y to the same canonic form, metric learning lowers the cost of edits which transform x to y, such that the edit distance between x and y is reduced (Paaßen et al., 2016). This makes it possible to smoothly regulate the emphasis on semantics, style, and syntax and keeps the legal move graph intact, such that state reification is unnecessary. On the other hand, metric learning is limited by the neighborhood structure in the legal move graph and thus can not easily map equivalent states which are far apart in the legal move graph to the same point. One potential approach is to first apply (mild) canonicalization in order to map distant but semantically equivalent states to the same or similar canonic forms and subsequently apply metric learning such that the resulting distance measure mostly takes semantics into account but does not disregard stylistic differences entirely.

In summary, the concepts of edit set, legal move graph, and edit distance permit us to define a hint policy that recommends the first edit on the shortest path to a solution (Zimmerman and Rupakheti, 2015). In order to ensure that such hints are helpful, we require an edit distance which roughly corresponds to the semantic distance, but also takes important stylistic features into account. Still, our definitions are insufficient to select helpful edits as hints reliably. In particular, there may be many equally short paths toward a correct solution, and we do not know which one to take. Consider the Snap programming example from Figure 1a again and assume that a student requests help without having written any code. In that case, to get to the closest correct solution in Figure 1a, we need to insert all code statements of the abstract syntax tree in Figure 3, and a priori each of these insertions is an equally valid hint. Past research suggests that we should focus on insertions which other students have performed in the same situation, which brings us to the notion of traces and interaction networks.

2.2. Traces and Interaction Networks

If we wish to capture what students have done in the past, we require a formal notion of student movement through the state space. This is offered by the notion of a trace suggested by Eagle et al. (2012)[3].

Definition 5 (Trace)**.** Let X be a set and Δ be an edit set on X. Then, a sequence $x_1, \delta_1, \ldots, \delta_{T-1}, x_T$ is called a *trace* if for all t $x_t \in X$, $\delta_t \in \Delta$ and $x_{t+1} = \delta_t(x_t)$.

We are particularly interested in how close a trace is to a new student's solution. To answer this question, we need to couple the notion of a trace with the notion of the legal move graph. This coupling is provided by the notion of an interaction network (Eagle et al., 2012) and a solution space (Rivers and Koedinger, 2014; Rivers and Koedinger, 2015).

Definition 6 (Interaction Network)**.** Let X be a set and Δ be an edit set on X. Further, let $\{(x_1^i, \delta_1^i, \ldots, \delta_{T_i-1}^i, x_{T_i}^i)\}_{i=1,\ldots,N}$ be a set of traces. The *interaction network* corresponding to

[3]Note that this definition is not exactly equivalent to the one given by Eagle et al. (2012). In particular, they do not require actions to be *deterministic*, that is, in their framework the same action applied to the same state may lead to different subsequent states. For the sake of brevity, we refrain from this probabilistic extension here.

this set of traces is defined as the graph $G = (V, E)$ where

$$V = \left\{ x_t^i \middle| i \in \{1, \ldots, N\}, t \in \{1, \ldots, T_i\} \right\} \tag{6}$$

$$E = \left\{ (x_t^i, x_{t+1}^i) \middle| i \in \{1, \ldots, N\}, t \in \{1, \ldots, T_i - 1\} \right\} \tag{7}$$

We also call V a *solution space*.

Consider the example shown in Figure 2a, which shows two traces in blue. These traces cover the strings "a," "aac," "b," and "bbc". Therefore, the interaction network for this case would only contain these four strings and the edges ("a," "aac") as well as ("b," "bbc").

Ideally, the edit set used by students exactly corresponds to the edit set of the legal move graph, but there are many cases where this condition may not hold. For example, the edit sets might be different because students work on a different representation compared to the representation used to compute edit distances, for example, due to canonicalization (Rivers and Koedinger, 2012; Rivers and Koedinger, 2015). Another reason may be that student's states may be recorded only at certain points in time (e.g., when they explicitly hit a "save" button) such that multiple actions may have been performed since the last recorded state. Finally, the concrete actions of a student may not be available because the students work on the task off-line and submit their current state only if they need help from the system. In these cases, we have to assume that students may "jump" in the legal move graph from state to state and the exact path they have taken needs to be inferred by the system (Piech et al., 2015).

With a formal notion of the actions of past students, we have now aggregated all concepts we need to provide an integrated view of existing hint policies in the literature.

2.3. Hint policies

Recall the definition of a hint policy (Definition 2). We are referring to a function which outputs an edit for each possible input state. In the remainder of this section, we are going to provide a short review of the hint policies that have been suggested in the literature.

The arguably simplest policy is the one of Zimmerman and Rupakheti (2015), which always recommends the first edit on the shortest path to the closest solution. Such an approach does not even require student data, except for at least one example of a correct solution of the task. A drawback of the Zimmerman policy is that it does not consider whether the edits towards the closest correct solution correspond to critical steps toward a solution or relatively unimportant stylistic differences. Rivers and Koedinger (2015) address this issue in their Intelligent Teaching Assistant for Programming (ITAP), an intelligent tutoring system for Python programming. Their technique involves the following steps: First, they retrieve the closest solution according to the tree edit distance on canonic forms. Second, they use the edits which transform the student state into the closest correct solution to construct intermediate states. Third, of these intermediate states, the one with the highest desirability score is selected for feedback, where the desirability score is a weighted sum of the frequency in past student data, the distance to the student's state, the number of successful test cases the state passes, and the distance to the solution (Rivers and Koedinger, 2015). Finally, an inverse canonicalization (state reification) step is applied to infer edits that can be directly applied to the student's state to transform it to the selected state. This approach has been shown to provide helpful edits in almost all cases for a broad range of tasks (Rivers and Koedinger, 2015). Note that the success of the Rivers policy

hinges upon meaningful frequency information. If no or little frequency information is available, the hints provided by the Rivers policy may not be representative of generic steps toward a solution but rather of specificities of the reference solution that was selected.

The approaches of Zimmerman and Rupakheti (2015) as well as Rivers and Koedinger (2015) have been part of a study by Piech et al. (2015) who compared several hint policies with expert recommendations on a large-scale dataset consisting of over a million states from the *Hour of Code* Massive Open Online Course (MOOC). The data was collected from two beginner's programming tasks in a block-based programming language. They found that the policy which agreed most with the recommendations of tutors was a variant of the Zimmerman policy, in which the cost of an edit δ was determined based on the inverse frequency of the target state in the dataset, that is, edits which lead to less frequent states were considered more expensive (Piech et al., 2015). This approach makes states appear closer when they can be reached by crossing states that have been visited often. Note that this approach critically relies on frequency information, which may not be available in sparsely populated spaces, where almost no state is visited more than once.

An alternative approach to approaching the closest correct solution directly is to guide students along a trace, as proposed by Gross and Pinkwart (2015) in the *JavaFIT* system[4]. They distinguish between two types of help-seeking behavior, namely searching for errors or searching for a next-step. In both cases, they retrieve the closest state to the student's state in the interaction network. If students are trying to find an error in their code, the system recommends an edit leading the student toward this reference state, thereby attempting to correct the error. If students assume that their current state is correct, but they are looking for a next step, the system recommends an edit toward the *successor* of the reference state, thereby guiding the student closer to a solution (Gross and Pinkwart, 2015). This policy can be seen as an instance of *case-based reasoning*, where recommendations are based on a similar case from an underlying case base. Freeman et al. (2016) have taken this view to analyze Python programs and used a weighted tree edit distance to retrieve similar cases. Also similar to case-based reasoning, Gross et al. (2014) proposed example-based feedback, in which the closest prototypical state in a dataset is retrieved and shown to the student to elicit self-reflection and sense-making in order to improve their own state. If the closest state in the case base is sufficiently similar to the student's state and corresponds to a capable student, such an approach can provide hints which emulate the actions of a capable "virtual twin" of the student. However, if only few reference solutions exist, the selected next state may still be fairly dissimilar and edits toward the next state may include not only error-correcting hints or next-step hints but also stylistic or strategic choices which do not correspond to the student's goals.

Lazar and Bratko (2014) propose yet a different approach by applying edits that have been frequent in past student traces to manipulate the current student's state until an edit is found that achieves better unit test scores. As with the Piech policy, the policy of Lazar and Bratko (2014) critically relies on frequency information, albeit for edits instead of states, which may not always be available. Furthermore, edits which may be generally important for a task may not necessarily be helpful in a specific situation.

An alternative view is provided by the Hint Factory, which analyzes the question of choosing the optimal edit in a given state in a mathematically precise fashion via Markov Decision Processes (Barnes and Stamper, 2008). In particular, the Hint Factory always returns the edit

[4] https://javafit.de/

which maximizes the expected future reward, where a reward is given whenever a student has achieved a correct solution. The Hint Factory was originally created as a hint-generation add-on to the *DeepThought* instruction system for deductive logic (Barnes and Stamper, 2008). Several studies have demonstrated that the Hint Factory reduces student dropout and helps students to complete more problems more efficiently (Stamper et al., 2012; Eagle and Barnes, 2013). The Hint Factory has also been applied to further domains, such as the serious game BOTS (Hicks et al., 2014) or the SNAP programming environment (Price et al., 2017).

Note that the Markov Decision Process model relies on an estimate of the transition probability distribution $P(x'|x, \delta)$ of moving to state x' from x via the edit δ. The Hint Factory estimates this probability distribution based on transition frequencies in the trace data and therefore requires meaningful frequency information. As such, the Hint Factory can provide hints only for states that are part of the interaction network, and for which a directed path to a correct solution in the interaction network exists. This has been dubbed the *hintable subgraph* (Barnes et al., 2016). In practice, students may move outside the hintable subgraph. Indeed, research has shown that for a reasonably small, open-ended programming task, over 90% of states are visited only once, indicating that future students will likely visit states that have not been seen before and may not even be connected to previously seen states in the legal move graph (Price and Barnes, 2015). Also note that the number of unique states remained high even after applying harsh canonicalization (Price and Barnes, 2015). This result matches our own two datasets, where 97.23% and 82.79% of states were visited only once. So how can the Hint Factory be extended to such sparsely populated state spaces? A first approach has been proposed by Price et al. (2016), who suggest *contextual tree decomposition* (CTD) which generates interaction networks only for small subtrees of the students' abstract syntax trees. Due to the size limitation, the state space for each subtree is significantly smaller and thus more densely populated with student data. However, the approach faces an ambiguity challenge in that one hint is generated for each (small) subtree of the student's state, and the student or the system has to select from these possible hints (Price et al., 2017a).

Overall, we observe that all previous approaches are either limited by their reliance on frequency data, namely the Hint Factory, the Piech policy, and the Lazar policy, or by generating hints based only on a single reference solution, namely the Zimmerman policy, the Gross policy, or the Rivers policy. Our approach is an attempt to generate hints based on *multiple* reference solutions, but without relying on frequency information. More specifically, we use a weighted average of multiple reference solution to express a virtual state to which the student should move, and we select the weights for this average such that the resulting virtual state corresponds to the probabilistically optimal next state of a capable student. As such, we use the same basic approach as the Hint Factory, in that we also try to bring the student closer to the next state of capable students in the same situation. However, we extend the Hint Factory by basing our prediction not on frequency counting, but on the movements of students in *similar* situations through the space of possible solutions. This state of possible virtual solutions, expressible as weighted averages of states we have seen before, is continuous; hence the name *Continuous Hint Factory* (CHF).

Note that embedding states in a continuous state has already been proposed by Piech et al. (2015), who constructed such an embedding via neural networks. The embedding is computed by executing the programs on example data and recording the variable states P before executing a block of code A as well as the variable states Q after A has been executed. Both P and Q are embedded in a common space via a single-layer neural network, yielding the representations f_P

and f_Q. Then, a matrix M_A is constructed which maps f_P to f_Q, that is, M_A is constructed such that $f_Q \approx M_A \cdot f_P$. This matrix M_A is the embedding of the code block A (Piech et al., 2015). However, this work has two crucial limitations: First, it relies on a task-specific representation of f_P and f_Q, which is generated via execution, whereas the CHF only relies on edit distances, which are not task-specific (Mokbel et al., 2013). Second, we provide a technique to convert the predictive result in the continuous space to an actual, human-readable edit, which the Piech approach lacks.

We also note connections to other approaches cited before. First, the CHF is connected to the work of Gross and Pinkwart (2015), in that we also recommend following the actions of students in a similar situation, but we integrate knowledge of more than one student. Second, similar to the work of Lazar and Bratko (2014), we recommend edits which are frequent in past student data but focus on those edits which have been applied in similar states. Finally, we incorporate many of the key concepts and approaches of Rivers and Koedinger (2015), in that we also apply canonicalization, and build upon the concept of path construction, a desirability score, as well as state reification to infer an edit which corresponds to the optimal hint in the embedding space. However, we extend this approach by considering not only edits toward the closest correct solution but edits toward all reference solutions and by replacing their desirability score with the distance to the recommended next state in the edit distance space. This alternative score incorporates the spirit of many of the criteria proposed by Rivers and Koedinger (2015), as it also punishes going too far away from the student's current solution, rewards getting closer to the goal, and represents what other students generally did, but it relies neither on frequency information, nor on an expert-chosen weighting between the different criteria.

In the next section, we introduce the CHF in more detail.

3. Continuous Hint Factory

The goal of the Continuous Hint Factory (CHF) is to identify a state which represents where students should move next on their way towards a correct solution and to recommend an edit which gets as close as possible to this selected state. In doing so, we also want to be able to select next states which are not contained in the data of past data but can be represented as mixtures of previously seen states.

To implement this goal, the CHF involves three steps. First, we embed past student data in a continuous space by means of an edit distance. In this space, we can represent any mixture of known states as a vector. Second, we develop a hint policy in this embedding space based on Gaussian process prediction for structured data (Paaßen et al., 2017). This policy returns what a capable student would do in the respective input state, represented as a mixture of states from traces of such capable students. In a final step, we transform this mixture into a human-readable edit by selecting the edit which brings us closer to all states with positive mixture coefficients and further away from all states with negative mixture coefficients.

In the remainder of this section, we describe each of these three steps - embedding in the edit distance space, prediction, and pre-image identification - in turn.

3.1. The Edit Distance Space

In a first step, we embed the states of past students in a Euclidean vector space, such that the edit distances between any two states x and y corresponds to the Euclidean distance between the

corresponding vectors $\vec{\phi}(x)$ and $\vec{\phi}(y)$. Such a space is necessary to ensure that "mixing" states becomes meaningful. If we have vectors $\vec{\phi}(x)$ and $\vec{\phi}(y)$ which correspond to x and y, we can easily mix those two vectors by adding them or subtracting them from each other.

A simple example of such an embedding is shown in Figure 2b (left). In this figure, the strings "a", "b", "ab", "aa", "aac", "bb", "bbc", and "abc" are embedded in a two-dimensional space, namely the two dimensions of the paper. The Euclidean distance between the strings on the paper approximately corresponds to their edit distance, that is, strings with an edit distance of 2, such as "ab" and "aac", are about twice as far away from each other on the paper compared to strings with an edit distance of 1, such as "ab" and "aa". Note that this embedding is not exact. For example, the distance on the paper between "a" and "b" is much larger compared to the distance between "a" and "ac", even though in both cases the edit distance is 1. Another example is shown in Figure 1b. This figure displays ten traces of students working on the guessing game task from Figure 1a, such that the distance between states roughly corresponds to the edit distance between them. In both cases we observe that finding a vectorial embedding, such that the edit distance corresponds exactly to the Euclidean distance in the embedding, is not trivial. Importantly, though, theoretical results show that such an embedding does always exist.

Theorem 1 (Latent Distance Space)**.** *Let X be some finite set and $d : X \times X \to \mathbb{R}$ be a function such that for all $x, y \in X$ it holds: $d(x, x) = 0$, $d(x, y) \geq 0$ and $d(x, y) = d(y, x)$. Then, there exists a vector space $\mathcal{Y} \subset \mathbb{R}^s$ for some $s \in \mathbb{N}$, and a mapping $\vec{\phi} : X \to \mathcal{Y}$, such that for all $x, y \in X$:*

$$d(x, y)^2 = (\vec{\phi}(x) - \vec{\phi}(y))^T \cdot \Lambda \cdot (\vec{\phi}(x) - \vec{\phi}(y)) \tag{8}$$

where Λ is a diagonal matrix with entries in $\{-1, 0, 1\}$.

Proof. Refer to Theorem 1 in Hammer and Hasenfuss (2010) as well as page 122 in Pekalska and Duin (2005). □

Note that if Λ has no negative entries, d corresponds to the Euclidean distance in the embedding space. Otherwise, there exist points in the latent vector space $\mathcal{Y}$ for which the pairwise distance becomes *negative*, which may cause errors in subsequent processing. These negative entries can be addressed by various *eigenvalue correction* techniques, such as setting the negative entries to zero (clip eigenvalue correction), replacing them with their absolute value (flip eigenvalue correction) or adding an offset to Λ, such that all entries become positive (Gisbrecht and Schleif, 2015). Note that this correction is an *approximation* and does distort the distances, but only to the extent to which negative entries are present.

Based on this embedding, we can prove the main theorem of our work, namely that for any symmetric edit distance we can find an Euclidean embedding, which we call the edit distance space.

Theorem 2 (Edit Distance Space)**.** *Let $V \subset X$ be a solution space taken from a state space X, let Δ be a symmetric edit set on X and let $C : \Delta \times X \to \mathbb{R}^+$ be a symmetric* edit cost function *on Δ. Further, let D be the eigenvalue-corrected version of the distance matrix with entries $D_{ij} = d_{\Delta,C}(x_i, x_j)^2$ for all $x_i, x_j \in X$.*

Then, there exists a vector space $\mathcal{Y} = \mathbb{R}^s$ for some $s \in \mathbb{N}$ which we call the edit distance space *for $d_{\Delta,C}$; and there exists a mapping $\vec{\phi} : X \to \mathcal{Y}$, such that for all $x, y \in V$ it holds:*

$\|\vec{\phi}(x) - \vec{\phi}(y)\|_2^2 = D(x, y)$, *i.e. the Euclidean distance in* $\mathcal{Y}$ *corresponds to* $d_{\Delta,C}$*, up to eigenvalue correction.*

Proof. We first show that, under our constraints on Δ and C, the resulting edit distance $d_{\Delta,C}$ fulfills the constraints of Theorem 1.

$d_{\Delta,C}(x, y) \geq 0$**:** If x and y are connected in the legal move graph, $d_{\Delta,C}(x, y)$ is a sum of non-negative contributions (because C is non-negative), and thus $d_{\Delta,C}(x, y) \geq 0$. Otherwise $d_{\Delta,C}(x, y) = \infty > 0$.

$d_{\Delta,C}(x, x) = 0$**:** For all x we can use the empty edit sequence to transform x to x. The cost of the empty edit sequence is 0, independent of the cost function C. As we have shown that $d_{\Delta,C}(x, x) \geq 0$, there can also be no cheaper edit sequence. Therefore, we obtain $d_{\Delta,C}(x, x) = 0$

$d_{\Delta,C}(x, y) = d_{\Delta,C}(y, x)$**:** Let $x, y \in X$ such that x and y are connected in the legal move graph. Let $\delta_1, \ldots, \delta_T$ be a sequence of edits that transforms x to y such that the cost is minimal. Because Δ is symmetric, we can construct the sequence of edits $\delta_T^{-1}, \ldots, \delta_1^{-1}$ which transforms y to x. Because C is symmetric we know that the cost of this path is equal to the cost of $\delta_1, \ldots, \delta_T$, which in turn implies $d_{\Delta,C}(x, y) \geq d_{\Delta,C}(y, x)$. We can do the same argument in the other direction (from y to x), such that $d_{\Delta,C}(x, y) \leq d_{\Delta,C}(y, x)$, which implies $d_{\Delta,C}(x, y) = d_{\Delta,C}(y, x)$. If there is no path from x to y in the legal move graph, then there is also no path from y to x, and it holds $d_{\Delta,C}(x, y) = \infty = d_{\Delta,C}(y, x)$.

Theorem 1 now yields the required embedding. Because of eigenvalue correction, this embedding is Euclidean. □

Note that the construction of the edit distance space depends on example data. How we select this example data is crucial for a viable edit distance space and, in turn, for a helpful hint policy. We suggest selecting example data with two heuristics. First, we should limit ourselves to data of successful students, meaning data of students who did end up in a correct solution to the task. Second, we should incorporate the goal-directedness criterion suggested by Rivers and Koedinger (2014), that is, we should only incorporate those intermediate solutions which get closer to the correct solution the student ended up in.

In our approach, we make extensive use of the edit distance space. In particular, we replace the problem of finding a hint policy for the original edit set of the edit distance by finding a hint policy in the edit distance space.

3.2. A Hint Policy in the Edit Distance Space

Due to Theorem 2 we know that, for a symmetric edit distance $d_{\Delta,C}$, there exists an embedding in a vector space $\mathcal{Y} \subset \mathbb{R}^s$, such that the edit distance corresponds to the Euclidean distance in $\mathcal{Y}$ after eigenvalue correction. The main advantage of the edit distance space $\mathcal{Y}$ is that constructing a hint policy for vectors is much easier compared to a hint policy for arbitrary states. In particular, we can define edits in a vector space as vectors which are added to the input states. Consider the example edit distance space in Figure 2b. In this space, string edits become vectors, displayed here as blue and orange arrows, and the resulting state after applying the edit corresponds to adding the vector to the original state.

In formal terms, we define the edit set $\Delta_{\mathcal{Y}}$ and the cost function $C_{\mathcal{Y}}$ in the edit distance space as follows.

$$\Delta_{\mathcal{Y}} = \{\delta_{\vec{\xi}} | \vec{\xi} \in \mathcal{Y}\} \quad \text{where} \quad \forall \vec{\phi} \in \mathcal{Y} : \delta_{\vec{\xi}}(\vec{\phi}) = \vec{\phi} + \vec{\xi} \quad \text{and} \quad C_{\mathcal{Y}}(\delta_{\vec{\xi}}, \vec{\phi}) = \|\vec{\xi}\| \tag{9}$$

The edit distance resulting from this definition of edit set $\Delta_{\mathcal{Y}}$ and edit cost function $C_{\mathcal{Y}}$ in the edit distance space is provably equivalent to the original edit distance, up to eigenvalue correction.

Theorem 3. *Let $V \subset X$ be some solution space from a state space X, let Δ be a symmetric edit set on X and let $C : \Delta \times X \to \mathbb{R}$ be a symmetric* edit cost function *on Δ. Further, let D be the eigenvalue-corrected version of the matrix with entries $D_{ij} = d_{\Delta,C}(x_i, x_j)^2$ for all $x_i, x_j \in V$. Finally, let $d_{\mathcal{Y}}(x_i, x_j)$ be the edit distance in the edit distance space according to $\Delta_{\mathcal{Y}}$ and $C_{\mathcal{Y}}$.*

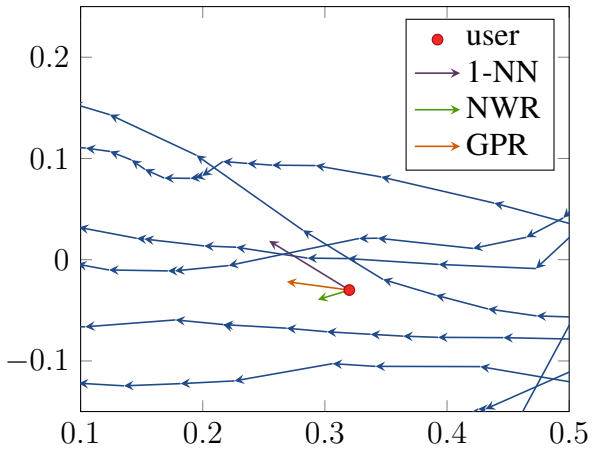
Then, for all $x_i, x_j \in V$ it holds: $D_{i,j} = d_{\mathcal{Y}}(\vec{\phi}(x_i), \vec{\phi}(x_j))^2$, i.e. the edit distance in the edit distance space is equivalent to the original edit distance $d_{\Delta,C}$, up to eigenvalue correction.

Proof. Recall that Theorem 2 already shows that $D_{i,j}$ corresponds to the squared Euclidean distance $\|\vec{\phi}(x) - \vec{\phi}(y)\|_2^2$. It remains to show that the edit distance in the edit distance space $\mathcal{Y}$ is equivalent to the Euclidean distance in the edit distance space as well.

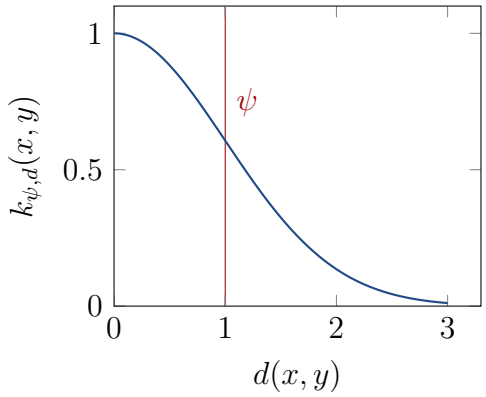
Let $\vec{\phi}(x), \vec{\phi}(y) \in \mathcal{Y}$. Then, the edit $\delta_{\vec{\phi}(y)-\vec{\phi}(x)}$ is in $\Delta_{\mathcal{Y}}$. Therefore, $d_{\mathcal{Y}}(\vec{\phi}(x), \vec{\phi}(y))$ is at most $\|\vec{\phi}(y) - \vec{\phi}(x)\|$. Due to the triangular inequality on the Euclidean distance we also know that there exists no point $\vec{\phi}(z) \in \mathcal{Y}$, such that $\|\vec{\phi}(x) - \vec{\phi}(z)\| + \|\vec{\phi}(z) - \vec{\phi}(y)\| < \|\vec{\phi}(x) - \vec{\phi}(y)\|$. Therefore, there can exist no sequence of edits which is cheaper than $\|\vec{\phi}(y) - \vec{\phi}(x)\|$, which concludes the proof. □

Thanks to this theorem we can replace a hint policy in the original space with a hint policy in the edit distance space. We can infer such a hint policy based on example data of successful students. In particular, assuming a dataset of traces of successful students, we denote any state in one of these traces as x_i and the next state in the trace as y_i. If x_i is the final solution of a trace, then $y_i = x_i$. Constructing a hint policy in the edit distance space corresponds to finding a function which outputs for any $\vec{\phi}(x_i)$ the edit vector $\vec{\phi}(y_i) - \vec{\phi}(x_i)$. In machine learning terms, this is a *regression problem*. The simplest approach to this problem would be one-nearest neighbor regression (1-NN), which looks for the closest data point in the database and returns the edit that has been done for that point, that is, $\pi_{\text{1NN}}(\vec{\phi}(x)) = \vec{\xi}_i$ where $d(x, x_i)$ is minimal. Unfortunately, such a policy tends to overestimate the importance of particularities of individual traces. Consider the example shown in Figure 2b. We wish to provide a hint for point $\vec{\phi}(x)$, shown in red. Our trace data consists of the points x_1 and y_1 as well as x_2 and y_2. Both x_1 and x_2 are equally close to x. Still, one-nearest neighbor has to arbitrarily choose between both points and predict the edit corresponding to only one. In practical examples, we observe similar problems. Consider the excerpt of the Snap dataset shown in Figure 4a and assume we want to provide feedback for a state located at the red point. In that case, a one-nearest neighbor policy would recommend a movement in upwards direction which is not representative of the overall movement in the data, which rather goes straight to the left.

We can address the overemphasis of individual particularities by integrating information from multiple student traces. In particular, we propose to construct a hint policy which returns a *weighted average* of all past edits, that is $\pi(x) = \sum_{i=1}^{M} \gamma_i(x) \cdot \vec{\xi}_i$, where $\gamma_i(x) \in \mathbb{R}$ are numeric weights. These weights should result in a compromise between past data, such that individual



(a) An excerpt of the embedding for the Snap guessing game task from Figure 1b (blue), a hypothetical new student state (red) and the outputs of a one-nearest neighbor hint policy (1-NN), a Nadaraya-Watson kernel regression hint policy (NWR) and a Gaussian Process hint policy (GPR) for this state, based on a length scale of $\psi = 0.5$.



(b) The output of the radial basis function (RBF) kernel (y-axis) for distances between two states x and y in the range $[0, 3]$ (x-axis) and a length-scale of $\psi = 1$ (red line).

Figure 4: Two figures illustrating regression techniques (left) and the radial basis function kernel (right).

particularities are averaged out, and general trends in the data are emphasized. In the example in Figure 2b, we could simply set the weights for both traces to an equal value, resulting in a viable edit (shown in orange).

The key question in such an averaging approach is how to obtain the weights $\gamma_i(x)$. The simplest way is to scale $\gamma_i(x)$ inversely to the distance $d(x, x_i)$ between x and x_i, i.e., such that states x_i which are close to x are emphasized. For that purpose, we can define $\gamma_i(x)$ as $k(x, x_i)$ divided by $\sum_{j=1}^{M} k(x, x_j)$ where k may be any function which decreases monotonically with rising edit distance $d(x, x_i)$. A particularly popular choice is the radial basis function (RBF) or Gaussian function which is defined as $k_{\psi,d}(x, y) = \exp(-0.5 \cdot \frac{d(x,y)^2}{\psi^2})$ (Rasmussen and Williams, 2005). The RBF assigns a value of 1 for a distance of 0 and assigns lower values for higher distance, quickly approaching 0. ψ is a hyper-parameter called *length-scale* which controls the distance value at which $k_{\psi,d}$ reaches its maximum slope (see Figure 4b).

This way of setting $\gamma_i(x)$ is called *Nadaraya-Watson kernel regression* (NWR) and oftentimes yields more accurate predictions compared to simple 1-NN (Paaßen et al., 2017). However, NWR is highly sensitive to the choice of ψ. In particular, for small ψ values, NWR is equivalent to 1-NN, and for large ψ values, NWR overemphasizes distant states, as is visible in Figure 4a. Here, NWR recommends a downward motion, which does not reflect the movement in the local environment. An alternative to NWR is *Gaussian Process Regression* (GPR), which is well-justified via probability theory (Rasmussen and Williams, 2005), and has been shown to be more accurate in prediction (Paaßen et al., 2017). To apply GPR, the function k needs to be a *kernel function*. For an exact definition of kernel functions, we point to the work of Rasmussen

and Williams (2005). For our purposes here, it suffices to state that the RBF is one such kernel function.

Let now $\vec{\mathrm{k}}(x)$ be the row vector of kernel values $k(x, x_i)$ for all $i = 1, \ldots, M$ and let $\boldsymbol{K}$ be the matrix of pairwise kernel values $k(x_i, x_j)$ for all $i = 1, \ldots, M$ and $j = 1, \ldots, M$. Then, for the point $\vec{\phi}(x)$, we define the result of the GPR hint policy as $\pi_{\text{GPR}}(x) = \sum_{i=1}^{M} \gamma_i(x) \cdot \vec{\xi}_i$, where the vector $\vec{\gamma}(x) = (\gamma_1(x), \ldots, \gamma_M(x))$ is computed as follows (Paaßen et al., 2017).

$$\vec{\gamma}(x) = \vec{\mathrm{k}}(x) \cdot \left(\boldsymbol{K} + \tilde{\sigma}^2 \cdot I^M\right)^{-1} \tag{10}$$

and where $\tilde{\sigma}$ is a hyper-parameter which quantifies the assumed amount of noise in our data and I^M is the $M \times M$ identity matrix. Increasing $\tilde{\sigma}$ tends to decrease the accuracy of GPR but enhances smoothness in the predictions as well as numerical stability (Rasmussen and Williams, 2005).

We highlight two key properties of GPR. First, if x is equal to some state x_i in the recorded trace data, then $\vec{\mathrm{k}}(x)$ equals the ith column in the matrix $\boldsymbol{K}$. The product $\vec{\mathrm{k}}(x) \cdot \boldsymbol{K}^{-1}$ is then equal to a vector of zeros which is only one at position i. So for $\tilde{\sigma} = 0$, the GPR policy will return exactly the vector $\vec{\xi}_i$. Second, if x is distant from all states x_i in the trace data, $\vec{\mathrm{k}}(x)$ is approximately a zero vector, such that the hint recommended by the GPR policy degrades to zero. In other words, if the student's solution is dissimilar to everything we have seen before, the GPR hint policy cannot provide feedback. While this limits coverage, it also implies that the GPR hint policy automatically detects where its hints may not be useful anymore, namely in case of truly novel strategies for which new example data are required.

As an example for the application of the GPR hint policy, consider the string edit distance example shown in Figure 2b. Note that the string edit distances are: $d_{\Delta,C}(x, x_1) = d_{\Delta,C}(x, x_2) = 1$ and $d_{\Delta,C}(x_1, x_2) = d_{\Delta,C}(x_2, x_1) = 1$. For the length scale $\psi = 1$ and a noise variance $\tilde{\sigma}^2 = 0$ we obtain

$$\vec{\mathrm{k}}(x) = (\frac{1}{\sqrt{e}}, \frac{1}{\sqrt{e}}), \quad \boldsymbol{K} = \begin{pmatrix} 1 & \frac{1}{\sqrt{e}} \\ \frac{1}{\sqrt{e}} & 1 \end{pmatrix}, \text{ and } \quad \vec{\gamma}(x) = \vec{\mathrm{k}}(x) \cdot \boldsymbol{K}^{-1} \approx (0.3775, 0.3775).$$

Thus, the recommended edit, shown as orange arrow, is $\pi_{\text{GPR}}(x) \approx 0.3775 \cdot \vec{\xi}_1 + 0.3775 \cdot \vec{\xi}_2$. For the Snap example, the result of the GPR hint policy is shown in orange in Figure 4a. As can be seen, the GPR policy is able to return a hint that represents the local trend in the data, thereby improving upon both the one-nearest neighbor (1-NN) and the Nadaraya-Watson regression (NWR) policy.

Via the hint policy π_{GPR}, we can now recommend edits in the edit distance space which are optimal according to a Gaussian Process model. However, this edit has the form of a coefficient vector $\vec{\gamma}(x)$, which is not directly interpretable to a student. So our last challenge is to derive a viable hint from our prediction in the edit distance space. More precisely, we wish to obtain an edit in our original edit set Δ which corresponds to the recommended edit in the edit distance space.

3.3. Edit pre-image problems

The problem of finding an unknown original object which maps to a known point in an embedding space is called a *pre-image problem* (Bakır et al., 2003), so the problem of finding the

edit which best corresponds to a recommended edit in the edit distance space can be described as an *edit pre-image problem*. We want to emphasize here that such pre-image problems are typically hard to solve (Bakır et al., 2003), and, to our knowledge, no approach exists to date which addresses edit pre-image problems. We also note a connection to the state reification (Rivers and Koedinger, 2014) because the mapping to the edit distance space $\vec{\phi}$ can be seen as a canonicalization, and we now try to align the edit returned by a hint policy in the edit distance space with the student's original state. In this section, we provide an approximate solution to the edit pre-image problem.

First, following Bakir et al., we re-frame our edit pre-image problem as a minimization problem: Starting from the student's current state x, we try to find an edit δ which brings us as close as possible to the recommended state of the Gaussian Process regression (GPR) hint policy in the edit distance space.

$$\min_{\delta \in \Delta} \| \vec{\phi}\big(\delta(x)\big) - \big(\vec{\phi}(x) + \pi_{\text{GPR}}(x)\big) \|^2 \tag{11}$$

This optimization problem is infeasible because it requires us to estimate the effect of an edit δ to the student's state x after mapping this state to the edit distance space. Fortunately, our established edit distance theory lets us replace this minimization problem with a simpler form.

Theorem 4. *Let $\vec{\gamma}$ be the weights applied by GPR, that is, $\vec{\gamma} = \vec{\mathrm{k}}(x) \cdot (\boldsymbol{K} + \tilde{\sigma}^2 \cdot I^M)^{-1}$. Then the maximization problem in 11 can be re-written as:*

$$\min_{\delta \in \Delta} d_{\Delta,C}(\delta(x), x)^2 + \sum_{i=1}^{M} \alpha_i \cdot d_{\Delta,C}(\delta(x), x_i)^2 \tag{12}$$

where $\alpha_i = -\gamma_i$ if x_i is the start point of a trace, $\alpha_i = \gamma_{i-1} - \gamma_i$ if x_i is an intermediate element of a trace, and $\alpha_i = \gamma_{i-1}$ if x_i is the end point of a trace.

Proof. In a first step, we note that we can re-write:

$$\vec{\phi}(x) + \pi_{\text{GPR}}(x) = \vec{\phi}(x) + \sum_{i=1}^{M} \alpha_i \cdot \vec{\phi}(x_i) \tag{13}$$

According to Theorems 3 and 4 in the paper by Paaßen et al., we can now re-write the distance $\| \vec{\phi}\big(\delta(x)\big) - \big(\vec{\phi}(x) + \pi_{\text{GPR}}(x)\big) \|^2$ as

$$\| \vec{\phi}(\delta(x)) - \vec{\phi}(x) \|^2 + \sum_{i=1}^{M} \alpha_i \cdot \| \vec{\phi}\big(\delta(x)\big) - \vec{\phi}\big(x_i\big) \|^2 - Z \tag{14}$$

where Z is a constant that does not depend on δ (Paaßen et al., 2017). Due to Theorem 2 we know that the Euclidean distance in the edit distance space corresponds exactly to the edit distance, which concludes the proof. □

This form of the minimization problem in Equation 12 has multiple key advantages. First, it does not require us to compute the vectorial embedding for any state anymore. Instead, we can infer the optimal edit sequence solely based on the edit distance $d_{\Delta,C}(\delta(x), x)$, as well as the edit distances $d_{\Delta,C}(\delta(x), x_i)$. Second, our revised form of the problem provides a useful re-interpretation. We need to find an edit δ, such that the resulting state stays close to the original

state x, gets closer to states x_i for which α_i is positive, and gets further away from state x_i for which α_i is negative. Note that this re-interpretation is consistent with the criterion of Rivers and Koedinger (2014) that a next state should stay close to the student's current state. Finally, the re-formulation shrinks our search space, because we only have to consider edits which bring us closer to states x_i with positive coefficients α_i. We can extract such edits from the shortest edit sequences between x and states x_i with positive coefficients α_i. For all these possible edits we can evaluate the error in Equation 12 and select the edit with the lowest error.

Consider the example illustrated in Figure 2c. Recall that the coefficients α resulting from the GPR hint policy are $\alpha_{x_1} = \alpha_{x_2} \approx -0.3775$ and $\alpha_{y_1} = \alpha_{y_1} \approx +0.3775$. So we need to find an edit which brings us closer to $y_1 =$ aac and $y_2 =$ bbc but further away from $x_1 =$ a and $x_2 =$ b. The cheapest edit sequence between x and y_1 is $\mathrm{rep}_{2,\mathrm{a}}, \mathrm{ins}_{3,\mathrm{c}}$, and the cheapest edit sequence between x and y_2 is $\mathrm{rep}_{1,\mathrm{b}}, \mathrm{ins}_{3,\mathrm{c}}$. Therefore, we need to consider the edits $\mathrm{rep}_{2,\mathrm{a}}$, $\mathrm{ins}_{3,\mathrm{c}}$, and $\mathrm{rep}_{1,\mathrm{b}}$. The resulting states of these edits would be aa, abc, and bb. Amongst these options, abc minimizes our error because it is closer to both aac and bbc, further away from both a and b, and stays close to ab. Therefore, we would recommend $\mathrm{ins}_{3,\mathrm{c}}$ as hint.

In practical examples, this approach would be limited by the number of edits to be considered. For many training data points with positive coefficients and long shortest edit sequences, this number can become unreasonable. One way to limit the number of edits is to incorporate more of the criteria suggested by Rivers and Koedinger (2014) and consider only edits which result in syntactically correct states, result in programs which fulfill at least as many test cases or get us closer to a correct solution. In addition, we propose to limit the search space to a reasonable size by using fewer coefficients to represent the recommended state. More precisely, we are looking for a coefficient vector $\tilde{\alpha}$ which is nonzero for at most m training states. Further, we propose to use only those states to represent the recommended state which are between the student's current state x and the next correct solution x^*. This is consistent with the criteria of Rivers and Koedinger (2014) that the recommended state should both be close to a correct solution and to the student's current state. Thus, we look for a coefficient vector $\tilde{\alpha}$, such that $\tilde{\alpha}_i$ is nonzero only if $d(x_i, x) \leq d(x, x^*)$ and $d(x_i, x^*) \leq d(x, x^*)$, such that at most m entries are nonzero, such that the sum over all entries of $\tilde{\alpha}$ is 1, and such that the state represented by $\tilde{\alpha}$ is as close as possible to the state represented by $\vec{\alpha}$. While this is an NP-hard problem, multiple simple heuristics exist which have been summarized by Hofmann et al. (2014). In our experiments, we apply both kernelized orthogonal matching pursuit and an approximation via the largest entries of $\vec{\alpha}$ and use whatever approximation is closer to the actual recommended state.

Consider the example illustrated in Figure 5. Here, the original coefficients α returned by the GPR hint policy are shown in orange and represent the state shown as an orange diamond. Now, assume that the student's current state is the string "ab" and the closest correct solution is the string "abcd". In that case, only the strings "ab," "aac," "bbc," and "abcd" fulfill the constraints $d(x_i, x) \leq d(x, x^*)$ and $d(x_i, x^*) \leq d(x, x^*)$ (indicated by dashed purple lines). If we now try to represent the recommended state by using only 3 of those four strings, this results in a representation via the strings "aac," "bbc," and "abcd" with roughly equal coefficients, resulting in a represented state (shown in purple) close to the original hint. The selected hint, in this case, would still be $\mathrm{ins}_{3,\mathrm{c}}$.

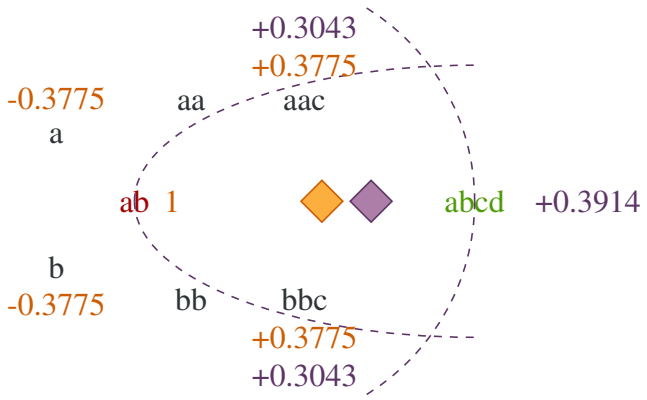


Figure 5: An illustration of the sparse representation of the recommended state for the string example of Figure 2. The student's current state is the string $x = \text{ab}$ (shown in red), the closest correct solution is the string $x^* = \text{abcd}$ (shown in green). The coefficients α_i and the represented state $\vec{\phi}(x) + \pi_{\text{GPR}}(x)$ returned by the Gaussian Process Regression (GPR) policy are shown in orange. The sparse coefficients and the corresponding represented state are drawn in purple. The constraints $d(x_i, x) \leq d(x, x^*)$ and $d(x_i, x^*) \leq d(x, x^*)$ are illustrated by dashed purple lines.

3.4. Summary

To conclude our description of the Continuous Hint Factory (CHF) we provide a short summary of all steps involved in the CHF hint policy. First, we need to perform the following preparation steps:

1. Collect trace data from successful students.
2. Remove all intermediate states in the traces which do not get closer to the goal.
3. Compute the canonic forms of the trace data and their pairwise edit distances.
4. Perform eigenvalue correction on the pairwise edit distances.
5. Compute the pairwise radial basis kernel values $\boldsymbol{K}$. The length scale parameter ψ, as well as the noise parameter $\tilde{\sigma}$, can be selected such that the predictive accuracy of the GPR model on unseen evaluation data is as high as possible.

Now, assume that a new student is in state x and requests help. In that case, the following steps need to be performed.

1. Compute the canonic form of x and the edit distance of this canonic form to all canonic forms in the trace data before.
2. Extend the eigenvalue correction to the new distances.
3. Compute the radial basis kernel values $\vec{\text{k}}(x)$ based on these corrected distances.
4. Compute the coefficients α of the GPR hint policy via the formulas in Theorem 4.

5. Optionally, sparsify these coefficients via one of the techniques of Hofmann et al. (2014).
6. Compute the cheapest edit scripts between x and all training states x_i for which α_i is positive.
7. Subselect edits δ from these edit scripts which result in states $\delta(x)$ that conform to further criteria, e.g., unit test fulfillment, or syntactic correctness (Rivers and Koedinger, 2014).
8. Compute the error term in Equation 12 for all remaining edits.
9. Select the edit with the lowest error as hint.

This concludes our description of the CHF. In the next section, we evaluate the CHF approach experimentally.

4. EXPERIMENTS

We consider two datasets for our analysis. First, a dataset collected in an introductory undergraduate computing course for non-computer science majors during the Fall of 2015 at a research university in the south-eastern United States. The course had approximately 80 students, split among six lab sections. The first half of the course focused on learning the Snap[5] programming language through a curriculum based on the *Beauty and Joy of Computing* (Garcia et al., 2015). Here, we focus on the "Guessing Game" task, which had the following description: "The computer chooses a random number between 1 and 10 and continuously asks the user to guess the number until they guess correctly." Students did not receive specific instructions regarding the form of the program. An example solution for the task is presented in Figure 1a. Students worked on this assignment during class for approximately one hour, with a teaching assistant available to assist them and the option of working in pairs. The class was conducted as normal, and the students were not informed that data was being collected. The state of the student's program was recorded after every edit. Students who did not correctly select the assignment they were working on were excluded from the analysis. The dataset consists of 52 traces with 8669 states overall.

Each of the final states was graded by two independent graders. The graders used a rubric consisting of nine assignment objectives and marked whether each state successfully or unsuccessfully completed each objective. The graders had an initial agreement of 94.5%, with Cohen's $\kappa = 0.544$. After clarifying objective criteria, each grader independently regraded each state where there was disagreement, reaching an agreement of 98.1%, with Cohen's $\kappa = 0.856$. Any remaining disagreements were discussed to create final grades for each assignment. As our aim is to predict what *capable* students would do, we kept only traces which successfully completed at least eight of the nine objectives. This left 47 traces with 7864 states.

As a second dataset, we utilize data collected in an introductory programming course for computer scientists at a German university in 2012. The students were asked to draw a UML activity diagram which described the process of adding two binary numbers. An example solution is shown in Figure 6. From the available student data, we extracted six typical strategies and created two correct traces and one erroneous trace for each strategy. Overall, the correct traces contained 364 states and the erroneous traces 115 states. We presented each state in the

[5] http://snap.berkeley.edu

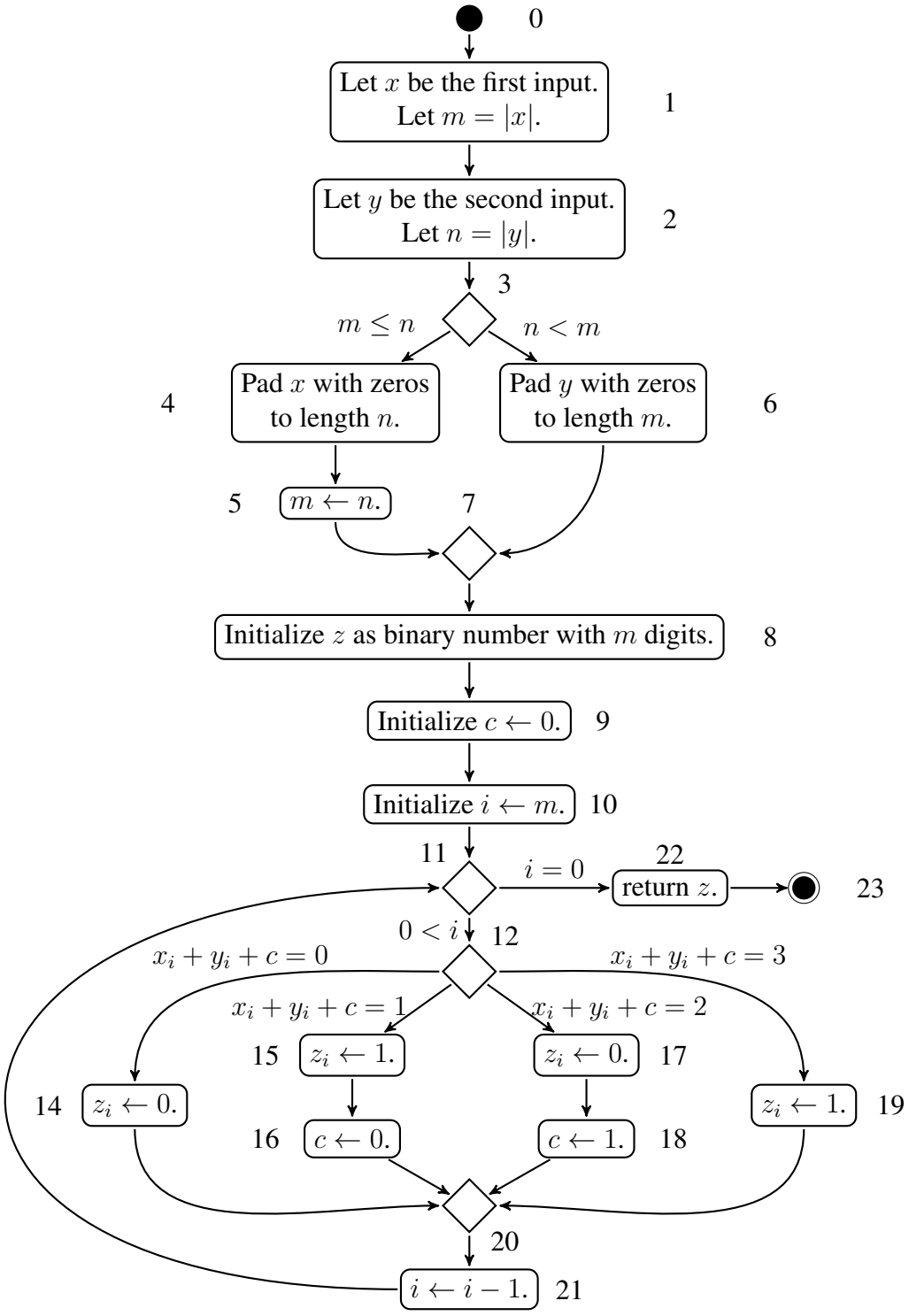


Figure 6: A correct example solution for the UML binary adder task. Numbers indicate the order in which nodes have been added to the UML diagram.

erroneous traces to three graders who independently were asked to suggest all possible edit hints that could be given to a student in the particular situation, taking past states into account. We also instructed the tutors to provide an estimate of hint quality in the interval $[0, 1]$ for each of their hints, taking into account the following criteria: 1) Does the hint follow the strategy of the student? 2) Does the hint conform to the student's current focus of attention or does it address a different part of the state? 3) Is the hint effective in addressing the problems in the student's state? 4) Is the hint effective in guiding the student toward a solution? In a second meeting, all tutors met to add ratings for the hints of the respective other tutors and to discuss discrepancies in the ratings. If after discussion at least one expert rated a hint with a grade below 0.5, the hint was excluded from the set. 1053 hints remained after this process. The average inter-rater correlation via Pearson's r was $r = 0.588$, indicating moderate agreement. This dataset is available online under the DOI 10.4119/unibi/2913083.

We represent the states of both datasets as trees. In the UML dataset, we removed back-references, such as the arrow from node 20 to node 11 in Figure 6, to obtain a tree structure. We labeled all tree nodes with the name of the respective syntactic construct (e.g., "doIf," "var," and "literal"). In case of the UML dataset, we also added the text of the respective node to the label, for example, "return z" for node 21 in Figure 6. In both datasets, we canonicalized the trees by normalizing variable names and literals, normalizing the order of binary relations, and removing non-executable code, as recommended by Rivers and Koedinger (2012).

As an edit distance, we employ the tree edit distance of Zhang and Shasha (1989) as implemented in the *TCS Alignment Toolbox* (Paaßen et al., 2015). For the Snap dataset we use a uniform edit cost of 1 for deletions, insertions, and replacements. For the UML dataset, we define deletion and insertion costs as 1, replacement costs between unequal node types as infinite, and replacement costs between action nodes (displayed as ellipses in Figure 6) as the string edit distance between the node text, normalized to the interval $[0, 1]$. We post-process the tree edit distances via clip eigenvalue correction (Gisbrecht and Schleif, 2015). Based on the tree edit distance, we excluded states which did not get closer to the final state in the respective trace, which left 1005 states. Of these states, 812 were unique, and of these unique states, 94.09% were visited only once. Similarly, 215 of the 354 training states in the UML dataset were unique, and of these unique states, 82.79% were visited only once. These numbers indicate that meaningful frequency information is only available for very few training states, which is consistent with the findings reported by Price and Barnes (2015) on similar data from an open-ended Snap programming task.

To evaluate the utility of the CHF fairly, we need to compare to existing reference hint policies. Due to the lack of meaningful frequency information in our data, however, we can neither apply the Hint Factory (Barnes and Stamper, 2008) nor the Piech policy (Piech et al., 2015). Furthermore, to keep the approach generic, we do not use task-specific syntactic or unit test information for our experiments, which rules out the policy of Lazar and Bratko (2014). There remain the policy of Gross and Pinkwart (2015), which uses the successor of the next state in the trace data to construct a hint, the policy of Zimmerman and Rupakheti (2015), which uses the closest correct solution to construct a hint, and the policy of Rivers and Koedinger (2015), which also uses the closest correct solution. The Zimmerman policy and the Rivers policy mainly differ in *how* hints are constructed from the closest correct solution. However, given that we use neither frequency nor syntactic or semantic correctness information, and consider only single edits instead of edit combinations, both policies become very similar, such that we only consider the Gross policy and the Zimmerman policy in this case.

Table 1: Mean RMSE $\pm$ standard deviation in predicting the next step and the final step of capable students for both the Snap dataset, as well as the UML dataset. The first column lists the different prediction schemes. Lower values are better, and a value of 0 is ideal.

	Snap		UML	
Prediction scheme	Next	Final	Next	Final
Do nothing	17.5 ± 3.89	39.3 ± 9.36	5.27 ± 0.53	28.9 ± 5.28
Successor-of-closest	23.7 ± 5.39	39.1 ± 9.49	7.89 ± 3.50	29.1 ± 6.00
Closest-correct	26.7 ± 5.46	43.0 ± 8.60	25.50 ± 1.23	19.9 ± 8.42
Gaussian Process	16.6 ± 4.09	37.8 ± 9.15	3.18 ± 1.66	27.8 ± 5.32

We implemented all hint policies in MATLAB® [6]. To optimize the kernel length scale ψ and the noise standard deviation $\tilde{\sigma}$ of the Gaussian Process model, we employ a random hyper-parameter search with 10 repeats as recommended by Bergstra and Bengio (2012). We set the maximum number of training states to represent the hint of the CHF policy to $m = 11$.

In our experiments, we investigate two research questions, which we will cover in turn. We evaluate statistical significance using a paired Wilcoxon sign-rank test. Further, we apply a Bonferroni correction to avoid type I errors due to multiple tests.

RQ1: How well does the Gaussian Process model capture the behavior of capable students, that is, can the Gaussian Process predict what a capable student would do?

To investigate RQ1, we consider two measures of predictive accuracy. First, we measure the distance between the predicted next state of the Gaussian Process model and the actual next state of the respective student (next-step error). Second, we measure the distance between the predicted next state and the *final* state of the respective student (final-step error). We square these distances, average them over a trace, and then compute the root, resulting in a root mean square error (RMSE), which estimates the standard deviation of the error distribution of the model and is a well-established measure for model evaluation (Chai and Draxler, 2014). We evaluate the next-step error and the final-step error in a leave-one-out crossvalidation over the traces, which means that in each fold we use all but one trace as training data for the prediction and the remaining trace to evaluate the model.

Note that RQ1 is only concerned with the prediction module of each hint policy, that is, the reference state based on which edits are generated, not the edits which are used as hints. As such, we do not directly compare with the Gross or Zimmerman policy but with the reference states they would use, namely the successor of the closest next solution (Successor-of-closest), and the closest correct solution (Closest-correct) respectively. Given the nature of these references, we would expect that the Successor-of-closest prediction would perform well in the next-step error but badly in the final-step error and that the Closest-correct prediction would perform badly in terms of the next-step error but good in terms of the final-step error.

Table 1 shows the RMSE averaged over the crossvalidation folds ($\pm$ standard deviation) for

[6] Our implementation of the GPR hint policy is available under the DOI 10.4119/unibi/2913104

both datasets where each column lists one error measure for all prediction schemes[7]. As an additional reference, we provide the error for the trivial prediction of staying in the same state, that is, $\pi(x) = x$. Statistical analysis reveals that the Gaussian Process is significantly better in predicting the next state compared to all other baselines for both datasets ($p < .01$). Further, the Gaussian Process is significantly better in predicting the final state compared to the "Do nothing" and the Successor-of-closest prediction for both datasets ($p < .01$), and better than the Closest-correct prediction for the Snap dataset ($p < .001$). Interestingly, the Successor-of-closest prediction does not perform better in predicting the actual next state of a student compared to staying in the same state, indicating that students in both data sets do not necessarily move along the same states, even though their directions may be consistent, which is consistent with the embedding in Figure 1b. Furthermore, we note that, counter to our expectations, the Closest-correct prediction has a higher final-step error on the Snap dataset than any other prediction scheme, which indicates that, on average, the closest correct solution of other students is far away from the student's actual final solution in this dataset. This effect is likely explained by the high strategic variability in an open-ended programming task such as the guessing game task. For such tasks, we expect that the averaging approach of the Gaussian Process to be particularly helpful, because the general trends in the datasets may be more akin to the student's actual plans than a single closest correct solution. Conversely, the UML dataset features less strategic variability, and the closest correct solution of another student is still close to the final state of the student for which the prediction is made, which is reflected in significantly better predictions of the Closest-correct prediction compared to all other prediction schemes ($p < 10^{-3}$). Overall, we can conclude that the Gaussian Process is more accurate in predicting the next state of students compared to other baselines on our example datasets and that this is especially the case for the Snap dataset, which is characterized by high strategic variability.

RQ2: Do the hints of the Continuous Hint Factory correspond to the hints of human tutors?

To investigate RQ2, we require a reference measure of hint quality, which is provided by the quality judgments of human tutors in the UML dataset. In particular, we iterate over every state in the erroneous traces of the UML dataset and generate a hint with each hint policy, using all correct traces as training data. If multiple edits achieve the lowest error rank, we resolve ties by selecting the edit as hint which is closest to the root of the tree. If the recommended hint of the policy matches at least one tutor hint, we assign the average quality rating of the human tutors for that hint. Otherwise we set the rating to 0. This is similar to the evaluation scheme suggested by Price et al. (2017a). We report five evaluation measures, namely the median and mean hint quality, the fraction of hints with a quality > 0, the distance between the policy hint and the closest human tutor hint in terms of RMSE, and the fraction of states for which a hint could be generated. In addition to the Gross and the Zimmerman policy, we also compare to a random policy, which selects a random reference state from the training state and recommends an edit on the shortest path towards that state as hint. Finally, we also provide the best-rated tutor hint as the gold standard.

The experimental results are shown in Table 2, where each column displays one evaluation measure, and each row lists the results for one hint policy. Regarding hint quality, we observe

[7]Note that the RMSE cannot be interpreted directly as the average number of edits between the predicted next state and the gold standard because the RMSE assigns higher weight to larger deviations due to the square (Chai and Draxler, 2014). Further, in this particular evaluation, but not for RQ2, Eigenvalue correction distorts the edit distances to become larger.

Table 2: The hint evaluation measures for all hint policies on the UML dataset. Mean hint quality and mean ambiguity are reported with standard deviation. For all measures except the RMSE, higher numbers are better with a value of 1 and 100% respectively being ideal.

	Hint quality			RMSE	Hintable
Hint policy	Median	Mean	> 0		
Random	0.0	0.360 ± 0.456	39.1%	1.42	83.5%
Tutor	1.0	0.994 ± 0.021	100.0%	0.00	100.0%
Gross	0.8	0.569 ± 0.465	60.9%	1.42	100.0%
Zimmerman	0.8	0.557 ± 0.431	64.3%	1.48	100.0%
CHF	0.9	0.590 ± 0.471	61.7%	1.36	97.4%

that the CHF performs significantly better compared to a random policy ($p < .01$), and significantly worse compared to human tutor hints ($p < .001$), but otherwise there are no significant differences between the hint policies. This indicates that for simple datasets like the UML dataset, which feature low strategic variability, single reference states are sufficient to generate viable hints. Interestingly, though, we could also observe cases where this was not the case. In particular, Figure 7 displays a UML diagram where the Zimmerman policy recommends appending a decision node close to the root (purple), which is outside the student's current focus of attention because the last node the student added was the "return z" node at the bottom of the diagram. Accordingly, the CHF recommends appending a "finish" node to that branch (orange).

Another interesting finding is that the CHF and the Gross policy consistently achieved perfect hint quality for the first three steps in each trace. This is important in light of the research of Price et al. (2017b), which indicates that students are more likely to seek help and follow hints if *early* hints provided by the system were useful.

5. Conclusion

This work makes three primary contributions. First, we have provided a mathematical framework for edit-based hints and placed prior contributions within this framework. Second, we have introduced the concept of the edit distance space, which is a continuous embedding of student states such that the edit distance corresponds to the Euclidean distance in the embedding space. Finally, we introduced the *Continuous Hint Factory* (CHF), a novel hint policy which provides edit hints to students by choosing an edit consistent with the general trend of capable students in similar states.

In our experiments, we have shown that the CHF model is able to predict what capable students would do better than other predictive schemes, especially on an open-ended programming dataset with high strategic variability. We also showed that the CHF reproduces human tutor hints about as well as existing hint policies on a simple UML diagram task. These results indicate that the averaging approach of the CHF is beneficial for prediction, but that this advantage is not necessarily reflected in higher hint quality, at least for a simple learning task with low

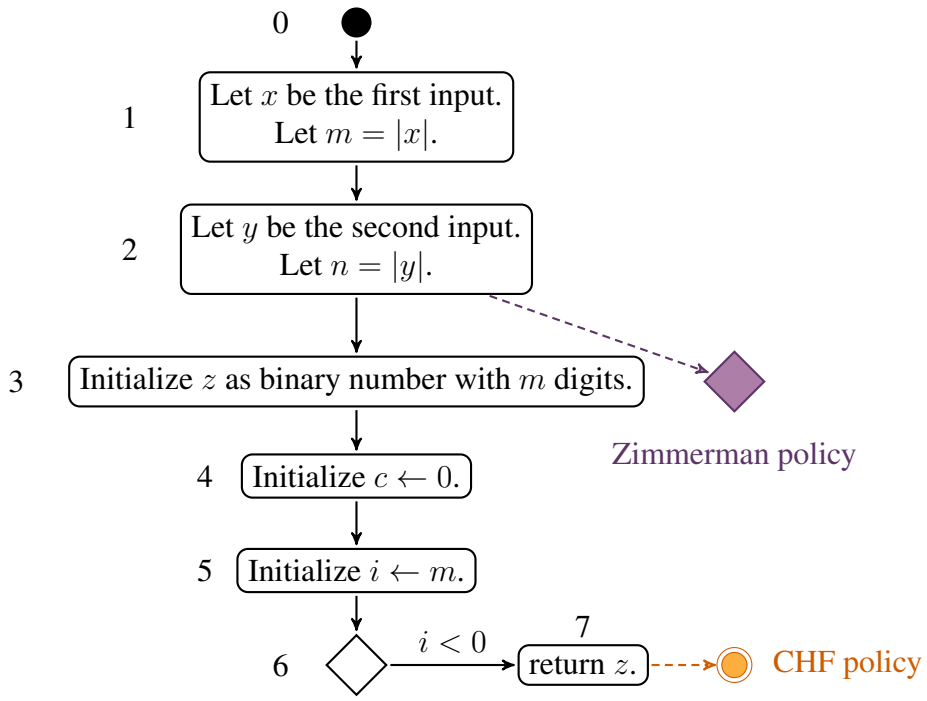


Figure 7: An example state from the UML dataset, where the Zimmerman policy generates a hint (purple) which is not in the student's current focus of attention. In contrast, the Continuous Hint Factory generates a hint (orange), which adds upon the last added node.

strategic variability.

We note that the Continuous Hint Factory still has several limitations. In particular, the CHF can only be applied if an edit distance is available which is efficient, takes syntax and semantics into account appropriately, and yields edits that are viable as next-step-hints for the student. This is not an issue for the domain of computer programming, as edit distances appear as a natural fit for syntax-tree-based representations of programs but may be an issue for other domains. Further, as any data-driven hint approach, hint quality will suffer if the strategy of a new student is substantially different from anything that the system has seen before. With regards to evaluation, our assessment of hint quality is not definitive, and it appears likely that our proposed approach only yields significant advantages compared to existing work on more complicated tasks compared to the ones we investigated. Further, we do not yet know how a difference in hint quality translates to learning outcomes in students. After all, better hints from the view of a tutor may not always yield better learning outcomes, due to difficulties in sense-making or lack of prior knowledge on the student's side (Aleven et al., 2016). Finally, we acknowledge that our evaluation is rather narrow, including only two learning tasks from different domains.

With regards to future work, it appears promising to integrate the CHF even better with prior work presented in the literature. In particular, we could take syntactic and unit test information into account (Rivers and Koedinger, 2014), combine multiple edits instead of single edits (Rivers and Koedinger, 2015), or apply more sophisticated edit distances as suggested by Mokbel et al. (2013), Paaßen et al. (2016), as well as Price et al. (2017a). Finally, it would be interesting to evaluate the CHF on more learning tasks, especially more open-ended learning tasks where the advantages of the CHF are more likely to be visible.

6. Acknowledgements

This research was funded by the German Research Foundation (DFG) as part of the project "Learning Dynamic Feedback for Intelligent Tutoring Systems" under the grant number HA 2719/6-2 as well as the Cluster of Excellence Cognitive Interaction Technology 'CITEC' (EXC 277), Bielefeld University and the NSF under grant number #1432156 "Educational Data Mining for Individualized Instruction in STEM Learning Environments" with Min Chi & Tiffany Barnes as Co-PIs. We also wish to express our gratitude to our anonymous reviewers and our editors who helped to increase the quality of our contribution substantially.

References

Aleven, V., McLaren, B. M., Sewall, J., and Koedinger, K. R. 2006. The cognitive tutor authoring tools (CTAT): Preliminary evaluation of efficiency gains. In *Proceedings of the 8th International Conference on Intelligent Tutoring Systems (ITS 2006)*, M. Ikeda, K. D. Ashley, and T.-W. Chan, Eds. Springer Berlin Heidelberg, Jhongli, Taiwan, 61–70.

Aleven, V., Roll, I., McLaren, B. M., and Koedinger, K. R. 2016. Help helps, but only so much: Research on help seeking with intelligent tutoring systems. *International Journal of Artificial Intelligence in Education 26,* 1, 205–223.

Augsten, N., Böhlen, M., and Gamper, J. 2008. The pq-gram distance between ordered labeled trees. *ACM Transactions on Database Systems 35,* 1, 4:1–4:36.

BAKIR, G. H., WESTON, J., AND SCHÖLKOPF, B. 2003. Learning to find pre-images. In *Proceedings of the 16th International Conference on Neural Information Processing Systems (NIPS 2003)*, S. Thrun, L. K. Saul, and P. B. Schölkopf, Eds. MIT Press, 449–456.

BARNES, T., MOSTAFAVI, B., AND EAGLE, M. J. 2016. Data-driven domain models for problem solving. In *Domain Modeling*, R. A. Sottilare, A. C. Graesser, X. Hu, A. M. Olney, B. D. Nye, and A. M. Sinatra, Eds. Design Recommendations for Intelligent Tutoring Systems, vol. 4. US Army Research Laboratory, 137–145.

BARNES, T. AND STAMPER, J. 2008. Toward automatic hint generation for logic proof tutoring using historical student data. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems (ITS 2008)*, B. P. Woolf, E. Aïmeur, R. Nkambou, and S. Lajoie, Eds. Montreal, Canada, 373–382.

BERGSTRA, J. AND BENGIO, Y. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13*, 281–305.

CHAI, T. AND DRAXLER, R. R. 2014. Root mean square error (RMSE) or mean absolute error (MAE)? - arguments against avoiding RMSE in the literature. *Geoscientific Model Development 7,* 3, 1247–1250.

CHOUDHURY, R. R., YIN, H., AND FOX, A. 2016. Scale-driven automatic hint generation for coding style. In *Proceedings of the 13th International Conference on Intelligent Tutoring Systems (ITS 2016)*, A. Micarelli, J. Stamper, and K. Panourgia, Eds. Zagreb, Croatia, 122–132.

EAGLE, M. AND BARNES, T. 2013. Evaluation of automatically generated hint feedback. In *Proceedings of the 6th International Conference on Educational Data Mining (EDM 2013)*, S. K. D'Mello, R. A. Calvo, and A. Olney, Eds. Memphis, Tennessee, USA, 372–374.

EAGLE, M., JOHNSON, M., AND BARNES, T. 2012. Interaction networks: Generating high level hints based on network community clustering. In *Proceedings of the 5th International Conference on Educational Data Mining (EDM 2012)*, K. Yacef, O. Zaïane, H. Hershkovitz, M. Yudelson, and J. Stamper, Eds. Chania, Greece, 164–167.

FLEMING, M. L. AND LEVIE, W. H. 1993. *Instructional Message Design: Principles from the Behavioral and Cognitive Sciences*. Educational Technology Publications, Englewood Cliffs, NJ, USA.

FREEMAN, P., WATSON, I., AND DENNY, P. 2016. Inferring student coding goals using abstract syntax trees. In *Proceedings of the 24th International Conference on Case-Based Reasoning Research and Development (ICCBR 2016)*, A. Goel, M. B. Díaz-Agudo, and T. Roth-Berghofer, Eds. Atlanta, GA, USA, 139–153.

GARCIA, D., HARVEY, B., AND BARNES, T. 2015. The Beauty and Joy of Computing. *ACM Inroads 6,* 4, 71–79.

GIEGERICH, R., MEYER, C., AND STEFFEN, P. 2004. A discipline of dynamic programming over sequence data. *Science of Computer Programming 51,* 3, 215 – 263.

GISBRECHT, A. AND SCHLEIF, F.-M. 2015. Metric and non-metric proximity transformations at linear costs. *Neurocomputing 167*, 643–657.

GROSS, S., MOKBEL, B., HAMMER, B., AND PINKWART, N. 2014. Example-based feedback provision using structured solution spaces. *International Journal on Learning Technologies 9,* 3, 248–280.

GROSS, S. AND PINKWART, N. 2015. How do learners behave in help-seeking when given a choice? In *Proceedings of the 17th International Conference on Artificial Intelligence in Education (AIED 2015)*, C. Conati, N. Heffernan, A. Mitrovic, and M. F. Verdejo, Eds. Madrid, Spain, 600–603.

HAMMER, B. AND HASENFUSS, A. 2010. Topographic mapping of large dissimilarity data sets. *Neural Computation 22,* 9, 2229–2284.

HEAD, A., GLASSMAN, E., SOARES, G., SUZUKI, R., FIGUEREDO, L., D'ANTONI, L., AND HARTMANN, B. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth ACM Conference on Learning@Scale (L@S 2017)*. ACM, Cambridge, MA, USA, 89–98.

HICKS, A., PEDDYCORD, B., AND BARNES, T. 2014. Building games to learn from their players: Generating hints in a serious game. In *Proceedings of the 12th International Conference Intelligent Tutoring Systems (ITS 2014)*, S. Trausan-Matu, K. E. Boyer, M. Crosby, and K. Panourgia, Eds. Honolulu, HI, USA, 312–317.

HOFMANN, D., SCHLEIF, F.-M., PAASSEN, B., AND HAMMER, B. 2014. Learning interpretable kernelized prototype-based models. *Neurocomputing 141*, 84–96.

KOEDINGER, K. R., BRUNSKILL, E., BAKER, R. S., MCLAUGHLIN, E. A., AND STAMPER, J. 2013. New potentials for data-driven intelligent tutoring system development and optimization. *AI Magazine 34,* 3, 27–41.

LAZAR, T. AND BRATKO, I. 2014. Data-driven program synthesis for hint generation in programming tutors. In *Proceedings of the 12th International Conference on Intelligent Tutoring Systems (ITS 2014)*, S. Trausan-Matu, K. E. Boyer, M. Crosby, and K. Panourgia, Eds. Honolulu, HI, USA, 306–311.

LE, N.-T. 2016. A classification of adaptive feedback in educational systems for programming. *Systems 4,* 2, 22.

LE, N.-T. AND PINKWART, N. 2014. Towards a classification for programming exercises. In *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science (AIEDCS)*, K. E. Boyer, N.-T. Le, S. I.-H. Hsiao, S. Sosnovsky, B. Di Eugenio, and B. Chaudry, Eds. Honolulu, Hawaii, 51–60.

LEVENSHTEIN, V. I. 1965. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10,* 8, 707–710.

LYNCH, C., ASHLEY, K. D., PINKWART, N., AND ALEVEN, V. 2009. Concepts, structures, and goals: Redefining ill-definedness. *International Journal of Artificial Intelligence in Education 19,* 3, 253–266.

MARIN, V. J., PEREIRA, T., SRIDHARAN, S., AND RIVERO, C. R. 2017. Automated personalized feedback in introductory Java programming MOOCs. In *33rd International IEEE Conference on Data Engineering (ICDE 2017)*. San Diego, CA, USA, 1259–1270.

MOKBEL, B., GROSS, S., PAASSEN, B., PINKWART, N., AND HAMMER, B. 2013. Domain-Independent Proximity Measures in Intelligent Tutoring Systems. In *Proceedings of the 6th International Conference on Educational Data Mining (EDM 2013)*, S. K. D'Mello, R. A. Calvo, and A. Olney, Eds. Memphis, Tennessee, USA.

MURRAY, T., BLESSING, S., AND AINSWORTH, S. 2003. *Authoring tools for advanced technology learning environments: Toward cost-effective adaptive, interactive and intelligent educational software*. Springer, Berlin/Heidelberg.

NGUYEN, A., PIECH, C., HUANG, J., AND GUIBAS, L. 2014. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web (WWW 2014)*. Seoul, Korea, 491–502.

PAASSEN, B., GÖPFERT, C., AND HAMMER, B. 2017. Time Series Prediction for Graphs in Kernel and Dissimilarity Spaces. *Neural Processing Letters*. epub ahead of print, https://arxiv.org/abs/1704.06498.

PAASSEN, B., JENSEN, J., AND HAMMER, B. 2016. Execution Traces as a Powerful Data Representation for Intelligent Tutoring Systems for Programming. In *Proceedings of the 9th International Conference*

on Educational Data Mining (EDM 2016), T. Barnes, M. Chi, and M. Feng, Eds. Raleigh, North Carolina, USA, 183–190.

PAASSEN, B., MOKBEL, B., AND HAMMER, B. 2015. A toolbox for adaptive sequence dissimilarity measures for intelligent tutoring systems. In *Proceedings of the 8th International Conference on Educational Data Mining (EDM 2015)*, O. C. Santos, J. G. Boticario, C. Romero, M. Pechenizkiy, A. Merceron, P. Mitros, J. M. Luna, C. Mihaescu, P. Moreno, A. Hershkovitz, S. Ventura, and M. Desmarais, Eds. International Educational Datamining Society, 632–632.

PAASSEN, B., MOKBEL, B., AND HAMMER, B. 2016. Adaptive structure metrics for automated feedback provision in intelligent tutoring systems. *Neurocomputing 192*, 3–13.

PANE, J. F., GRIFFIN, B. A., MCCAFFREY, D. F., AND KARAM, R. 2014. Effectiveness of Cognitive Tutor Algebra I at scale. *Educational Evaluation and Policy Analysis 36,* 2, 127–144.

PEKALSKA, E. AND DUIN, R. P. W. 2005. *The Dissimilarity Representation for Pattern Recognition: Foundations And Applications (Machine Perception and Artificial Intelligence)*. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

PIECH, C., HUANG, J., NGUYEN, A., PHULSUKSOMBATI, M., SAHAMI, M., AND GUIBAS, L. 2015. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*, F. Bach and D. Blei, Eds. International Conference on Machine Learning. Lille, France, 1093–1102.

PIECH, C., SAHAMI, M., HUANG, J., AND GUIBAS, L. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second ACM Conference on Learning @ Scale (L@S 2015)*, G. Kiczales, D. M. Russel, and B. Woolf, Eds. Vancouver, BC, Canada, 195–204.

PRICE, T. W. AND BARNES, T. 2015. An exploration of data-driven hint generation in an open-ended programming problem. In *Workshops Proceedings of the 8th International Conference on Educational Data Mining (EDM 2015)*, O. C. Santos, J. G. Boticario, C. Romero, M. Pechenizkiy, A. Merceron, P. Mitros, J. M. Luna, C. Mihaescu, P. Moreno, A. Hershkovitz, S. Ventura, and M. Desmarais, Eds. Madrid, Spain.

PRICE, T. W., DONG, Y., AND BARNES, T. 2016. Generating data-driven hints for open-ended programming. In *Proceedings of the 9th International Conference on Educational Data Mining (EDM 2016)*, T. Barnes, M. Chi, and M. Feng, Eds. Raleigh, NC, USA.

PRICE, T. W., DONG, Y., AND LIPOVAC, D. 2017. iSnap: Towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM Technical Symposium on Computer Science Education (SIGCSE)*. Seattle, Washington, USA, 483–488.

PRICE, T. W., ZHI, R., AND BARNES, T. 2017a. Evaluation of a data-driven feedback algorithm for open-ended programming. In *Proceedings of the 10th International Conference on Educational Datamining (EDM 2017)*, X. Hu, T. Barnes, A. Hershkovitz, and L. Paquette, Eds. Wuhan, China, 192–197.

PRICE, T. W., ZHI, R., AND BARNES, T. 2017b. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *Proceedings of the 18th International Conference on Artificial Intelligence in Education (AIED 2017)*, E. André, R. Baker, X. Hu, M. M. T. Rodrigo, and B. du Boulay, Eds. Springer, Wuhan, China, 311–322.

RASMUSSEN, C. E. AND WILLIAMS, C. K. I. 2005. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

RIVERS, K. AND KOEDINGER, K. R. 2012. A canonicalizing model for building programming tutors. In *Proceedings of the 11th International Conference on Intelligent Tutoring Systems, (ITS 2012)*, S. A. Cerri, W. J. Clancey, G. Papadourakis, and K. Panourgia, Eds. Chania, Greece, 591–593.

RIVERS, K. AND KOEDINGER, K. R. 2014. Automating hint generation with solution space path construction. In *Proceedings of the 12th International Conference on Intelligent Tutoring Systems (ITS 2014)*, S. Trausan-Matu, K. E. Boyer, M. Crosby, and K. Panourgia, Eds. Honolulu, HI, USA, 329–339.

RIVERS, K. AND KOEDINGER, K. R. 2015. Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. *International Journal of Artificial Intelligence in Education 27,* 1, 37–64.

SAMMON, J. W. 1969. A nonlinear mapping for data structure analysis. *IEEE Transactions on Computers 18,* 5, 401–409.

SHIH, B., KOEDINGER, K. R., AND SCHEINES, R. 2008. A response time model for bottom-out hints as worked examples. In *Proceedings of the 1st International Conference on Educational Datamining (EDM 2008)*, C. Romero, S. Ventura, M. Pechenizkiy, and R. Baker, Eds. Montreal, Quebec, Canada, 117–126.

STAMPER, J. C., BARNES, T., CROY, M., AND EAGLE, M. 2012. Experimental evaluation of automatic hint generation for a logic tutor. *International Journal of Artificial Intelligence in Education 22,* 1, 3–18.

VAN LEHN, K. 2006. The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education 16,* 3, 227–265.

YIN, H., MOGHADAM, J., AND FOX, A. 2015. Clustering student programming assignments to multiply instructor leverage. In *Proceedings of the Second ACM Conference on Learning @ Scale (L@S 2015)*, G. Kiczales, D. M. Russel, and B. Woolf, Eds. ACM, Vancouver, BC, Canada, 367–372.

ZHANG, K. AND SHASHA, D. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing 18,* 6, 1245–1262.

ZHANG, K., STATMAN, R., AND SHASHA, D. 1992. On the editing distance between unordered labeled trees. *Information Processing Letters 42,* 3, 133 – 139.

ZIMMERMAN, K. AND RUPAKHETI, C. R. 2015. An automated framework for recommending program elements to novices (N). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, M. Cohen, L. Grunske, and M. Whalen, Eds. Lincoln, NE, USA, 283–288.