Building Educational Technology Quickly and Robustly with an Interactively Teachable AI

Daniel Phillip Weitekamp

CMU-HCII-24-111
September 2024

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This dissertation research was supported in part by the training grant (R305B150008) from the Institute of Education Sciences. Additional thanks to Google for providing a 2018 Google Faculty Research Award (for Ken Koedinger) in support of this work. Opinions expressed in this dissertation are the author's alone and do not represent the views of the U.S. Department of Education or Google.

Keywords: Interactive Task Learning, Machine Teaching, Interactive Machine Learning, Programming by Demonstration, Intelligent Tutoring Systems, Self-Aware Learning, ITS Authoring Tools

Abstract

Interactive task learning (ITL) is a machine-learning paradigm that envisions AI that can learn whole programs directly from the natural instructions of untrained users. In this dissertation, I present a system called AI2T that improves upon an ITL sub-paradigm called authoring-by-tutoring, whereby highly adaptive educational technology known as intelligent tutoring systems (ITSs) are authored by teaching an agent with rapid human-like learning capabilities. In the course of about 20-30 minutes authors can tutor AI2T with demonstrations and interactive feedback instead of needing to program an ITS by hand; a process which typically requires 200-300 developer hours per hour of instruction.

Authoring-by-tutoring presents a significant opportunity to democratize the authoring of ITSs. The defining characteristic of an ITS is the automatic delivery of detailed step-by-step feedback and hints characteristic of human-to-human tutoring. ITSs are typically more effective than traditional instruction and in some cases even more effective than human tutors. Authoring-by-tutoring is a path toward building the cognitively focused, precisely engineered, and reliably accurate behaviors of traditional ITSs without needing to hand-program behaviors or rely upon costly pretrained AI systems like large language models (LLMs) that are prone to hallucinating incorrect solutions and feedback. Toward this aim, this work innovates on methods of machine learning that robustly learn complex behaviors via rapid bottom-up induction, instead of by mimicking patterns in big data.

In this dissertation, I present two novel machine-learning algorithms that enable data-efficient and robust interactive task learning, whereby correct and complete rule-based programs can be induced from interactive instruction. First I present STAND, a highly data-efficient algorithm for inducing preconditions for rules from binary reward signals. STAND out-performs algorithms like random forests and XGBoost known for their data-efficient learning on tabular data. STAND also enables a measure called *instance certainty*, an estimate of prediction probability that is more highly correlated with actual increases in holdout set performance than methods that rely on weighted ensembles. I show in simulation and with users that instance certainty can help authors estimate when AI2T has induced 100% complete programs, and show that it can provide active-learning support, helping authors identify the most helpful problems to tutor AI2T on next. Second, I introduce a method for learning hierarchical task networks (HTNs) from action sequences that helps AI2T induce simpler and more robust hierarchical programs than past systems. This approach is agnostic to action sequence lesson ordering, and induces HTNs with features like unordered groups and conditional actions that are useful for ITS rules.

The machine learning and interactions design innovations of this work improve upon the authoring-by-tutoring implemented in past systems like SimStudent and the Apprentice Learner (AL) framework. I evaluate these improvements in two user studies each with 10 users. In study 2, half of our participants succeeded at teaching AI2T 100% complete and accurate ITS behavior for two K-12 math domains when we evaluated the induced programs on a large holdout set of 100 problems. These first-time participants worked with AI2T for a median time of just 22 minutes per domain, half the time reported in our prior work.

Acknowledgements

I would like to extend my gratitude to my parents for all that they have done to support my upbringing and my several decades of education. To my mother for standing by me through difficult times, for teaching me to care for myself and others, and for encouraging my interests in the humanities. To my father for instilling in me the inquisitive and inventive disposition of a scientist, for his wit, for his sense of humor, and of justice. To my two older brothers—my role models in everything. To the peers in my PhD cohort for the time we shared, especially to Sujeath Pareddy who we miss dearly. And finally, I would like to extend my gratitude to my colleagues, my committee members, my advisor Ken for 6 great years of talking shop.

Table of Contents

Chapter 1

Introduction

This dissertation presents an authoring tool called AI2T that enables non-programmers to build adaptive educational technology by interactively teaching an AI. AI2T is an agent-based system that rapidly generates knowledge structures through bottom-up induction from the author's instruction, enabling them to quickly build educational software that would typically require hundreds of dedicated programmer hours to produce. This dissertation presents several innovations in highly data-efficient and robust machine learning that enable AI2T to be taught interactively in a web interface. In our final study, we demonstrate that non-programmers can author correct and complete programs with AI2T in around 20-30 minutes.

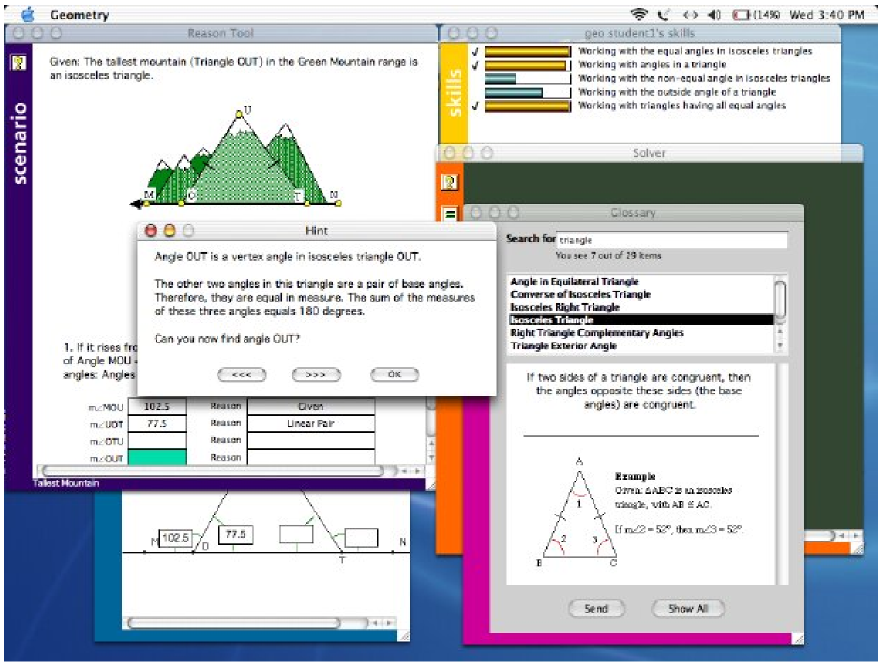
In this dissertation I consider a tutoring system induced by AI2T to be correct and complete if it exhibits the same grading behaviors and solution flexibility as a ground-truth hand-programmed tutoring system. Specifically, we look at whether tutoring systems built with AI2T permit exactly the same set of acceptable next student actions for all reachable tutor states when compared to a ground-truth reference tutoring system. We focus on tutoring system behaviors that permit many possible solution paths, meaning the set of permitted actions and reachable states typically differ between instances of the same kind of problem. Half of our users in our final study succeeded at teaching AI2T programs that are 100% correct and complete in this regard when evaluated against a large holdout set of problems. Among those that did not succeed in authoring 100% correct and complete programs, several participants came very close (within less than 12% of perfect) and two participants failed to catch mistakes that led to buggy tutoring system behavior.

The core features of AI2T span several machine learning paradigms including machine teaching [126], active learning [106], and interactive task learning (ITL) [46]. At its core AI2T builds upon the programming-by-demonstration (PBD) paradigm, where non-programmers can demonstrate the behavior of a program instead of writing code in a programming language [22]. Many demonstrations of PBD have shown robust performance in automating simple single action or sequential behaviors [27, 51, 59] that require minimal generalizations from users' demonstrations. However, the challenges of PBD multiply when applied to building larger multi-faceted applications [72, 87]. AI2T is able to very robustly induce the core behaviors of complex applications known as Intelligent Tutoring Systems (ITS)—educational technologies known for their comprehensive adaptive student support features. Authors train AI2T with a set of interactions that go beyond PBD. These interactions are better described as programming-by-tutoring [70]—the author is not only responsible for demonstrating solutions to AI2T but also interactively checking its behavior as it attempts to solve problems on its own. In this relationship, the author is the tutor, and AI2T is the tutee. Therefore AI2T is best described by Laird et. al.'s vision of interactive task learning (ITL), the idea of AI systems that can be taught complex and robust new capabilities using a variety of natural interaction methods that are intuitive to non-programmers.

1.1 Intelligent Tutoring Systems

ITSs are highly adaptive educational technology that attempt to mimic the kinds of interactive feedback that students receive in one-on-one tutoring. The development of ITSs was motivated by Bloom's famous observation that tutoring is about two standard deviations more effective than traditional instruction in terms of expected learning gains [10]. Many ITSs have been shown to match or exceed the benefits of human tutors [44, 113]. ITSs are distinguished from basic online educational content like quizzes and videos by features that adapt to students as they engage in practice [112].

Historically ITSs have implemented many overlapping paradigms of complex adaptive support. For instance, constraint-based modeling (CBM) approaches characterize correct solutions by the satisfaction of several independent requirements that when violated trigger highly specific feedback [78], conversational agents engage students in interactive dialogues [26], and approaches like example-tracing and model-tracing track students solutions step-by-step to deliver correctness feedback, hints, and other forms of support that are aligned with particular problem steps, solution strategies, and common mistakes [4].



Given: The tallest mountain (Triangle OUT) in the Green Mountain range is an isosceles triangle.

✓ Working with the equal angles in isosceles triangles
✓ Working with angles in a triangle
Working with the non-equal angle in isosceles triangles
Working with the outside angle of a triangle
✓ Working with triangles having all equal angles

Angle OUT is a vertex angle in isosceles triangle OUT.

The other two angles in this triangle are a pair of base angles. Therefore, they are equal in measure. The sum of the measures of these three angles equals 180 degrees.

Can you now find angle OUT?

1. If it rises fr
of Angle MOU
angles: Angles

m∠MOU	102.5	Reason	Given
m∠UOT	77.5	Reason	Linear Pair
m∠OTU		Reason	
m∠OUT	🟩	Reason	

Search for triangle
You see 7 out of 29 items

Angle in Equilateral Triangle
Converse of Isosceles Triangle
Isosceles Right Triangle
Isosceles Triangle
Right Triangle Complementary Angles
Triangle Exterior Angle

If two sides of a triangle are congruent, then the angles opposite these sides (the base angles) are congruent.

Example
Given: ΔABC is an isosceles triangle, with AB ≅ AC.
If m∠2 = 52°, then m∠3 = 52°.

Fig. 1: The Geometry Cognitive Tutor. The interface breaks the problem down step-by-step and an adaptive hint is provided in a pop-up. In the top right the student's current knowledge trace is shown by yellow and green bars indicating varying degrees of knowledge component mastery (image as appeared in [1]).

No single adaptive feature distinguishes an ITS from other forms of online instruction, although roughly speaking an ITS deliberately adapts instruction to students in ways that could be considered "intelligent"—replicating the reasoning processes of a good teacher or tutor [93]. Typically the learning objectives of an ITS are characterized by a detailed breakdown of the expert knowledge it aims to teach. Toward supporting optimal learning, ITSs also often include tools for estimating individual students' mastery of components of that expert knowledge and implement interactions that support students in learning those components individually within a tutoring interface. Tutoring systems' adaptive features span multiple time scales. Outer-loop adaptivity controls the selection of next practice problems to ensure optimal learning. Outer-loop adaptivity relies upon knowledge tracing, a model of a student's current knowledge of individual knowledge components—particular facts, skills, or principles—to select next problems that will help them learn new things. Inner-loop adaptivity controls how the tutoring system provides support within each problem [112]. Inner-loop features include replications of many of the forms of support that a human tutor would provide in one-on-one practice sessions including hints, step-by-step breakdowns of problems into more granular steps [73, 98], elaborations on why certain step responses are correct and incorrect [79], and elicitations for self-explanations [21, 100], just to name a few [112].

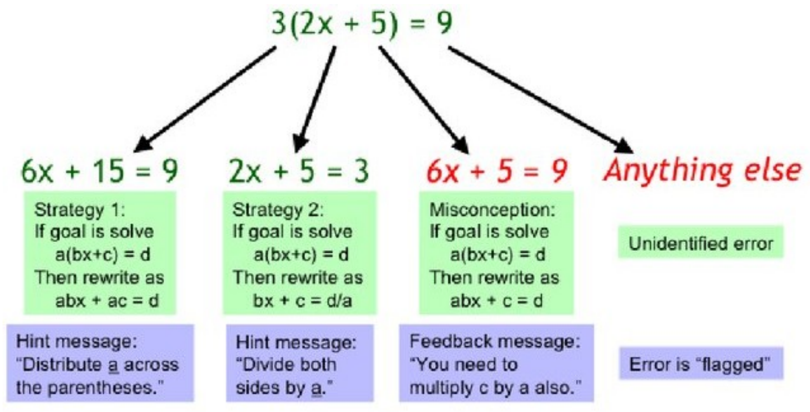
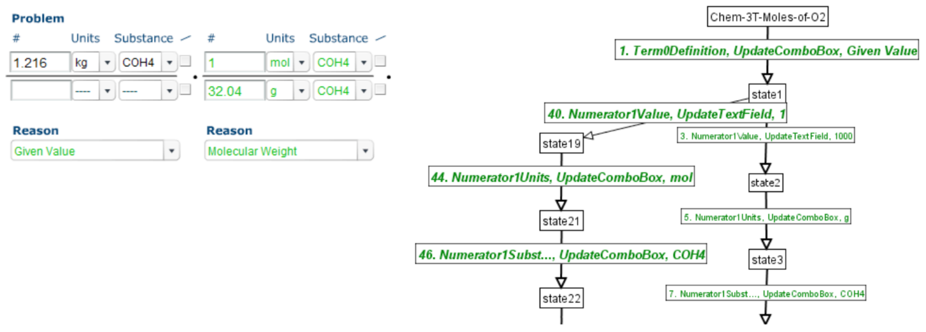

3(2x + 5) = 9
6x + 15 = 9
2x + 5 = 3
6x + 5 = 9
Anything else
Strategy 1: If goal is solve a(bx+c) = d Then rewrite as abx + ac = d
Strategy 2: If goal is solve a(bx+c) = d Then rewrite as bx + c = d/a
Misconception: If goal is solve a(bx+c) = d Then rewrite as abx + c = d
Unidentified error
Hint message: "Distribute a across the parentheses."
Hint message: "Divide both sides by a."
Feedback message: "You need to multiply c by a also."
Error is "flagged"

Fig. 2: Model tracing production rules to trace three different possible student actions. Different hint messages are available depending on the student solution path or mistake (image as appeared in [39]).

Both inner- and outer-loop adaptivity depend upon a detailed representation of what an expert in the target domain knows. Model-tracing tutors track students' step-by-step responses to determine in what ways their current knowledge aligns with or diverges from the knowledge of an expert. ITSs, like Cognitive Tutors [5] and the Andes physics tutor [114] rely upon model-tracing to ensure that the forms of support that students experience address individual misconceptions and unmastered knowledge. A large number of adaptive features can be built upon a model-tracer, and determining which adaptive features students should experience creates a challenging design space of features that may interact with one another to produce multiplicative improvements in learning students' rates [40]. Determining which combinations of features best support learning for particular domains creates a challenging experimental problem, where researchers must deploy multiple versions of the same ITS. Thus, having tools for rapidly building ITSs is important both for the practical purpose of delivering content with highly effective ITS features, and for speeding up the pace and coverage of learning engineering research to explore which features are most effective in which circumstances.

1.2 Authoring Tools

AI2T's primary purpose is to make it possible for non-programmers to very quickly build model-tracing ITSs simply by interactively tutoring AI2T on a handful of problems. Typically programming a model-tracing ITS requires 200-300 hours of development time per hour of instruction [2], and requires authors to do so by writing expert-system-like rules—a now uncommon paradigm of programming expertise. AI2T aims to significantly reduce authoring times so that the core functionality of tutoring systems can be built in a short 20-30 minute session.


Problem
#
Units
Substance
1.216
kg
COH4

Reason
Given Value
#
Units
Substance
1
mol
COH4
32.04
g
COH4
Reason
Molecular Weight
Chem-3T-Moles-of-O2
1. Term0Definition, UpdateComboBox, Given Value
state1
40. Numerator1Value, UpdateTextField, 1
state19
3. Numerator1Value, UpdateTextField, 1000
44. Numerator1Units, UpdateComboBox, mol
state2
state21
5. Numerator1Units, UpdateComboBox, g
46. Numerator1Subst..., UpdateComboBox, COH4
state3
state22
7. Numerator1Subst..., UpdateComboBox, COH4

Fig. 3: A CTAT example-tracing tutoring system and behavior graph. The author has demonstrated part of two solution paths that diverge after the first action (image as appeared in [2]).

Many efforts have been made to build ITS authoring tools that can be easily used by non-programmers, however, most methods place considerable limitations on the kinds of step-by-step support that the authored ITSs can provide to students. For instance, many tutors built with the ASSISTments authoring tool are only simple quizzes of question-answer pairs [29]. OATutors offer slightly more support for building more granular step-by-step adaptivity through the availability of strictly sequential "tutoring pathways" that break problems into multiple steps [95]. Cognitive Tutor Authoring Tool (CTAT) example-tracing allows users to demonstrate problem steps to generate a graph structure of states and actions that can diverge, re-converge, and manifest unordered groups, essentially limiting them to a slightly larger class of finite state machines-like control structures [2]. Both CTAT example-tracing and OATutor enable means of mass producing problems within these fixed control structures via a template filling approach, where special variable strings are replaced by problem specific content detailed in spreadsheets. In practice, this method of mass-producing problems still requires some programming effort, for instance, by writing programs to fill in the content of each step in a spreadsheet formula language [2, 95].

1.3 Authoring by Tutoring

The development of AI2T is motivated by the objective of building an authoring tool for non-technical users that shoulders some of the burden of authoring by exploiting methods of data-efficient induction that mimic human learning. This capability allows AI2T to induce more complex control structures than other tools intended for non-programmers, enabling forms of step-by-step support in ITSs that are typically only implementable by programming a model-tracing tutor in a production rule language. For many kinds of problems, AI2T can also eliminate the need to undergo additional programming steps to mass-produce problems since it automatically generalizes from the author's demonstrations and feedback. This research aims to democratize the development of highly adaptive tutoring systems by enabling non-programmers to build complex ITSs and by considerably reducing authoring times.

AI2T significantly improves upon the standard set by several prior attempts to build ITS authoring systems that directly generalize from natural interactions with an author. Demonstr8 is a very early example of a PBD system that allowed authors to build simple arithmetic domains in a graphical user interface that could partially generalize from an author's demonstrations [9]. Later authoring systems prototyped the possibility of more natural authoring-by-tutoring interactions driven by human-like machine learning systems called simulated students or simulated learners (all humans are "learners" even if they are not students). SimStudent is a simulated learner which is one of the earliest systems to implement authoring-by-tutoring functionality [70]. Later work by Maclellan et. al. demonstrated some benefits of using simulated learners from the Apprentice Learner (AL) framework for authoring tasks. This work demonstrated AL's ability to induce imperfect tutors for a large variety of domains [68]. Following these works we conducted a user study in which non-programmers trained AL agents in a web interface geared toward training simulated learners for ITS authoring [117]. This was the first user study to demonstrate that novice users could author ITSs with a simulated learner trained using authoring-by-tutoring. We showed a large 7x improvement in authoring efficiency over CTAT example-tracing in a single arithmetic domain known to be tedious to represent in example-tracing's finite graph structure.

Authoring-by-tutoring, like example-tracing, begins with a blank HTML interface for demonstrating tutoring system behavior. This blank interface may be programmed or built with a graphical drag-and-drop interface builder. In the case of AI2T, any blank HTML interface produced by any means typically suffices. The author first fills in the interface with an initial problem that they will solve for the simulated learner. The author provides a series of demonstrated actions detailing a step-by-step solution as a student would fill in when solving the problem themselves. The author may add additional clarifying information along with each demonstrated action, like indications of the arguments they used in their computation [70] or natural language descriptions of how they performed each action [119] to help the AI induce correct production rules that reproduce the demonstrated actions.

When the user writes in the initial state for subsequent problems, the AI agent may try to apply its induced production rules, called skills, and suggest next-step actions automatically. The user can then give correctness feedback to each suggested action. If an action is correct and the way it was computed is correct, then the author gives positive feedback, otherwise, they give negative feedback. This feedback helps the agent refine its skills so that they are only applied in appropriate situations. By tutoring the agent on several problems, and giving it demonstrations and feedback where appropriate, the AI refines its induced skills until they can be trusted as production rules for a model-tracing ITS.

1.4 Contributions of This Dissertation

In this dissertation, I demonstrate that AI2T improves upon the authoring efficiency measured in our prior study with AL. Through improvements to AI2T's learning capabilities over AL and improvements in the authoring interface, tutoring systems can be authored in less time. More importantly, while prior demonstrations with SimStudent and AL consistently exhibited imperfect performance, AI2T introduces several solutions that help authors more reliably build tutoring systems that exhibit 100% *model-tracing complete* [117] ITS behavior. A tutoring system is 100% model-tracing complete if it aligns exactly with a ground-truth tutor's model-tracing behaviors: it should only permit correct actions and no incorrect actions at each problem state. As experienced by a student, this means the tutoring system would mark the student's correct actions as correct as they proceed along any correct solution path or incorrect if they don't, and as a side effect of predicting correct solution paths also produce next step actions as requestable "bottom-out" hints. This work does not evaluate the presence of auxiliary features beyond model-tracing like the inclusion of conceptual hint messages, that might be added on top of core model-tracing behavior. Model-tracing completeness only evaluates the completeness of the core hard-to-program behavior of a rule-based ITS, on which additional support systems are typically added.

AI2T implements two significant improvements in machine learning algorithms over prior work with SimStudent and AL that help it more robustly induce 100% model-tracing complete programs from less training data. One of these new algorithms introduces a new data-efficient and performant learning mechanism called STAND for learning the preconditions of induced production rules. The second algorithm introduces a learning mechanism for hierarchical task network induction not present in SimStudent and AL, that was explored in a much earlier simulated learner system called Sierra [110, 111]. This mechanism enables AI2T to induce knowledge structures that are more reliable, more explainable, and more concise than those induced by previous simulated learner systems. Compared to Sierra's HTN induction approach, our method is agnostic to the order that training problems are presented and induces an HTN representation that captures forms of solution flexibility that are desirable in an ITS authoring setting including unordered groups and optional or conditionally applicable actions.

Both of these two new algorithms implemented in AI2T contribute to a solution for overcoming a difficult issue for authors: knowing when the simulated learner has been trained on a sufficient number of problems such that its induced behavior can be trusted on new untrained problems. AI2T's two new algorithms, especially STAND, enable it to have very accurate self-awareness of its learning progress. STAND allows AI2T to produce a certainty score called *instance certainty* that estimates prediction certainty as the degree of agreement in a space of possible induced programs. We show preliminary evidence that users can successfully use instance certainty as a heuristic for determining when AI2T has achieved correct and complete behavior. In simulations of interactive authoring, we show that this new measure has several advantages that make it a more desirable heuristic than traditional estimations of prediction probability, especially in an interactive task learning (ITL) setting where users expect the AI to induce correct knowledge structures from just a few instances of interactive instruction. STAND's instance certainty measure is strongly correlated with actual performance improvements on holdout data, and it has very high utility in an active-learning [106] setting. In other words STAND is very good at picking potential training examples that will help AI2T learn most effectively.

This dissertation is organized into 9 chapters including this introduction. In chapter 2 we will review several simulated learner systems that have preceded and inspired the development of AI2T with special attention to the common collection of learning mechanisms that they each implement—a collection of mechanisms that I have called Decomposed Inductive

Procedure Learning (DIPL). In chapter 3 we summarize the results of our 2020 prototype interface design for authoring with AL. In chapter 4 we describe how the interface for AI2T was developed in response to the lessons learned from our earlier work and interactions with users across several pilot sessions and two user studies. Chapters 5 and 6 describe STAND and process-learning respectively.

In chapter 7 we report on a pair of user studies where participants taught AI2T two math domains: multicolumn addition and fraction arithmetic. These studies form a pseudo-experiment between two different versions of AI2T which differ along several features. This pseudo-experiment sheds some light on the backend and frontend components of AI2T that are essential to its effectiveness and usability.

Finally, in Chapter 8 we address an elephant in the room: what place does an interactive task learning method like AI2T serve in the rapidly advancing world of pretrained large language models (LLMs)? In short, we address some of the opportunities that LLMs offer interactive task learning methods like AI2T and outline some of the opportunities, dangers, and limitations of employing LLMs in education to implement adaptive features. Chapter 9 concludes by summarizing the contributions of this dissertation and outlines directions for future work.

Chapter 2

Decomposed Inductive Procedure Learning: An Overview

When students practice skills in controlled learning environments like intelligent tutoring systems (ITS)—educational technologies that mimic one-on-one tutoring interactions through highly adaptive step-by-step instructional support—their learning proves to be remarkably quick [38]. The data collected from these learning environments make it clear that when given proper support humans are orders of magnitude more data-efficient learners than today's data-driven machine learning (ML) systems. While data-driven ML may require thousands to millions of examples to replicate human performance capabilities (and often take many passes through that data), humans can typically learn general procedural tasks incrementally from fewer than a dozen practice opportunities—learning steadily from being complete novices to reaching a point of task mastery characterized by error rates of less than 10%.

Recent advances in ML for mathematical problem solving show remarkable performance capabilities on challenging problems [48]. For instance, Large Language Models (LLMs) have succeeded on challenging word problems [14] from the MATH dataset [30] including recent GPT models trained step-by-step which have demonstrated 78% accuracy [61]. While these systems have arguably achieved human-like performance, in terms of data-efficiency they are about as far from human learning as one could imagine. The quantity of learning experiences required to achieve these capabilities is many orders of magnitude greater than a human would experience in a lifetime, and the process of learning from those experiences is relegated solely to the sorts of pattern recognition and fuzzy mimicry achieved by fitting to data in a data-driven manner. Humans learn by engaging a wide array of knowledge construction strategies that are not well captured by simple data-driven prediction. Instances in past AI systems include constructing knowledge through interpretation of and disambiguation of natural language instruction [33, 119], representation learning [57], inductive generalization from examples [111], and analogical reasoning [36]—just to name a few—all of which have typically been achieved through acquiring or utilizing symbolic representations.

A shortcoming of many prior attempts to replicate various human learning capabilities has been the need for researchers to pre-engineer an AI system's prior knowledge and/or environment in domain-specific ways—limiting practical applicability beyond isolated experiments. Much work in logic programming [69] and cognitive architectures like ACT-R [103] and SOAR [45], for instance, could be held to this criticism. Data-driven ML, although impressive in its ability to replicate patterns in data without hard-coding knowledge or features, has analogous drawbacks due to the engineering effort required to prepare training environments, or collect, clean, and preprocess data, in addition to the effort of selecting, training, and tuning models. Laird and colleagues [46] have promoted a research vision to overcome these shortcomings via systems of Interactive Task Learning (ITL) where "untrained humans" as opposed to programmers "teach agents new tasks and task knowledge". In partial alignment with this end, LLMs have demonstrated successes at enabling untrained users to clarify or refine requests to better utilize their numerous pre-trained generative capabilities [102]. Yet the broader ITL vision of ML that can efficiently learn rich representations and performance capabilities in a mostly bottom-up constructive manner from natural instruction is still an ongoing project within the field of AI.

2.1 Simulated Learners

In this work, we build upon a body of work that has set out to study human learning computationally by developing ML methods that learn rapidly from interactive natural instruction instead of from large datasets or well-structured environments. We focus on the commonalities between three simulated learner (SL) systems: Sierra [111], SimStudent [70], and the Apprentice Learner (AL) architecture [66], which have been used in both computational modeling and interactive task learning of academic procedural tasks. We also discuss our own implementation of these mechanisms in AI2T. These systems share a common breakdown of learning mechanisms for induction from examples and feedback situated in step-by-step display-based instructional environments (i.e. ITSs). VanLehn [111] has shown that this form of inductive learning underlies the acquisition of early mathematical skills. Later efforts with AL have shown that it can closely match student performance changes (i.e. their learning curves) practice opportunity by practice opportunity [66,120]. When these SL systems experience examples they induce production rule-like skills that may initially produce errors, but are rapidly refined toward a state of errorless mastery through supervised practice.

The procedure learning implemented in these inductive SL systems differs from the various forms of speed-up learning achieved by systems like SOAR [45] and related approaches like Neves et. al.'s approach to simulating math learning [90]. In these examples learning is assumed to be "a side effect of performance" [47] of a planning system executed on pre-built environments. Inductive SL systems by contrast learn by interpreting external instructional experiences situated in natural supervised practice environments, like for instance a tutoring system in an online interface. Thus, inductive SL systems do not expect a well-structured environment ready-made with primitive actions, or easily verifiable goal states. In these systems the primitive operations of a procedure and the means of verifying its completion are induced from examples and feedback. Inductive SL systems learn these capabilities by inducing minimal sets of production rule-like skills from demonstrated worked examples and refining those skills through supervised practice. This induction can leverage domain-general prior knowledge like arithmetic functions, or domain-specific prior knowledge if available, but does not rely on pre-specified domain-specific affordances within an environment. AL agents, for instance, can learn interactively from instruction that begins in blank HTML interfaces [117] instead of from specialized environments pre-programmed with strong representational commitments, such as for instance in [69]. This general purpose bottom-up induction is essential in ITS authoring applications where we expect many of the potential users to be non-programmers.

2.2 Decomposed Inductive Procedure Learning

Advances in data-driven machine learning have led to a generation of AI technologies built almost exclusively with artificial neural networks. While there are many kinds of neural network architectures and training methodologies, essentially all neural networks are learned via the same single learning mechanism: regression by gradient descent. The back-propagation algorithm [28] enables loss gradients of the weight of these models to be computed over large multi-layered network structures, enabling flexible training of monolithic multi-purpose deep-learning models. Using collections of differentiable network elements, enormous quantities of data, and copious computing power and time, deep learning has been shown to learn a wide variety of things with some expected rate of error. This approach is much slower than the rapid incremental learning that humans employ and requires neural network models to slowly tweak weights as they review datasets in multiple passes.

Reinforcement Learning (RL) is an approach to learning procedural capabilities by learning policies $\pi(s) \rightarrow a$, directly or indirectly, that map states to actions to maximize reward over task episodes. RL can maximize overall collected rewards in environments where reward signals are delayed between states, or even with non-deterministic actions. As RL algorithms have evolved and come to rely largely on deep learning, they have shown successes at challenging games [80] and robotics tasks [105], and have become a go-to choice even in deterministic procedural domains that require symbolic manipulation including theorem proving [35], and for learning mathematical tasks such as geometry [123], and early K-12 mathematics like fractions and long arithmetic [97] (as we do in this work). However, these approaches, like most deep learning-based ML, require many examples collected over typically at least thousands of task episodes, and generally learn by attempting tasks in specially formatted environments with pre-specified action spaces.

Inductive simulated learners like Sierra, SimStudent, and the Apprentice Learner also learn procedural capabilities, but are remarkably more data-efficient learners than RL methods—about as data-efficient as human learners. They can acquire capabilities from just dozens of examples by utilizing a combination of three to four core learning mechanisms, and sometimes more, that work together to induce structured knowledge representations. To highlight the inductive power of decomposing learning processes into multiple learning mechanisms, we coin the name Decomposed Inductive Procedure Learning (DIPL) to refer to this collection of mechanisms. The data-efficient induction achieved by DIPL-based simulated learners arises precisely because the learning processes are decomposed into multiple parts that have well-defined, but intertwined roles, as opposed to being handled by end-to-end single-mechanism approaches.

2.2.1 Decomposing RL into DIPL

A good way to understand DIPL, and why it is about as data-efficient as human learning [66] and orders of magnitude more data-efficient than RL, is to consider how its multiple learning mechanisms break up the highly unconstrained induction problem that gradient descent-based RL methods attempt to solve. A key element of DIPL's advantage is that it induces skills (which are like production rules in development). An agent induces multiple skills to produce actions instead of selecting among predefined grounded actions [65] like RL methods. These skills are learned through the induction and generalization of distinct parts characterizing *how* certain kinds of actions are taken, in which circumstances (i.e. *where*) it is possible to take those actions, and *when* (i.e. under what circumstance and in what order) those actions must be taken to correctly execute a target behavior. For each of

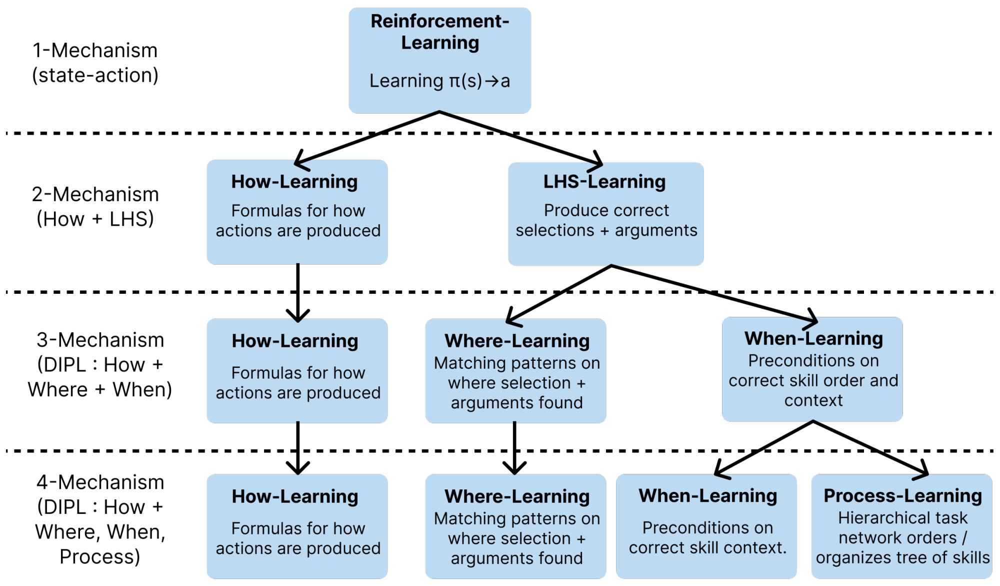

1-Mechanism (state-action)
Reinforcement-Learning
Learning π(s)→a
2-Mechanism (How + LHS)
How-Learning
Formulas for how actions are produced
LHS-Learning
Produce correct selections + arguments
3-Mechanism (DIPL : How + Where + When)
How-Learning
Formulas for how actions are produced
Where-Learning
Matching patterns on where selection + arguments found
When-Learning
Preconditions on correct skill order and context
4-Mechanism (DIPL : How + Where, When, Process)
How-Learning
Formulas for how actions are produced
Where-Learning
Matching patterns on where selection + arguments found
When-Learning
Preconditions on correct skill context.
Process-Learning
Hierarchical task network orders / organizes tree of skills

Fig. 4: Decomposition from 1-mechanism learning, like RL, that maps states-to actions to DIPL's 3-mechanism or 4-mechanism learning. A 2-mechanism system bridges the difference and uses how-learning but combines where- and when-learning into LHS-learning.

these how-, where- and when- parts of a skill there is a corresponding learning mechanism. Between Sierra, Simstudent, AL, and AI2T, the implementations of these mechanisms vary considerably, yet the division of mechanisms is largely the same. Unless otherwise stated, our descriptions will align with the particular implementation we use in AI2T.

By analogy to RL's choice of actions via a global policy $\pi(s) \rightarrow a$ DIPL's induced skills produce actions via multiple symbolic pieces $When(s, Where(s)) \rightarrow How(Where(s)) = a$. Ablating from RL's 1-mechanism learning we can produce DIPL's 3 or 4-mechanism structure by incrementally introducing learning mechanisms. First, we can consider a two 2-mechanism system by introducing how-learning and a combined mechanism which we'll call left-hand-side (LHS) learning that learns the if-part in the if-then structure of a production rule which is typically called the left-hand-side.

How-learning serves a similar role to action model learning [6] in the planning system literature, except that instead of inducing simple operators that describe the addition or deletion of grounded predicates between states, how-learning is used in situations where the computations underlying an action are somewhat ambiguous. How-learning finds the most likely program that can reproduce values within an action. A mathematical example is finding a composition of known functions that can reproduce an example value from other available values in a problem state (e.g., how could the value 10 in a demonstration for a step be produced from the numbers 2, 3, and 4). We'll focus on this numerical framing of how-learning, as it suites many academic tasks, but the idea of how-learning captures a broader idea, of learning *how* different kinds of actions are performed. For instance, a mechanism that determines how a system of robotic actuators might carry out an atomic action would also be a sort of how-learning. In general, how-learning solves the problem of abducing a program (i.e. finding the most likely one) that reproduces one or more demonstrated actions.

We need another learning mechanism to decide where and when potential applications of the generalizations learned by how-learning ought to be applied. One way is with a combined left-hand-side (LHS) learning mechanism that learns the correct object to act upon (called the selection) and the correct objects to use as arguments. LHS-learning can be a simple multi-class classifier that predicts a selection and set of arguments from a problem state. A how-part and LHS form a simple kind of skill that can produce actions on a state.

To get a 3-mechanism model we can break down LHS-learning even more. Instead of using a single classifier, we can use a where-learning mechanism to learn matching patterns that pick out candidate selections and arguments, and a when-learning mechanism—a binary classifier that learns when a candidate matching set should be applied with a how-part formula to produce an action that is correct given the current problem state. Breaking down LHS-learning into these two parts enables where-learning to modify matching patterns to produce spatial generalizations, and when-learning uses variables introduced by the matching pattern to learn relational preconditions that gate how individual skills can be applied. Note that a particular SL agent may learn many skills each with an independent how-, where- and when- part.

To get a 4-mechanism model we can relegate part of the role of when-learning so to a 4th mechanism called process-learning that organizes skills into a hierarchical task network. Without process-learning, when-learning learns conditions that control both the order that skills are allowed to produce actions and the context in which individual skills are applicable. Process-learning takes over the role of learning the order that skills should be applied by ordering them explicitly (or not) within its induced hierarchical task network structure. Within this structure, the application of every primitive skill—a skill that directly produces an action—is gated by the application of higher-order skills representing an intention to achieve higher-level goals. This structure reduces the role of when-learning so that it must only ensure that the correct skills in this hierarchical task network structure are applied if there is a disjoint choice of possibilities. SimStudent and AL use a 3-mechanism model, whereas Sierra and AI2T use the 4-mechanism model that includes process-learning.

This idea of decomposing or distributing learning across mechanisms with unique, but complementary roles has much precedent in AI research. For instance, actor-critic algorithms [43] typically divide learning between an actor that performs the target task and a critic that helps to train the actor. The important elements of DIPL's decomposition, which we hypothesize contributes to its data-efficiency, are that each learning mechanism assists in or simplifies the learning of others, and that each learning mechanism is instantiated separately across each unique learned skill (i.e., production-rule). Instead of resigning task learning to a single global mechanism, each instance of each individual learning mechanism serves a bite-sized and well-defined role; each one responsible for acquiring particular kinds of generalizations for individual skills—where each skill is built up from these induced pieces and is responsible for performing particular kinds of actions.

2.2.2 How-Learning

How-learning acquires the then-part of a skills if-then-structure, and is responsible for determining *how* actions are produced. This mechanism takes a goal value from a demonstrated worked example and finds a composition of domain-general prior-knowledge functions (like

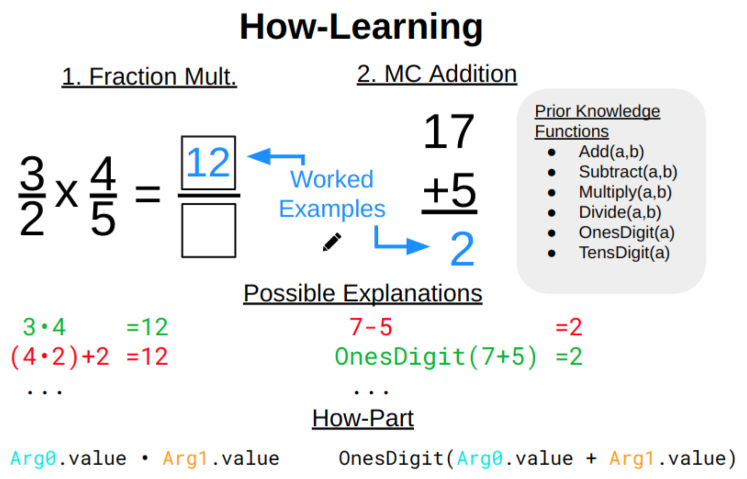


Fig. 5: How-learning explanations produced for worked examples for the first steps of fraction multiplication and multi-column addition problems.

arithmetic functions and string operations) that reproduces the value given the set of available values visible in the current problem state. How-learning synthesizes short programs via an abductive explanation process. It finds the shortest explanation for how a value in a demonstrated worked example came to be produced. To do so it searches over compositions of known primitive functions in an effort to reproduce the worked example. In principle, this search may produce multiple candidate compositions that reproduce the worked example, some of which may be incorrect.

Among the candidate compositions that reproduce the worked example the most parsimonious (having the fewest operations and arguments) is chosen. If several are equally parsimonious, then one is selected randomly. The chosen explanation is generalized by replacing the constants in the grounded composition with variables. For instance, `OnesDigit(7+5)` in the above example (Fig. 5) may be generalized to depend upon two argument variables `Arg0=Var(TextField)` and `Arg1=Var(TextField)` which select TextField type interface elements. Additional conversion functions may be added to the composition so that the `.value` fields of those elements are evaluated as floats.

Set-Chaining A constraint of ITL systems is that the implemented algorithms must be efficient enough that users do not experience significant delays. How-learning can become an issue in this regard because it engages in a form of search that grows exponentially with composition length. Prior implementations of how-learning have composed primitive functions in an iterative-deepening fashion [70]. At each deepening, the composition depth (e.g., Add(a,b) is depth=1, Divide(Add(a,b), Subtract(c,d)) is depth=2) is increased, and each composition of the current depth is formed and executed on all permutations of the source values. This continues up to a maximum search depth. Executions that reproduce the goal value can be stored as possible explanations, or search can stop short at the first reproduction of the goal value.

The how-learning implementation used in AI2T utilizes a search process which we call Set Chaining (Algorithm 1), which we have found to have efficiency benefits over serially com-

Algorithm 1 Set Chaining

```
Input: goal $g$, values $V_0$, functions $F$
Parameter: maximum search depth $d$
Output: composition set $C$
 1: $S_0 = UniqueSet(V_0)$.
 2: for $k \in \{1, \ldots, d\}$ do
 3:     $S_k = copy(S_{k-1})$.
 4:     for $f \in F$ do
 5:         $n$ = number of args in $f$
 6:         for each permutation $\{a0, ..., a_n\}$ of $S_{k-1}$ do
 7:             $v = f(a0, ..., a_n)$
 8:             $S_k[v] = [Record(f, a0, ..., a_n), ...S_k[v]]$
 9:     if $g \in S_k$ then
10:         return $ExplanationIterator(S_k[g])$
11: return null
```

posing compositions as in prior iterative-deepening approaches. Set Chaining is amenable to SIMD (Single Instruction, Multiple Data) parallelization because it applies primitive functions over a growing set of intermediate values computed at successive depths. Only the unique values generated from all previous execution depths are used, instead of using every value instance individually. This reduction of potential arguments greatly reduces the combinatorial explosion of the search. By a common characterization of search methods, SetChaining is akin to a graph search method while serially composing functions is akin to a tree search method. SetChaining trades the memory overhead of storing intermediate values, for the efficiency benefits of reducing repeated work, and parallelizing execution.

Each individual function execution in SetChaining is tracked with a lightweight record data structure, that is inserted into a linked list stored in a hash map keyed by values. When the goal value is found, Set Chaining returns an `ExplanationIterator` which incrementally enumerates all solutions up to the depth where the goal was found. At instantiation this iterator works back through the record structure, building up a tree of records that lead to the goal value. The iterator then uses a depth-first traversal of this record tree to interactively generate compositions and arguments that explain the goal value. The `ExplanationIterator` can generate subsets of explanations if there happen to be more than are computationally feasible to enumerate. This means that in an interactive training setting users can be made aware if their worked examples produce very large numbers of explanations. This can help them decide if additional instruction is necessary to clarify the correct explanation.

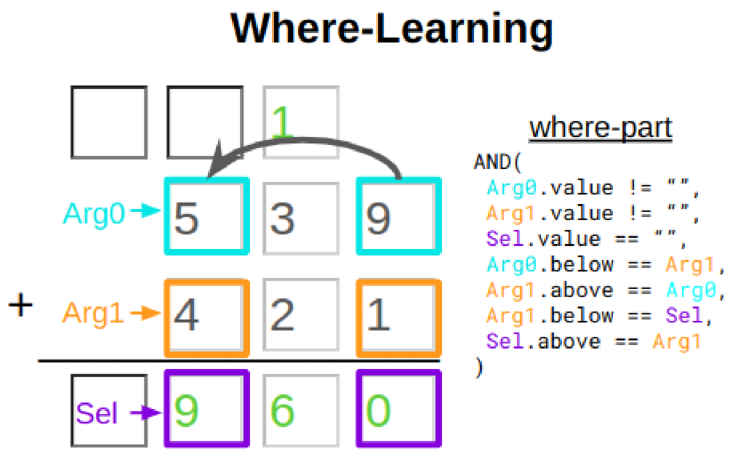
Clarifying Correct Explanations Several methods of annotating worked examples with additional instructional information have been used in prior work to narrow how-learning's selection of explanations. For instance, explanations with incorrect arguments can be culled from consideration by explicitly pointing out the arguments used to compute the worked-example (called foci-of-attention in prior work) [70]. Another method is to label each worked example as an instance of a particular skill, and then consider only those compositions that jointly explain multiple worked examples annotated with the same skill label. Our more recent approaches enable how-learning to be guided by natural language instruction [119]. Instructors are able to provide a hint describing how their worked examples were produced, and this accompanying hint is processed into a policy that guides the search for a correct composition.

Each of these clarifications support distinct forms of mutual disambiguation between worked examples and other kinds of instruction. For instance, our prior work that added natural language processing to how-learning showed that hint annotations and worked examples were more likely to reproduce correct generalizations than when provided with either alone [119]. Since our ablation analyses will compare these methods to reinforcement-learning—which typically relies solely on agent generated or expert provided examples annotated with reward—we restrict ourselves to using just argument annotations in one of our two task domains.

2.2.3 Where-learning

Where-Learning discovers matching patterns that can pick out candidate applications of skills. While how-learning is responsible for producing the operational generalizations within skills, where-learning is responsible for building spatial generalizations that determine the situations where skills can be applied. Where-learning is similar to the data description problem in programming by demonstration literature [60]: the problem of producing generalizations (e.g., data descriptions) that uniquely select correct data to operate on. However, where-learning is a more restrictive concept than data descriptions since it explicitly captures just the spatial relationships that are common between demonstrations of the same skill, but not necessarily preconditions that would select only correct candidate application of a skill, or pick one over another (this is the role of when-learning).

For instance, in a multi-column addition task, where-learning could support generalizing a skill for computing the one's digit of a partial sum so that it can be applied across columns. The patterns learned by where-learning are expressed in terms of a set of argument variables (generated from how-learning, like `Arg0` and `Arg1` in the example above) and a single selection variable (e.g., like `Sel=Var(TextField)`) which matches the interface element that will be acted upon.



```
AND(
 Arg0.value != "",
 Arg1.value != "",
 Sel.value == "",
 Arg0.below == Arg1,
 Arg1.above == Arg0,
 Arg1.below == Sel,
 Sel.above == Arg1
)
```

Fig. 6: An example of a where-part that has been generalized to act across columns in multi-column addition.

The learned where-part pattern consists of a logical statement, expressing necessary conditions and spatial relationships that must be upheld between its variables. The where-part pattern produced from an initial worked example is typically highly specific and constrained such that it may only bind to a limited set of selections and arguments. However, if subsequent examples are not captured by a skill's current where-part then it is generalized to capture the new selections and arguments, either by generalizing individual literals in the statement or by removing them entirely. How-learning has a supporting role in identifying the sets of arguments where-learning generalizes from. How-learning attempts to explain each new worked example using existing skills' how-parts. If there are any candidate explanations, the one with arguments that would make the minimal change to an existing skill (quantified by a structure mapping score that measures explanation structure similarity) is used for where-part generalization. Otherwise, how-learning generates a new skill from the example.

In some implementations, including AI2T's, where-parts can have additional variables that match neighbors or parents of the selections and argument variables. The structure of these additional variables can depend heavily on whether the problem state is structured hierarchically, or if the agent is supplied with prior representation parsing knowledge. One example, used in past work [57], is knowledge for structuring equations into hierarchies of expressions, terms, coefficients, and variables. In the interest of eschewing representation engineering, which would require some programming in an ITS authoring setting, and to maintain greater parity with the purely flat state representation expected by the typical deep-RL methods, we limit ourselves to simple adjacency relationships (i.e. right, left, above, below) computed directly from the static positions of HTML elements.

2.2.4 When-Learning

When-learning identifies the contexts or order in which skills ought to be applied. The when-part of a skill consists of necessary preconditions on the applicability of the skill. When-learning is typically implemented by a binary classifier. This classifier is often symbolic and relational. For instance, prior work has employed inductive logic programming, decision trees, and incremental concept learning approaches [66, 70]. In a given state, the learned when-part preconditions distinguish whether a particular candidate application of a skill matched by the where-part pattern ought to be applied as an action or not. Collectively the where- and when-parts make the left-hand-side of skills. While how- and where-learning operate purely over positive worked examples, when-learning utilizes both positive and negative feedback on attempted actions. When-learning uses the features directly available in the problem state, and those inferred using additional prior knowledge feature functions, like Equals(a,b), which recognizes that pairs of values are equal.

When-learning takes positive and negative examples of candidate applications of the skills applied to particular problem states. Thus, where-part processing in each skill assists when-learning by generating candidate selections, and sets of arguments for each state, action, reward triple. This allows when-learning to construct when-parts as relational concepts consisting of literals that express features in the state as they relate to the selection (e.g., Sel) and arguments (e.g., Arg0, Arg1) instead of as they relate to particular interface elements. This enables generalization across spatially distinct instances of the same skill. For instance,

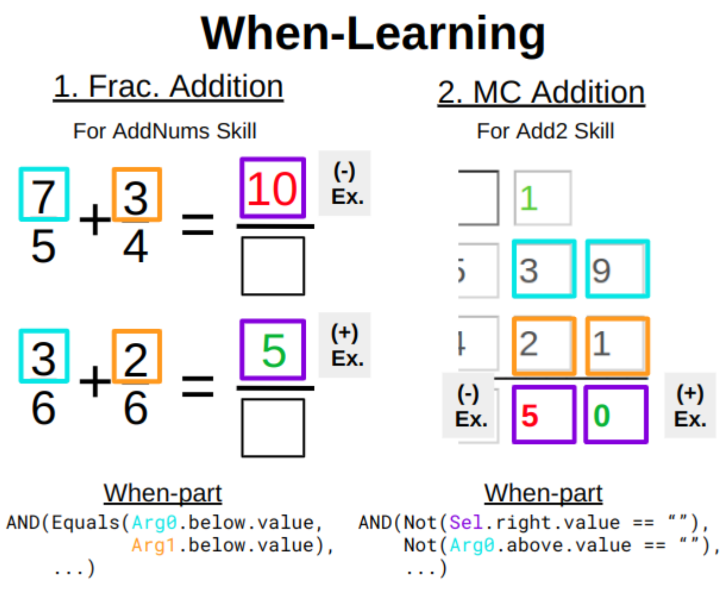

When-Learning
1. Frac. Addition
For AddNums Skill
2. MC Addition
For Add2 Skill
(-) Ex.
(+) Ex.
When-part
AND(Equals(Arg0.below.value, Arg1.below.value), ...)
When-part
AND(Not(Sel.right.value == ""), Not(Arg0.above.value == ""), ...)

Fig. 7: Positive and negative examples of an AddNum skill in fraction addition, and an Add2 skill in Multi-Column addition. Partial when-parts that accept the positive examples but reject the negative example are shown expressed with relational features generated by relative featurization.

the when-part in the fraction example in Figure 7 includes a literal `Equals(Arg0.below.value, Arg1.below.value)` which references the values of the elements below `Arg0` and `Arg1`.

Prior work has used FOIL, an inductive logic programming (ILP) method, to learn concepts with relational constraints like those above. FOIL searches for and combines new literals like `Below(Arg0, A)` expressed in terms of source variables like `Arg0`, or invented variables like `A`. Searching for such statements can be computationally demanding, posing a risk of latency experienced by an end user training a SL interactively. Instead we introduce a streamlined means of restating features in the problem state relative to the selection and argument variables. The shortest path through adjacency relationships is found between interface elements using the Bellman-Ford algorithm. Then each feature in the state is relabeled with the shortest path from the selection or arguments (in the dot-notation of Figure 4). This method of "relative featurization" also allows us to keep the when-learning classifier independent of relational feature generation.

2.3 Task Domains

We build on the TutorGym environments presented by Maclellan et. al. [65] to wrap two different ITS domains into RL gym environments. (1) A Fraction arithmetic domain (Figure 8) which randomly selects among: adding same denominator fractions, different denominator fractions, and multiplying fractions. (2) A multi-column addition tutoring system for adding 3-digit numbers (Figures 6,7). In both domains agents can request demonstrated worked examples (demos), and receive immediate reward signals (1 for correct, -1 for incorrect) on attempted actions. These particular tasks can tractably be learned via RL without needing to heavily engineer extra affordances into the environment. The action spaces consist only of,

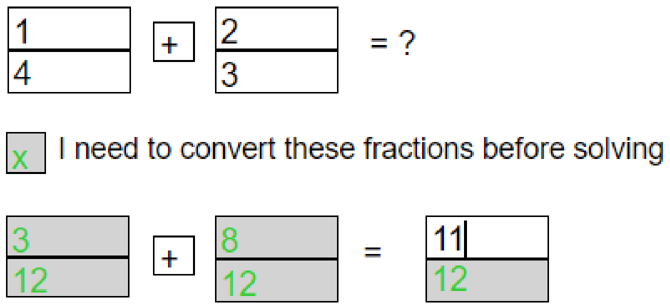


Fig. 8: Fraction arithmetic tutoring system for teaching multiplying and adding fractions. Learner insert an 'x' to the 'check_convert' field if the fraction must first be converted.

checking boxes, placing numbers in fields, or pressing the 'done' button, meaning there are finitely many unique primitive actions, and a discrete state space with features that overlap between problems.

In the fractions tutor, the agent must perform the correct fraction arithmetic procedure step-by-step (multiplying, adding, or converting then adding) based on the two starting fractions and operator. In the RL-Gym wrapper for this environment the agent is able to fill in each of 6 number fields with the numbers 1-450, fill in the 'check_convert' field with an "x" or press the done button, for a total of 2,702 unique actions. The multi-column addition domain, see Figures 6 and 7, has 7 fields that can be filled with the digits 0-9, plus a done button for a total of 71 unique actions. In this domain the agent must compute the sum of two 3-digit numbers by computing each partial sum in right-to-left order by placing the ones digit and then carrying the ten's digit when necessary. In both domains the state is encoded into a vector with 0.0 or 1.0 in each element via one-hot encoding (size 2,000 in fractions and 240 in multi-column addition). The one-hot encoding maps each unique interface element-attribute pair to a slot in the state vector.

The non-RL-based agents instead experience the state in its original object-based representation, where each object has a unique identifier, type, position, shape, and value. Additionally no predefined action space is provided to these agents. Instead, they learn how to produce actions by applying how-learning to the on-demand worked example demos. These agents are instantiated with the primitive domain-general prior knowledge functions necessary to compose how-parts for each task: `Add(a,b)` and `Multiply(a,b)` for fractions and `OnesDigit(a)`, `TensDigit(a)`, `Add3(a,b,c)`, `Add(a,b)` for multi-column addition. In multi-column addition, extraneous how-learning explanations are common thus we aide how-learning by annotating each demo with its arguments. We do not provide these annotations for fractions. In the fractions domain we provided a single feature function `Equals(a,b)`, enabling the agent to identifying equal values, which is necessary for learning to check for equal denominators. Toward the objective of building an SL based authoring tool, these ready-made ITSs are not just learning environments, but also essentially simulations of the ideal tutoring interactions an author would provide to the SL. The ideal ML approach for this purpose should show high data efficiency for quick training and low long-tail error for robust grading behavior.

2.4 Ablation Analysis

2.4.1 Methods

We apply two RL approaches: an off-policy Deep-Q-Network (DQN) model [81] and an on-policy Proximal Policy Optimization (PPO) model [105]. PPO has become a popular RL approach for its relative stability, data-efficiency, and consistent convergence without hyper-parameter tuning. We additionally train agents with or without automatically provided demo worked examples. In the former case (indicated by "+Demos"), each incorrect action is followed by training on the current step's demo worked example. This mimics the capability of DIPL-based simulated learners to request demos when no next action can be produced. Unfortunately, this method only works with the DQN models, as there is not a simple method for training on-policy methods like PPO with actions not produced by its current policy. All models were implemented using OpenAI's stable baselines library and were trained for 500,000 timesteps. For a symbolic comparison we additionally train a decision tree using the "+Demos" training modality.

Our 2-mechanism model uses a Set Chaining how-learning mechanism to learn individual skills, but only a single Left-Hand-Side (LHS) learning mechanism that predicts where and when those skills should be applied. The LHS-learning uses a decision tree as a multi-class classifier that predicts the ids of the selection and arguments for the correct next action from features of the problem state.

Finally we utilize AI2T with 3-mechanisms (how, where, when) as a DIPL-base agent. This setup is essentially a re-implementation of the mechanisms used in the Apprentice Learner (AL) Architecture. Set-Chaining is used for how-learning. Where-learning uses a MostSpecific implementation that recalls sets of selections and arguments from past examples. Finally, when-learning is achieved with a decision tree. We train both with and without relative featurization to illustrate the effects of utilizing where-part processing in when-learning.

2.4.2 Results

		Fractions	MC Addition
1-mech	PPO	Not Converge	30,642
	DQN+Demos	11,315	9,496
	DT+Demos	1,944	7,816
2-mech	How+LHS	17	270
3-mech	DIPL (no rel. feat.)	33	38
	DIPL	20	19
	Human Data	∼9-14	N/A

Table 1: Number of problems before $< 10\%$ average error.

Our results are summarized in Table 1 and select learning curves are shown in Figure 9. The DQN models converged only in the "+Demos" condition. PPO successfully converged only in multi-column addition. Among the RL methods the "DQN + demos" approach achieved the best data-efficiency up to the <10% error mastery point requiring about 10,000

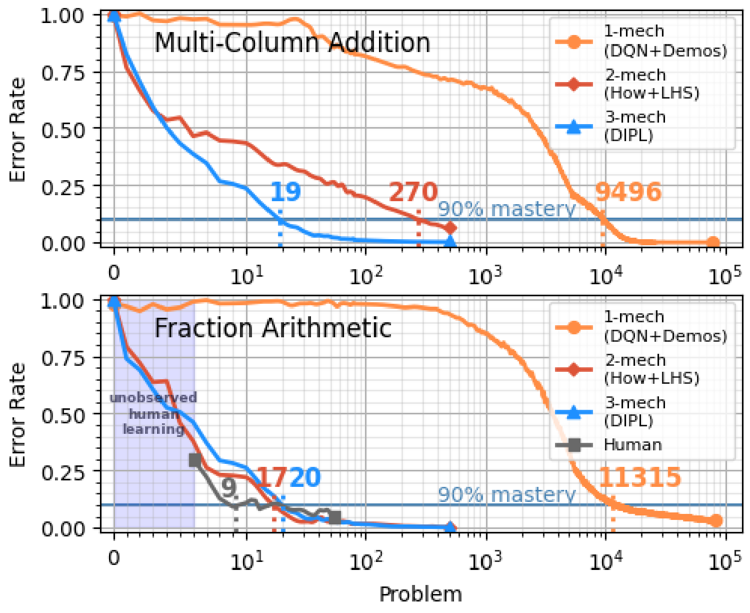


Fig. 9: Learning curves for DQN-Demos, How+LHS, and DIPL. Human curves are included for fractions offset to account for unobserved learning opportunities.

problems in each task. Decision trees were more data-efficient than the RL methods, but still required 1,944 problems in fractions and 7,816 problems in multi-column addition. The 2-mechanism model showed a dramatic improvement in data-efficiency, taking just 17 problems to master fractions but 270 for multi-column addition. The 3-mechanism DIPL model by contrast mastered both in 20 problems or less. Turning off relative featurization resulted in a 13-19 problem worsening of data-efficiency. We additionally include human data collected by Patel et. al. [96] from the same fractions ITS. We shift this data by 6 problems to align the initial 30% student error-rate with the most efficient SL, leading to an adjusted mastery intercept of around 9-14 problems. The DIPL agents and humans show similar initial learning rates, but the DIPL SLs improve more rapidly beyond the mastery threshold up to less than 1% error after about 130 problems.

2.4.3 Discussion

Data-Efficiency Our 1-mechanism models cover only a small number of RL training approaches yet are fairly representative of the data-efficiency of RL—a level far from acceptable for ITL purposes. One contribution to inefficiency is that the RL learns how to pick correct numbers instead of how to compute them (similar to how LLMs approach math). However, in similar experiments in which RL agents were given domain-specific primitive actions equivalent to what an SL would induce through how-learning, training still required thousands of episodes [65]. Thus, there is not much hope that these approaches will be useful for ITL purposes even with knowledge engineering assistance.

The decision tree's relatively better data-efficiency demonstrates an advantage of symbolic versus sub-symbolic learning. However, learning decomposition proved to be the more essential condition for achieving human-like data-efficiency. In the 2-mechanism model how-learning helped produce near human-like efficiency for fractions, but in multi-column addition the further decomposition of separating where- and when-learning was essential, since where-learning allows when-learning to spatially generalize across multiple uses of the same skill (like across columns).

Representation Prior academic task learning methods have leveraged rich pre-programmed domain representations (e.g., Neves [90] and Manhaeve et. al [69]). By contrast our approach succeeds despite using relatively simple representations drawn from static-HTML pages. While this is a compelling ITL demonstration, applying DIPL in domains like algebra for instance requires supplying prior hierarchical representation parsing knowledge. Li et. al. [57] have explored offline representation-learning in SimStudent. This approach is relatively data-efficient, but needs to be trained in advance of regular procedure learning. Efficient online representation-learning that works along side procedure learning remains an open problem.

Decomposing Further with Process-Learning In this section we have discussed the 3-mechanism approach used by the Apprentice Learner (AL) framework, and SimStudent. In chapter 6 we add a 4th mechanism, process-learning that learns a hierarchical structure of skills. VanLehn's Sierra [111] is an early example of this 4-mechanisms approach. We show in chapter 6 that this additional learning mechanism produces yet-greater data-efficiency, and more robust learning that goes beyond achieving the high ($> 90\%$) success rates characteristic of human mastery, to achieve programs that are 100% correct and complete (with respect to model-tracing performance) on large holdout datasets.

Scope of Applicability As the name suggests, Decomposed Inductive Procedure Learning is an approach for learning procedural tasks—typically ones that can be performed within a structured user interface. Additional performance and learning mechanisms would be needed to extend DIPL to tasks that require parsing or producing images or natural language. Nonetheless, DIPL is generally applicable for interface-based programming by demonstration, and excels at learning programs that require some contextual decision making. For instance, correct execution of the two domains we show in this work require applying different sequences of actions between tasks, and decisions need to be made within the process of executing each task, and not just as triggers for deciding when to apply an overall task or not. While I focus on just two domains here—as some effort is required to prepare domains to work with reinforcement-learning—prior work has employed the DIPL approach to dozens of domains including several math domains including other forms of long-arithmetic and algebra [70, 111], scientific procedures like stoichiometry [64], games [20], and simple language domains like Chinese character translation and English article selection [64].

2.5 Conclusion

We show that data-efficient academic task learning can be achieved by decomposed procedure learning so that sets of action-producing skills are induced via a collection of interconnected learning mechanisms. This result illustrates an instance of symbolic induction that flexibly produces general-purpose structured knowledge representations without relying upon domain-specific knowledge engineering. Our results suggest that learning mechanism decomposition is a more important consideration for data-efficiency than simple symbolic/sub-symbolic distinctions. Learning decomposition may be an essential component to building ML that readily and flexibly constructs rich operational knowledge structures bottom-up purely from natural instruction.

For the purposes of describing our AI2T authoring tool this section has achieved two aims. First we have described three of the 4 mechanisms that make up AI2T—the three mechanisms (how, where, and when) that AL and SimStudent implement—and motivated the development of a 4th: process-learning. Second, we have established the data-efficiency that this variety of approaches can offer in an interactive task learning (ITL) setting like ITS authoring. The several orders of magnitude difference between the training effort that an author would need to put into training our approach versus a deep-learning based approach, illustrates the inductive power of simulated learner systems. In chapters 5 and 6, we expand upon the simulation experiments we outline here by introducing a new implementation of when-learning and an implementation of process-learning, and evaluate these in simulation environments that replicate instruction from an ITS author. This simulation environment differs from the one reported in this section in that the agent is shown multiple solution paths and is evaluated on total completeness instead of single path correctness. In the following two chapters 3 and 4 we describe the front-end development for a prototype authoring tool we designed in 2020 and our new interface for AI2T.

Chapter 3

Prior Investigation: An Interaction Design for Machine Teaching to Develop AI Tutors

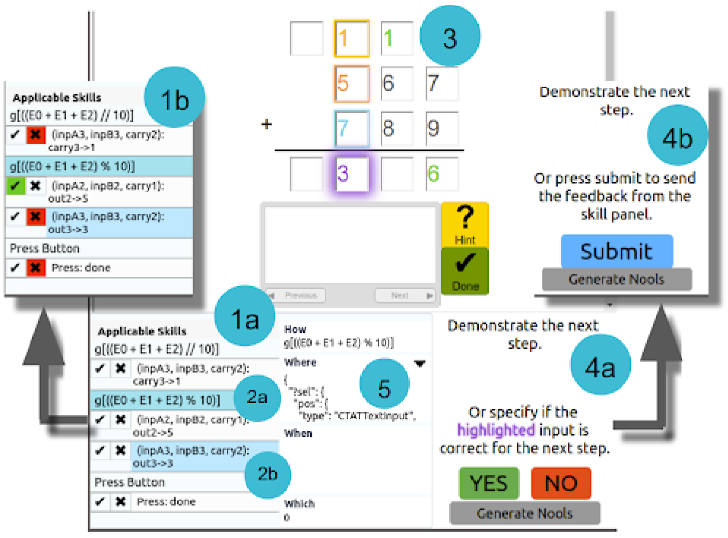
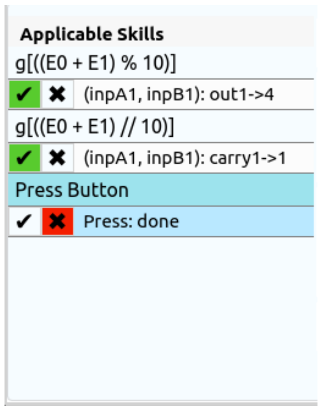

Applicable Skills
g[((E0 + E1 + E2) // 10)]
(inpA3, inpB3, carry2): carry3->1
g[((E0 + E1 + E2) % 10)]
(inpA2, inpB2, carry1): out2->5
(inpA3, inpB3, carry2): out3->3
Press Button
Press: done
1b
1a
2a
2b
3
4a
4b
5
Hint
Done
Previous
Next
How
Where
When
Which
Demonstrate the next step.
Or press submit to send the feedback from the skill panel.
Submit
Generate Nools
Or specify if the highlighted input is correct for the next step.
YES
NO

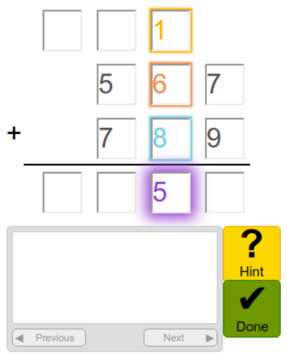
Fig. 10: Authoring interface for AL (2020). The tutor interface is embedded in the training interface (3). For each problem state users see AL's conflict set (1a), and provide feedback to the whole set at a time (1b). Users are also able to give feedback one action at a time (4a).

In our 2020 CHI study: "An Interaction Design for Machine Teaching to Develop AI Tutors" [117] we prototyped the core training interactions of a simulated learner based ITS authoring tool using AL. The objective of this study was to prototype AL as an authoring tool with actual participants. An early attempt to replicate SimStudent's Java training interface in HTML for use with AL revealed some significant limitations in the way SimStudent framed training. The design of these early interfaces for SimStudet and AL were framed heavily around human-to-human tutoring in which the author provides examples or feedback to the synthetic tutee one action at a time. While debugging our reimplementation of this general design we found that training one action at a time was limiting because it provided the user with only a very narrow view into what the agent had learned, and the scope of possible actions it thought were correct.

The aim of our redesign was to build a more efficient and robust interface for training AL agents that was explicitly designed around the objective of inducing fully complete model-tracing behavior. The main new feature of this interface was the skill window (Figure 11.a) which showed all of the agents proposed next actions at each training step, and information concerning the underlying skills that produced them (full view in Figure 10). Since each action is produced from an induced skill, every action produced by the simulated learner is actually a **skill application**: a particular way that some skill is applied on an interface elements with certain arguments. Henceforth we will refer to actions and skill applications interchangeably. The skill window allows the user to give feedback for every candidate skill application that the agent predicts to be correct for a particular problem state. The skill applications shown in the skill window are those generated by matching the patterns induced by each skill's where-learning mechanism to the current problem state. These skill applications also satisfy the skill's preconditions induced by the when-learning mechanism. In this version of the interface we did not have the user see information about the when- and where-components of each induced skill to demonstrate that authors do not need to engage in program checking. In this interface the user sees just the how-part formula for the induced skill, and the interface elements that each skill application is applied with. This information allows the user to evaluate not just whether the action is correct but whether it was produced using the correct formula and arguments. Seeing the set of skill applications in this window also provides the author a broad overview of which actions the agent currently considers correct or incorrect, helping them evaluate and correct deficiencies in model-tracing completeness directly.


Applicable Skills
g[((E0 + E1) % 10)]
✓ ✖ (inpA1, inpB1): out1->4
g[((E0 + E1) // 10)]
✓ ✖ (inpA1, inpB1): carry1->1
Press Button
✓ ✖ Press: done

(a



(b)

Fig. 11: (a) The skill window with user feedback filled in for two correct actions and one incorrect one. (b) A proposed skill application (i.e. action) is highlighted in purple, arguments are highlighted in yellow, orange, and blue.

The benefit in this interface over verifying single actions is simple: the user can give feedback to all the proposed actions because they can see all the proposed actions. Other useful features of this interface include element highlighting to show the argument interface elements employed by the proposed skill application (Figure 10 right), and the ability to demonstrate new actions at any point, instead of only when the agent has no proposals for what to do next.

To make a direct comparison with conventional authoring tools we had our participants work with both AL and CTAT example-tracing to compare their authoring efficiency. Building on previous authoring investigations with SimStudent and AL [67, 70], our 2020 CHI study was the first full comparison between a simulated learner-based and conventional ITS authoring tool, and the first to evaluate multiple participants beyond the researchers that designed the tool simulated learned based authoring tool.

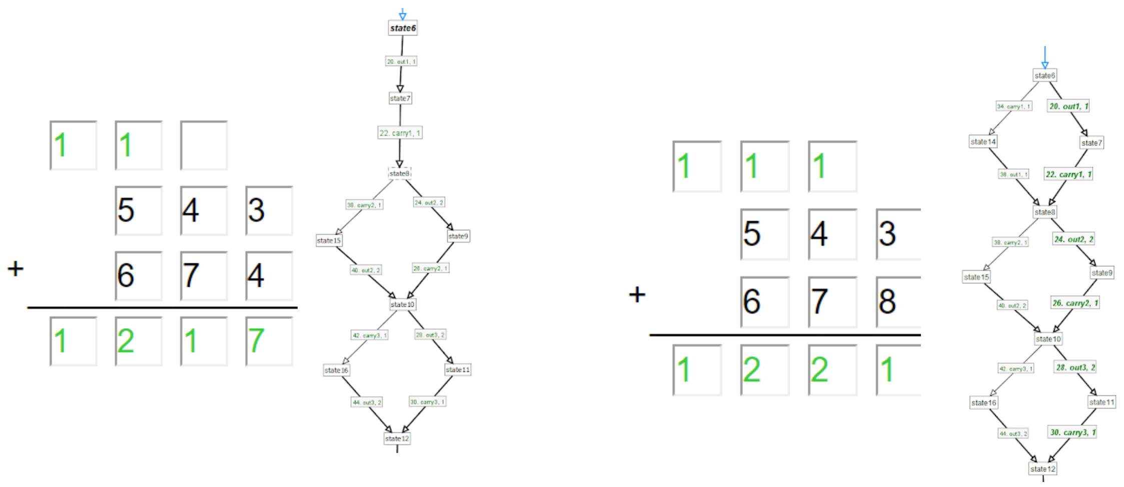


Fig. 12: Two kinds of multi-column addition problems that require two structurally different behavior graphs in CTAT example-tracing.

Our participants consisted of ten masters and PhD students studying educational technology. Eight of our participants had used CTAT example-tracing before. Participants each spent 45 minutes trying to build a three-digit multi-column addition tutoring system with CTAT and with AL. Multi-column addition was selected as the authoring domain because almost everyone remembers how to do it, yet it has a relatively complex control structure. It cannot be authored in CTAT with a single example-tracing graph, and the number of behavior graphs necessary (Figure 12) increases exponentially with the number of digits (e.g., 8 for 3x3-digits, 16 for 4x4-digits etc.). Thus, we predicted that AL's ability to induce control structures would provide a more efficient authoring experience. The tool used first by participants, CTAT or AL, was selected randomly. Four participants used CTAT first and six used AL first. The table in Figure 13 summarizes the results.

User	*First*	*AL Complete*	*AL Time*	*CTAT Complete*	*CTAT Time*
1	CTAT	30%	15min	11%	45min
2	AL	85%	55min	N/A*	N/A*
3	AL	98%	45min	13%	45min
4	CTAT	83%	30min	7%	45min
5	CTAT	92%	42min	50%	45min
6	AL	91%	25min	27%	45min
7	CTAT	92%	35min	11%	45min
8	AL	98%	41min	23%	45min
9	AL	85%	41min	13%	30min**
10	AL	99%	45min	23%	45min
Average		85%	37min	20%	43min
Median		92%	41min	13%	45min

Fig. 13: Time spent and levels of authoring completeness achieved by participants using CTAT example-tracing and AL.

For CTAT example-tracing completeness was evaluated as the proportion of the eight required behavior graphs that were fully demonstrated, variablized, and mass produced into multiple problems within 45 minutes. Partial credit was given in the CTAT condition for partially finished graphs. We measured completeness for AL slightly differently. There is not a simple way of counting the absolute completeness of AL's induced skills, since two skills with different constituent parts may be functionally equivalent. Instead after 45 minutes of training or after users self-reported that they believed they were finished training the agent, we ran the agent on a holdout set of problem states. We evaluated model-tracing completeness on these states: the proportion of states in which the agent only produced all of the correct next actions. Note that this measure is considerably harder to succeed at than training the agent to only have a high accuracy (i.e. select a correct action from its set of proposed actions at each step), or have a high step score (i.e. have a high proportion of proposed correct actions). Additionally the holdout set comprised a meaningful subset of problems needed to achieve absolute mode-tracing completeness. The holdout set consisted of every intermediate state on a variety of problems including a complete set of known edge cases.

Our main quantitative result was that users were able to exhibit about a 7-fold improvement in median authoring efficiency (i.e. 13% to 92% over 45 minutes) while using AL, over CTAT example-tracing. This main result is promising. It empirically demonstrates that for domains with complex control structures using a simulated learner like AL can be significantly faster than a more traditional authoring tool. Additionally users generally conveyed that they enjoyed using AL more than CTAT. However, the qualitative results from our observations of participants paint a broader picture of challenges that needed to be overcome to build a full-fledged simulated learner-based authoring tool. Some remaining issues include that users were unable to effectively estimate when they were finished training the agent, and struggled to make consistent plans for tutoring it through varied enough sequences of problems and solution paths to achieve 100% model-tracing completeness. These issues are symptoms both of issues in the interaction design and in the backend AI. In the following chapter, I discuss the remaining issues with this interaction design, and some potential solutions.

Chapter 4

Designing an Interface for Interactively Teaching an AI Agent: Lessons Learned

Our initial 2020 study revealed several of the challenges of building an ITS authoring tool around interactively teaching an AI agent. A good way of understanding these interaction challenges is to consider how they differ from the simple experience of solving individual problem instances. Even if authors lack the skills to program the rules of an ITS or have little understanding of what would make for a good ITS, they ought to at least know how to solve the kinds of problems that they intend for students to practice in the ITS. Thus, solving problems is a good baseline interaction around which to design a system intended for non-programmers. One benefit of authoring with an AI agent is that the agent can alleviate much of the tedium of filling in answers for every step across several problems. When the agent believes that it knows how to solve a problem step it can propose an answer that the author can check instead of needing to write solutions themselves each time. If the author comes to trust that the agent has induced the correct general ITS behavior, then they might trust the agent's induced program to tutor students for problems they never tutored the agent on. The fact that the AI agent generalizes from the author's instruction imposes some extra responsibilities on the author beyond providing simple step-by-step solutions:

1. **Demonstration Explanations**: The author needs to ensure that the agent has the correct understanding of how each of their demonstrated actions were produced. The tool should reveal the agent's explanations and support amending those explanations if they are wrong.
2. **Per-State Completeness**: The author needs to verify the correctness of all of the actions that an agent proposes for each problem state. It is not enough to simply give confirmatory feedback; respond Yes for the first correct action proposed by the agent, since for many model-tracing domains each step may permit multiple correct next actions. The interface should reinforce checking every proposed agent action, and encourage authors to demonstrate correct actions that the agent did not propose.
3. **Solution Path Navigation**: The author may need to ensure that the agent produces the correct behavior along multiple solution paths. The author is accountable for the agent's performance over a potentially large space of possibilities. Every correct action leads to a potentially unique new problem state, and the author can greatly assist the agents inductive processes if they give feedback along a large fraction of these states in at least a handful of problems. It should be easy for the author to navigate among these possibilities and know which ones have not yet been checked.

The interaction design for AI2T was developed to resolve issues that we observed among the users in our initial 2020 study [117]. For the purposes of this dissertation work, our revised interface was developed and refined over a process of several rounds of piloting, and two user

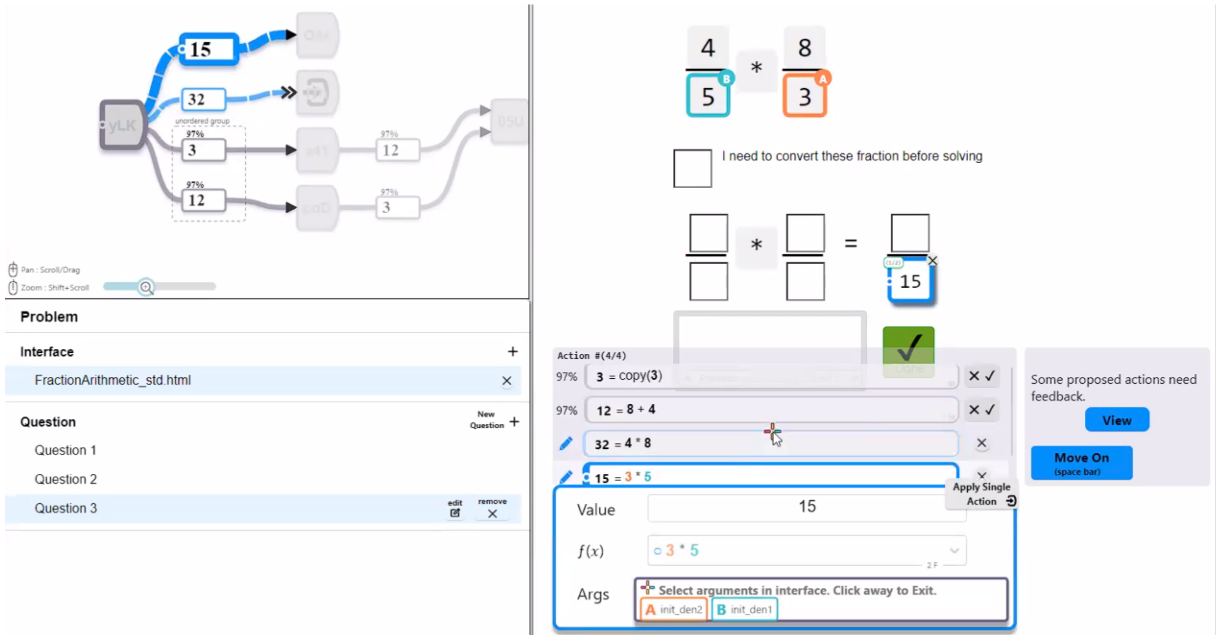
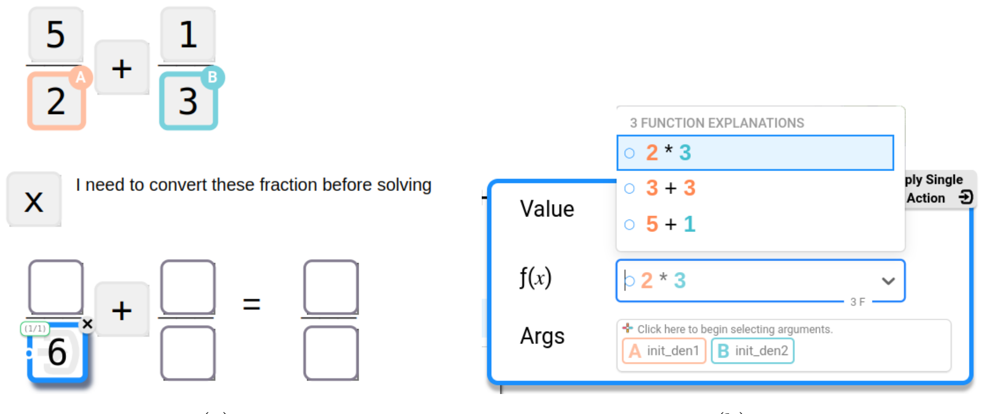


Fig. 14: Interface for AI2T (study 2 version). The user has just demonstrated two actions. They show up as blue dashed edges in the behavior graph (top-left) and are indicated by ✎ in the skill application window (middle-bottom). One of the demonstrations for multiplying the denominators of the expression 4/5 * 8/3 is selected and is previewed as a 15 in the tutor interface overlay. Demonstrations can be removed by clicking the X button in the skill application window or on the tutor interface. Two other actions were proposed by the agent as part of an unordered group (grey edges in dashed box). They both have high certainty scores of 97%, but are incorrect. The user can add correctness feedback by clicking the ✗or ✓icons on the toggler in the skill application window.

studies. We discuss these two users studies in Chapter 7. In this section we outline how our interface was refined through qualitative observations of users interactions and think-aloud over the course of these evaluations. We will refer to various stages in the development of AI2T's final interface with respect to the prototype in our 2020 study, piloting prior to user study 1, user study 1, piloting prior to user study 2, and the final interface for user study 2.

4.1 Visualizing and Assisting Demonstration Explanations

Each AI2T agent begins as a blank slate. The author teaches the agent the skills necessary to complete problems within the ITS that they are authoring by first demonstrating step-by-step solutions to problems. An AI2T agent tries to self-explain these initial demonstrations using its how-learning mechanism. One of the author's responsibilities is to ensure that the compositions of primitive functions that the agent proposes as an explanations for their demonstrations are correct. For instance, in a simple case, the author may demonstrate 6 for the converted denominator of a sum of fractions problem 5/2 + 1/3 (Fig. 15). The agent may come up with the correct explanation 2 * 3, or some incorrect explanation like 3 + 3 or 5 + 1.



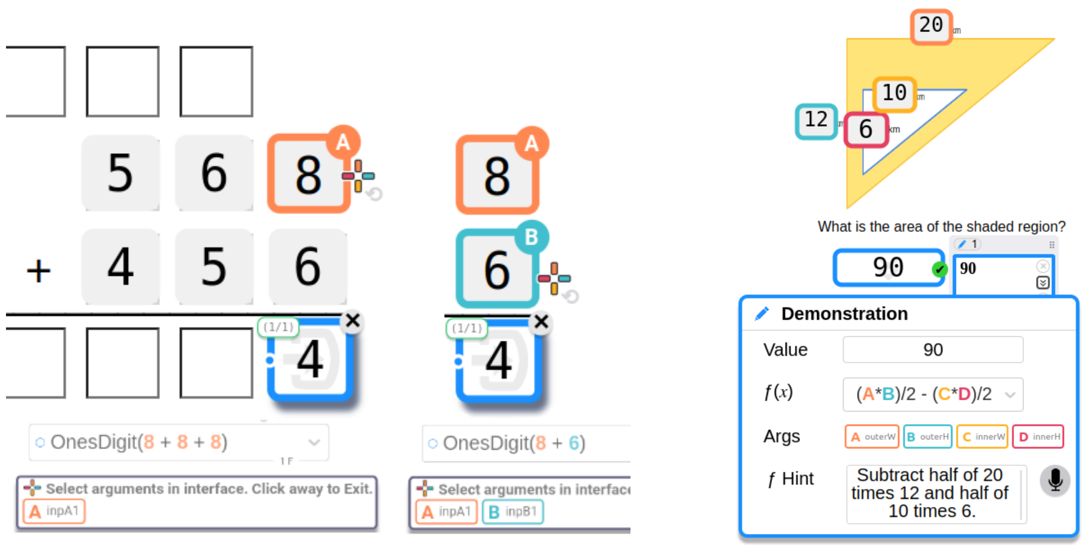
(a) (b)

Fig. 15: (a) The user has demonstrated the converted fraction 6. (b) The agent displays several possible explanations for this demonstration in a drop-down. The arguments of the correct explanation are currently highlighted in the tutor interface.

Users need to understand what composition of functions the agent is proposing, and what interface elements that formula takes as arguments. One mistake we made in our 2020 prototype interface was showing a representation for induced formulae with too much information. Under the hood the agent must account for details such as the attributes of each interface element being used and conversions between numbers and strings. For instance, to induce a skill that multiplies two numbers the agent must come up with a how-part formula like: string(float(A.value) * float(B.value)). Where float() and string() convert between numbers and text strings, and A.value retrieves the string in the *value* slot of an interface element bound to variable A.

To simplify the formula displayed to the user we drop all conversion operations and attribute de-references, reducing each formula to its simplest form (i.e. A * B), and then express each formula in terms of color coded argument values instead of variables: 2 * 3. This color-coding is mirrored in the interface by highlighting the borders of each argument interface element with corresponding colors (Fig. 15.a). This approach makes it clear what values are being using to explain the author's demonstrated action, and resolves ambiguities between repeat values. If the agent comes up with multiple explanations for the same demonstration then they appear in a drop down (Fig. 15.b). Mousing over each item in this drop down highlights the arguments for one formula option in the interface, and clicking selects a formula as the correct one.

The agent can come up with just a few explanations or sometimes thousands. In either case it is time consuming to check and select among candidate explanations, so we enable the user to directly clarify the arguments and operations of the intended formula by two additional methods. The user can clarify the arguments they used to compute a demonstrated value by simply selecting interface elements on the screen. When the user writes in a demonstration they enter an *argument selection mode* indicated by replacing their mouse with a multi-colored cross-hair that toggles selected arguments when clicked (Fig. 16.a below). The selected arguments constrain the search for an explanation. Selecting arguments very often reduces the set of possible explanations to just one correct explanation. Users also have the option of stating the formula directly in natural language or in mathematical notation (Fig. 16.b below). These natural language explanations are mutually disambiguated with the demonstration using the method we describe in [119], to further narrow down the candidate explanations.



OnesDigit(8 + 8 + 8)

Select arguments in interface. Click away to Exit.

A inpA1

OnesDigit(8 + 6)

Select arguments in interface

A inpA1 B inpB1

Value: 90

f(x): (A*B)/2 - (C*D)/2

f Hint: Subtract half of 20 times 12 and half of 10 times 6.

(a) (b)

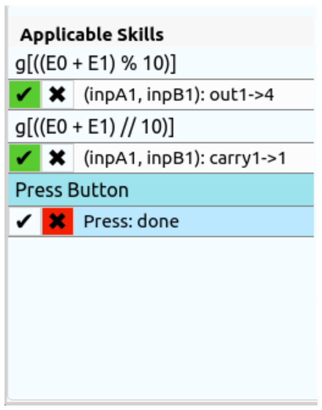
Fig. 16: (a) The user selects the interface elements that they used as arguments to compute the demonstrated 4. They first select the 8 and then the 6, and the agent's explanation for how the 4 was generated is immediately updated. (b) The user describes the formula that produced their demonstration 90 in natural language, and this is interpreted by the agent to guide the search for a formula that explains it.

4.2 Supporting Per-State Completeness

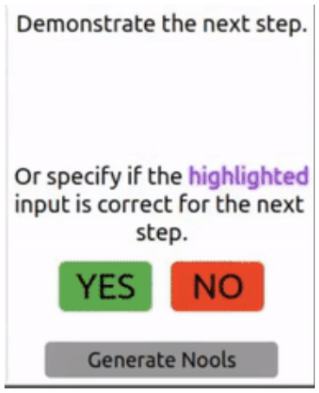
A common issue in our 2020 study was that some users fell into a pattern of validating the first correct action suggested by the agent, but failed to give feedback to any other actions that it proposed for the same state. For getting an agent to solve problems with high accuracy this is not a bad strategy. However, for authoring purposes we want the agent to be able to track all possible correct ways of solving problems. In this case, we aim to achieve 100% model-tracing completeness which means the agent should suggests all correct next actions and no incorrect actions for every reachable problem state. When users do not give feedback to all proposed actions the agent is deprived of important feedback toward achieving model-tracing completeness.

It is natural that users verify single actions and move on since this is similar to solving problems or tutoring someone to solve them—to get a problem right you typically only need to solve it one way—and so we hypothesized that making the multiplicity of proposed action more visible to the user would help reinforce more comprehensive checking behavior.

4.2.1 Prototype Design


Applicable Skills
g[((E0 + E1) % 10)]
✓ ✖ (inpA1, inpB1): out1->4
g[((E0 + E1) // 10)]
✓ ✖ (inpA1, inpB1): carry1->1
Press Button
✓ ✖ Press: done

(a)


Demonstrate the next step.

Or specify if the highlighted input is correct for the next step.

(b)

Fig. 17: (a) The skill window and (b) Yes/No buttons from the design used in our 2020 CHI study.

In our 2020 design the set of proposed actions was made visible to the user in a skill application window (Fig. 17.a) in the bottom-left corner of the screen, however many users ignored this box and gave feedback by answering the Yes and No prompts (Fig. 17.b) in the bottom-right side of the screen. One poor design choice in this early version was that the Yes button provided both positive feedback to the selected action and applied that action as well, moving the user on to a new state, and effectively preventing additional feedback. This design choice was motivated by trying to make teaching the agent similar to the experience of tutoring someone working on paper. However, this choice made it difficult for users to give comprehensive feedback to all of the agent's proposed actions if they flipped through the skill window and ignored the Yes/No buttons entirely.

4.2.2 Skill Application Pop-ups (a removed feature)

We initially hypothesized that we could improve the interface by replacing the skill application window with pop-up windows that would appear in an overlay on top of the tutor interface. We believed that these pop-ups would give the user an immediate sense of how many actions the agent was proposing and where they were being applied. Similar to the skill application window this pop-up display showed the action value of each skill application and information about their skills' how-part formulae. In this display each skill application is shown in a card, and when multiple skill applications apply on the same interface element multiple cards can be scrolled through. A hypothesized advantage of this pop-up approach was that it would allow users to quickly visualize the context of each skill application by displaying their information immediately next to their corresponding interface elements.

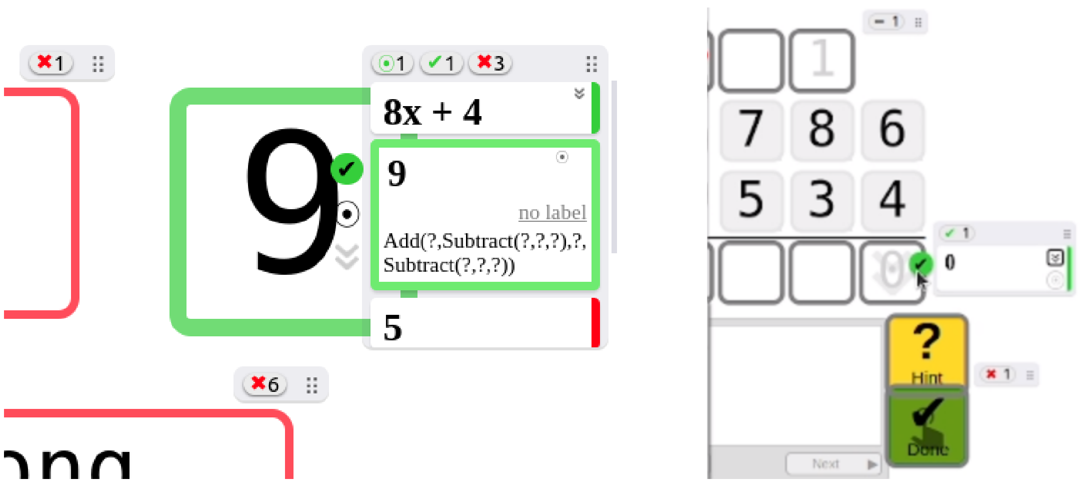
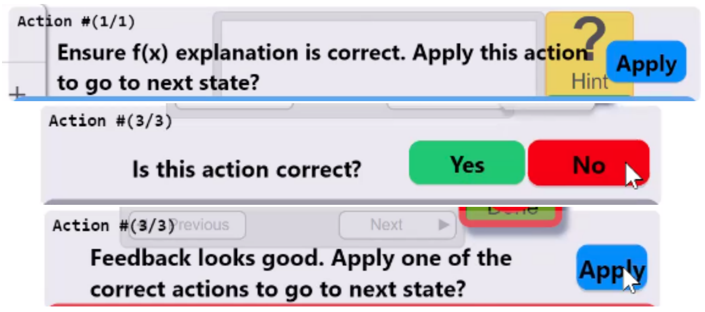


Fig. 18: Two designs for skill application popups—a feature removed in the final design. The left image shows multiple skill applications scrolled through in the same popup group. In both images, the group for the selected skill application is expanded.

However, what we found in our first round of piloting was that the purpose of the pop-ups was not immediately clear to users, and at times they would cover other important information like other interface elements and other pop-up windows. We ended up dropping this feature and keeping the information for each skill application in a single window at the bottom of the screen that simply displayed information about just the selected skill application.

Another feature which we removed in our pilot design was that users were able to see multiple skill applications displayed in the interface at once as semi-translucent values. Clicking on these semi-translucent values was one ways of selecting particular skill applications in this version, in addition to moving through them by clicking the Yes and No prompts and directly clicking on edges in the behavior graph visualization which we discuss in a later section. One issue with having multiple semi-translucent actions visible at once is that users can confuse these proposed actions with committed values and misunderstand which step in the problem they are on. Prior to study 1 we streamlined the interface to only show a single skill application at a time—only the selected skill application.

4.2.3 Prompting for Complete Feedback


Action #(1/1)
Ensure f(x) explanation is correct. Apply this action to go to next state?
Apply

Action #(3/3)
Is this action correct?
Yes
No

Action #(3/3)
Feedback looks good. Apply one of the correct actions to go to next state?
Apply

Fig. 19: Several prompts that users are given in the AI2T interface.

One approach we took to guiding our users through using our interface was simply to prompt them for whatever information was needed at each step. We added prompts asking users to demonstrate new actions when appropriate, select arguments after giving explanations, and most importantly we added prompts that would ask the user whether each proposed action was correct. The interface asks: "Is this action correct?". To which they can respond by clicking the Yes or No buttons to give positive and negative feedback. After pressing Yes or No the next skill application is selected for feedback, and this repeats until all of the skill applications have been given feedback. After clicking Yes/No for the last skill application that needs feedback, one of the correct skill applications is selected and the prompt is displayed: "Feedback looks good. Apply this action to go to the next state?". The user can click an Apply button to apply the newly selected correct action and move on to the next step of the problem.

4.2.4 Visualizing the Set of Proposed Skill Applications

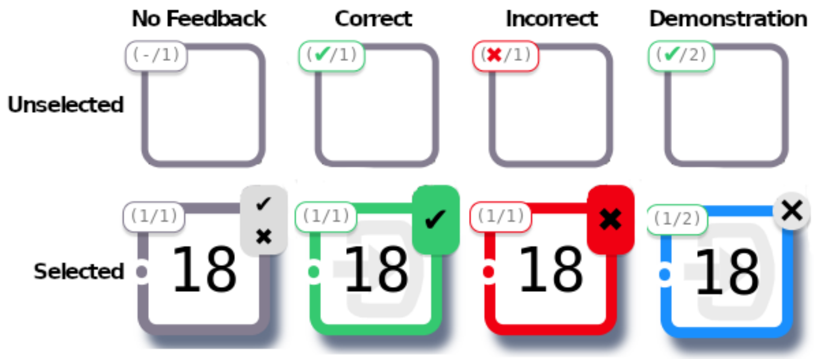


Fig. 20: Interface element overlays have an action count indicator when one or more proposed actions apply to that element. (Top) none of the actions are selected. (Bottom) one of those actions has been selected. The indicator and border are colored grey (left) when none of the skill applications have been given feedback, green (middle-left) when positive feedback has been given, red (middle-right) when negative feedback has been given, and blue when the selected skill application is a demonstration (right).

Users are made aware the set of actions proposed by the agent via several methods. One way is via the edges of the behavior graph visualization which we describe in the next section. Another way is by small indicators superimposed on the tutor interface showing the number of proposed actions for each interface element (Fig. 20). Clicking on an interface element that has this indicator selects one the first of its skill applications. These indicators are helpful for getting a quick sense of what interface elements the agent has proposed actions on, and which actions have been given feedback. The indicator's border is colored grey with a dash when no feedback has been given, green with a ✓ if any positive feedback has been given, and red with an ✗ if all of the provided feedback is negative.

When an action is selected this indicator shows the index number of the selected action. Also when an action is selected either a small toggler is shown next to its interface element to allow the user to quickly toggle positive and negative feedback, or if the action was

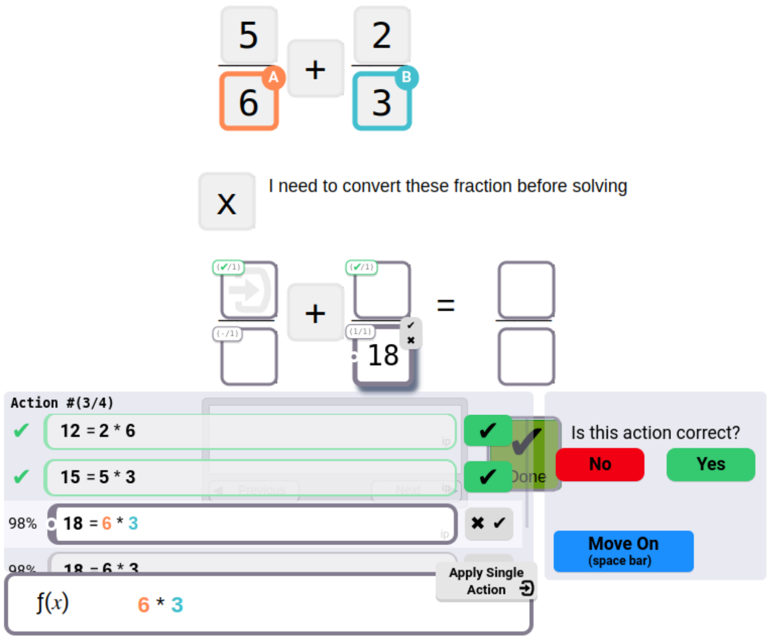


Fig. 21: The interaction design used in study 2. The skill application window shows each of the actions proposed by the agent. Currently, action 3 of 4 is selected.

produced by a demonstration then a small x button is shown to allow the user to remove that demonstration.

In the interface version we deployed in study 2 we reintroduced a revised version of the skill application window as an additional way for the user to see and select among the set of proposed skill applications. In user study 2 we found that users interacted with this window heavily (which was not the case for the 2020 prototype design), and that they tended to follow a simple interaction loop of checking each item in the list and assigning it feedback. Unlike our prototype design, each click of the Yes/No buttons queued positive or negative feedback and selected the next action for consideration. In this design users could change each item's correctness feedback directly with toggle buttons inside the skill application window. This design differs from our 2020 prototype, in that the skill application window is centrally located, more easily readable, and functionally aligned with the Yes/No prompts. Pressing Yes stages positive feedback and moves the user through the list of skill applications instead of applying the selected action.

Our observations of users in piloting and in user study 1 ultimately disabused us of the idea that displaying comprehensive information about multiple skill applications super-imposed on the interface would help users attend to giving feedback to multiple proposed skill applications. These observations led us to remove the semi-translucent candidate actions and pop-ups which we had in our pilot design prior to study 1, and we reintroduce the skill application window prior to study 2. We found that these design choices were beneficial simply because with less information on the screen users were less likely to misinterpret the information that was being displayed. In addition limiting the information super-imposed on the tutor interface meant that the interface was visually better aligned with what users expected to see if they were just solving problems.

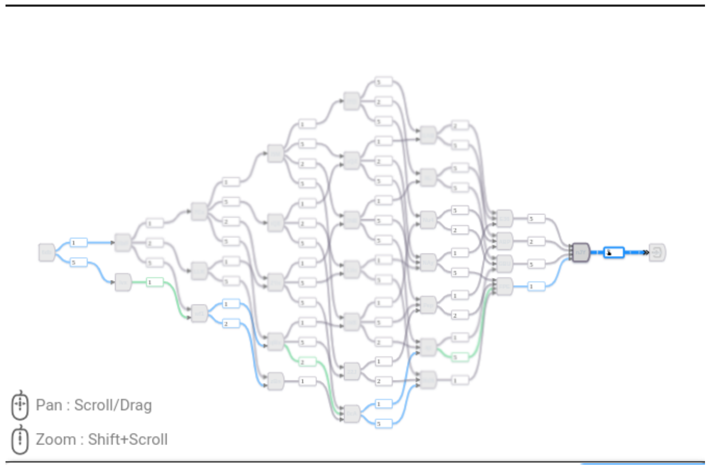
4.3 Supporting Solution Path Navigation with Behavior Graph Generation

Authors need to be able to see which solution paths they have given feedback on and which ones are untouched. In our 2020 prototype design authors needed to restart problems from the beginning and take a different path if they wanted to show an alternative solution path. However, when the space of allowed solution paths is very large, or not strictly sequential—including paths that can diverge and re-converge in a graph—it can become difficult for the author to keep track of what paths they have and have yet to tutor the agent on.

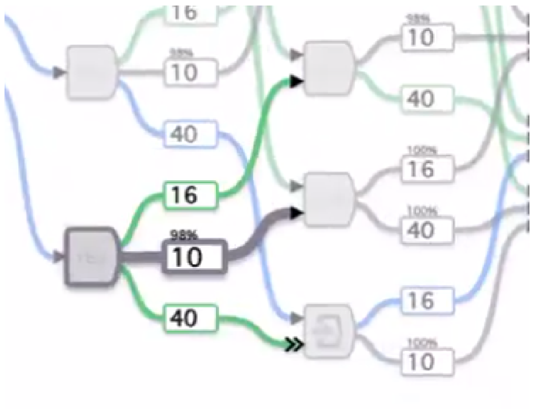
To support users in visualizing the solution paths that they have covered, and allow them to easily navigate between states, AI2T implements a feature similar to CTAT's example-tracing behavior graphs. In CTAT users demonstrate actions one at a time to build up a state-machine-like structure of states and actions. In AI2T behavior graphs are automatically generated for each problem by repeatedly querying the agent for next actions starting with the start state of the problem, following each proposed action into a new state, and terminating when an action would press the done button.

In general since an AI2T agent learns rule-like knowledge structures instead of strict state-machines, each generated behavior graph is simply a visualization of the AI2T agent's induced program applied to a particular problem—not a direct visualization of its internal knowledge structure. The behavior graph shows all of the diverging action sequences the agent believes are correct, and allows the author to navigate between these possibilities by clicking on particular nodes (for states) and edges (for actions).

To assist the author in tracking which paths have been tutored and which paths have not, each edge of the behavior graph is labelled with the value of its action and given a color indicating its feedback state. Blue edges indicate actions that the author demonstrated to the agent, grey edges indicate proposed actions that have not yet been given feedback, and green and red edges indicates actions that have been given positive and negative feedback



(a)



(b)

Fig. 22: (a) A generated behavior graph and (b) a behavior graph zoomed into the current state with the 2nd of 3 proposed actions selected. Demonstrations are blue, actions with positive feedback are green, and grey edges are proposed actions that still need feedback.

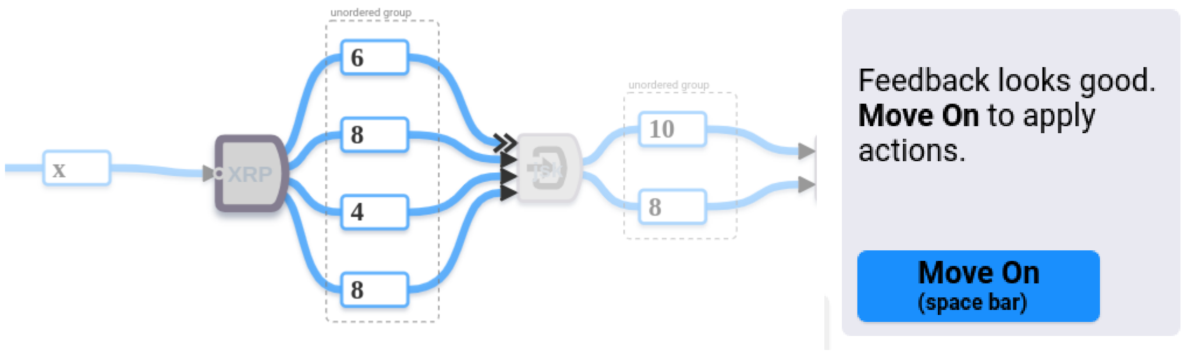
respectively. We utilize a pallet which uses a particularly light green, dark red, and slightly blue-tinged grey, to reduce mix-ups common to various forms of color-blindness. We verified this pallet using an application that simulates protanopia, deuteranopia, and tritanopia, and checked with some colorblind labmates.

Users can pan through AI2T's generated behavior graph by scrolling or dragging, and zoom in and out with shift+scroll. Entering a new state for any reason including clicking on a node, selecting an edge in a different state, or applying an action in the main interface, automatically animates the graph view so that it is centered on the new state and its downstream actions. This feature keeps the graph aligned with the current problem state. In piloting and both user studies users did not generally have much trouble with the mechanics of navigating the through the behavior graph.

4.4 Simplifying Graphs and Navigation with Induced Unordered Groups

One issue we did encounter was that some users in study 1 struggled with connecting the graph visualization to the notion of actions that can be applied in alternative orders. In a very simple case two actions that can be applied in either order make a pattern of two edges (indicating two choices for the first action) diverging into two separate states each with one edge (the other action) converging again into a common state. Some users had trouble understanding why they effectively had to show the same actions multiple times across three different states to make this pattern. In general there are $n!$ permutations for any n number of actions that can be performed in any order, so the combinatorics of checking graphs of unordered actions becomes unwieldy very quickly. For instance, in fraction arithmetic four actions are needed to convert a pair of fractions and fill in their converted numerators and denominators, making for the very complex graph shown in Figure 22.a.

In chapter 6 we describe a learning mechanism called process-learning that induces hierarchical task networks from sequences of actions and includes support for inducing unordered groups. In the version of AI2T we deploy in study 2, we utilize this learning mechanism to



Feedback looks good. **Move On** to apply actions.

(a) (b)

Fig. 23: (a) A behavior graph with two visible unordered groups. (b) The Move On button moves the author to a new state by applying a single action or multiple actions in an unordered group.

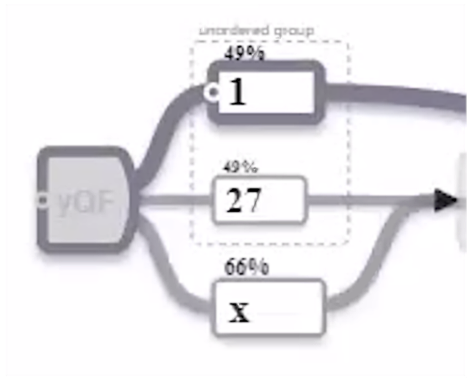
recognize groups of unordered actions. This allowed us to simplify our behavior graph representation considerably. For reasons that are made clear in chapter 6, without this process-learning mechanism the agent really needs to be tutored on states along a large fraction of possible orders to allow actions to be applied in any order without permitting other incorrect actions as well. Training the agent to 100% complete behavior can be tedious in this case.

With process-learning however, the agent really only needs to experience demonstrations of actions performed in one order and then again in the reverse order to recognize that it should generalize those actions to be part of an unordered group. When this kind of generalization occurs we can directly display the presence of an unordered group in the graph visualization (Fig. 23.a). This makes the graph easier to understand, but also discourages the user from unnecessarily navigating through the combinatorial branching structures necessary when there is no process-learning to induce unordered groups. Whenever the user encounters an already induced unordered group as they are navigating through the problem, moving on to the next state applies all of the actions in the unordered group by default. Along with this change, instead of giving authors the option to go to the next state with an Apply button associated with the selected action, users instead press a button labelled “Move On” that either applies all actions in an unordered group, or just the single next correct action.

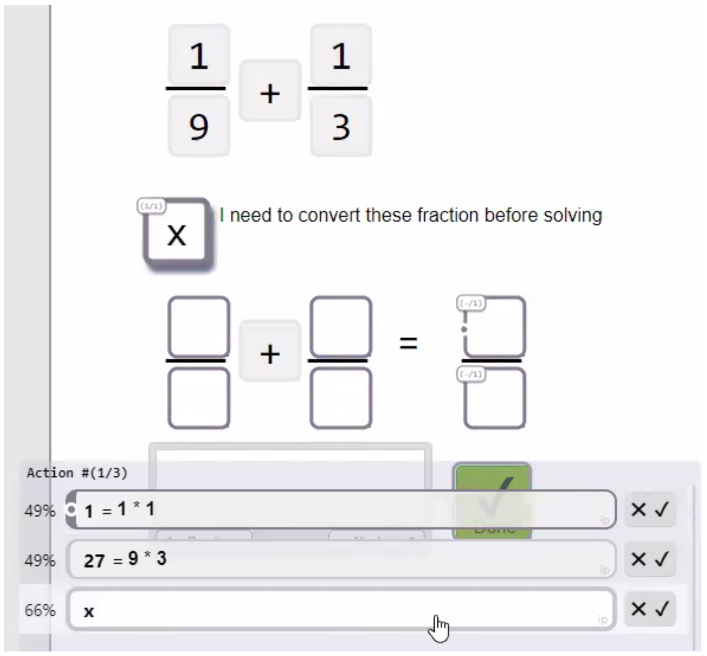
With these changes, we make it so that users can demonstrate unordered groups explicitly by demonstrating multiple actions at once in the same state. By default multiple actions are interpreted as a demonstration of an unordered group which the agent automatically experiences as a sequence of demonstrations plus a second sequence in the reverse order. In principle, multiple valid actions in the same state can also indicate that two actions are mutually exclusive instead of unordered. This was not a distinction that was necessary to make in the domains we tested AI2T on, so we leave interactions for making this distinction to future work.

4.5 Visualizing Action Certainty

Finally within our revised skill application window, and on each edge of the behavior graph visualization, we added a continuous certainty score ranging between -100% and 100%. These scores indicate how sure the agent is that each proposed skill application is correct or incorrect. Negative values indicate that the agent is mostly certain that an action is incorrect and positive values indicate varying degrees of certainty that the proposed action is correct. As we describe in chapter 5 these values come from our *instance certainty* measure which we will show is a fairly reliable indicator of prediction certainty that is theoretically and empirically a strong reflection of the agent’s actual learning progress—something we show is not true of many alternative methods of estimating prediction probability. Roughly speaking if the agent proposes only actions with 100% certainty scores for a particular problem state, then this is a fairly strong indication to the author that the agent will exhibit 100% model-tracing complete behavior in similar situations. Mixtures of lower certainty scores are a fairly strong indication that the user should continue to train the agent on more problems.



(a)



(b)

Fig. 24: Three actions proposed for the initial state of 1/9+1/3. Certainty scores are shown over edges in the behavior graph (a) and in the skill application window (b). The first proposed action (49% certainty) would fill the numerator with 1=1*1. The first action is selected (indicated by a small white circle), but the third action is hovered over and previewed in the interface. This is the correct next action and has a higher certainty of 66%.

Chapter 5

STAND: Improving Precondition Learning by Embracing Inductive Ambiguity

Participants in our initial authoring study only needed to train AL agents on a few problems before the agent could correctly produce all solution paths on most of the holdout problems. Yet, none of these participants succeeded at training agents that exhibited 100% model-tracing complete tutoring systems behavior [117]. In other words they did not succeed at training agents that would reliably only permit correct actions along any valid solution path. In our own testing we had verified that it was possible for the agent to achieve 100% model-tracing completeness if it was trained on a selection of fewer than 20 problems that exhibited important edge cases. However, none of our participants succeeded at training agents to this level of completeness.

We can look at this deficiency in two ways: (1) The agents were arguably not data-efficient enough to learn from users' limited feedback, and (2) the users were arguably not sufficiently optimal teachers; they did not know how to select good problems that would be helpful to the agent's inductive learning processes, and they could not self assess when they had trained the AI on a sufficient number of problems. In this section, I address both of these problems with a new machine learning method called STAND that is more data-efficient than the decision-trees used for when-learning in our initial study. STAND is able to self-reflect on its learning to estimate the certainty of its predictions—an estimate that authors can use as a heuristic to assess the agents total learning progress.

Toward addressing these issues, especially the issue of data-inefficiency, it is important to note that typically only the when-learning mechanism is at fault when it comes to late-stage agent mistakes. The structures within an AL-agent's skills for producing actions, the how- and where-parts, converge quickly to their final states from just a handful of examples. But the when-learning mechanism, which is responsible for learning the preconditions for applying each skill, requires considerably more training to converge to correct generalizations. The when-learning mechanism faces the difficult problem of determining which features within a problem state are sufficient indicators that a particular candidate application of a skill—with a particular selection element and set of arguments—should be applied. When-learning must learn which conditions to impose from just the limited set of positive and negative examples that users are able to provide interactively. From these small training datasets it is not difficult to construct conditions that separate correct and incorrect skill applications. However, producing a set of conditions that also succeeds at selecting correct skill applications in all states of all unseen problems is extremely challenging.

5.1 Overcoming the Limits of Supervised Learning by Embracing Ambiguity

Traditional supervised machine learning frames the problem of learning generalizations that perform well beyond a training set as a matter of mitigating overfitting and underfitting.

In a small-data setting the properties of the distribution from which the data was sampled cannot be estimated precisely and thus overfitting becomes especially problematic. Overfitting is typically mitigated by methods of regularization that constrain how specialized or complex models are allowed to become when fit to training data. Regularization methods often sacrificed some training data performance to maximize performance on unseen data.

This framing of supervised learning where a theoretically optimal, yet fundamentally imperfect predictor is fit to noisy data works well when data is numerous and when there is some stochasticity in features or sample labels. However, in an interactive task learning (ITL) setting [46], data is small because it is generated interactively from a single user, and the aim is to enable the user to automate intended behaviors through various forms of instruction. In this context the user may reasonably expect that they can produce a program that reliably works in every situation. In principle this should be possible since in many cases we can expect that the features available within the task are noiseless, and that the program that the user intends to produce has consistent well-defined behavior (like tutoring systems).

The best practices of traditional supervised-learning do not translate well to ITL tasks. For instance, it is rarely feasible to collect enough well balanced data that one could reliably model variations in a user's generated data as samples of a common distribution, without permitting major biases. For instance, in our authoring tool, training data is generated from user's feedback on the agent's proposed actions, so a typical training sequence will accumulate far more positive examples than negative examples as the agent improves in performance. We have seen that users typically only accidentally produce important edge cases, and rarely produce them as a result of employing good teaching strategies. Thus, important edge cases are rarely covered during training, if at all. Many data-driven approaches that rely heavily on the relative frequencies at which patterns occur in data would perform poorly on this small, poorly balanced, and sparse data. For instance, data-hungry deep-learning methods are likely useless in this context.

Additionally many of the tricks for reducing the overfitting of symbolic learning mechanisms, like subsampling and pruning for tree classifiers, are irrelevant by their very principle. After all if our goal is to find conditions that are 100% accurate at separating positive and negative skill applications on arbitrary problem steps of non-stochastic domains, then the training set must have 100% accuracy as well. No sacrifice in performance on the training set can be made in an effort to optimally generalize beyond it since we are always aiming for, and expect to achieve, perfect performance. Of course this expectation assumes that authors catch all of their mistakes, which is not something that can be guaranteed, but can certainly be supported with good interaction design choices, like affordances for reviewing past training interactions.

In a small-data setting fitting a classifier that has perfect performance on the training set is often so easy that any generated solution will be almost arbitrary. There are typically more perfect solutions than are feasibly generatable or even countable, and we can at best employ classifiers with inductive biases that select among the more parsimonious ones. In when-learning finding a set of conditions that achieves 100% accuracy in all unseen cases is like finding a secret special needle in a pile of needles—we would not know if we happened to select the correct one even if it was right in front of us, and any attempt to guess would just be an arbitrary choice.

The key to solving this problem is to abandon the idea that we can rely upon statistical learning at all, and embrace that among the classifiers that could be learned from limited data, the choice of the right one is an ambiguous one. Each user interaction provides evidence of a program that the user intends to demonstrate to the AI agent. This intended program remains ambiguous until the user provides sufficient evidence to disambiguate the intended program from other possibilities. To support this perspective we can at best try to map out all of the programs that the user may be intending to demonstrate and cut out large swaths of possibilities as new evidence proves possibilities to be impossible. Approaching learning in this way allows us to do a great deal more with limited data, both because we are no longer arbitrarily guessing at solutions from limited evidence and because knowing all of the possibilities means we can support users in making decisions that can maximally reduce the ambiguity of selecting the right generalization.

The notion of learning by updating possible spaces of classifiers is not a new one. The theory of version spaces [75,76] dates back to the early days of machine learning, and describes a general theory for efficiently representing spaces of all generalizations consistent with a set of examples. For a particular representation language, a version space bounds the set of all consistent examples with two boundary sets. Version spaces' boundary sets can enclose a space of consistent generalizations far larger than can be feasibly enumerated, and provides a means of updating the space to cut out large sets of possible generalizations as they prove to be inconsistent with new training examples. Version spaces require a definition of relative generality between generalization hypotheses. For instance, in the generalization language of logical statements, a statement consisting of one predicate $X_5 = True$, is more general than a conjunction of two predicates $AND(X_5 == True, X_3 == False)$. The two boundary sets consist of a maximally general set G, and a maximally specific set S. The set of all hypotheses contained in the version space is the set of hypotheses in S or more general than S, but no more general than any hypothesis in G.

The seminal work on version spaces [75] describes them as "generalization as search". Although "search" is a somewhat misleading analogy. Typically searching or planning is a process of looking through individual possibilities in pursuit of a known target, like finding a needle in a haystack. Version spaces do something far more powerful. They cut out vast sets of possibilities as new evidence proves them to be impossible. They systematically cut out possibilities in pursuit of the particular good needle in the needle stack, throwing out large sets of generalizations which are inconsistent with new examples. A version space may narrow down to a single possible generalization when its general set G and specific set S converge to a single generalization. However in practice, especially when data is limited, convergence to a single generalization never occurs. This is not a weakness of version spaces, so much as a realistic treatment of the ambiguity of inducing correct generalizations from limited data.

Unfortunately, version spaces are intractable to learn under the typical cases that when-learning is employed. Specifically, in the 3-mechanism learning employed by AL, the role of when-learning is quite broad and almost always requires learning preconditions that are in disjunctive normal form. For instance, a disjunctive normal logical statement is one that disjoins two or more conjunctions with an OR. For instance:

$$OR(AND(X_5 == True, X_3 == False), AND(X_1 == "1", X_2 == True)) \tag{1}$$

The most common version space implementation is the version space over conjunctive logical statements. However, it is well known that extending version spaces to disjunctive normal logical statements is computationally intractable [31].

To overcome this issue, this section introduces a method that I call STAND, that succeeds at building an approximate version space over this known intractable representation language. Unlike typical algorithms for learning version spaces, STAND's approximate versions space learning does not fail catastrophically when it encounters noisy or mislabelled data; it does not suffer from version space collapse. More importantly, as I will demonstrate in the following sections, STAND is much more useful than a single classifier or even an ensemble of classifiers, both because it tends to achieve higher performance from less data, and because it can produce very reliable estimates of its learning progress that are useful for users in an ITL setting.

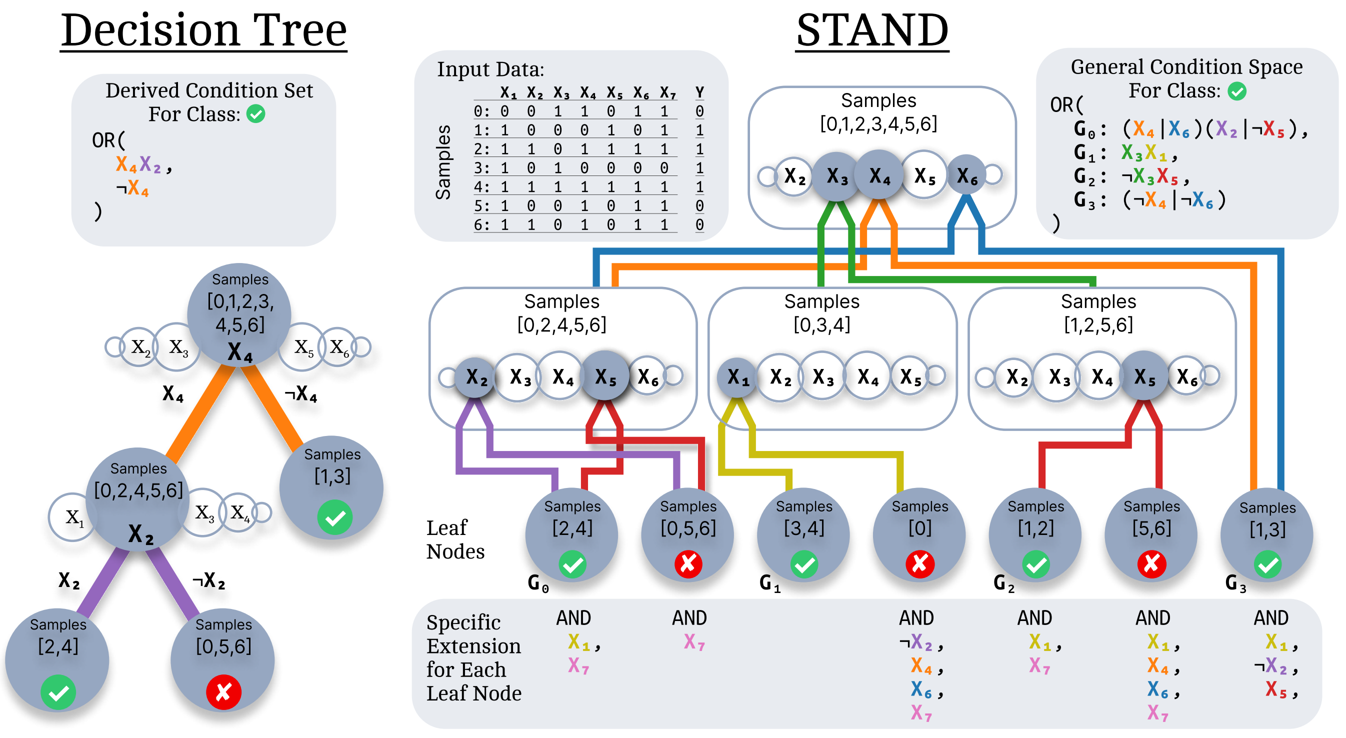
Symbolic classification methods that learn a single set of conditions can only accept or reject a new unlabeled example, but STAND can learn an approximate version space of possible condition sets that produce competing predictions about an example's correctness. Much like an ensemble this allows STAND to estimate the certainty of its predictions based on a set of competing hypotheses. While many statistical machine-learning methods are able to make continuous probabilistic predictions about the class labels of unseen examples, they are often too data-inefficient to be helpful in an ITL setting. STAND by contrast requires remarkably little data to accurately estimate the certainty of its predictions. Additionally STAND's estimates of prediction certainty have a structurally meaningful counterfactual interpretation that differs from typical statistical estimates of class probability. If an example is accepted by a part of STAND's approximate version space but rejected by another, then one of the two parts will be eliminated when the label is revealed. This allows STAND to quantify how much of its approximate version space will change as a result of receiving feedback from a user. STAND can essentially predict which training examples will help it learn, and which will not.

5.2 STAND: Building A Complete Space of Classifiers for the Cost of One

In ecology a stand is a contiguous region of trees that share similar characteristics. An ecological stand is a habitat that is typically mutually beneficial to the trees that comprise it. Analogous to its namesake concept, STAND is a method for learning a compact collection of classifiers embedded in a shared data-structure. The compact representation learned by STAND holds every classifier that would be generated by a randomized greedy learning process. In our explanation we'll use decision trees as an example for this underlying classifier but the same method can be applied to other greedy strategies as well, including sequential covering methods common in inductive logic programming [99].

To efficiently produce every classifier that would be produced by a greedy learning process, STAND expands every decision whenever an arbitrary choice would be made between nearly equally good options. For instance, when applied over decision trees, STAND is structurally similar to an option-tree [42]: a variation of decision trees where each node splits every feature with the highest split criterion reduction simultaneously, instead of splitting on just one (often randomly selected) best feature. Like an option-tree, instead of only expanding one feature that optimally splits data at each node, STAND expands every split that would decrease the impurity criterion nearly as well as the best split $\Delta C_{imp}(X_i) >= \alpha \Delta C_{imp}(X_*)$; where the parameter $\alpha \in (0, 1]$ varies the rejection rate for splits relative to the best split. Choosing $\alpha = 1.0$ to only accept splits with utility equal to the best split often works well for small non-noisy datasets. Like an option-tree, each node in STAND has $2n$ edges each leading to child nodes, where n is the number of best splits for that node. By contrast, normal decision trees typically have strictly 2 child nodes per non-terminal node.

While regular option-trees are unwieldy or intractable to compute without imposing constraints, such as limits on node depth or the number of expansions at each node, STAND can efficiently generate a complete compressed option-tree-like structure by caching nodes by the set of subsets of samples that they select. Since the set of expanded splits at each node depends entirely on the training samples selected by that node, we can route all edges that select the same subset of samples to the same shared node. Reusing nodes in this manner allows STAND to learn a complete space of possible decision trees, often in only a little more time than it takes to learn a single decision tree. This trick also makes it easy to support partial incremental learning over streams of examples. Highly ranked splits tend to remain highly ranked when new examples arrive, so when new examples are filtered into the tree only those nodes with changes in their set of best splits need to be refit.



	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Y
0:	0	0	1	1	0	1	1	0
1:	1	0	0	0	1	0	1	1
2:	1	1	0	1	1	1	1	1
3:	1	0	1	0	0	0	0	1
4:	1	1	1	1	1	1	1	1
5:	1	0	0	1	0	1	1	0
6:	1	1	0	1	0	1	1	0

Fig. 25: An example of a Decision Tree and STAND fit to the same input data. In STAND multiple splits (filled grey circles) are expanded per node. STAND builds a general condition space (top-right) that is bounded below by a specific extension (bottom).

Figure 25 shows an example of a decision tree and STAND fit on the same data. Each sample 0-6 in the training data has seven binary features $X_1, ..., X_7$. In the decision tree X_4 is selected randomly for the root node from among the best features for splitting the data. $\neg X_4$ (i.e. $X_4 = 0$) selects a pure subset of two positives samples and X_4 (i.e. $X_4 = 1$) selects an impure subset that is then split further by X_2 into leaves with purely positive or negatively samples. By contrast, STAND splits X_4 at the root, but also X_3 and X_6 as well. STAND's sample caching trick makes it so that the 6 edges formed by these 3 splits only lead to 4 nodes (two of which are reused by two edges) instead of 6 nodes (one per-edge) like in a normal option-tree. The 4 nodes downstream of the root are reached by following edges that select sample sets [0,2,4,5,6], [0,3,4], [1,2,5,6], and [1,3] respectively. The last one is a leaf because it only selects positive samples, while the others are still impure, and are further split into pure leaves. Note that unlike a typical decision tree, in STAND a single sample can filter into multiple leaves. For instance, the 3rd leaf [3,4] and last leaf [1,3] in Figure 25 both contain sample 3.

From a conventional decision tree, one can derive a disjunctive normal logical statement that only selects training samples of a particular class. If all leaves are pure, which can be expected for non-stochastic data, then edges along paths from the root of a decision tree to its positive leaves form conjunctions of literals that each only select positive training examples. For instance, in Figure 25 the decision tree's derived statement for positive samples is $OR(X_4X_2, \neg X_4)$. In STAND, multiple edges can lead into the same node, meaning there are multiple paths of literal sequences (i.e. conjuncts) that select the same sub-samples. In the top-right of Figure 25 we represent these options in parentheses separated by the | symbol. For instance, the choice of literals $(X_4|X_6)$ corresponds to the two edges leading into the left-most node that selects [0,2,4,5,6]. STAND's caching trick helps account for alternative conjuncts that select the same subsets of training samples, and thus it is not just an optimization, but also a means of compressing sets of alternative generalizations. For instance, the left-most leaf in Figure 25 is reached by any conjunct in the Cartesian product of options represented by $\mathcal{G}_0$:

$$\mathcal{G}_0 = (X_4|X_6)(X_2|\neg X_5) = X_4X_2 \mid X_6X_2 \mid X_4\neg X_5 \mid X_6\neg X_5 \tag{2}$$

It is important to keep in mind that STAND's node caching trick is only helpful in limited circumstances. For instance, it would likely not be effective in many data-driven prediction settings where classifiers are learned over large datasets with noisy features or labels. In these cases there would be very little consistency between the subsets of samples selected by different splits leading to limited potential for node reuse. Large training sets would also make subset hashing and comparison computationally expensive. However, this approach thrives in non-stochastic small-data environments, including many ITL applications. This includes precondition induction like when-learning where the target generalization is a set of hard requirements and not a probabilistic predictor. Preconditions after all must be expressible in a representation language of non-stochastic predicate-like features. It is admissible for stochastic features to be present in the dataset so long as they are not necessary for discriminating the target preconditions.

5.2.1 A Space of Classifiers for the Cost of One

Any AI approach used in an interactive setting should execute quickly to avoid subjecting users to considerable lag. In our authoring tool several skills' when-learning mechanisms may have their `.fit()` sub-routines called after the users submits their demonstrations and feedback by pressing the "move on" button. Since the when-learning mechanism has an instance for each skill its `.predict()` sub-routine may be called many hundreds of times to recalculate rollouts for building a behavior graph visualisation. Thus the duration of `.fit()` and `.predict()` should be kept to a minimum. Ideally re-fitting and then calculating a new rollout should take less than half a second.

In this context STAND provides a great deal of benefit with almost no efficiency drawback. On average fitting and predicting with STAND takes only marginally more time than fitting and predicting with a single decision tree. Averaging over a long 100 problem training sequence refitting STAND takes about 5.30 ms whereas a single decision tree takes about 4.42 ms. On average for predicting the correctness of a new action STAND takes 0.35 ms and a decision tree takes 0.27ms. With either of these when-learning mechanisms the total time it takes to re-fit each skill and update the behavior graph after the user presses "move on" rarely exceeds 300ms. Replacing these with an ensemble method like random forests or XG Boost leads to update times of a second or more—long enough to be noticeable to users or even disruptive their training process.

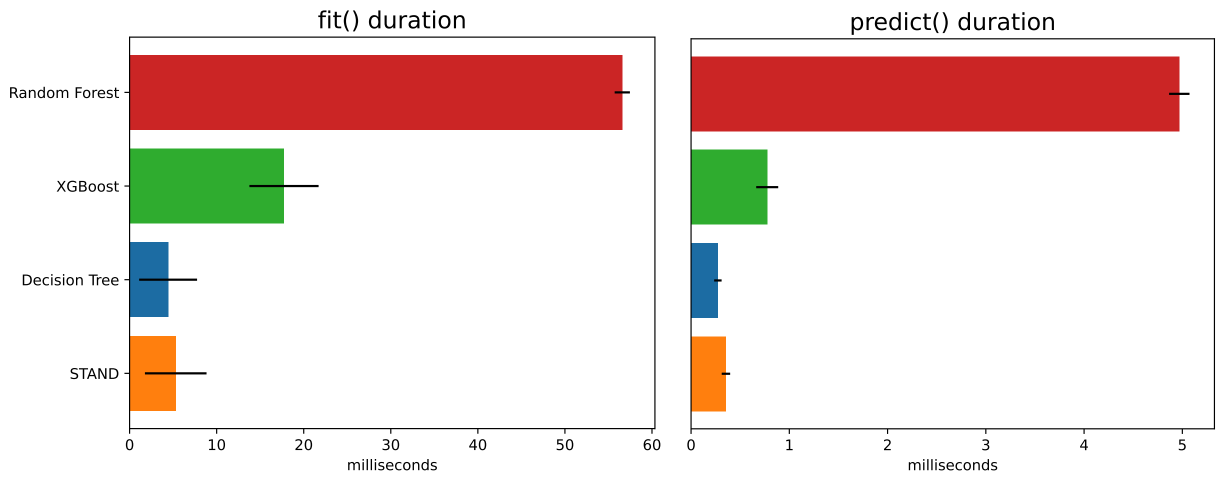


Fig. 26: Average `.fit()` and `.predict()` durations for Random Forest with 100 estimators, XG Boost, Decision Tree and STAND used as when-learning mechanisms for agents trained on 100 multi-column addition problems. Timed on an Ubuntu 22 laptop with an 11th generation Intel i7-1165G7 processor and 16 GB of RAM.

STAND's caching trick is a big part of why it is nearly as efficient as a single decision tree. Typically decision trees calculate the utility of every possible way of splitting each node's sub-samples. STAND simply hashes the indices of the two child subsets generated by each split and routes equivalent child subsets to the same node. When data is small and mostly noiseless STAND's total number of nodes is not more than a small factor greater than a normal decision tree's. Very little extra work is performed per-node since each node only needs to calculate its split utilities once.

5.2.2 STAND as an Approximate Version Space

STAND's compressed option-tree-like structure of cached nodes is akin to the general set G of a version space. Since a true G over disjunctive normal logical statements is intractable to compute, it goes without saying that STAND's structure is only an approximation—a strict subset of a true general set G. Nonetheless, this structure shares important properties of a true version space's general set G:

1. The members of G are not any more specific than they need to be
2. The members of G cover all of the most-general possibilities (by some definition).

The first property depends on the choice of greedy classifier underlying STAND, but is certainly true of decision trees and of sequential covering. These processes construct conditions one split or literal at a time and do not grow generalizations any more than necessary to separate training examples by label. The second property follows from the fact that STAND expands all options that could be randomly constructed by repeatedly rerunning one of these greedy construction processes. This is of course a much looser definition than the typical theoretical notion of a G set which includes all of the most-general consistent generalizations expressible within a representation language. Nonetheless, STAND's approximate G is useful in practice because it spans a well-defined space of *good* choices within a disjunctive representation language.

STAND's general set G is encoded in a distributed manner over its leaf nodes. Examples filter into one or more leaves because they satisfy one or more of the alternative literal statements associated with each of the ancestor nodes along some path leading from the root. The paths leading to each leaf node i form a set of alternative conjunctive statements $\mathcal{G}_i$ that select all of the training samples associated with that leaf node. As in the example above, the set of all conjunctions in $\mathcal{G}_i$ is the Cartesian product of each alternative. For instance, in Figure 25, leaf $i = 0$ forms a space of alternative conjunctive statements $\mathcal{G}_0$:

$$\mathcal{G}_0 = (X_4|X_6)(X_2|\neg X_5) = X_4X_2 \mid X_6X_2 \mid X_4\neg X_5 \mid X_6\neg X_5 \tag{3}$$

where $|$ represents an alternative choice. Let $\mathcal{L}_{cov}$ be the set of all minimal subsets of leaves that cover the positive training examples. For each covering set $L_{cov} \in \mathcal{L}_{cov}$ a portion of the general set $G_{L_{cov}}$ is formed by disjoining every combination of alternative conjunctions for all $i \in L_{cov}$. The set of disjunctive statements generated by L_{cov} is then:

$$G_{L_{cov}} = \{g_1 \vee \ldots \vee g_n | (g_1, \ldots, g_n) \in \mathcal{G}_0 \times \ldots \times \mathcal{G}_n\} \tag{4}$$

Where $\vee$ indicates disjunction and $\times$ indicates the cartesian product of all $\mathcal{G}_i$ associated with L_{cov}. The total set of most general disjunctive statements covered by STAND's effective G set is then simply:

$$G = \bigcup_{L_{cov} \in \mathcal{L}_{cov}} G_{L_{cov}} \tag{5}$$

STAND's specific set S is formed by extending the generalizations associated with each leaf node i so that they select any additional features that are common between the samples selected by each leaf. The *specific extension* s_i for each leaf i is a conjunction of literals selecting all common features in the leaf's subset of samples that do not overlap with any of the literals comprising $\mathcal{G}_i$. Each pair of $\mathcal{G}_i$ and s_i define a mini-version space of conjunctive statements within the whole. Any conjunction in $\mathcal{G}_i$ can be extended by adding literals from s_i to form a new conjunction that selects all of the training samples that filter into leaf i.

The classic candidate-elimination approach for learning conjunctive version-spaces [75] can fail catastrophically when it encounters examples that are logically inconsistent with its enclosed generalizations. This is called version space collapse, and it makes traditional version space approaches brittle to training on noisy data. STAND does not suffer from this issue. It is as robust to noise as whatever greedy algorithm it is applied to. For instance, when applied to decision trees, as we have describe above, a mislabelled example can prevent STAND from converging to perfect performance, but it will not cause STAND to break entirely. If a user mislabels the correctness of an example in this case, then STAND will very likely introduce new disjunctions into its tree structure. In the best case the mislabelled example may filter into its own leaf isolated from the rest, which would produce minimal changes to model behavior. Or in the worst case it may filter into a leaf that captures several other properly labelled examples, which could alter the literals that select those examples, or cause the specific extension s_i for the leaf to over-generalize.

5.3 Estimating Model Ambiguity and Instance Certainty

Interpreting STAND as an approximate version space allows us to quantify various notions of ambiguity and certainty. We define *model ambiguity* as the size of the approximate version space. It captures how ambiguous the target generalization remains given all of the generalizations that are equally consistent with the current training data. *Instance certainty* captures how unambiguous the label prediction of an example is given all of the generalisations that capture the example. Low instance certainty indicates high disagreement between the predictions of alternative generalizations in a STAND model's version space.

Model ambiguity is loosely analogous to the inverse of the posterior distribution $P(\theta|X)$ of a Bayesian statistical model and instance certainty is loosely analogous to the posterior predictive distribution $P(y|x, X, Y)$. However, these probabilistic concepts are imperfect analogs since STAND is not nearly as sensitive to the distributional properties of data as a typical parameterized statistical model. Each element in STAND's space of generalizations is equally consistent with the training data, and possess no notion of relative likelihood between them, nor do generalizations have parameters derived from the frequencies of patterns in the data. Consequently, STAND does not need to be trained on large sets of independent and identically distributed examples. STAND requires diverse examples to learn well, but not necessarily numerous or well-distributed ones.

5.3.1 Model Ambiguity

In practice, the true values of model ambiguity and instance certainty are prohibitively expensive to compute since they require generating G from all minimal spanning sets $\mathcal{L}_{cov}$. This runs the risk of combinatorial explosion. For practical purposes, it is more helpful to use heuristics that change over the course of training in a manner reflective of changes in model ambiguity and instance certainty. As a simplification, we can calculate a heuristic A for the total model ambiguity by summing a heuristic A_i representing the size of each leaf's independent mini version space. The true total size of each leaf i's mini-version-space is on the order of:

$$size_i = (\prod_{g_{ji} \in \mathcal{G}_i} |g_{ji}|)(1 + |s_i|)! \tag{6}$$

Estimating this size precisely is not particularly useful, since the magnitude of $size_i$ is highly sensitive to the size of the specific extension s_i, leading to a number that can vary wildly between leaves. It is far more useful to simply sum the number of literals in $\mathcal{G}_i$ and s_i to make for an easy to compute, numerically stable heuristic that reflects immediate changes to the boundary sets G and S. We'll define A_i and A to be:

$$A_i = (\sum_{g_{ji} \in \mathcal{G}_i} |g_{ji}|) + |s_i| \tag{7}$$

$$A = \sum_i A_i \tag{8}$$

In the above, $|g_{ji}|$ represents the number of alternative choices of literals in ancestor node j of leaf i. Because of node caching each node may have multiple parents, and thus finding all $|g_{ji}|$ for leaf i involves traversing several branching possibilities back to the root.

5.3.2 Instance Certainty

To compute instance certainty we must consider how examples may be filtered into several leaves that disagree in label prediction. We must consider three varieties of disagreement that may occur when an example is compared to STAND's approximate version-space.

First, an unlabelled example can simultaneously filter into multiple positive leaves and multiple negative leaves. This disagreement is similar to the prediction disagreement between classifiers within an ensemble.

Second, within each of the leaves that accepts the example, if the true correctness label agrees with the leaf's label, the mini-version-space formed by $\mathcal{G}_i$ and s_i of leaf i may reduce in size to accommodate the new example. Subsets of the literals in each $g_{ji} \in \mathcal{G}_i$ and literals of the specific extension s_i may be inconsistent with the new example, and thus will be dropped. This will reduce the total heuristic size A_i of leaf i's mini-version-space.

Third, if a user's stated correctness label disagrees with the leaf's label then refitting the option-tree structure will result in extensions or rearrangements of nodes in order to achieve purity in all leaves. These sorts of changes are largely unpredictable without speculatively refitting with alternative example labels, and are likely to effectively increase the size of A, so it is better to ignore them. Part of the curse of trying to approximate a version-space over disjunctive concepts is that the space must grow as it entertains new disjunctions. Consequently A is unlikely a useful heuristic of learning progress on its own. However, instance certainty is still useful if we only focus on how new examples may reduce the size of each known A_i.

Given these considerations, we can calculate the certainty that a new example x belongs to the positive class or negative class independently. For a set of positive leaves $L_+(x)$ that accept an unlabelled example x we can find the average disagreement of their mini-version-spaces. If A'_i is the value of A_i after shrinking from accommodating the new example then the proportion of literals bounding i's mini-version-space that accept the example is A'_i/A_i. Averaging over each leaf in $L_+(x)$ we get:

$$IC(x)_+ = \frac{1}{|L_+(x)|} \sum_{i \in L_+(x)} (A'_i/A_i) = \frac{1}{|L_+(x)|} \sum_{i \in L_+(x)} \frac{(\sum_{g'_{ji} \in \mathcal{G}'_i} |g'_{ji}|) + |s'_i|}{(\sum_{g_{ji} \in \mathcal{G}_i} |g_{ji}|) + |s_i|} \tag{9}$$

If we also compute $IC(x)_-$ from the negative leaves that accept x we can define $IC(x)$ as a value ranging from -100% to 100% that can be easily placed within an interface and interpreted by a user.

$$IC(x) = \begin{cases} IC(x)_+ & \text{if } IC(x)_+ \geq IC(x)_- \\ IC(x)_- & \text{otherwise} \end{cases} \tag{10}$$

Compared to other measures of model probability, instance certainty is particularly informative to a user in an ITL setting since it captures prediction certainty, example-by-example learning utility, and indirectly indicates learning completion when $IC(x) = 100\%$. Many statistical models can produce continuous probability predictions from the contributions of either a single classifier's internal weights or the competing outcomes of multiple models in an ensemble. By contrast, $IC(x)$ accounts for all of the predictions of alternative consistent generalizations within STAND's version space. Relative to STAND, most methods for estimating predictions probabilities do not rely on a particularly complete account of alternative predictors. Bayesian estimations of posterior predictive distributions $P(y|x, X, Y)$ by integration over a posterior $P(\theta|X, Y)$ are a notable exception, although in practice these measures are hard to compute, and lack some of $IC(x)$'s desirable properties as a helpful signal to users.

For instance, when $IC(x) = 100\%$ on an unseen example x that means that STAND's version space encloses no positively labelled generalizations that reject x, meaning that nothing can be learned by verifying that x is correct. $IC(x) = 100\%$ may be a false positive if the users' examples are very similar to one another or if they have not yet provided many negative examples.

5.4 Evaluating Learning Performance and Certainty Estimation Quality

To assess STAND's capabilities as a when-learning mechanism in an authoring setting, we evaluate its performance against several alternative when-learning mechanisms in two tutoring system domains. We evaluate each model on its overall predictive performance per-problem, and performance on measures of prediction quality.

5.4.1 Methods

We train AI2T agents with several competing when-learning mechanisms using an automated training system that mimics the demonstrations and feedback that an ideal user would provide while authoring. We apply this special *authoring* training approach in the two domains introduced in chapter 2: multicolumn addition and fraction arithmetic.

In this setup each agent receives ideal on-demand demonstrations and correctness feedback. At each state all proposed actions are given correctness feedback. If an action is missing then it is demonstrated to the agent with annotations that make the underlying reason for the action unambiguous. Each demo is annotated with the formula for producing the action's value, and the arguments used. This replicates the behavior of an ideal user of first selecting the correct formula that explains each demo (among the several abduced possibilities) before moving on. These annotations enable how- and where-learning to produce error-less generalizations almost immediately, meaning almost all errors can be attributed to when-learning. No annotations are provided to assist when-learning besides the correctness labels of each action. Just like an ideal author, the training system trains the agent on all alternative solution paths for each problem.

We compare several classifiers with STAND:

1. **Decision Tree**: A decision tree using gini impurity [13] as the impurity criterion. We use STAND's implementation, expanding just one random split at each decision point.
2. **Random Forest**: Scikit-learn implementation of random forest ensemble [12] of 100 decision trees. Random forests use bagging [11] to independently train several decision trees on subsets of the data.
3. **XG Boost**: An ensemble method that trains multiple decision trees one at a time. This method uses gradient-based sampling to re-weight the samples for subsequent trees [17].

These tree-based methods are chosen because they excel at learning from small datasets of structured data. In all models no limits are set on tree depth or leaf size since for these condition-learning tasks the agents are provided with features that are sufficient for separating correct and incorrect candidate skill applications perfectly. Since condition learning is noiseless the trees will already tend to not become more complex than the ideal solution, and limiting their depth could only prevent the ideal solution from being discovered.

The two ensemble methods are included for comparison with STAND's comprehensive verison-space-based approach, and to compare the utility of their prediction probabilities with instance certainty $IC(x)$. Each model is re-trained on 40 repetitions on a sequence of 100 randomly generated problems. However, in the active-learning conditions described below, agents self-select their next training problems. After each completed training problem each agent is evaluated on a holdout set of 100 problems. All agents are evaluated on the same holdout set for each domain.

In the active learning conditions agents assign a certainty score to each problem in the random problem pool. For each candidate next problem the agent rolls out every sequence of actions that they predict to be correct using the same act_rollout() sub-procedure used to dynamically generate behavior graphs. For each problem each action produced along this rollout is given a certainty value—$IC(x)$ for STAND and prediction probability for the other models. The certainty score for a problem is the minimum certainty value assigned to all actions produced in this rollout that are predicted to be correct. The minimum is used instead of the average so that problem selection is not biased by number of problem steps. After each training problem the problem in the pool with the lowest certainty value is selected as the next problem. Then the selected problem and the highest certainty 50% of problems are replaced with randomly regenerated problems. This resampling ensures that the pool tends to contain a high proportion of uncertain problems.

5.4.2 Evaluating Prediction Performance and Stability

STAND's raw predictive performance compared to alternative methods is only one element of evaluating its usefulness in an interactive task learning setting. An ideal when-learning mechanism should rapidly converge to a state of 100% holdout set accuracy and alter its prediction behavior conservatively—changing predictions only when new examples provide evidence to suggest the change. Methods like decision trees that randomly pick among alternatives choices when fitting can produce an issue where predictions on unlabeled examples change dramatically between training events. In an authoring setting, users may find that the actions suggested at each problem state change spontaneously to include new incorrect actions or exclude correct one. Thus, in addition to raw model performance, it is important to evaluate the stability of when-learning's predictions. With these considerations in mind we evaluate STAND's prediction behavior to the comparison models on per-problem completeness, per-problem errors by type, and error re-occurrence rate.

Per-Problem Completeness Below we report each model's completeness performance on a holdout set of 100 problems after each training problem. We define completeness here as "model-tracing completeness" [117]: the average number of problem states along all correct solution paths where the agent would only suggest every correct action, and no incorrect actions. Completeness reflects the proportion of problem states in the holdout set where when-learning produces 100% correct predictions.

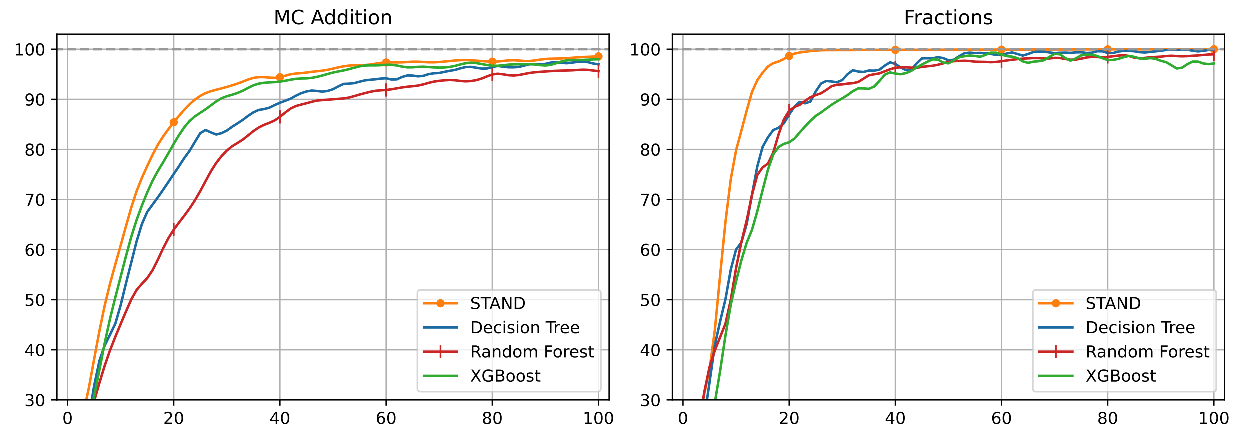


Fig. 27: Average holdout completeness by problem.

Table 2: Average Holdout Completeness at Problem N, and Number of 100% Complete Repetitions at problem 100.

	MC Addition				Fractions			
	N=20	N=50	N=100	100% Reps	N=20	N=50	N=100	100% Reps
STAND	**85.45%**	**96.10%**	**98.62%**	**19/40**	**98.72%**	**99.91%**	**99.99%**	38/40
Decision Tree	75.75%	91.86%	96.97%	10/40	88.15%	97.24%	99.88%	38/40
Random Forest	64.16%	90.02%	95.53%	0/40	88.13%	97.44%	98.97%	11/40
XG Boost	81.20%	95.40%	98.01%	3/40	81.12%	96.20%	97.34%	27/40

In both domains STAND's average completeness is higher than the competing models throughout the training sequence. This implies that STAND has better overall model performance, and data-efficiency since it can achieve greater levels of completeness with fewer training problems. In 19 of 40 MC addition repetitions STAND achieved 100% completeness after training on a sequence of 100 problems compared to 10 of 40 repetitions for decision trees. In fractions 38 of 40 repetition achieved 100% completeness with STAND and decision trees. The relative performance of the decision tree, random forest, and XG Boost varies between domains. Notably the random forest was the worst in multicolumn addition, likely because its bagging approach of sampling subsets of the data had the effect of dropping important edge cases, which are particularly important in this domain.

The relatively poor performance of random forests' highlights that models that tend to work well in a data-driven machine learning setting do not necessarily work well in an interactive task learning setting. When fitting an imperfect predictor to a large noisy datasets fitting on sub-samples can create helpful diversity in an ensemble. In our case however fitting on sub-samples most likely discarded important edge cases that could have provided valuable evidence about the true preconditions, thus many of the tree instances in the random forest most likely underfit the data.

5.4.3 Per-problem Errors by Type

A when-learning mechanism can make errors of omission where a correct action is considered incorrect (i.e. a false negative), and errors of commission where an incorrect action is predicted to be correct (i.e. a false positive). Errors of commission are easy for users to fix and hard to miss: the user must simply mark a proposed action as incorrect. Errors of omission are harder to notice, and take slightly longer to fix: the user must demonstrate a correct action missing from the set of proposed actions.

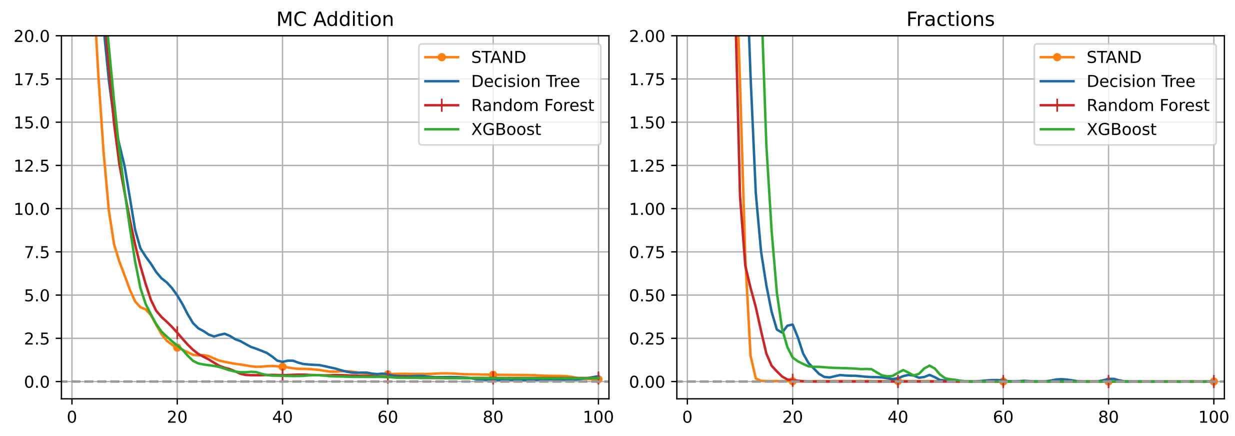


Fig. 28: Average omission errors by problem.

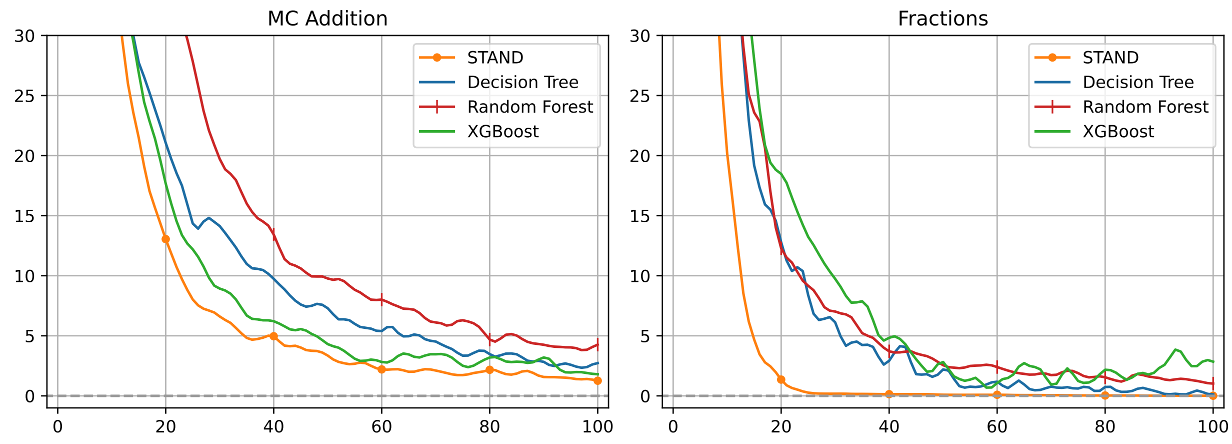


Fig. 29: Average commission errors by problem.

Our results show that STAND tends to make strictly fewer errors of omission and commission than the other models, except that in multi-column addition it does not make fewer errors of omission than the two ensemble methods.

Error Re-occurrence Rate Error re-occurrence rate is defined as the proportion of correct predictions prior to a training event that transition into being incorrect after the training event. A low error re-occurrence rate implies that errors strictly decrease with more training examples and do not spontaneously reappear in problem states that have already been given feedback. Error Re-occurrence Rates can also be broken down by type. The omission re-occurrence rate is the proportion of true positives that transition into false negatives, and the commission re-occurrence rate is the proportion of true negatives that transition into being false positives.

Table 3: Total Error Re-occurrence Rates

	MC Addition			Fractions		
	Total	Omission	Commission	Total	Omission	Commission
STAND	0.53%	0.43%	0.86%	**0.05%**	0.04%	0.08%
Decision Tree	**0.38%**	**0.00%**	1.48%	0.06%	**0.00%**	0.23%
Random Forest	1.28%	1.37%	0.98%	0.41%	0.52%	**0.07%**
XG Boost	0.74%	0.73%	**0.81%**	0.81%	0.96%	0.34%

Our results indicate that overall STAND does not succeed at reducing error re-occurrence rates over single decision trees, although it is slightly better in fractions. However since STAND makes fewer errors overall this result is not necessarily an indication that STAND is less desirable on this front. STAND generally produces fewer error re-occurrence events than the two ensemble methods, and has a low ratio of omission re-occurrence events to commission re-occurrence events. A low rate of omission re-occurrence events is desirable since this implies that authors are less likely to find that proposed actions spontaneously disappear.

5.4.4 Evaluating Instance Certainty

In order to be informative to users, it is important that STAND's estimates of certainty reflect actual learning progress and eventual completeness. Instance certainty $IC(x)$ should reflect STAND's learning trajectory: it should be low when receiving the correctness label of an example would cause STAND to learn a lot, and high when STAND achieves a state of complete mastery. Additionally, increases in certainty estimates should reflect changes in performance on unseen problems.

We report several measures of desirable properties along these lines: precision at high certainties, productive monotonicity, and normalized active learning utility. We compare STAND with only those models that can produce prediction probabilities (i.e. the two ensembles), and where applicable each comparison models' prediction probabilities are negated when an action is predicted to be incorrect. This maps their values into to $IC(x)$'s range of $[-100\%, 100\%]$.

Precision at High Certainties If a when-learning classifier predicts that an action is correct with a high certainty of 90%-100% then there should be a very low probability that the user must inform the agent that the proposed action is actually incorrect.

Table 4: Total Precision at High Certainties

	MC Addition		Fractions	
	$\geq 90\%$	$= 100\%$	$\geq 90\%$	$= 100\%$
STAND	93.19%	99.81%	95.70%	100.00%
Random Forest	97.15%	94.79%	95.19%	93.72%
XG Boost	98.35%	100%	99.39%	100.0%

Our simulations show that XG Boost has the highest precision at high certainties. For predictions of 100% STAND is nearly as precise as XG Boost in multicolumn addition and equally 100% precise in fractions. For predictions of $\geq$ 90% STAND's precision is closer to 90%, which is arguably a desirable property—as it indicates some alignment of instance certainty $IC(x)$ with actual ground-truth precision.

Productive Monotonicity Productive monotonicity is defined as the proportion of changes in certainty estimates for actions in a holdout set that move toward 100% when the action is correct and -100% when the action is incorrect. High productive monotonicity reflects the degree to which changes in certainty estimates mirror actual learning gains.

Table 5: Total Productive Monotonicity

	MC Addition	Fractions
STAND	**56.74%**	**78.54%**
Random Forest	51.26%	50.61%
XG Boost	52.58%	50.90%

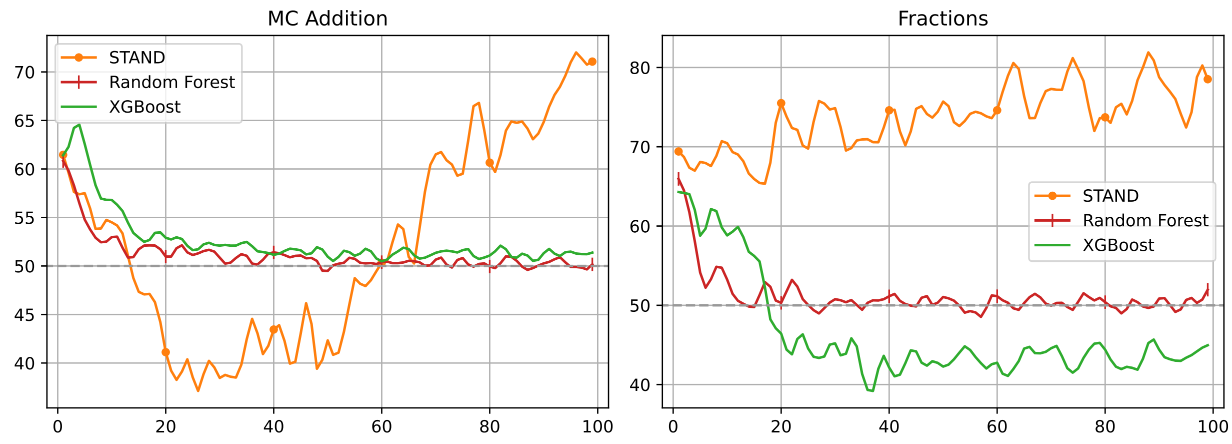


Fig. 30: Productive Monotonicity By Problem

Our results show that STAND's instance certainty $IC(x)$ has considerably higher overall productive monotonicity than the two ensemble methods' prediction probabilities. The random forest and XG Boost's prediction probabilities align with actual changes in holdout performance not much more than 50% of the time—they are not much better than chance.

In multi-column addition STAND's productive monotonicity is $< 50\%$ for the first 60 training problems and $> 50\%$ thereafter. This may be the case in this domain because in the early stages of training STAND's version-space is still growing from permitting new disjunctions. This growth introduces new leaves that reduce $IC(x)$ but increase prediction performance. In the later stages of training the version-spaces enclosed by each leaf tend to gradually shrink as possible generalizations are eliminated. Fractions may not show a similar pattern because purely conjunctive preconditions tend to suffice in this domain, and so STAND's effective version space is covered by fewer leaves, and tends to shrink monotonically.

Normalized Active Learning Utility If the agent proposes an action with low certainty then this should indicate high expected learning gains when the user verifies the action's correctness. Consequently, we can use instance certainty $IC(x)$ and estimates of prediction probability as heuristics in an active-learning scenario where the agent self-selects next training problems from a pool of random problems. We define normalized active-learning utility as the average difference in completeness between agents that can and cannot self-select problems divided by the total completeness deficit of the agents that cannot:

$$\hat{U}_{active} = (C_{active} - C_{normal})/(1.0 - C_{normal}) \tag{11}$$

Active learning utility is a measure of the expected proportion of agent errors that can been eliminated by allowing the agent to self-select the problems it is instructed on. The denominator normalizes the completeness benefit of active learning by the total completeness deficit of the basline model in order to controls for differences in baseline model performance.

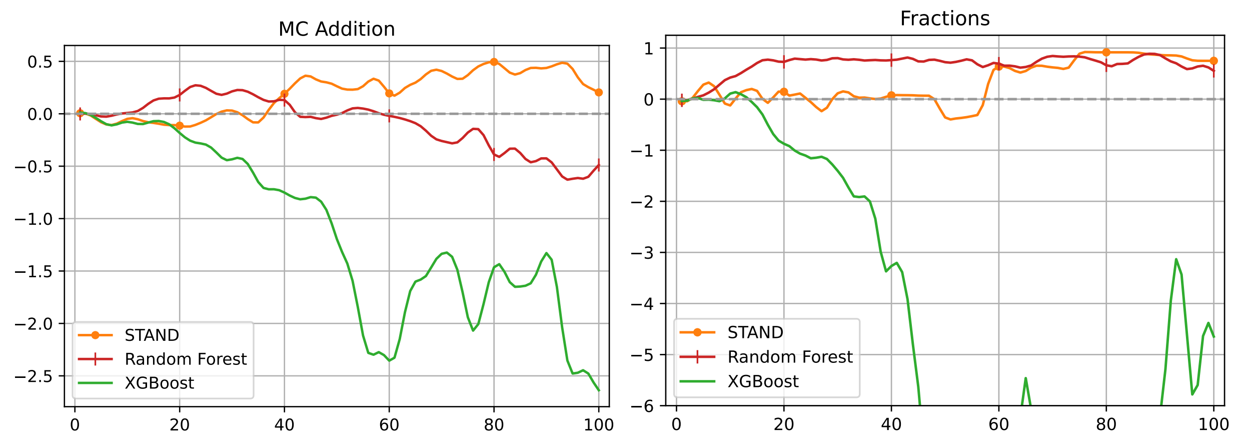


Fig. 31: Normalized Active Learning Utility by Problem

STAND shows positive active learning utility after problem 40 in multi-column addition with a peak of nearly .5 at problem 80. An active learning utility of .5 indicates that half of the remaining completeness deficit was made up by being able to self-select new problems with $IC(x)$. STAND shows high positive active learning utility in fractions after problem 60. The random forest shows some active-learning utility for early problems in multicolumn addition, and throughout training for fractions. XG Boost consistently shows negative utility on both domains.

STAND may only benefit from active learning in the latter stages of training in both domains because it is good for identifying edge cases to round out the final stages of training, but less effective when nearly any new example would be helpful.

5.5 An Improvement, but is STAND Enough?

Using STAND for when-learning resulted in higher levels of completeness with less training than the competing models. However, on its own STAND appears to be insufficient for making authoring-by-tutoring a practical method for building ITSs. Our simulation results show that STAND is able to achieve 100% model-tracing completeness in both domains, although not consistently. In the best case fewer than 30 problems are needed for 100% complete behavior in multi-column addition, assuming they cover relevant edge cases. In this domain if the author randomly picked next problems they would have a 50% chance of achieving 100% complete tutoring behavior if they trained AI2T on a huge set of 100 problems, and a small roughly 5% chance of achieving 100% complete tutoring behavior when training on a more manageable but still very large training set of about 30 problems. Compared to big-data machine learning these results are rather impressive, and indicate at least some chance of overcoming the issues that previous simulated learner based authoring tools have had with asymptotic completeness [118]. However, these results are still short of what would be desirable for practical use.

In the following chapter we improve upon AI2T's data-efficiency and robustness by relegating much of the role of when-learning to a new learning mechanism that learns hierarchical task networks. Later in chapter 7 we show that STAND's instance certainty measure can be displayed to users, helping them estimate when they are finished training an AI2T agent.

Chapter 6

Process-Learning: Hierarchical Task Network Induction from Action Sequences

In chapter 2 we discuss why constructing simulated learners' from a collection of multiple learning mechanisms is beneficial to data-efficiency. SimStudent and the Apprentice Learner (AL) employ a collection of 3-mechanisms: how-learning, where-learning, and when-learning. In this chapter we show that yet-greater data-efficiency and robustness can be achieved by further decomposing the role of when-learning by introducing a fourth learning mechanism called process-learning that learns the hierarchical control structure of tasks. We will show that this learning mechanism makes it possible to more reliably produce correct and complete ITS behavior in our two ITS domains.

6.1 The Problem with Purely Display-Based Induction

In SimStudent and AL, when-learning effectively learns two things that could be treated separately: 1) the order that skills should be applied and 2) the necessary preconditions for applying each skill. If we look at the set of preconditions learned by a particular skill we can often pick out literals that act as ordering constraints or precondition constraints. For instance, in the fraction arithmetic example, the preconditions for the induced skill AddNumerators that adds the numerators for two fractions together might look like:

```
AddNumerators(a,b) :=
    a.below.value == b.below.value and          # The denominators are the same
    a.right.value == "+" and                    # The operator is "+"
    a.value != "" and b.value != "" and         # The numerators are not empty
    a.below.value != "" and b.below.value != "" # The denominators are not empty
```

The first two literals are precondition-like since they control whether or not the skill should be applied depending on the context of the problem: we only add numerators if the operator is "+" and only if the denominators of the fractions are equal. The remaining literals are order-like conditions. They simply prevent this skill from being applied before all of the values of a converted fraction are filled in.

In SimStudent and AL, when-learning is responsible for learning both precondition-like and order-like conditions. In this setup each skill's preconditions are learned independently from one another, meaning similar kinds of conditions may need to be learned across different skills. Each of these conditions embodies a form of display-based reasoning [50]. They specify what values certain relational features of the state should have when it is the correct time to apply each skill. However, this display-based approach can quickly prove insufficient. For instance, in order for the agent to learn multi-column addition by this approach, we need to give the agent feature predicates that elaborate on the state by specifying the values that would be filled in by other candidate actions. For instance, a plain English translation of preconditions for the skill Add2(a,b) that adds two numbers and takes the ones digit might be:

```
Add2(a,b) :=
    The partial sum slot for the previous column is filled and
    Carry2(a,b) would not carry a 1 into this column and
    Carry3(a,b,c) would not carry a 1 into this column and
    Carry2(a,b) would not carry a 1 into the previous column and
    Carry3(a,b,c) would not carry a 1 into the previous column and
    A value has not already been carried into this column
```

The Carry2(a,b) skill which carries the tens digit of the sum of two numbers would need to independently learn a similar set of preconditions. The need to learn the order that skills are applied by inducing these complicated preconditions makes when-learning much more complex problem than it needs to be and makes agents highly prone to producing errors where otherwise correct actions are suggested out of order. These kinds of errors can persist far into the late stages of training, and make up the vast majority of mistakes that agents make.

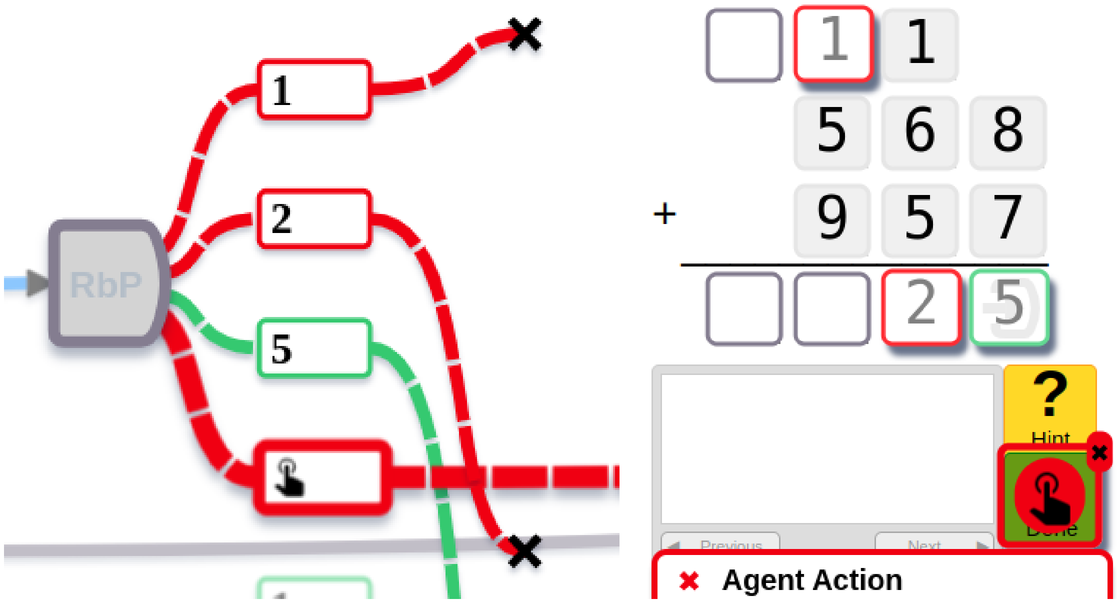


Fig. 32: A problem state where the agent suggests 3 out-of-order actions and one correct action. Actions have been marked correct (green) or incorrect (red) by the user. All incorrect actions are incorrect because they are applied too soon.

Relying upon when-learning to produce ordering constraints produces an initial inductive bias for permitting skills to produce actions in any order—so long as a set of arguments for applying the skill are present, the action produced from those arguments will be predicted to be correct. An author can only reign in this overly-general behavior by rejecting incorrect ordering behaviors as they occur. This policy of: *anything goes until proven incorrect*, makes for a rather frustrating and tedious user experience of constantly rejecting unusual behaviors as they unpredictably crop up. We can produce a more streamlined and predictable training experience by giving the agent an inductive bias for performing actions in the order that the user originally presented them, and only generalize from that strict ordering when the user demonstrates new action orders. A simple implementation of this bias would be to implement something similar to CTAT's example tracing tutors, where there is a graph of states with legal actions connecting them. However, this approach enables only a limited variety of control structures.

6.2 Hierarchical Task Networks

A more general approach to imposing control structures on procedural tasks is with hierarchical task networks (HTNs): a control structure of tasks and subtasks that permits strict action and subtask ordering, recursion, and alternative solution pathways gated by preconditions. Sierra implements HTN induction with a fourth mechanism referred to as skeleton induction [110] or control structure learning [111]. Although, we will instead refer to mechanisms that induce hierarchical control structures as *process-learning*. We use the term *process-learning* to generalize the idea beyond Sierra's particular implementation and avoid confusion with the field of control in robotics, which is the study of directing the continuous movements of objects and actuators [92].

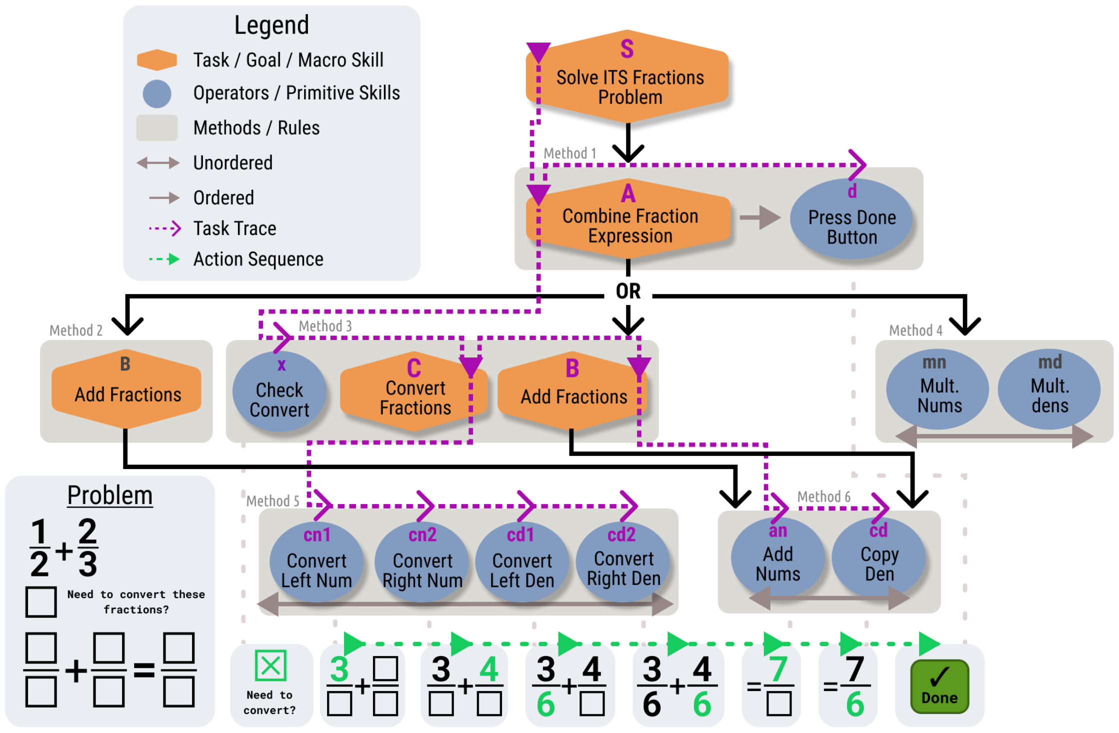


Fig. 33: A possible HTN for a fraction arithmetic ITS. A task trace (dotted-purple) and action trace (dashed-green) are shown of the solution to an addition problem of fractions with different denominators.

Process-learning acquires HTN that may recursively divide an overall target task into subtasks, and those subtasks into further subtasks. These learned HTNs capture the general process of performing a task, and their hierarchy terminates in primitive operator-like skills that collectively produce a sequence of grounded actions sufficient to achieve the overall task. The structure of a hierarchical task network is much like a grammar, except that each symbol may have argument variables, much like a function, and the expression of those symbols may be gated by preconditions, consisting of predicates over those variables.

An HTN consists of a set of hierarchical rules called methods. A method is a rule in the HTN's grammar structure that expands a task into subtasks [23]. A method has a head that represents the task that it achieves, a set of preconditions, and a sequence of resulting subtasks or primitive operators that should be executed if the preconditions hold. If multiple methods have the same task in their heads then this represents a disjunction—there are alternative choices of methods that achieve the same task. Preconditions may prevent the execution of certain method options for expanding a task. A task in an HTN represents a high-level objective, but does not necessarily represent a well-defined outcome or goal as a particular change to working memory or a particular change to the world. Higher-order tasks signify the decomposition of a process into more granular parts, and primitive tasks are operator-like rules that produce changes to the world.

The following table consists of terms that are roughly interchangeably with reference to structures shared between HTNs, traditional production systems, grammars, and the different kinds of skills of a simulated learner that implements process-learning:

HTNs	Production Systems	Grammars	Simulated Learners
primitive task, operator	operator	terminal symbol	primitive skill
(goal) task	goal	non-terminal symbol	macro skill
method	N/A	rewrite rule	method skill

Table 6: Terms that describe similar structures in different kinds of representation languages.

In our language describing process-learning we largely align our vocabulary with HTN literature with the exception of the term macro skill. In HTN literature the word task is often used to refer to both primitive and higher-order symbols interchangeably [24]. Our term macro skill refers to higher-order symbols in place of the term task or goal since our induced higher-order symbols lack the semantic certainty a hand-written task or goal with a meaningful name signifying the objective to produce a particular outcome. From the perspective of inducing HTNs from action sequences, an induced high-order task is not necessarily such a solid and semantically meaningful thing. The induced higher-order task (i.e. a macro skill) simply unifies some apparent disjunction in observed action sequences, and may change or restructure throughout the training process. While a user could certainly provide meaningful names, or even structured information to label or even assist the induction of a macro skill, process-learning as we frame it here does not require that a user interact with or structure the induced HTN in any way. Instead we would like to induce HTNs simply from the users' demonstrations.

6.2.1 Prior Work: Hierarchical Task Network Induction

Many approaches to inducing HTNs exploit highly structured inputs and background knowledge that can directly reveal elements of a target HTN's structure. For instance, a task trace (Fig. 33) provides a top down description of a decomposition of an action sequence into a set of tasks, subtasks, and operators that produced it. Many HTN induction methods require task trace annotations as an input [32,49,85], which directly specify a particular target

HTN structure. Other approaches exploit the rich representations available in planning environments to induce HTNs from sequences of operators. For instance, a method by Nejati et. al. [88] utilizes the existing preconditions of primitive operators in a planning domain to deduce the need for higher order tasks by backwards chaining from known goal. These approaches are not particularly useful for the sake of non-programmer ITS authoring, as they operate within heavily structured environments. ITS authors begins with at most a blank HTML interface, and so there is no ready-made environment for providing task traces, well-formed operators with preconditions, or even well defined goals. Some methods have explored interaction methods where a user directly articulates a recursive decomposition of task and subtasks in a top down manner similar to a task traces [33, 52, 59]. However, in this work we'll focus on the harder (and more ambiguous) problem of inducing HTNs in a bottom up manner from action sequences alone.

Learning HTNs from just action sequences is a fairly under-represented problem. A few approaches have emerged within planning and robotics, although many methods have tradeoffs that are ill-suited to ITS authoring. For instance, many approaches lack a means of learning multiple alternative solution paths [91]. Several methods utilize probabilistic context-free grammars to handle alternative solution paths [56,109]. However, these methods tend to produce large, overly specific HTNs [56] that vary greatly in structure from ground-truth HTNs and permit incorrect action sequences [16]. One approach that produces more constrained generalizations, and shares some similarity with the approach we outline in section 6.4, applies reductions that treat diverging sequences of actions as analogies to paths in a resistive circuit [16].

Sierra provides an early HTN induction approach for learning hierarchical control structures in mathematical tasks. Sierra was designed with the intent of simulating human learning of hierarchical control structure from tutorial demonstrations, and for modeling the errors that result from this process. Sierra's accompanying analyses of student errors support the idea that induction from tutorial instruction plays a dominant role in the learning of early elementary school mathematics skills compared to direct verbal instruction [111]. This conclusion is consistent with the almost universal phenomena that students learn best by doing, and less effectively when only by being told [18,41,115]. Sierra's approach to HTN induction succeeds at reproducing the human capability of learning control structure from action sequences in a several arithmetic domains. However, this approach comes with one very large caveat: Sierra's method is highly sensitive to the order that action sequence lessons are presented. Sequences must begin with very simple tasks that capture basic sub-procedures and introduce just one new sub-procedure per new lesson [110]. This is a reasonable constraint for the purposes of modeling learning within highly curated curricula, however for the purposes of ITS authoring this constraint would put a large burden on authors to curate ideal curricula for the AI. If not, authors would quickly fall into hard to debug failure modes.

In this chapter we outline a method for learning HTNs from arbitrary action sequences in the context of ITS authoring, without imposing constraints on the order that action-sequence lessons are presented. Additionally, the methods we outline in the following sections enable the induction of an HTN language that is well-suited to capturing control structures that are useful in an ITS environment such as alternative solution paths, undordered action sequences, and optional or conditional actions.

6.2.2 The Benefits of Action Sequence Driven Induction

There are several reasons why in an authoring setting it is desirable for an agent to be able to replicate the human ability to learn hierarchical control from action sequences instead of from more explicit annotations like task traces or direct HTN descriptions, and several reasons why studying this form of induction is interesting in general:

Expert Blind-Spot While qualified authors will undoubtedly have tacit expertise of the domains they aim to author, they may lack meta-cognitive awareness of the full structure of that tacit expertise. Experts are often only able to explain their reasoning case-by-case upon deliberate reflection. In the absence of structured prompting to elicit these reflections experts often fail to to immediately articulate as much as 70-80% of what they know implicitly [124]. Methods for explicitly eliciting the structure of an expert's tacit knowledge are collectively called cognitive task analysis (CTA) [19]. A large body of literature overlapping with early work in AI investigates how CTA can be used to build programs called expert systems that reproduce human expertise across varied situations [116]. Since authors may be blind the structure of their tacit knowledge, or that of an external expert, they may find it challenging to produce accurate subtask traces that reflect an HTN structure sufficient for supporting full solution flexibility, and other forms of adaptivity in an ITS.

Novelty There are quite a few works that demonstrate methods for recursively prompting users to produce task trace-like annotations. Although, the kinds of domains that these sorts of methods have been demonstrated in are typically quite a bit simpler than the kinds of ITS domains that would require a rule-based implementation, and are often implemented within environments that implement many domain specific support structures beyond what could be expected in an authoring environment.

INSTRUCTO-SOAR [33] is an early example of this approach applied to teaching hierarchical tasks to a simulated robotic agent in a well-structured symbolic environment. A task like “press the green button” can be described to INSTRUCTO-SOAR in terms of primitive actions that it understands such as “move above the green button”, “move the arm up”, “move the arm down”. Rosie extends upon this SOAR-based system with capabilities for learning the words for primitive features, objects, and actions using situated instruction [84], and learning hierarchical tasks from situated instruction. SUGILITE [59], learns hierarchical programs for automating tasks within a smartphone UI, that implement simple forms of generalization from situated examples. VAL [52] is a recent example applied within game environments that uses LLMs to adapt flexibly to diverse phrasings of user instructions.

These systems rely upon a human instructor to essentially articulate task hierarchy, or demonstrate single action sequences that easily cover all variations in task instances with simple generalizations that replace constants with variables. This work is unique both because it induces hierarchy from action sequences alone, and because it does so in domains that also require inducing precondition that handle complex contextual decision making.

Novice Friendly The benefit of learning HTNs from step-by-step action traces alone is that authors need only initially exercise their tacit expertise by producing step-by-step solutions. By contrast, an authoring method that required subtask traces would likely impose steeper skill requirements on the author. Authors would need some familiarity with CTA-like thinking, and knowledge of how to articulate that thinking within an open-ended interface. By contrast, authoring by solving and checking problems requires minimal expertise beyond knowledge of the ITS domain itself.

Concrete Editable Artifacts Users may want to revise the HTN induced by our system, or simply see their tacit knowledge displayed in a well-structured format for their own learning purposes. An HTN induced from action sequences offers an opportunity to reveal its induced solution in its own representation language (or a simplification of it). An induced HTN is a concrete artifact that can be displayed with editing affordances—it may be easier to tweak an HTN than it is to produce one by scratch.

Broader Machine Learning Applications Deep reinforcement learning (RL) has captured a great deal of attention within the field of machine learning applied to procedural tasks [7]. Reinforcement learning learns a policy that performs tasks by selecting among a predefined set of grounded primitive actions for each world state. The method I report here may set a path for far more data-efficient learning in domains that would typically be approached with RL. Action sequence-based HTN induction could also serve as an explainable alternative to, or method of augmenting, black-box neural network models.

6.2.3 An HTN Representation for Flexible ITS Behavior

It is helpful to highlight that the hierarchical structure of an HTN is akin to a grammar—one that parses action sequences instead of sentences or character strings. The terminal symbols of the grammar are primitive operator-like rules that produce individual actions. The non-terminal symbols are tasks that decompose further into subtasks and operators. In an HTN each method is like a rewrite rule in a grammar, it indicates one possible decomposition of a task into subtasks or operators.

A key difference between an HTN and a grammar is that each of the symbols (tasks or primitives) in the HTN have variables, and each of the methods has both variables and preconditions. In our implementation, we permit certain tasks and primitives to also have preconditions (although this is somewhat unconventional). A successful application of a method can pass arguments to its child items, or a potential application may be rejected because its preconditions fail. Neglecting variables and preconditions (which we do not show), the HTN in Figure 33 could be represented with the following grammar-like shorthand:

$S \to A\ d$	mn : multiply numerators	md : multiply denominators
$A \to B \mid C\ B \mid mn\ md$	an : add numerators	cd : copy denominator
$B \to an\ cd$	cn1 : convert left numerator	cn2 : convert right numerator
$C \to cn1\ cn2\ cd1\ cd2$	cn1 : convert left denominator	cn2 : convert right denominator

Here the | symbol represents a disjunction of alternative methods. For instance, A is a goal task (i.e. a non-terminal) that can be achieved by applying one of three disjoint methods: B, $C\ B$, or $mn\ md$. In this domain each of these methods is mutually exclusive, meaning the preconditions for their methods should permit only one of them to ever be applied. In other cases one may permit multiple methods to be executed for the same problem, indicating multiple acceptable solution strategies.

By contrast AL and SimStudent's 3-mechanism learning approach induces skills that are akin to just the terminal lowercase symbols in this grammar notation. They represent strictly primitive single-action-producing rules (i.e. operators). SimStudent and AL's skills, are initially unconstrained and will consider any match they find in the problem state. Since matches are not inherited from any higher-order structures, matching is somewhat computationally expensive relative to an HTN and can produce a broader set of possibilities. In this case, since valid matches are not constrained by a higher-level control structure, skills tend to be erroneously applied out-of-order until sufficient positive and negative examples enable when-learning to induce sufficiently constraining preconditions.

Typically HTNs are hand-programmed to guide planning systems that find a single sequence of operator applications that achieve a goal from a set of initial conditions. In an ITS authoring environment our HTN representations must be flexible enough to permit a variety of student solutions. In an authoring setting an ideal HTN must be able to handle alternative solution strategies, the presence or absence of optional and or preconditioned actions, and enable loose constraints on the order that actions are performed (i.e. unordered groups).

In our HTN representation, we allow methods to be marked as unordered. In our grammar shorthand, we represent an unordered method by underlining the method's item sequence. For instance in the fractions example:

$$\begin{aligned}
&S \rightarrow A\ d \\
&A \rightarrow B \mid C\ B \mid \underline{mn\ md} \\
&B \rightarrow \underline{an\ cd} \\
&C \rightarrow \underline{cn1\ cn2\ cd1\ cd2}
\end{aligned}$$

mn : multiply numerators	md : multiply denominators
an : add numerators	cd : copy denominator
cn1 : convert left numerator	cn2 : convert right numerator
cn1 : convert left denominator	cn2 : convert right denominator
d : press done button	

Optional and conditioned items are indicated by an asterisk. For instance, a recursive HTN for multi-column addition of two numbers could look like:

$$\begin{aligned}
&A \rightarrow B\ d \\
&B \rightarrow C\ B^{*} \\
&C \rightarrow \underline{a2\ c2^{*}} \mid \underline{a3\ c3^{*}} \mid cp
\end{aligned}$$

a2 : write ones-digit of a+b	c2 : carry tens-digit of a+b
a3 : write ones-digit of a+b+c	c3 : carry tens-digit of a+b+c
cp : copy down digit	d : press done button

Here applying the asterisk to a symbol U is equivalent to replacing it with V such that $V \rightarrow U \mid \epsilon$, where ϵ indicates the null string (i.e. do nothing). This annotation simplifies the notation; so that grammars tend to require fewer disjuncts, and side-steps the burden of implementing a special ϵ no-op primitive. Functionally a symbol annotated with an asterisk indicates that an application of a skill might be rejected by its preconditions, and should be skipped over if that is the case.

In the above example, this asterisk annotation for marking symbols as conditional serves two roles. First, it controls the recursion of the symbol B, which is responsible for computing the partial sums one column at a time from right to left. This is equivalent to the more conventional recursion notation $B \rightarrow C \mid CB$. Second, it allows the two primitive skills for carrying the ones-digit to be applied conditionally. Preconditions should be learned that only apply these skills if the partial sum is greater than 10.

6.2.4 HTN Induction: Lessons from Sierra

Sierra's approach to learning HTNs from action traces treats an HTN like a grammar and makes incremental changes to that grammar so it can parse new sequences of primitive actions. Sierra first parses a new action sequence as far as it can in a top-down manner, recursively executing each method in its HTN. This process can produce several alternative parses that may align with the new sequence to varying degrees. If top-down parsing fails to reproduce the input sequence then Sierra amends its grammar with a process called parse completion. It tries to parse the sequence bottom-up as far up as it can to meet the top-down parse trees. The hole between the top-down and bottom-up parses is filled by adding a new method.

In practice there can be many different choices of ways to complete a parse. Sierra imposes some inductive biases that inform this choice that were found to explain common patterns of errors observed in student data. Sierra imposes biases to fill the holes between top-down and bottom-up parses that favor 1) adding new methods to the deepest subtasks in the top-down parse, 2) creating new methods with the fewest downstream symbols, and 3) calling upon the highest subtasks in the bottom-up parse (in that order). Roughly speaking, these biases tend to amend the HTN at its lowest symbols, and make minimal changes. Each new method added as a result of parse completion adds disjunction to the grammar: some subtask in the grammar gets a new alternative downstream method.

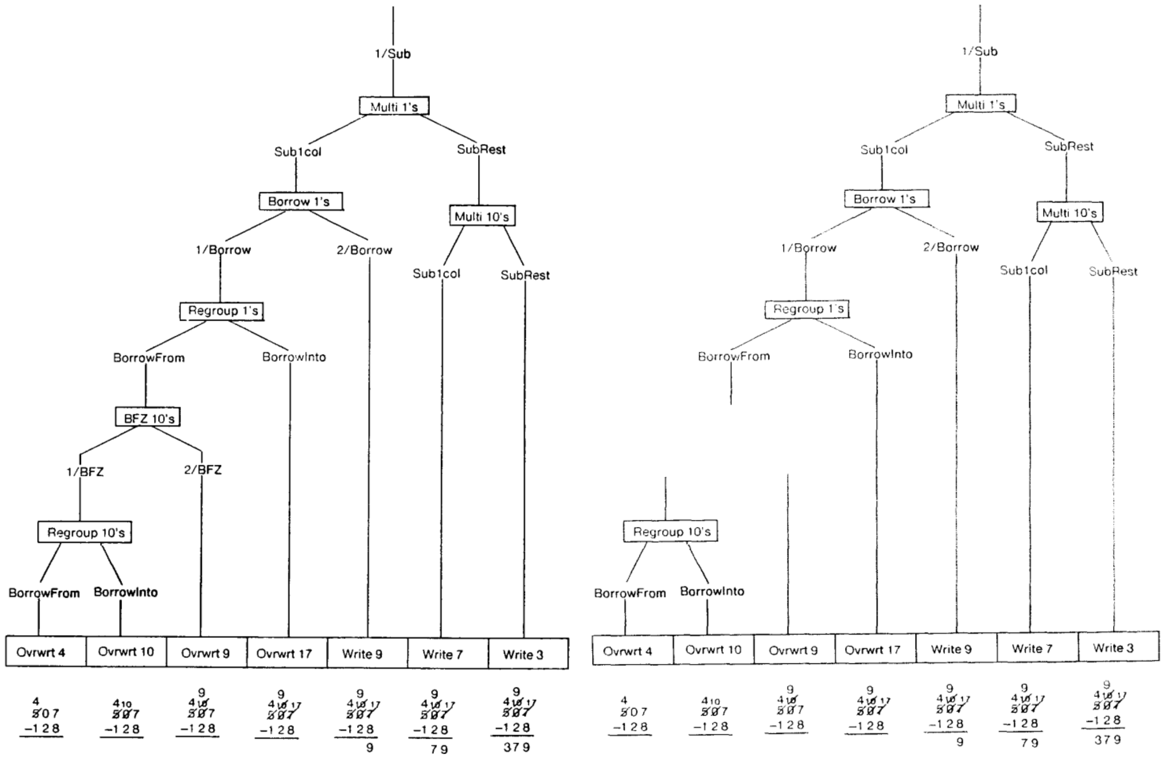


Fig. 34: (left) A trace tree of the true HTN for a solution to a subtraction problem where you need to subtract by borrowing from zero. (right) An incomplete trace with a hole. The agent's current HTN lacks the BFZ 10's goal. The top-down parse covers the last four actions, and a bottom-up parse can cover the first two. Parse completion attempts to fill the hole.

Early iterations of Sierra's parse completion algorithm implemented a broader accounting of possible grammars that constructed the space of all grammars (under some constraints) that could parse a particular action sequence. This is called the sequence's derivational version-space [110]. This version-space-based approach produces a very large collection of candidate methods that could be part of the ideal grammar. However, ultimately VanLehn concluded that imposing biases on the grammar construction process, such as adding the fewest new disjuncts per lesson, constrained the learning process such that only one grammar needs to be maintained at a time, instead of a whole space of possible grammars.

A drawback of Sierra's HTN induction that makes it cumbersome in an authoring environment is that it can only learn "one sub-procedure per lesson", or in our language: one method per action sequence. Sierra's capabilities have only been reported on curated sequences of lessons in a handful of arithmetic domains, so it is hard to know if its true requirements are even more limiting than this. For instance, parse completion's inductive biases control the subtasks that new methods are assigned to. It is difficult to know how likely it is for these biases to produce unrecoverable mistakes in uncurated curricula by assigning new methods to the wrong parent subtasks.

In any case, the interactive nature of an authoring environment favors more flexibility and robustness than Sierra's approach to HTN induction is likely to provide. It is not unreasonable to ask authors to attempt to curate lesson plans for the agent that are similar to the kinds of lessons that Sierra has been shown to succeed on—ones that begin with short simple problems and build up new edge cases one lesson at a time. However, it is unclear whether authors would find this easy, or that they could reliably succeed at producing sufficient lesson sequences on their first try. A bad lesson sequence could easily trip up the agent's inductive processes opening up a class of failure modes that are better avoided entirely if possible.

We can however draw a few lessons from Sierra's approach:

1. We probably do not need to track a space of multiple grammars. Updating a single grammar to accommodate each new action sequence should suffice.
2. We should only generalize incrementally and conservatively, imposing strong inductive biases.
3. Generalizing properly requires aligning each action sequence with existing skills in the current HTN. A combination of top-down and bottom-up parsing is a good approach for finding candidate locations for grammar edits.

6.3 A Complex Initial Approach: Minimal Grammar Updates

We want our HTN induction system to be robust to the order that action sequences are received. The simplifying power of Sierra's one sub-procedure per lesson requirement is that much of the ambiguity of how operators should be grouped into methods is eliminated. If only one new method can be expressed in a lesson then any actions that cannot be parsed must be part of the same new method.

In abandoning the one sub-procedure per lesson requirement, we open ourselves to the possibility that each new action sequence can introduce several new methods. New action sequences may also introduce generalizations on existing methods by applying one of our new unordering and conditional symbol annotations—an ordered method may become unordered or one of its items can be marked as conditional.

In this section, I outline several considerations that arose in my attempt to implement a grammar induction algorithm that works by generalizing a grammar with each new action sequence. Ultimately I abandoned this approach, as it became far too complex to implement in a way that would succeed with any order of action sequences. However, it is helpful to highlight some of the considerations that arose during this process, as they inform the more robust approach that I outline in the next section.

6.3.1 An Example Target HTN Grammar

Consider a non-recursive target HTN sufficient for adding two 3-digit numbers:

$S \rightarrow A\ B\ C\ cp\ d$	a2 : write ones-digit of a+b	c2 : carry tens-digit of a+b
$A \rightarrow \underline{a2\ c2^*}$	a3 : write ones-digit of a+b+c	c3 : carry tens-digit of a+b+c
$B \rightarrow \underline{a2\ c2^*} \mid \underline{a3\ c3^*}$	cp : copy down digit	d : press done button
$C \rightarrow \underline{a2\ c2^*} \mid \underline{a3\ c3^*}$		

Now consider an action sequence, where each action is notated by the pattern {prim}{x} where primitive skill "prim" is applied to column "x". For instance, let a21 indicate adding two digits in the ones-column and writing the ones-digit of that partial sum below, and c33 indicate carrying the tens-digit of the sum of three numbers in the hundreds column to the next column. So for instance we can consider a few action sequences that might be produced by a few kinds of problems:

777+777	a21 c21 a32 c32 a33 c33 cp d
	c21 a21 c32 a32 c33 a33 cp d
772+776	a21 c21 a32 c32 a33 d
	c21 a21 c32 a32 a33 d
333+333	a21 a22 a23 d
238+179	a21 a22 c22 a33 c33 cp d
	a21 c22 a22 c33 a33 cp d

The target ITS behavior encoded in our target HTN permits certain groups of actions to be performed in any order, so several of the problems may produce multiple action sequences.

This example HTN grammar may appear a bit unusual since it is non-recursive, meaning it would only work for pairs of 3-digit numbers. It also disjoins some primitive operations that could in principle be replaced with a single operation. For instance, we might express each of the instances of *cp*, $a2$, $a3$ as simply: OnesDigit(Sum(column)).

The lack of recursion in this example is partially for illustrative purposes. It is easier to consider how the basics of the two approaches described in this section and the next, would work without introducing the problem of inducing recursive structures. Additionally, at the time of writing this, we haven't perfected the details of inducing recursive programs by these approaches, although I share some considerations on the matter in the latter part of the next section.

The choice of using a plurality of primitive operators *cp*, $a2$, $a3$ is also partially to illustrate the generalization capabilities of these approaches since modeling to a domain in this way requires a grammar with some instances of disjunction. Although even for practical purposes, this disjunctive execution of multiple similar primitives is arguably a good choice for an ITS environment. It is not unreasonable to expect that a student may accidentally just add two numbers (i.e. $a2$) in a column, but forget to add the carry value as well (i.e. $a3$). Having this behavior explicitly expressed in the HTN provides a means of recognizing this kind of mistake so that adaptive instruction can be provided in response to it. When it comes to programming an ITS, the simplest program is not necessarily the best program, the best program is the one that adapts to students appropriately.

6.3.2 Abducing Skill Applications from Actions

When an author provides an action sequence, they are providing information with a more impoverished representation than our shorthand implies. For instance, the action for a21 in the first sequence of 777+777 would initially be experienced by the agent as Update-TextField(out1, 4) meaning a 4 was placed below the line in the ones column. The agent must abduce a skill with the correct how-part formula OnesDigit(a+b) and where-part pattern to explain how this action was produced. In the second action trace a21 would be experienced as the action UpdateTextField(out1, 6) and the agent would need to abduce that this action was produced by the skill it learned from the previous problem. Thus, the two instances of a21 are in fact different actions, but the agent's ability to self-explain actions allows it to interpret them as essentially the same skill application (i.e. the same skill applied with the same arguments). We'll assume in the following examples that the agent always succeeds at abucing the correct skill application for each action, and thus we'll assume that each item in an action sequence uniquely implies a particular skill application.

6.3.3 Initialization

Next, we must consider how we can induce an HTN that covers a set of action sequences. First, we must decide upon a grammar that covers the first sequence. There are an infinite number of grammars that could parse any finite sequence. But since a single sequence has no evidence to indicate any disjunction, unordering, or conditional items, a reasonable inductive bias is to begin with the flattest possible grammar that would only produce the initial sequence. For instance, if the first sequence a21 c21 a32 c32 a33 c33 cp d was seen from the problem 777+777 would produce an HTN:

$S \rightarrow a2\ c2\ a3\ c3\ a3\ c3\ cp\ d$

Here our HTN can be expressed as a flat grammar with a single method. Each item in this initial method indicates an operator-like primitive skill (i.e. they produce one atomic action). In our notation, we drop the column index for each item to differentiate skill applications from primitive skills, since a single skill may be applicable in multiple situations.

Note that unlike the "'one subprocedure per lesson" approach this initial HTN contains none of the methods of the target grammar. To produce the target grammar, or something functionally equivalent to it, we'll need to edit this flat grammar to include new symbols and disjuncts as new sequences provide new evidence for generalizing the grammar.

6.3.4 Sequence Alignment and Parsing

Updating the grammar from an action sequence requires determining which items in the sequence are applications of existing primitive skills already in the grammar and which ones are new. Similar to Sierra's approach, in our approach we first apply the HTN in a top-down manner to produce a tree structure of skill applications. The skill applications produced from the application of primitive skills in the top-down parse are compared pair-wise with the skill applications from the action sequence. A pair of skill applications can overlap to varying degrees. 1) The skill applications' how-part formulas may agree or not, and 2) some proportion of their match elements—the particular elements bound to their selection and argument variables—may be shared.

The average of these two forms of overlap creates a total overlap score from 0.0 to 1.0 between a pairing of skill applications between those produced by the top-down parse and those abduced from the observed action sequence. Any perfectly overlapping pairs are recorded for later. Then a bottom-up parse is performed starting with the deepest methods in the HTN, moving upward. During bottom-up parsing each method is aligned with the input sequence. The alignment is a mapping, built greedily by adding the pair of primitive skills applications with the highest overlap score one at a time. This alignment process associates some subset of the methods in the HTN with non-overlapping spans of the input sequence.

Each method with any non-empty alignment span will need to be edited to accommodate the discrepancies between the method and the skill applications in its span. If all the alignment spans do not completely cover the input sequence, then the holes are filled by choosing a method somewhere upstream from the hole to expand its span and fill the hole. The overlap score of elements at the edges of each span are one heuristic for choosing between them. If there is a tie the lowest method is chosen.

We might consider whether this process could work only with a bottom-up parse, and no top-down parse. The top-down parse is necessary for two reasons:

1. Top-down parsing allows us to generate virtual skill applications that can be compared to the observed skill applications from the action sequence. This is helpful for generating overlap scores.
2. Parsing top-down allows us to find all the skill applications that the grammar can already produce before assigning method spans in the bottom-up parse. This gives the highest methods in the grammar the opportunity to reserve coverage of any skill applications that they cover so that they are not erroneously assigned to the spans of downstream methods.

6.3.5 Insert, Delete, Disjoin, and Unorder Edits

If a method's alignment with the input sequence is not a one-to-one mapping then the method can be edited in one of four ways so that it can cover the subsequence in its span:

1. **delete**: If one of the method's items has no pair in the subsequence then we can apply a delete edit. Add the asterisk annotation to that item to identify it as conditionally applicable.
2. **insert**: If one of the skill applications in the subsequence has no paired item then we can apply an insert edit. Add a new item (with an asterisk annotation) to the method that covers the skill application.
3. **disjoin**: If some subsets of the method's item sequence and skill application subsequence cannot be reconciled then apply a disjoin edit. Turn each of the two subsets into a new

method. Add the new methods to a new non-terminal symbol and replace the item subset with that symbol.

4. **unorder**: If some subset of the method's item sequence and skill application subsequence have perfect overlap scores, but the alignment shows that they are out of order, then make a substitution with a new non-terminal symbol with one child method consisting of the item subset which is marked as unordered. If the item subset covers the whole method then just mark the original method as unordered.

6.3.6 The Trouble with Bad Generalization Commitments

Sometimes grammar edits can produce incorrect new structures in the grammar that are challenging to amend. Recall our ground-truth grammar for reference:

$S \to A\ B\ C\ cp\ d$	a2 : write ones-digit of a+b	c2 : carry tens-digit of a+b
$A \to \underline{a2\ c2^*}$	a3 : write ones-digit of a+b+c	c3 : carry tens-digit of a+b+c
$B \to \underline{a2\ c2^*} \mid \underline{a3\ c3^*}$	cp : copy down digit	d : press done button
$C \to \underline{a2\ c2^*} \mid \underline{a3\ c3^*}$		

Consider a grammar initialized from the problem 777+777, $S \to a2\ c2\ a3\ c3\ a3\ c3\ cp\ d$, that will be generalized from a sequence from 333+333: a21 a22 a23 d. The single method in the grammar only perfectly aligns with the first and last skill applications a21 and d. Consider a few possibilities for how this grammar could be edited to incorporate the new sequence.

Modifying $S \to a2\ c2\ a3\ c3\ a3\ c3\ cp\ d$, to incorporate: a21 a22 a23 d

(1)	(2)	(3)	(4)
$S \to a2\ A\ d$	$S \to a2\ c2^*\ A\ cp^*\ d$	$S \to a2\ c2^*\ A\ B\ cp^*\ d$	$S \to a2\ c2^*\ A\ B\ cp^*\ d$
$A \to c2\ a3\ c3\ a3\ c3\ cp \mid a2\ a2$	$A \to a3\ c3\ a3\ c3 \mid a2\ a2$	$A \to a3\ c3 \mid a2$	$A \to a3 \mid a2$
		$B \to a3\ c3 \mid a2$	$B \to c3\ a3\ c3 \mid a2$

The first option only permits the two example sequences that have been seen so far. It would be generated by a simple disjoin edit. The second option permits only a few more sequences than the two example sequences. It would be generated by dividing off two insert edits from the disjoin edit in the first example to introduce $c2^*$ and $cp*$. The third option is the closest to our target grammar. However, it makes several premature generalizations that allow it to cover several correct sequences that have not yet been encountered. The fourth illustrates that the way the third option was partitioned was a lucky choice, this option would permit some incorrect sequences.

These four possibilities illustrate a difficult trade-off. Ideally, we would like to make edits that minimally generalize the grammar so that it accepts as few new example sequences as possible. Conservative generalizations reduce the risk of inducing grammars that may permit incorrect sequences. However, even minimal changes can impose structures on the grammar that can cause issues later on.

For instance, to converge to our target grammar from the first example we would need to at some point move the first $c2$ from the first method of A and put it in the same method as $a2$. However, it is challenging to consider how this sort of **redistribution edit** could be implemented conservatively. If such an edit were allowed then in principle there could be many possible items in an HTN's methods that could be redistributed, and it is would be difficult to formulate a policy for when such an edit should be applied and how alternative possibilities should be chosen between.

An alternative that works for this example is to impose biases that prefer edits that produce the second grammar over the first. One bias is to prefer edits that keep items together in the same method that have high match overlap—ones that draw from and act upon the same objects. Each pair of $a2$ $c2$ or $a3$ $c3$ in the target grammar draws upon digits from the same column, so the correct pairings always have a greater match overlap. This is a bias that is most likely reasonable to impose in almost any domain. If two primitive skills should belong to the same method then they share a parent symbol (i.e. a goal task/macro skill) with arguments that should be passed to multiple items in each of its child methods.

Unfortunately, these inductive biases cannot completely eliminate the possibility of generating an HTN structure that can only be fixed through an implementation of redistribution edits. We saw our failure to find an inductive bias that consistently eliminated these kinds of bad inductions as a considerable drawback to this approach. In general, grammar editing can become quite complex.

6.3.7 Merge Edits

A particularly complex form of edit, the merge edit, is required when we consider the changes needed to transform the second grammar into something closer to our target grammar. Consider the edits that must be made to the second grammar to incorporate an example from the problem 238+179, with sequence a21 a22 c23 a33 c33 cp d :

Modifying	$S \rightarrow a2\ c2^*\ A\ cp^*\ d$	to incorporate: a21 a22 c23 a33 c33 cp d
	$A \rightarrow a3\ c3\ a3\ c3 \mid a2\ a2$	

(5)	(6)
$S \rightarrow a2\ c2^*\ A\ cp^*\ d$	$S \rightarrow a2\ c2^*\ CD\ cp^*\ d$
$A \rightarrow C\ D$	$C \rightarrow a3\ c3 \mid a2\ c2^*$
$C \rightarrow a3\ c3 \mid a2\ c2^*$	$D \rightarrow a3\ c3 \mid a2$
$D \rightarrow a3\ c3 \mid a2$	

This example highlights some new considerations. In this example the last two items of A's first method: a3 c3 **a3 c3**, align with parts of the new sequence and the first item of the second method: **a2** a2, aligns with one of the skill applications in the sequence. However since these two methods are disjoint so the grammar must be modified so they are no longer disjoint if it is to permit the subsequence a22 a33 c33. To do this we can apply a **merge edit** where we combine two disjoint methods into one method that redistributes their items into new subtasks C and D. This gives us grammar (5). Merging A's two methods leaves it with just one method: C D, so we can substitute any instance of A with the sequence C D, to get the slightly simpler grammar (6).

The need for merge edits adds another layer of complexity to this approach. There are several edge cases that need to be considered for a robust implementation, including the fact that any merge edit can also trigger additional insert, delete, disjoin, or, unorder edits. For instance, in this example, an insert edit adds $c2^*$ to C's second method.

6.3.8 Why Abandon This Approach?

In our initial testing, this approach appeared to work quite well. For instance, in both multi-column addition and fractions it is easy to converge to a structure similar to a target grammar if one simply initially provides two action sequences from the same problem where the second is presented with any unordered subsequences applied in the reverse order. For instance, consider taking two different sequences from the problem 777+777.

Modifying $S \rightarrow a2\ c2\ a3\ c3\ a3\ c3\ cp\ d$, to incorporate: c21 a21 c32 a32 c33 a33 cp d

$S \rightarrow A\ B\ C\ cp\ d$
$A \rightarrow \underline{a2\ c2}$
$B \rightarrow \underline{a3\ c3}$
$C \rightarrow \underline{a3\ c3}$

Three unorder edits are applied to the initial grammar. After this point, very little can go wrong since the HTN grammar already has a correct partitioning of subtasks. However, if we do not begin with helpful sequences like these that make this partitioning clear then we may find ourselves in situations where method items are incorrectly partitioned and need to be redistributed. As noted before, implementing a redistribution edit opens up many difficult considerations.

One reason that it may be interesting to continue pursuing this approach is that these hard-to-fix grammar induction problems may have analogs in human learning. Perhaps some class of students' misconceptions are of a similar nature: some action in an example may be misinterpreted as belonging to an unrelated subtask that happens to occur contiguously with it in an example lesson. This may cause a class of errors where students make the mistake of taking an action when it is not necessary or neglecting to take an action as part of its true subtask. How does one remediate such a misconception?

For authoring purposes, however, the fact that this approach falls into these kinds of failure modes is problematic. Without more work, this approach does not succeed at achieving our goal of being agnostic to action sequence order. Moreover, with seven or so different kinds of edits (insert, delete, disjoin, unorder, merge, simplify, redistribute) it is quite complicated to implement and maintain. It would be nice to have something simpler.

6.4 A Robust Approach: Order Agnostic HTN Induction

One of the key insights in our implementation of STAND was that making any sort of arbitrary inductive commitment from a collection of evidence limited our inductive power. HTN induction presents a similar, but perhaps harder problem: in making arbitrary inductive commitments we run the risk of making incorrect commitments that we cannot recover from. It is likely possible that a similar version-space-like solution could be applied to HTN induction where multiple diverging commitments produce multiple independent grammars or a bounded space of grammars. The literature surrounding Sierra's development dives quite deeply into this sort of possibility [110]. However, entertaining a plurality of grammars would only complicate the existing grammar editing approach, and may be computationally expensive to the point of being useless for interactive task learning.

6.4.1 An Atomic Evidence Representation

Instead, we'll start by attempting to characterize the problem of inducing HTNs from action sequences more precisely. Our first consideration should be to understand the kinds of evidence that action sequences provide. Relative to task traces they are a very implicit sort of evidence. Task traces directly provide pieces of the target HTN abstraction. By contrast, action sequences only provide atomic forms of evidence that disambiguate elements of the target HTN, but do not reveal its structure directly.

Let us take account of some forms of evidence that an action sequence can provide, and define a concrete representation for those forms of evidence. A new action sequence may introduce:

1. **New Primitives:** An action that cannot be explained with known primitive skills is evidence that a new primitive skill (i.e. operator) is needed to form the target HTN.
2. **Order (A^{min}, A^{max}):** Evidence about the orders that skill applications can occur. Skill applications that are part of the same method tend to be adjacent or close to one another. We can encode this evidence in two matrices A^{min} and A^{max}. If skill application i has position index $P_k(i)$ in sequence k then $A^{min}_{ij} = \min_k P(i) - P(j)$ and $A^{max}_{ij} = \max_k P(i) - P(j)$.
3. **Co-Occurrence (C):** Some skill applications may always co-occur. The items of a method produce skill applications that always co-occur unless one is conditionally applicable. The matrix $C_{ij} = 1$ if all action sequences where skill application i occurs skill application j also occurs, otherwise $C_{ij} = 0$. It can be the case that $C_{ij} = 0 \wedge C_{ji} = 1$ if, for instance, j always occurs with i, but i can occur on its own.
4. **Disjunction (D):** Some skill applications may never co-occur. This can be evidence that two skill applications result from two disjoint methods of the same task. The matrix $D_{ij} = 1$ if skill applications i and j never co-occur in all action sequences and $D_{ij} = 0$ otherwise.
5. **Match Overlap (O):** Evidence about the match overlap between skill applications. It is a reasonable inductive bias to assume that skill applications that belong to the same method will draw upon and act upon some of the same objects. O_{ij} is the match overlap score for skill applications i and j.

Instead of directly building and updating a single grammar, our order-agnostic approach will accumulate pair-wise evidence about the relationships between different skill applications within the matrices A^{min}, A^{max}, C, D, and O. When a new action sequence produces a change in these matrices we can induce the simplest HTN that follows from the updated evidence. Since the evidence matrices are order-agnostic—they will be the same regardless of the order that action sequences were evaluated—our grammar construction process will be order-agnostic as well.

6.4.2 Two Steps: Primitive Sequencing and Iterative Construction

This new approach to HTN induction is performed in two steps. The first step is **primitive sequencing**: we order all of the unique primitive skills into one sequence where: 1) primitive skills that are likely to be part of the same method stay together, 2) disjoint skills that likely share the same parent task stay together, and 3) the presentation order between co-occurring primitive skills is maintained. The second step is **iterative construction**. In this step, the sequence is partitioned into subsequences that are likely to be part of common methods, and then any disjoint method subsequences are grouped into common macro skills (i.e. goal tasks). If this joining process produces a reduced sequence with multiple remaining macro or primitive skills then the process is repeated again recursively until the remaining symbols require no grouping into new macros. Iterative construction should end when the remaining symbols do not have any neighboring disjoint items.

It is easiest to understand this two-step process by example. Let us begin by showing what an ideal sequence of primitive skills would look like in our multi-column addition example. For simplicity, we will assume that the implementation creates one primitive skill per unique skill application so we can use our {prim}{x} notation for skill applications to indicate each primitive skill. For target grammar:

$$\begin{array}{l} S \rightarrow A\ B\ C\ cp\ d \\ A \rightarrow \underline{a2\ c2^*} \\ B \rightarrow \underline{a2\ c2^*} \mid \underline{a3\ c3^*} \\ C \rightarrow \underline{a2\ c2^*} \mid \underline{a3\ c3^*} \end{array}$$

a2 : write ones-digit of a+b	c2 : carry tens-digit of a+b
a3 : write ones-digit of a+b+c	c3 : carry tens-digit of a+b+c
cp : copy down digit	d : press done button

The ideal ordering of primitive skills in stage 1 would give:

$a21\ c21\ a22\ c22\ a32\ c32\ a23\ c23\ a33\ c33\ cp\ d$

Which ideally would be partitioned into methods as:

$(a21\ c21)\ (a22\ c22)\ (a32\ c32)\ (a23\ c23)\ (a33\ c33)\ (cp)\ (d)$

Then the disjoint methods can be grouped into macros skills as:

$[(a21\ c21)]\ [(a22\ c22)\ (a32\ c32)]\ [(a23\ c23)\ (a33\ c33)]\ (cp)\ (d)$

Note that $(a21\ c21)$, (cp), and (d) are single candidate methods with no parent disjunction. If the ordering matrices A^{min} and A^{max} showed that the order in $(a21\ c21)$ could be permuted then it will be grouped into its own macro so that a new unordered method can be introduced. (cp), and (d) imply no disjunction or unordering, so there is no reason to introduce new macros or methods to cover them. The remaining new grouped sequence would be $A\ B\ C\ cp\ d$. Assuming the evidence matrices A^{max}, A^{min}, and C had converged to their final states at this point the methods of the grouped task could also be given their asterisk and unorder annotations: $A \rightarrow \underline{a21\ c21^*}$, $B \rightarrow \underline{a22\ c22^*} \mid \underline{a32\ c32^*}$, $C \rightarrow \underline{a22\ c22^*} \mid \underline{a32\ c32^*}$.

At this point, the evidence matrices on this new set of symbols must be recomputed. However, since every pair of new items is non-disjoint $D_{ij} = 0$, we do not need to do another full round of iterative construction. At this point, the evidence matrices may suggest that (cp) does not always co-occur with the other skills in which case it must be given an asterisk annotation to mark it as conditional. An asterisk is given to skill i when there exists j in the new sequence such that $C_{ji} = 0$.

6.4.3 Step 1: Primitive Sequencing

In the previous example, we showed the iterative construction step operating on an already prepared primitive sequence. The primitive sequencing step ensures that iterative construction begins with a well-formed primitive sequence. From the first action sequence, the primitive skill sequence is simply the sequence of skills abduced from that initial action sequence. When a new action sequence is incorporated the primitive sequence must be reordered to include any newly abduced skills. The new ordering is generated greedily by adding primitive skills one at a time. The first skill can be chosen from any primitive skill that never strictly comes after some other skill. The next skills are greedily added to the sequence one at a time. Skill i is chosen next which maximizes the utility value U_{oi} on the current order o. This next primitive item is chosen on the basis of item $l \in o$, the right-most item in o that is non-disjoint with i (i.e. $D_{li} = 0$).

The utility U_{oj} is a tuple with the following slots in order of importance.

1. **Order Violation Count (v_i):** The negative of the number of ordering violations that would be produced by choosing primitive i next. An order violation occurs if $(\exists k \in o, k \neq i : A^{max}_{ki} < 0) \vee (\exists k' \notin o, k \neq i : A^{max}_{ik'} < 0)$. In practice, if a non-violating ordering exists then the chosen next skill will always have $v_i = 0$.
2. **Strong Adjacency Score (s_i):** Two skills are strongly adjacent if they have ever been adjacent: $A^{min}_{li} <= 1 \wedge A^{max}_{li} >= 1$, and always occur together $C_{il} = 1 \wedge C_{li} = 1$. If l and i are strongly adjacent then $S_i = (.5 + O_{il}) / \max(A^{max}_{li}, 1)$, otherwise $S_i = 0$. S_i is high when the match overlap between i and l is high, and when their maximum distance from each other is small.
3. **Weak Adjacency Score (w_i):** Two skills are weakly adjacent if they have ever been adjacent: $A^{min}_{li} <= 1 \wedge A^{max}_{li} >= 1$, and one always occurs when the other occurs $C_{il} = 1 \vee C_{li} = 1$, but not necessarily the other way around. If l and i are weakly adjacent then $W_i = (.5 + O_{il}) / \max(A^{max}_{li}, 1)$, otherwise $W_i = 0$.
4. **Match Overlap O_{li}:** The match overlap between l and i.
5. **Disjoint Score d_i:** If l and i are disjoint then $d_i = O_{li}$ otherwise $d_i = 0$.

Greedily choosing i that maximizes U_{oi} greedily produces primitive sequences that do not violate order constraints, that keep adjacent and co-occurring skills together, that keep skills together that tend to have high match overlap, and that keep disjoint spans of primitives together. In other words, subsequences of primitives likely to be part of the same method stay together, and disjoint method subsequences tend to stay together—an ideal sequencing for the iterative construction step. All of the sub-heuristics of U_{oi} incorporate O_{li} to some degree. This makes the greedy choice for the next primitive biased toward keeping items that match the same objects together.

6.4.4 Step 2: Iterative Construction

Iterative construction proceeds in two phases. First, the primitive sequence is partitioned into candidate methods. Each span (n, m) of the primitive sequence is given a method score M_{nm}. The method score M_{nm} has value 0 if any pair $i, j \in [n, m]$ are disjoint $D_{ij} = 1$. Otherwise, its value is computed from the other evidence matrices. We can break down the calculation for M_{nm} into four parts:

1. Weak Co-occurrence Score (M_{nm}^{weak}): $M_{nm}^{weak} = \sum_{i,j \in [n,m]; i \neq j} C_{ij} \vee C_{ji}$.
2. Strong Co-occurrence Score (M_{nm}^{strong}): $M_{nm}^{weak} = \sum_{i,j \in [n,m]; i \neq j} C_{ij} \wedge C_{ji}$.
3. Overlap Score ($M_{nm}^{overlap}$): $M_{nm}^{overlap} = \sum_{i,j \in [n,m]; i \neq j} O_{ij}$.
4. Fully Unordered (u_{nm}): $u_{nm} = 1$ if A^{min} and A^{max} permit all $i, j \in [n, m]$ to be permuted, otherwise $u_{nm} = 0$

Combining these parts:
$M_{nm} = \frac{(m-n)*(M_{nm}^{strong}+u_{nm})+M_{nm}^{weak}+M_{nm}^{overlap}}{(m-n)^2-(m-n)}$
This metric rewards partitions of methods that span long sets of strongly co-occurring items and that can be unordered. Weak co-occurrence and match overlap are given less weight. The denominator divides the total value by the number of off-diagonal elements in each sum so that long spans are not artificially weighted more highly. The set of method spans is chosen greedily by starting from $i = 0$, selecting the span (i, m) such that $\max_m M_{im}$, then incrementing i to the end of the span and repeating to the end.

Next, any subsequences of method spans that are completely mutually disjoint, meaning each pair of spanning items between them are mutually disjoint $D_{ij} = 0$, are grouped into common subtasks. The evidence matrices for the new sequence are computed, and the next construction iteration begins if needed.

This two-step process succeeds at inducing HTNs for arbitrary action sequences for both the 3x3 digit multi-column addition example and for our fraction arithmetic example.

6.4.5 Considerations for Inducing Recursive Grammars

One of the undesirable qualities of our multi-column addition example is that it only works for problems where we add two three-digit numbers. Of course, we would prefer a method that induces a recursive grammar that works for any pair of numbers. Several forms of information could be used to hypothesize potential recursions. The most obvious hint is the presence of contiguous skill applications that have the same how-part formula. Consider a recursive grammar for multi-column addition:

$$\begin{aligned} &A \to B\ d \\ &B \to C\ B^* \\ &C \to \underline{a2\ c2^*} \mid \underline{a3\ c3^*} \mid cp \end{aligned}$$

a2 : write ones-digit of a+b	c2 : carry tens-digit of a+b
a3 : write ones-digit of a+b+c	c3 : carry tens-digit of a+b+c
cp : copy down digit	d : press done button

Now consider an example like 333+333, with sequence $a21\ a22\ a23\ d$. In this case, there is a fairly clear candidate for recursion: $B \to a2\ B^*$, since $a2$ is repeated. To avoid spurious over-generalizations we might hold off until it is clear that this recursion is needed since we do not know that there are not always exactly 3 instances of $a2$. For instance, 33+33 or 3333+1234 would indicate a variable number of repeats.

Of course, other problems would reveal that $a2$ is not the only primitive that is repeated, meaning we need a means of slotting in other primitives into the body of the recursive rule appropriately. One helpful heuristic is to consider sets of symbols where there is a consistent relationship between the skill applications at each recursion step. Recall that each of our symbols is really a skill with variables and so a recursion like $B \to a2\ B^*$ could be restated with variables. Depending on how rich our interface representation is we could express this as one of the following:

$B(a, b, c) \to a2(a, b, c)\ B(a.\text{left}, b.\text{left}, c.\text{left})^*$
$B(col) \to a2(col)\ B(col.\text{left})^*$

Where a.left is an expression for de-referencing the symbol to the left of variable a, and where *col* is a variable that captures a collection of interface elements grouped into a column. If we encountered a new problem with new primitives like $238 + 79$ with sequence $a21\ a22\ c22\ a33\ c33\ cp\ d$, then many of the primitives may be new, but pairs of them may share spatial relationships that are a hint that they ought to be descendants of the same macro skill in a recursion relationship. For instance, the *col*.left relationship is maintained by partitioning the sequence above into:

$(a21)\ (a22\ c22)\ (a33\ c33)\ (cp)\ d$

Several details remain to be worked out considering how these forms of evidence could be incorporated into our current system, and used to induce recursive structures.

6.4.6 Integration with Decomposed Inductive Procedure Learning

The addition of process-learning as a fourth learning mechanism necessitates some reconsideration of the relationship between different learning mechanisms. An agent that implements process-learning should be able to determine where it currently is within its HTN's control structure on the basis of the current problem state. While the three-mechanism version of Decomposed Inductive Procedure Learning (DIPL) could operate in a purely display-based manner—reflexively responding to new problem states by attempting to apply each of its skills—the inclusion of process-learning requires us to consider how skills are applied as part of a higher-level control process.

For instance, the agent must have a means of accounting for an internal hierarchical planning state. Since actions can be externally demonstrated by a user, and since the HTN induced so far is not necessarily complete, the agent cannot necessarily rely upon a stored internal planning state updated one action at a time. Instead, it is sometimes necessary for the agent to self-explain how a particular problem state can be reached by following the process outlined in its current induced HTN. The agent can use its HTN grammar to parse a sequence of actions leading to the state in question in a top-down manner and determine if the sequence can be produced by its grammar. If this parsing process succeeds then the grammar can yield a set of multiple possible next primitive skill applications. If the parsing process fails then the sequence diverges from the grammar in some way, in which case any unparsable actions are considered *out-of-process*.

For instance, consider that the agent has experienced four actions in a solution to the multi-column addition problem 773+668:

a21 c21 a32 c32 a33 c33 cp4 d

The parse path for this series of actions puts the HTN in a planning state where there are four actions that could be produced next.

$S \to A\ B\ \mathbf{C}\ cp^*\ d;\quad C \to \underline{\mathbf{a2}\ c2^*}$
$S \to A\ B\ \mathbf{C}\ cp^*\ d;\quad C \to \underline{a2\ \mathbf{c2}^*}$
$S \to A\ B\ \mathbf{C}\ cp^*\ d;\quad C \to \underline{\mathbf{a3}\ c3^*}$
$S \to A\ B\ \mathbf{C}\ cp^*\ d;\quad C \to \underline{a3\ \mathbf{c3}^*}$

Taking one more action:

a21 c21 a32 c32 a33 c33 cp4 d

There are three actions that can be taken next.

$S \rightarrow A\ B\ \mathbf{C}\ cp^*\ d;\quad C \rightarrow$ ~~a2~~ $\underline{\mathbf{c2}^*}$
$S \rightarrow A\ B\ C\ \mathbf{cp}^*\ d$
$S \rightarrow A\ B\ C\ cp^*\ \mathbf{d}$

Of course, in both of these problem states only a subset of the highlighted actions are actually correct next actions. The when-learning mechanism is responsible for learning preconditions that permit only correct next actions. With an induced HTN the role of when-learning is immensely simplified. Without a guiding HTN, when-learning is responsible for discovering preconditions that enforce the execution order of each primitive skill. By inducing an HTN with process-learning, ordering constraints are enforced by default and relaxed as the HTN generalizes.

Even in cases where the HTN produces a choice of multiple next actions, when-learning generally does not need to learn preconditions with cross-skill dependencies. For instance, in the second example, two primitives with a conditional annotation precede d, the skill for pressing the done button to finish a problem. Without process-learning the preconditions for pressing the done button are very complicated. They must check that several other skills must not first be applied, which requires special feature predicates that indicate the applicability of those other skills. However, with the HTN, primitives have a clear order of priority. In our first example above, all of the next candidate primitives have the same priority since they are part of two disjoint unordered methods. In the second example, cp^* is only applicable if $c2^*$ is not applicable, and d is only applicable if cp^* is not applicable. Thus, the HTN produces a situation where the primitive d no longer requires any preconditions at all—it is simply the action that is taken after all of the other actions that should have been taken.

In general, the HTN makes it so that we only need to learn preconditions at points of effective disjunction in the grammar. Preconditions are needed for choosing between disjoint methods, and for deciding whether a symbol with a conditional annotation U^* ought to be applied. In principle, these could be treated as the same situation, since U^* is the same as $V \rightarrow U|\epsilon$. Although, for the purposes of debugging and for potentially communicating HTN structure to a user there is an elegance in maintaining a structure with as few symbols as possible.

6.4.7 Performance with Process-learning

When integrated as a fourth learning mechanism in Decomposed Inductive Procedure Learning (DIPL) our order-agnostic method for building HTNs from action sequences largely resolves prior issues with training agents that achieve 100% correct and complete tutor system behavior.

Table 7 and Figure 35 compare the holdout performance of agents with and without a process-learning mechanism. In both multicolumn addition and fraction arithmetic the average holdout performance over 40 repetitions for agents with a process-learning mechanism is

Table 7: Average Holdout Completeness at Problem N, and Number of 100% Complete Repetitions at problem 100.

	MC Addition				Fractions			
	N=20	N=50	N=100	100% Reps	N=20	N=50	N=100	100% Reps
STAND	85.45%	96.10%	98.62%	19/40	98.72%	99.91%	99.99%	**38/40**
STAND + HTN Ind.	**99.72%**	**100.00%**	**100.00%**	**40/40**	**99.69%**	**99.94%**	**99.99%**	**38/40**

higher than those without it. In multicolumn addition, 20 problems are sufficient to achieve 100% model-tracing completeness.

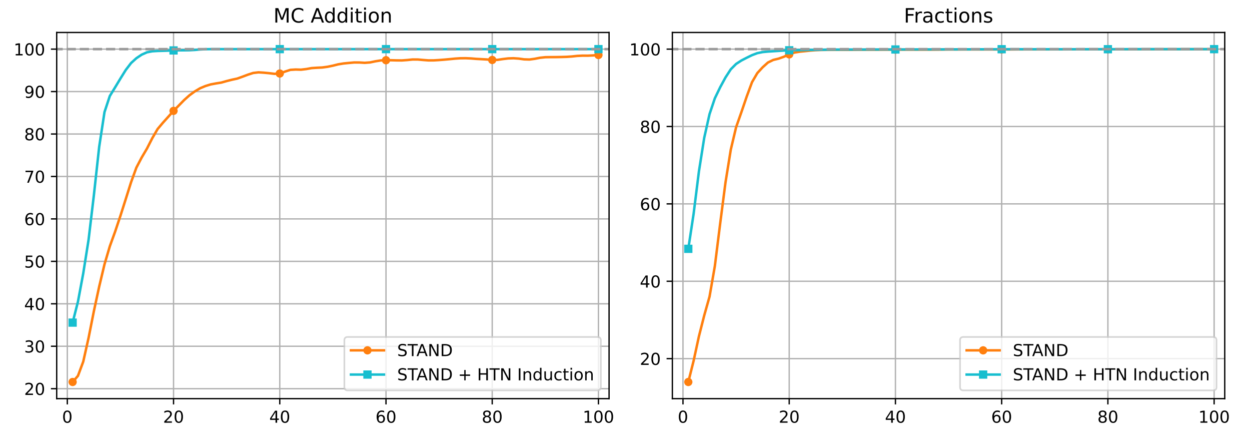


Fig. 35: Average completeness trajectories for AI2T agents using STAND with and without a process-learning mechanism for HTN induction.

Chapter 7

Evaluating AI2T With Users

We evaluated AI2T in two studies each with 10 users in which participants authored two tutoring system domains. In each domain participants tutored an AI2T agent on several problems until they were convinced that the agent could produce correct and complete behavior for any new problem instance. When participants self-reported that they believed the agent had achieved a state of absolute completeness, we scored their agents' model-tracing performance on a large holdout set of 100 problems. After being scored, participants moved on to the next domain.

The central aim of these studies was to evaluate what configurations of the agent and interface design best support authors in teaching the AI to induce correct and complete programs. Beyond qualitative observations of usability, a core element of this evaluation is to determine whether our interaction design supports authors in building model-tracing complete tutoring systems. This support includes assisting authors in self-assessing when they have sufficiently trained agents to a point of absolutely complete behavior.

Very little prior work explores interactions for supporting users in self-assessing the learning progress of interactively teachable AI agents that learn completely bottom-up from users. This is largely because most machine learning systems lack the data-efficiency or flexibility to be taught entirely from interactive training. Our interactively teachable simulated learners are a notable exception, as they learn domain-specific skills entirely from an author's instruction within arbitrary HTML interfaces instead of within specialized environments. These studies evaluate not just how well our interaction design supports users in engaging in effective training, but also in gaining trustable estimates of the agents' learning progress via our certainty score indicators which are enabled by STAND's instance certainty measure.

7.1 Methods

Domains The participants in our two studies taught one of two versions of AI2T multi-column addition and fraction arithmetic, the same two domains we used in our simulation experiments. Authors were given a blank interface for each of these domains and tutored an AI2T agent on several problems. In multicolumn addition, participants taught the agent the algorithm for summing large numbers together by computing partial sums and carrying each tens digit. We limited this domain to just summing pairs of 3-digit numbers. In fraction arithmetic, participants taught the agent how to add and multiply fractions, including how to convert fractions before adding when the two fractions have unequal denominators. In this domain, fractions are converted by simply multiplying their denominators, and then multiplying crosswise to find the converted numerators. Participants did not need to train the agent to reduce the converted fractions or simplify the final answer.

Domain Variants In both studies the fraction arithmetic domain was the same. However, the version of multicolumn addition that we had users teach AI2T differed between the two studies. In study 1, the agent was not given a process-learning mechanism. Recall that in chapter 5 we showed that agents without a process-learning mechanism struggle to learn multicolumn addition programs that are 100% model-tracing complete relative to a ground-truth tutoring system. Since our goal is to evaluate whether authors can self-assess completeness, we used a simpler version of multi-column addition in study 1 where this was feasible to achieve with the available agents.

The version of multicolumn addition we had users teach in study 1 requires that the tens digit of partial sums are carried even when it has a value of zero (i.e. when the partial sum is less than 10). We call this a *zero-carry* variant of multicolumn addition. This version is much easier for the 3-mechanism variety of AI2T (without process-learning) to learn because it has a much simpler control structure than the normal multicolumn addition algorithm in which zero-carries are omitted. The key reason for this is that explicitly performing zero-carries helps the agent keep track of where it is in the process of summing the two numbers. The agent can reason in a display-based manner from right to left, column-by-column by simply learning preconditions that ensure that the carry and output boxes of the previous column's partial sum have already been filled in. For the 3-mechanism variety of AI2T to learn the normal addition algorithm, the agent must learn much more complex between-skill preconditions that must sometimes check that other skills should not first carry a non-zero value into the current column.

In study 2, the AI2T agent is configured with a process-learning mechanism making learning the normal version of multi-column addition over 3-digit numbers much easier. However, since our current process-learning mechanism is still limited to non-recursive grammars it must encounter examples where every combination of primitive skills are applied in every column to achieve 100% holdout performance. In study 2 we helped users by presenting them with a set of 7 target problems that cover a wide variety of these cases. We asked that users train the agent on these 7 problems, and then continue to make up their own problems until they believed the agent's learning was complete.

Instruction In both studies, participants authored multicolumn addition before fraction arithmetic. None of the participants had used AI2T before, so in both studies we can think of this first domain as a warm-up attempt to practice using the tool. We first gave participants a short tutorial on how to use the tool by showing them how to demonstrate each step of the problem 777+777, and showed them how to give feedback to the agent on the subsequent problem 222+222. Prior to having participants begin authoring each domain, we described the behavior that we expected the final tutoring system to have, and we asked that participants engage in a think-aloud: "say whatever you are thinking as you work with the tool". Participants always began authoring with a blank agent with no prior training.

While users worked with AI2T we made ourselves available to answer questions and made participants aware if they began to teach the agent a procedure that differed from the target ITS behavior. We refused to give participants any advice concerning when they should stop training the agent. We limited ourselves to suggesting that they train the agent on a variety of problems and suggested that they should at least keep training AI2T until it seemed like it had stopped getting things wrong. We never explicitly pointed out the availability of certainty scores in the interface so that we could assess whether users noticed, understood, and utilized these indicators on their own.

At the end of each session, we asked users to give their overall feedback about the system. We simply asked: “What worked well and what didn’t work well as you were using the tool?”. In study 2 we also explicitly asked participants if they noticed the certainty score indicators and whether they considered them when deciding whether or not to stop training the agent.

Participants All users participated in these studies remotely via Zoom, and screen shared as they worked in AI2T’s web interface. Participants filled out online forms indicating their consent to be recorded. For their participation in these IRB-approved studies users were compensated with a $30 Amazon gift card. We limited all sessions to a maximum of 90 minutes. Participants for piloting prior to study 1 included labmates and colleagues who had volunteered their time. The 10 paid participants for study 1 were all graduate students recruited from Carnegie Mellon University (CMU), and 8 of these 10 participants were part of graduate programs that specialized in educational technology. 4 more CMU graduate students were recruited and compensated as part of piloting prior to study 2. The participants for study 2 included 5 graduate students from CMU, 3 of which specialized in educational technology, 2 human-computer interaction graduate students, 4 biology graduate students from Arizona State University, and 1 professional specializing in the authoring of instructional technology for a major ITS project unaffiliated with CMU. Participants’ self-reported genders were roughly equally male and female in both studies.

Our participant population of mostly graduate students is fairly well aligned with the educational background of professionals who would typically use an ITS authoring tool. Many of our participants explicitly study the design of educational technology (although not necessarily the programming elements of it), and professional learning engineers typically have some graduate-level education. Since our tool is intended for use by non-programmers, teachers are another potential target population. K-12 teachers are typically more inclined toward seeking out ready-made materials than making new material themselves, although we have certainly encountered a handful of teachers who have conveyed interest in this kind of tool. While our recruited participants are not in-service teachers they do have similar backgrounds to a typical teacher. All have bachelor’s degrees, and 6 of the 10 participants in study 2 were non-programmers. Several of the study 1 participants were non-programmers as well but we did not collect this data.

For study 2 we asked participants to score their programming experience on a Likert scale from 1 to 5. We asked: “Would you describe yourself as a proficient programmer (i.e. you have the ability to write scripts/software)” where 1 is labeled as “No. I have little or no programming experience.”, and 5 is labeled as “I have extensive programming experience. I believe I could program professionally.” For the purposes of our analyses, we consider a score of 1 or 2 to be a non-programmer.

7.2 Study 1: 3-Mechanism Agent

The aim of study 1 was to test several features of AI2T’s interface design that distinguish it from our 2020 prototype and to test this interface with an agent that used STAND for when-learning, allowing for better learning performance, and for showing the user a certainty score from -100% to 100% for each agent proposed action. In the study 1 version of AI2T, the agent was configured with a 3-mechanism structure like AL and SimStudent, with no

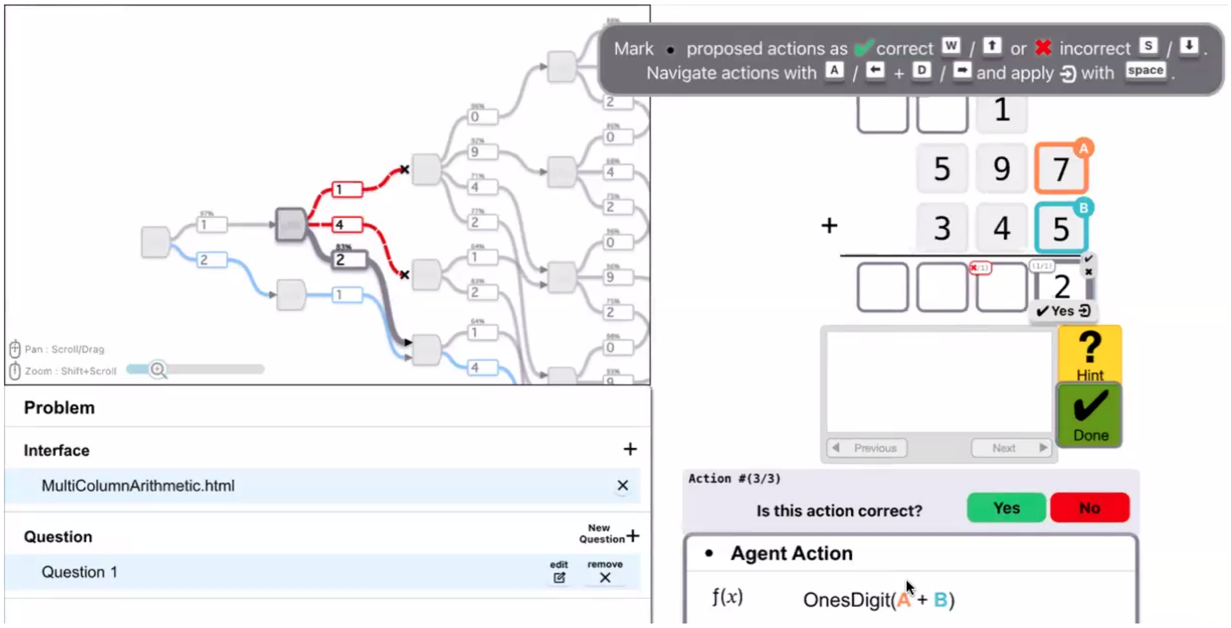


Fig. 36: Study 1 Interface. An intermediate state of problem 597+346. Two of three actions (outgoing graph edges in top-left) have been given negative feedback (marked red). A third action (grey) is selected, it adds 7+5 and takes the ones digit resulting in the value 2. Pressing the Yes button will mark this selected action as correct.

process-learning mechanism. The interface for this version had automatic behavior graph generation (without induced unordered groups), and users were able to cycle between the actions that the agent proposes at each problem state by selecting among the behavior graph edges, or by following along with the dialog prompts that ask "Is this action correct? Yes or No."

7.2.1 Quantitative Results

Table 8: Results for Study 1

	MC Addition (zero-carry)		**Fraction Arithmetic**	
User#	Completeness	Minutes	Completeness	Minutes
1	90%	55	-	
2	99.80%	33	64.69%	32
3	71.43%	31	100%	24
4	100%	33	92.76%	34
5	91.45%	24	100%	33
6	90.60%	24	38.05%	32
7	100%	22	85.75%	35
8	100%	23	95.29%	23
9	99.15%	28	76.43%	29
10	57.07%	50	-	-
Mean	90%	30.33	82%	30.25
Median	95%	28	89%	32

The quantitative results for study 1 are outlined in Table 8. For the *zero-carry* version of multicolumn addition, 3 of 10 participants taught agents that achieved a 100% model-tracing completeness score on the holdout set of 100 random problems. Two of our participants took more than half of the allotted 90 minutes for the first domain, and we did not have them complete the second. In fraction arithmetic just 2 of the 8 participants achieved 100%. In both domains, the median authoring time was about 30 minutes. This is the elapsed time spent by participants between beginning authoring the domain and self-reporting that they believe the agent has achieved correct and complete behavior.

7.2.2 Qualitative Results

The main goal of study 1 was to evaluate whether the interface improvements from our 2020 prototype succeeded in assisting authors. Automatic behavior graph generation was one major improvement over our prototype design. Users generally had little trouble panning and selecting states and actions in the behavior graph. However, some users had difficulty connecting patterns in the behavior graph with our instructions that certain subsets of actions should be permitted in any order. Alternative orders are displayed as diverging paths in the graph. A key element of users' difficulty was that actions essentially needed to be demonstrated or given feedback multiple times since they could appear as distinct edges along different paths with different orderings of actions.

For the fraction arithmetic domain, some users found it tedious to give feedback to the agent in 24 unique problem states generated from each permutation of the 4 steps associated with converting two fractions. In reality, users may have needed to only give feedback on a small subset of these states per problem to achieve 100% completeness. However, since the behavior graph made it very easy for users to see what states they had and had not given feedback on, and since we had set them upon the objective of 100% completeness, most participants tended to grade the agent along all generated paths.

Behavior graphs also showed certainty scores above each edge in the behavior graph. However, this presentation did not appear to be effective as none of the users mentioned the certainty scores in their think-alouds or follow-up interviews, and those that we asked explicitly said they did not notice them.

Many of the participants in study 1 had used other ITS authoring tools before, including CTAT's example-tracing authoring tool. In follow-up interviews, several participants commented that they found AI2T easier to use than CTAT example-tracing because after demonstrating solutions to a single problem, the agent would suggest step-by-step solutions automatically for the remaining problems. For instance, one participant remarked "This is a lot nicer than CTAT… I like that it mostly does the problems for you." These users commented that checking the agent's step-by-step solutions was much easier than demonstrating several problems themselves, and noted that the fact that the agent induced a program from their teaching saved them from needing to mass produce step-by-step problem solutions in a spreadsheet.

7.3 Study 2: AI2T with Process-Learning

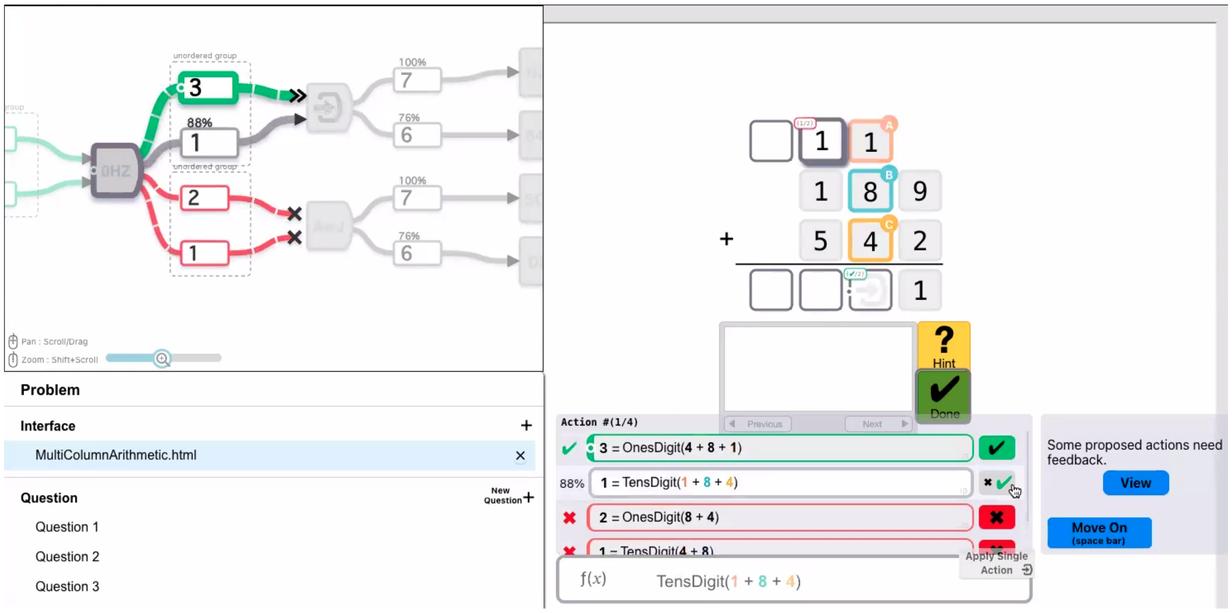


Fig. 37: Interface for study 2. An intermediate state of 189+542. There are 4 proposed actions, shown as skill app window items (middle-bottom) and graph edges (top-left). In the graph, they are grouped into two unordered groups (dashed boxes). Two actions have been given negative feedback (✗), and one has been given positive feedback (✓). The mouse hovers over an action without feedback that was proposed with 88% certainty. When the ✓ is pressed on the toggle button it will be given positive feedback. When "Move On" is pressed it will move to the state after both the correct actions in the top unordered group are applied.

The configurations of both the backend agent and frontend interface differ between study 1 and study 2. Taken together they form a loose pseudo-experiment, in which study 1 establishes a baseline with several issues and study 2 implements several fixes to remedy those issues. The version of AI2T deployed in study 2 implements three major improvements over study 1:

1. The AI2T agents have a process-learning mechanism that induces hierarchical task networks (HTNs) that organize skills into higher-order skills that can execute sub-skills—eventually terminating in primitive skills that produce actions. Process-learning learns an explicit control structure for each ITS domain instead of relying on each primitive skill to induce complex preconditions that gate their individual application.
2. When process-learning induces that a sequence of actions is unordered this information is displayed within the generated behavior graphs (Fig. 37 below). Users can automatically demonstrate an unordered group by demonstrating multiple next actions at once in the same state.
3. A skill application window is displayed listing every action that the agent believes is correct. This window prominently shows the certainty score for all proposed actions that have not yet been assigned feedback (Fig. 37 below).

Since participants were not randomly assigned to conditions and since there are multiple differences between the two studies we cannot precisely quantify the benefits of each improvement. However, as we will show in the following subsections participants in study 2 were far more successful at producing 100% model-tracing complete tutoring systems. The simulation results from chapters 6 would lead us to believe that this difference is largely due to the more rapid and robust learning that can be achieved with the inclusion of a process-learning mechanism. However, our qualitative observations and interviews with participants led us to believe that our interaction design improvements played a large role as well.

7.3.1 Quantitative Results

Table 9 outlines the results for study 2. 6 of our 10 participants reported their programming experience as a 2 out of 5. These 6 non-programmer participants included our 4 biology graduate students, and 2 graduate students specializing in design.

8 of 10 participants succeeded at training agents that achieved 100% holdout completeness in on the *normal* version of multicolumn addition (not the *zero-carry* version). This is the same version of multicolumn addition that participants were asked to author in our 2020 prototype, in which users had a median model-tracing completeness rate of 92%, and no user achieved 100%. Our study 2 results also showed users completing training in about half the time compared to our 2020 study: a median of 22 minutes instead of 41 minutes. For multicolumn addition, all users first solved the same 7 problems and selected their own problems thereafter. In most cases, two or three additional problems after the initial 7 were sufficient to achieve 100%. The two participants who did not reach 100% made mistakes during training that they did not succeed in tracking down and fixing.

Table 9: Results for Study 2

		MC Addition (normal)			**Fraction Arithmetic**				
User#	Prog. Exp.	Completeness	Minutes	N prob.	Completeness	Minutes	N prob.	Notice	Use
1	2	100%	22	13	100%	21	20	n	n
2	2	100%	20	9	100%	32	18	y	y
3	2	100%	30	11	96.31%	30	16	n	n
4	5	100%	14	11	88.52%	17	13	y	y
5	2	90.96%	30	14	40.18%	19	9	n	n
6	5	100%	22	11	98.87%	16	10	n	n
7	5	100%	28	11	100%	27	14	y	y
8	3	89.16%	36	11	38.73%	18	14	n	n
9	2	100%	22	10	100%	25	18	y	y
10	2	100%	18	9	100%	23	21	y	n
Mean	3	98%	24.2	11	86%	22.8	15.3		
Median	2	100%	22	11	99%	22	15		

In fraction arithmetic, participants self-selected all of their own problems, and 5 out of 10 participants succeeded at training agents that achieved 100% holdout completeness. The two lowest-performing participants made mistakes during training that prevented them from achieving more than 50% completeness. The median completeness in this domain was 99%. Participants trained the agent on 9 to 21 problems in this domain in 16 to 32 minutes with a median of 22 minutes.

In our follow-up interviews we asked participants if they noticed the certainty score indicators, and whether they considered them when deciding when to stop training the agent. 5 of 10 participants said that they did notice the certainty scores, and 4 indicated that they considered them when deciding whether or not to stop training the agent. Specifically, these participants indicated that they took the presence of a low certainty action as an indication that the agent needed additional training on similar problems. 3 of these 4 participants achieved 100% model-tracing completeness in both domains.

7.3.2 Qualitative Results

In our follow-up interviews several participants remarked on how quickly they learned to use our tool, and how they could succeed at using AI2T to author two tutoring systems in less than an hour. As in study 1, several users remarked on how quickly the agent was able to learn from their instruction and how the authoring process became much easier once they entered the stage of mostly checking the agent's behavior on new problems. For instance, one of our biology graduate student participants remarked: "This is wild, I could teach it all that math in like 20 minutes [per topic]."

While some study 1 participants had commented on issues with the smoothness of the "interaction loop", very few study 2 participants had constructive negative feedback. Our observations of users led us to believe that the inclusion of the skill application window in the study 2 design was helpful in this regard. Some study 1 participants jumped between problem states without fully giving feedback to all of the agent's proposed actions, whereas study 2 participants very consistently fell into a pattern of looking through actions in the skill application window and giving them all feedback before moving on. The inclusion of unordered groups meant that there were far fewer states in the study 2 version for users to navigate through. For instance, much of the users' time in study 1 was spent going through many diverging states in large behavior graphs, especially for the fraction arithmetic domain, but unordered groups spared study 2 users from the tedium of grading combinatorial paths. Some study 1 users expressed that they had become disoriented while navigating between problem states, but study 2 participants did not report any similar confusions.

Our follow-up interviews provided strong evidence that the users who noticed the certainty scores used them successfully to gauge when they should stop training the agent. For instance, participant 2 in study 2 said, "I definitely would have stopped teaching it earlier if I hadn't seen the low confidence on some problems that I thought it already knew how to do."

7.4 Discussion

Overall our study 2 results show that our redesign produced a considerable improvement over study 1. Half of the study 2 participants succeeded at training agents with 100% complete tutoring system behavior on both domains, usually in under half an hour. Our interviews with users also confirmed that displaying STAND's instance certainty measure was useful for assessing the AI2T agent's learning progress toward 100% completeness. Several participants in study 2 indicated that this indicator influenced their decision of when to stop training the agent on new problems. This is a strong preliminary indication that the certainty score indicators had the intended effect. A future randomized experiment would be able to lend stronger statistical evidence for the connection between the availability of this indicator and high authoring completeness. Although the productive monotonicity measure we report in chapter 5 already establishes that this measure accurately reflects agent learning, so it is reasonable to conclude that if users were explicitly trained to interpret it, they could use it successfully as a heuristic for estimating holdout completeness.

These two studies were conducted across two points in the development of AI2T, starting from a version that did not consistently achieve 100% model-tracing completeness, to a point (with the inclusion of process-learning) where correct and complete programs could be induced fairly consistently from being tutored by untrained users. In study 2, when users did not achieve 100% model-tracing completeness they either made clear mistakes during authoring (e.g., users 5 and 8) or trained AI2T on too few problems. Thus, improving AI2T's robustness may largely come down to better support for training users and helping them catch mistakes. Participants 3, 4, and 6 likely fell short of 100% because they trained AI2T on too few problems in fraction arithmetic (we didn't find any uncaught training upon reviewing their screen recordings). Participant 4 was the only user among these three who claimed to use certainty scores, and the only participant familiar with the trajectory of the AI2T project—specifically that AI2T had become more data-efficient in the months prior to study 2—and thus they may have had a skewed belief of how little training was required. When participants strictly interpreted a less than $< 100\%$ certainty score as an indication of incompleteness they trained AI2T on a sufficient number of problems. Consequently, a little bit of training to check for mistakes, and correctly interpret certainty scores may go a long way toward helping authors use AI2T effectively.

7.5 Future Work

While our results are very promising, our user observations indicate that elements of our system could be improved. For instance, a few of our study 2 participants made training mistakes that produced errors that they did not identify and fix, and several others trained the agent on too few problems to achieve 100% model-tracing complete behavior. It is reasonable that users interacting with a tool for the first time make these mistakes, however, it may be possible to better support users with these kinds of issues.

While our interaction design allows users to fix errors by deleting demonstrations or toggling feedback on proposed actions, users must still find an error in order to fix it. Usually buggy skills are easy to identify since they will persistently propose unusual actions. Experienced users may find it easier to find bugs if they were able to select individual skills, and browse through each example supporting its current induced generalizations. We could also help users choose good training problems by adding tools for auto-generating pools of problems and then using STAND's instance certainty measure to choose ones with low certainty. Recall that we showed in simulation in Chapter 5 that instance certainty can have a very high active-learning utility. This would help users produce more diverse training sets, and avoid a potential issue where the reported certainty scores are misleadingly high because the training problems do not cover important edge cases.

Chapter 8

LLMs: Opportunities and Risks in the Classroom and for Complementing Authoring-by-Tutoring

The elephant in the room is that since the proposal of this dissertation, the AI world has evolved tremendously, raising the question of whether the innovations reported here are obsolete. In the span of a year, the capabilities of Large Language Models (LLMs) have improved by leaps and bounds and spurred a wide range of applications. Pretrained language models have been employed in grading systems [25,53,74,86], educational content generation [125], and have even been suggested as all-purpose tutoring chat bots [82,89,108]. While these developments have attracted a great deal of attention, it is important to understand the scope of LLM's potential and risks. LLMs enable many exciting possibilities for generating content at a moment's notice and for responding flexibly to varied student inputs. Nonetheless, I am not convinced that many of the proposed applications of LLMs will produce significant improvements in the quality of online education in the near term. To understand my overall pessimism toward the use of LLMs in education, yet optimism in select cases, we must understand both how LLMs learn and perform, and consider whether they could play a role in building educational technology like ITSs that actually show considerable improvements over traditional instruction.

LLMs like ChatGPT are able to generate text, solve math problems, and write code because they have been trained on gargantuan datasets: hundreds of billions of words of text, hundreds of thousands of math problems, and large fractions of open-source code repositories like GitHub. An LLM's training dataset consists of more data of any variety than any human would consume in a lifetime. Whether it generates sentences, math solutions [61], or code [8], the performance of an LLM is typically imperfect, because it is essentially a very complex form of auto-complete that outputs text by predicting next characters one at a time. The more variety of human data the model is trained on, the more likely it is to generate an output that we recognize as an acceptable human-like response in more diverse situations.

Surprisingly, this approach can partially work in math on fairly challenging problems, yet always with imperfect performance (78% for instance in the case of [61]). Training a neural network to solve math problems is a bit of a Sisyphean project. Regardless of the model's size or training set, the approach of using neural networks to directly generate responses produces brittle performance [121]. Much of a neural network's acquired capabilities in these domains are simply memorized problem-solution pairs. The way that the neural network compresses these solutions into its many internal weights produces some generalization that enables the model to guess at unseen problems. However, since it does not actually induce a program that leverages arithmetic functions, as we report in this work, its induced behavior is far from reliable. It is quite easy, to come up with a problem that an LLM will undoubtedly get wrong. For instance, problems with very large numbers that are unlikely to be part of the LLM's training set often do the trick. Additionally, since LLMs use some stochasticity in their generation process and many are constantly retrained causing them to forget many of their prior capabilities, LLMs can have highly variable performance between repeated queries and over time.

8.1 Static Content Generation

The major value proposition of generative models is that they enable people to produce new content on a whim. Although for mathematical content, where precision is essential, it is difficult to see the benefit of using an LLM for question or answer generation when it will inevitably generate errors. It seems silly to favor buggy content over curated content, like for instance, the presumably errorless dataset that the LLM was trained on. After all, a small fraction of the problems from that gargantuan dataset would suffice for covering a large number of topics. For static content creation, the perceived value of the LLM is the convenience of instantly drawing upon a great deal of prior human labor.

In an educational context, generating text with an LLM is a much safer prospect than generating mathematical problems or solutions, as text is much easier to spot-check for mistakes. An LLM that produces occasional errors can still be useful if the time and effort required to double-check the model's output is less than writing the text from scratch. And there is a great deal of value in simply generating any material, even if it is of poor quality, as a starting point. Drawing from auto-generated content can greatly lessen the burden of needing to generate ideas, or provide an overall outline of what one wants to cover.

In a recent work, Pardos et. al. [94] found that ChatGPT 3.5 produced erroneous hints 30% of the time for algebra problems, yet they claim that they could mostly eliminate these errors by querying the model 10 times per problem, and taking the most frequent response. The filtered ChatGPT 3.5 hints were compared to expert-generated hints in a randomized experiment with students showing that both produced similar learning gains. However, it is worth considering if Pardos et. al.'s trick achieves something somewhat absurd in the grand scheme of things. The LLM is able to generate hints because it was trained on a dataset with lots of examples of algebra hints. If we poll the model multiple times, the expected output would be, on average, similar to the examples it was trained on, especially in a K-12 domain like algebra which is probably well-represented in the LLM's training set. This raises the question: was this experiment an evaluation of ChatGPT's generative capabilities or a comparison between the human-generated hints from ChatGPT's training set and another set of human-generated hints? An LLM user cannot escape the fact that the generative model was built upon other people's labor, which they may or may not be legally entitled to.

A key drawback of LLMs is that they simply compress existing content and regurgitate it back to users, meaning they cannot generate content of any higher quality than the average poor-quality content already available on the internet. In the absence of high-quality curated training data, the expected output of an LLM will be the fatefully mediocre mean of the most frequent patterns in its training set. One can often tease higher quality content from an LLM with "prompt engineering", in which one provides more specific requests to an LLM to constraint its generative capabilities, perhaps kicking it into drawing from parts of its network trained from higher quality training examples [122]. But, it is not as if one could not have searched for those high-quality materials themselves. High-quality publicly available materials for any conceivable university-level topic are available in Massive Online Open Courses (MOOCs), many of which are offered free of charge by world-class institutions.

Platforms like Khan Academy and IXL serve the same purpose for K-12 content. And of course, search engines allow us to locate countless documents about just about anything.

While many of the materials across the internet and in platforms like MOOCs and Khan Academy are of high quality in terms of accuracy, fluency, and comprehensiveness they are not necessarily optimal learning tools since they most often consist of static text and videos which are known to be among the least effective tools for of learning [18, 41]. If we want to consider how AI can be used to improve education we need to disabuse ourselves of the idea that content creation is the main hurdle to providing quality education. The internet has no shortage of high-quality content. LLMs simply make it easier for those starting from scratch to borrow or steal from existing content with plausible deniability. What we should really consider is how we can provide students with the kinds of active and adaptive one-on-one tutoring characteristic of ITSs, that rival the teaching performance of human tutors [44]. Delivering this kind of content at scale is the most compelling path toward improving online educational systems with AI.

8.2 On-Demand Tutoring

The idea of using LLMs directly as interactive tutors has attracted a great deal of attention, and many classrooms have already experimented with incorporating tools like ChatGPT, and Khan Academy's Khanmigo variant of GPT into their curricula. The perceived benefits of using an LLM as a tutor are inarguable. ChatGPT is considerably more flexible than any other chatbot before it and can answer queries on a wide range of topics, and solve problems step-by-step along with detailed explanations. This approach to on-demand tutoring presents a considerable opportunity.

Yet LLM-based chatbots have many glaring risks. Any on-demand response generated by an LLM may be riddled with errors that could do more harm than good. LLM mistakes have been likened to "hallucinations" [34]—a term that simultaneously captures the illogic of many of their gaffs and the polite but self-assured manner in which they utter statements that are clearly incorrect or logically inconsistent. However, I believe the most harmful hallucinations are those that are not obviously incorrect; those that are so nearly correct that they appear authoritative, yet confuse some important detail that would surely produce a misconception in a novice. An LLM's purpose is to produce text that sounds like what a human would say, yet it engages in no explicit reasoning to produce its responses. It is not difficult to imagine an LLM, for instance, providing advice that would be appropriate in one situation, that is completely incorrect in another. In one instance, I observed ChatGPT 3.5 refer quite confidently to the opposite sides of a triangle, which is of course a geometric impossibility. Student's misconceptions and mis-articulations lead them to say similarly odd things. An LLM-based chatbot could very well be prone to re-affirming those illogical notions instead of catching and correcting them.

There are several ways in which LLMs' hallucination-prone generative abilities can be reigned in. For instance, Retrieval Augmented Generation (RAG) is an approach that largely constrains the outputs of an LLM to content present in a reference text [54]. Other approaches to building educational chatbots with LLMs have taken the approach of generating static chat dialog flows [104]. Another relatively safe approach is to use an LLM as the basis for classifiers that grade or decide which of several fixed feedback options to provide to students on the basis of their responses [62]. As self-contained elements within more structured programs, the risks of using LLMs can be mitigated. For instance, some risks of an unconstrained chatbot are that students may abuse it by convincing it to do work for them, or distract them by entertaining their off-task queries. These sorts of issues have historical analogs in prior educational technologies. There are many cases of educational technologies with engaging and open-ended features that were shown to ultimately just waste student's time [63, 71]. Similar issues will no doubt become apparent as unconstrained LLM chatbots are evaluated in classroom studies.

8.3 Model-Tracing and Cognitive Supports

One reason to favor highly structured tutoring system designs over the sorts of open-ended systems that an unconstrained LLM might enable, is simply that many of the greatest ITS successes in terms of measured learning gains have been built through a highly detail-oriented cognitivist approach that tends to focus students on precisely what they need to learn at each point of their learning trajectory [44, 113]. The tried-and-true methods of ITS engineering involve using cognitive task analyses (CTAs) to design these very focused educational experiences [19]. CTAs of experts help learning engineers understand the nitty gritty details of the specific knowledge they intend for students to master, and CTAs of students help learning engineers understand what elements of that knowledge are difficult to learn so that they can design very targeted adaptive features. There is no conceivable reason to believe that in the absence of these efforts, an LLM could make similarly well-informed instructional decisions.

Model-tracing is the essential ITS feature that this dissertation has sought to make easier to implement: the ability of an ITS to track student solutions and mistakes step-by-step. Model-tracing enables ITSs to provide targeted step-specific feedback for particular solution strategies and mistakes and track when students utilize certain knowledge components in particular problem steps.

One issue with trying to replicate model-tracing with an LLM is that it is difficult to design a prompt that reliably extracts information about steps that are more fine-grained than the LLM's usual outputs. In my attempts so far, ChatGPT has consistently failed when answering my best attempts at designing model-tracing-like queries. For instance for the following prompt to ChatGPT 3.5:

> **Prompt:** "A student has partially solved the problem 1/4 + 2/3. They have found the common denominator 12 for the two fractions. What should they put as the numerator of the left converted fraction?"

The model responds by giving a long step-by-step explanation for solving the problem which concludes with:

> **ChatGPT:** "Now, we add these fractions:
>
> $\frac{1}{4} = \frac{1x3}{4x3} = \frac{3}{12}$
> $\frac{2}{3} = \frac{2x4}{3x4} = \frac{8}{12}$
>
> Therefore, the numerator of the left converted fraction is $\boxed{11}$"

This error is not surprising. The language model is almost certainly only trained to reproduce step-by-step solutions as single continuous blocks of text. It has likely not been trained to answer more specific granular queries about individual steps and thus defaults to providing the numerator of what is typically the final answer.

The following example highlights the fact that while ChatGPT's responses can be very detailed, it very often applies detailed feedback incorrectly:

> **Prompt:** "A student has partially solved the problem $1/4 + 2/3$. They have converted the two fractions into $2/4 + 3/6$. What would you tell a student who has made this mistake?"
>
> ChatGPT's response includes suggestions with four steps, the first of which is:
>
> **ChatGPT:** "**Incorrect Conversion**: Converting $\frac{1}{4}$ to $\frac{2}{4}$ assumes that they multiplied both the numerator and the denominator of $\frac{1}{4}$ by 2. However, the correct conversion to a common denominator for $\frac{1}{4}$ and $\frac{2}{3}$ should be to find the least common multiple of 4 and 3 which is 12."

There is an obvious mistake in this response's diagnosis of the student's error. Only the numerator was multiplied by 2, not the denominator as well. The fact that the model has even attempted to explain the student mistake at this level of detail suggests that it has been trained on examples that produce this very specific kind of feedback, yet in this case, the feedback was applied incorrectly.

ChatGPT did not make it very hard to illustrate these issues. These two prompts were the very first two that I attempted while writing this section, and both had errors. Perhaps in the course of minutes or hours, I might come up with prompts that more reliably produce the behavior I am after. However, the possibility of designing a prompt that works 100% of the time is likely a remote one.

The other three steps that ChatGPT suggests in this second response were without error and included "**Finding the Correct Common Denominator**", addressing the "**Calculation Error**" that $\frac{2}{4} + \frac{3}{6} = 1$, and emphasizing the "**Learning Point**" that the student should start by finding the common denominator.

These are all potentially helpful suggestions. However, it is worth considering whether there is not a more direct way of instructing a potential student than giving them long-winded explanations after they have made several errors. The common approach in model-tracing ITSs is to give immediate step-wise correctness feedback that would, for instance, catch this kind of error as soon as a student placed an incorrectly converted numerator or denominator. It is difficult to say for certain without experimental evidence which of the two approaches would be more effective: letting students play out their mistakes and then explaining those mistakes back to them at length, or simply keeping the student focused on executing a particular solution strategy in a more constrained environment. However, the latter would undoubtedly be more time-efficient for students, which is often a good strategy for optimizing student learning gains.

Of course, not every domain is well-suited to model-tracing-like supports, and we might consider if LLMs could give detailed feedback in these cases. There are some instances in prior literature of tutoring systems that largely diagnose errors in student solutions without relying heavily on within-problem step-by-step scaffolding. Mitrovic's SQL-Tutor [77] is one well-studied prior example along these lines. However, it's difficult to imagine an LLM achieving tutoring capabilities anywhere near the specificity and precision of a system like the SQL-Tutor that relies upon some 600 rules. Where would the training data for such detail-oriented feedback come from?

LLMs consume and reproduce text, information that is intended to be consumed passively. It is inconceivable that a system built upon static content would be able to provide the kinds of detailed cognitive supports that have historically distinguished ITSs from simple content delivery systems and quizzes.

On the other hand, I think there are many exciting hybrid opportunities for augmenting the cognitive supports of a traditional ITS with an LLM. For instance, many of the risks of LLM hallucinations can be mitigated if they are only used to generate static content like hints and feedback messages, that can be checked before they are deployed. Another application is the building of item-specific or behavior-specific detectors that identify predefined patterns in students' natural language responses. Having an LLM give detailed domain-specific feedback to incorrect student responses out of the box is beyond their current capabilities, but fine-tuning them to explicitly identify well-defined classes of errors is considerably more feasible, and if the feedback follows a relatively fixed template then the risks of providing misleading feedback can be mitigated.

8.4 Authoring-By-Programming Support

One possibility that has been explored with CTAT is using ChatGPT-4 to generate the code for rule-based model-tracing ITSs. In this work, an author with expertise in engineering rule-based model-tracing ITSs gave GPT4 examples of rules written in a language called nools [Aleven, 2024 venue pending]. The author described several domain-specific rules in plain English for ChatGPT to translate into nools code to build a CTAT tutoring system. The major downside of this approach is that the author needed to know how to write nools rules for CTAT-based ITSs, and moreover, the specific rules that they wanted to write. They described the contents of these rules to ChatGPT in detailed plain English and succeeded at getting ChatGPT-4 to produce rules that were correct. However, this required a great deal of back-and-forth with the language model where the author needed to describe a considerable number of corrections to the generated outputs. For one domain more than 100 clarifications were needed to write 22 rules.

An often overlooked issue with trying to finesse LLMs to behave properly with prompt engineering is that in order to test and refine a prompt one must already have a desired result in mind. In some cases, a well-written prompt can improve an LLM's performance at automating tedious repetitive tasks. However, when prompt engineering is used to produce very particular single-purpose results, the time required to check the generated output and then engage the LLM in multiple prompt cycles to refine the result can far outweigh the effort that would have been required to just write the generated content by hand.

8.5 Authoring-By-Tutoring Support

AI2T's authoring-by-tutoring approach is decidedly accessible to non-programmers. The author does not need to know anything about how to write rules, only how to solve problems.

Yet inducing programs that authors never explicitly check or edit has its drawbacks. For instance, in the fraction arithmetic domain, one of the preconditions of the induced rules is to check whether or not the starting fractions have the same denominator. It should be trivially obvious, even to a non-programmer, that this precondition needs to be somewhere in the induced program. After all, to solve fraction addition problems all of our participants needed to check for this and apply the appropriate procedure in turn. Authors could surely convey this knowledge directly to an authoring-by-tutoring agent if the agent could understand it, instead of forcing every precondition to be blindly induced from several example problems.

Allowing authors to go under the hood and write code would present many challenges. The authors may know what they want to accomplish, but not how to represent it in AI2T's internal representation language, and where to place the code within the overall induced program. The authoring interface could lighten some of this burden by grounding the writing of the precondition in a context in which it is relevant: within a step of a particular problem, with particular referenceable interface elements, being acted upon by a particular skill. Tools like SUGILITE [59] and Rosie [84], for instance, take this kind of approach where the demonstration of a program is situated in an example. Yet the problem remains for the non-programmer writing the actual precondition in an unfamiliar representation language.

One way LLMs could assist non-programmers with writing preconditions is by translating their natural language instructions: "Convert the two fractions when they have unequal denominators", into code. LLMs could help by querying for situated piece-wise instruction and ask the user: "What are the denominators?", similar to approaches like [84] and [52]. Or the LLM could be used to simply write an example predicate that tests for "unequal" things, and mutual disambiguation with induced predicates could narrow down which two unequal things across problems are most likely the necessary ones to check. Even if induction or LLM translation produces an incorrect predicate, it may provide a natural way for the user to see what a predicate looks like as expressed with reference to the variables of particular skills. LLMs could also translate predicates back to natural language so that non-programmer users can understand induced ones [52].

Of course, I am not suggesting that natural language instruction should replace induction entirely. In the ideal case, they should complement one another. Some balance should be struck between having users describe a program in its entirety and having an AI induce the program in its entirety. A solution at the intersection of these modalities makes for a large space of design considerations.

Another kind of natural language instruction that users could provide is the structure of a hierarchical program. I've described this approach, employed in systems like Instructo-Soar, Rosie [83], SUGILITE [59], and VAL [52] as potentially troublesome in cases where users aim to build very complex programs. However, combined with HTN induction, there may be a middle ground where users describe constraints, features, or sub-routines of a larger program that an inductive system can then connect together. This could avoid the issue with purely top-down instruction where the user must essentially know the hierarchical structure of their intended program from the start, yet still leverage the user's ability to describe elements of the intended program.

Chapter 9

Conclusion

Well beyond prior attempts at implementing authoring-by-tutoring [68, 70, 117] this dissertation has demonstrated a path towards methods of ITS authoring that are both acessible to non-programmers and enable the authoring of flexible and robust ITS behaviors that are typically only implementable with hand-programmed production rules. Prior works with SimStudent and AL have only shown imperfect ITS induction, even in the hands of their creators. By contrast, our evaluations of AI2T show half of our untrained participants, who were mostly non-programmers, succeeding at producing model-tracing complete ITSs with authoring-by-tutoring, and several others achieving nearly 100% model-tracing completeness. Toward this aim, STAND and our HTN induction approach mark major improvements toward building data-efficient and robust simulated learners that can be taught interactively, and our interaction design evaluations show how users can be supported in succeeding at this interactive training.

This work reports on the technical and design elements of AI2T that make 100% model-tracing completeness possible, yet is limited to evaluating just multi-column addition and fraction arithmetic. Focusing on just these two domains maintained some consistency between our two studies and prior work [117], limited participation time to less than 90 minutes, focused on domains that most participants would remember how to do, and limited the number of domains we needed to hand-program in specialized environments in order to evaluate model-tracing completeness against a ground-truth program (this is not general requirement of authoring, just for our evaluations). However, AI2T is not necessarily limited to these two domains, and its capabilities may be considerably broadened with the addition of a few additional features.

9.1 Potential Scope of Tutoring System Domains

Prior work with Sierra, SimStudent, and the Apprentice Learner, which learn in a similar manner as AI2T, have trained agents on dozens of domains including algebra, stoichiometry, geometry, simple linguistic tasks like Chinese character translation and article selection, a few games, and other arithmetic tasks like subtraction and multiplication. Many of these domains have control structures that could only typically be programmed in production rule-based implementations. AI2T may well succeed where these prior systems have fallen short of inducing correct and complete behavior. STAND is more data-efficient than the precondition induction approaches used in those systems, and AI2T can learn HTN-based control structures that may more naturally capture their control behaviors.

In general, AI2T is designed for authoring procedural domains similar to those typically targeted by rule-based model-tracing tutors like Cognitive Tutors [39]. These domains tend to have well-defined correct and incorrect solution paths with distinct steps, and with AI2T's rule induction approach these steps can vary based on the context of the problem or student solution strategy instead of being fixed within the structure of static sequences or graphs.

Naturally science and math skills tend to fall into this category of procedural skills. However, any domain that involves procedural decision making and can be expressed symbolically within an interface is a good candidate. Domains like essay grading that permit unconstrained natural text responses are out of scope of AI2T's abilities. However, there are certainly possibilities for using systems better suited to these kinds of assessments in combination with AI2T. AI2T may add value in these cases if the free response pieces needed to be situated within some procedural decision making process that AI2T could help induce.

9.2 Who can use AI2T?

The major aim of this work was to prototype a method whereby the authoring of complex ITSs is made simple, fast, and accessible to non-programmers. The typical users of an ITS authoring tool include instructional designers, learning-engineers, teachers, and researchers—all professionals who may or may not have programming expertise, but who all certainly benefit from being able to author ITSs quickly. Many of our participants succeeded at authoring complete model-tracing behavior in less than half an hour. Studies of in-service professionals working in a more open-ended authoring study may provide interesting design perspectives that this work has overlooked. However, in terms of usability, there is every reason to believe that the typical users of an ITS authoring tool should have similar success using AI2T as our mostly graduate student participants.

9.3 Interface Design and The Design Loop

One initial challenge of authoring a domain is building an interface for it. In our studies we provided interfaces for our users that were each ready-made for a particular approach to step-by-step problem solving. ITS authoring typically requires planning the design of an interface around its step-by-step support, and these designs may need to be revised as they are tested with students. This cyclic process of revision has been referred to as the design loop of ITS authoring [3]. AI2T does not provide a solution for building HTML interfaces, although it can certainly use interfaces built with any of the several options already implemented for other ITS authoring tools [2], including tools where interfaces can be built from natural language descriptions [15].

AI2T may play a beneficial role in the ITS design loop as it tends to learn more effectively from granular step-by-step solutions. Step-by-step scaffolding tends to benefit student learning, especially for students with low prior knowledge since step-by-step problem solving tends to reduce cognitive load and provide more opportunities in the ITS for adapting to student mistakes [37]. Instead of authors relying on classroom studies to assess if their cognitive support is appropriate, AI2T can be used as a simulated student that can partially guide this process. It is easier for AI2T to learn procedures in a step-by-step display-based manner where steps are granular and can be easily self-explained from its current function library, and where intermediate results are visible for use in later steps. For instance, the zero-carry version of multi-column addition that we used in study 1 can be trained with fewer examples than the normal algorithm (like study 2) where zeros are never carried. Since AI2T learns entirely bottom-up from authors' instruction, what works well for AI2T may very well work well for low prior knowledge students, and authoring with AI2T may encourage authors to build tutoring systems with more fine-grained support than they may otherwise consider on their first design attempts.

9.4 Future Work for Broader Authoring with AI2T

One challenge of using AI2T for general-purpose authoring is that it needs to have primitives in its function library to explain and generalize from users' demonstrated actions. Multi-column addition and fraction arithmetic can be authored with a library that has just a few arithmetic primitives like adding, multiplying, and isolating ones and tens digits from integers. Yet, many domains would require additional pre-built or author-programmed primitives for AI2T to compose into complete programs. There is some precedent for authors learning to write these primitives fairly quickly. At the LearnLab summer school, a week-long workshop on edTech tools developed at CMU, we have had several dozen participants succeed at building interfaces (with drag and drop tools) and writing primitive functions for a wide array of domains that they then succeeded at training a pre-cursor to AI2T on, in the course of about 2 dedicated work days of deliberate effort. These summer school participants have taught agents to carry out algorithms taught in CS curricula, special approaches to arithmetic, pre-algebra procedures, and skills in music theory, just to name a few. Of course, when it comes to writing primitives for complex domains, the scope of what authors can build depends on their prior programming experience. LLM's code generation capabilities could go a long way toward speeding up and helping less-experienced users with writing these primitive functions. Authors' demonstrations could also provide a natural means of unit testing these short sub-routines, and interface affordances could be designed around these features.

Adding yet more performance and learning mechanisms to AI2T could also expand the scope of what it can be used to author with simple primitive functions. For instance, Li et. al. added a representation learning mechanism to SimStudent [57] that greatly simplified the primitive functions required for it to learn algebra equation solving. The representation learning mechanism acquired a grammar for parsing short algebraic expressions, which eliminated the need to hand-write domain-specific primitives, like ones for extracting constants and coefficients from expressions. A limitation of this work was that the representation learning mechanism needed to be pretrained offline, not interactively. A representation learning mechanism that learns interactively and in tandem with procedure induction is an open, and potentially very difficult problem (perhaps PhD dissertation worthy). There are other opportunities along these lines for using pretrained systems like LLMs to parse content like text and images into a structured representation that AI2T can reason over. These features could extend AI2T's authoring scope to include domains with word problems and problems that involve reasoning over figures (this would be a relatively quick project to prototype).

Additionally, some minor improvements in AI2T's process-learning mechanism, like the ability to induce recursive HTNs, would go a long way toward broadening the generality of problem types that can be authored. For instance, in this work, we tested 3-by-3 digit multi-column addition, but recursive HTN induction would naturally extend to arbitrary addition problems, and assist in inducing many other domains as well (this could likely be perfected in a matter of months).

9.5 Other Time-Saving Benefits

Moreover, AI2T's authoring-by-tutoring approach could in principle replicate and speed up the authoring of domains similar to those that have been built with tools like CTAT example-tracing or OATutor that only allow for much simpler control structures. After an initial set of demonstrations, AI2T mostly just requires authors to check problem solutions. Authors do not necessarily need to solve every problem that they mass produce, and since AI2T has the potential to induce complete programs from a handful of examples, authors may not even need to check every problem they generate. As we've shown in this work, this benefit applies to building the core model-tracing features of an ITS like correctness evaluations of step-by-step actions, and the generation of "bottom-out" hints that produce next-step answers. However, in principle, AI2T's ability to induce rule-based programs can partially assist with some of the effort of authoring additional ITS features. For instance, it is much easier to write a few templates for conceptual hints and feedback messages that are associated with particular rules, than it is to write unique hints and messages problem-by-problem or generate messages with LLMs problem-by-problem and check them one at a time [94]. Automatically inducing rules for a domain also naturally produces a domain model, a model of the skills needed to master a domain—something which is typically hand-built or derived in a data-driven manner, and used in knowledge tracers that track student mastery [55].

9.6 Additional Elements of Feature Complete Tutoring Systems

The model-tracing behavior that AI2T induces in this work suffices for enabling automatic correctness feedback (e.g., marking actions as correct or not) and bottom-out hints (hints that give the answer for individual steps). Beyond this, authoring a typical fully featured model-tracing ITS would additionally involve:

1. Building a tutoring interface. This can be done with drag-and-drop UI tools like the one for CTAT [2] among others [15, 107].
2. Writing step-specific hints and feedback messages that go along with each induced skill.
3. Teaching AI2T several buggy skills to adapt to common mistakes. Currently AI2T has no explicit interaction method for this. Although, the interface design for doing this could be as simple as introducing a toggler to flag demonstrations or skills as buggy. The backend AI would simply need to be adjusted to explicitly ignore these buggy skills in some instances.
4. Assigning knowledge component labels to steps to enable knowledge tracing. Prior work has demonstrated that simulated learners can automate this process quite well [58]. Each independent application of an induced skill can serve as step-to-knowledge component mapping.

9.7 Considerations for Editing, Reusing, and on Building Upon Existing ITSs

The benefits of authoring ITSs quickly are numerous. ITSs can quickly become obsolete by shifts in technology. For instance, changing trends in programming language popularity and the shift from desktop applications to on-demand web applications have produced rapid changes in the tools used to program and deliver ITSs, often resulting in their reimplementation. Making ITS authoring easier may also enable more education technology to be like ITSs and provide fine-grained cognitive support. Additionally perfecting adaptive instruction requires researchers to implement multiple versions of tutoring systems so they can be tested against each other in A/B studies. All of these use cases benefit from tools for authoring ITSs quickly from the ground-up.

Yet, another possibility is that end-users like teachers and researchers may want to tweak existing ITSs to specialize them for their own purposes. For instance, a teacher may want to add, remove, or replace the solution strategies supported by an already-built ITS. In principle, this could be facilitated in AI2T by interface affordances that allow users to simply select and remove entire methods or primitive skills—a features which would also be helpful for cleaning up incorrect skills induced from erroneous demonstrations in normal authoring. Instead of just allowing for the removal of demonstrations, which AI2T already supports, this feature could remove whole skills and all of their supporting examples. To add new skills to an existing ITS, authors could simply demonstrate new solutions as needed. In principle, AI2T's process-learning mechanism reduces much of the complexity of this. With induced HTNs multiple strategies are naturally expressed as multiple disjoint methods of the same task, and thus new methods can be easily inserted by simply demonstrating them wherever they should be applied. In general, this should rarely interfere with the rest of the HTN structure.

An author may also want to make edits to an ITS or reuse skills that AI2T learned when making another ITS. In these cases, authors may want to use induced skills in interfaces different than the ones they were learned in. One compelling application along these lines is taking higher-order skills (e.g., methods in an HTN) learned in an interface with fine-grained steps, and compressing them into skills that perform more complex multi-step actions without placing intermediate results in a display. Toward making more complex tutors from simpler ones, and for supporting curricula with faded scaffolding in which step-by-step supports are removed as students gain greater mastery [101], chunking together and reusing composite skills is a desirable feature. Composing skills is fairly straightforward, although robustly generalizing skills to work in new interfaces is somewhat more difficult, and potentially error-prone. In principle, where-learning already provides a solution for cross-interface generalization, but we have not evaluated it extensively along these lines.

From an author's perspective, cross-interface transfer could work in multiple ways. In the ideal case the AI agent could propose applications of its current skills in new interfaces by automatically recognizing their applicability. In a less desirable case, the author may need to demonstrate the skill again in the new interface and have AI2T use where-learning to abduce how the new demonstrations map onto a prior skill, and how this mapping will generalize the where-part constraints for that skill. In the case where the agent proposes its own candidates for skill transfer, each proposed skill application may need to violate some of the skill's current where-part constraints. This requires applying partial matching—something AI2T already has a good implementation for, but which has some potential to cause erratic behavior. Partial matching is typically more computationally expensive than fully constrained matching and can generate a large number of candidate matches. So there are likely subproblems that need to be solved to find the sweet spot between helpful suggestions for skill transfer, and erratic behavior where the agent produces many spurious suggestions. Inputs from a user like hints about when skill transfer should be attempted may help with this.

9.8 Broader Applicability

In the broader context of interactive task learning, the innovations of this work—including its novel machine learning algorithms and interaction design—open up possibilities that could be applicable in robotics, web automation, and human-AI collaboration. In the last few decades, the field of machine learning has had a heavy focus on data-driven learning. More recently AI research has focused heavily on finding ways to utilize pretrained generative models. A smaller group of researchers have focused methods of ITL where AI capabilities are learned directly from human instruction [46]. However, many ITL approaches envision interactions where a user gives instructions to an agent in a mostly top-down manner. The objectives of a program are stated, and then the user describes steps to achieving that objective. By analogy to human learning, instructing an agent in this top-down manner is like learning from being told—an approach to learning that often produces poor learning outcomes in humans compared to learning-by-doing.

AI2T, like other simulated learners before it, simulates learning-by-doing in the context of supervised tutoring interactions situated in practice, including demonstrations and correctness feedback on attempted actions. This work is an example where the kinds of instruction that are often effective for human learners are employed effectively in tutoring an artificial learner. The consideration of looking to human learning as an inspiration for designing better machine learning and interactions for teaching artificial learners is one that has fallen out of practice in recent decades, perhaps in part because of the rise of data-driven methods like deep reinforcement learning (a learning-by-doing approach of sorts) that are decidedly not human-like and orders of magnitude less data-efficient than human learners. The fact that we achieve data-efficient and robust induction opens a number of considerations for improving systems that currently operate by reinforcement-learning, and for broadening the kinds of interactions that can be used when training agents interactively.

References

1. Aleven, V., McLaren, B., Roll, I., Koedinger, K.: Toward tutoring help seeking: Applying cognitive modeling to meta-cognitive skills. In: Intelligent Tutoring Systems: 7th International Conference, ITS 2004, Maceió, Alagoas, Brazil, August 30-September 3, 2004. Proceedings 7. pp. 227–239. Springer (2004)
2. Aleven, V., McLaren, B.M., Sewall, J., Van Velsen, M., Popescu, O., Demi, S., Ringenberg, M., Koedinger, K.R.: Example-tracing tutors: Intelligent tutor development for non-programmers. International Journal of Artificial Intelligence in Education **26**(1), 224–269 (2016)
3. Aleven, V., McLaughlin, E.A., Glenn, R.A., Koedinger, K.R.: Instruction based on adaptive learning technologies. Handbook of research on learning and instruction **2**, 522–560 (2016)
4. Anderson, J.R., Boyle, C.F., Corbett, A.T., Lewis, M.W.: Cognitive modeling and intelligent tutoring (1990)
5. Anderson, J.R., Corbett, A.T., Koedinger, K.R., Pelletier, R.: Cognitive tutors: Lessons learned. The journal of the learning sciences **4**(2), 167–207 (1995)
6. Arora, A., Fiorino, H., Pellier, D., Métivier, M., Pesty, S.: A review of learning planning action models. The Knowledge Engineering Review **33**, e20 (2018)
7. Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine **34**(6), 26–38 (2017)
8. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)
9. Blessing, S.B.: A programming by demonstration authoring tool for model-tracing tutors. International Journal of Artificial Intelligence in Education **8**, 233–261 (1997)
10. Bloom, B.S.: The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. Educational researcher **13**(6), 4–16 (1984)
11. Breiman, L.: Bagging predictors. Machine learning **24**, 123–140 (1996)
12. Breiman, L.: Random forests. Machine learning **45**, 5–32 (2001)

13. Breiman, L.: Classification and Regression Trees. Routledge (2017)
14. Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y.T., Li, Y., Lundberg, S., et al.: Sparks of artificial general intelligence: Early experiments with gpt-4. arXiv preprint arXiv:2303.12712 (2023)
15. Calo, T., MacLellan, C.J.: Towards educator-driven tutor authoring: Generative ai approaches for creating intelligent tutor interfaces. arXiv preprint arXiv:2405.14713 (2024)
16. Chen, K., Srikanth, N.S., Kent, D., Ravichandar, H., Chernova, S.: Learning hierarchical task networks with preferences from unannotated demonstrations. In: Conference on Robot Learning. pp. 1572–1581. PMLR (2021)
17. Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. pp. 785–794 (2016)
18. Chi, M.T., Wylie, R.: The icap framework: Linking cognitive engagement to active learning outcomes. Educational psychologist **49**(4), 219–243 (2014)
19. Clark, R.E., Feldon, D.F., Van Merrienboer, J.J., Yates, K.A., Early, S.: Cognitive task analysis. In: Handbook of research on educational communications and technology, pp. 577–593. Routledge (2008)
20. Codel, C., Koedinger, K., Harpstead, E.: Revisiting rumbleblocks with apprentice learning: Examining and learning gameplay with al (2020)
21. Conati, C., VanLehn, K.: Teaching meta-cognitive skills: Implementation and evaluation of a tutoring system to guide self-explanation while learning from examples. In: Artificial intelligence in education. pp. 297–304. IOS Press (1999)
22. Cypher, A., Halbert, D.C.: Watch What I Do: Programming by Demonstration. MIT press (1993)
23. Erol, K., Hendler, J.A., Nau, D.S.: Semantics for hierarchical task-network planning. Citeseer (1994)
24. Erol, K., Hendler, J.A., Nau, D.S.: Umcp: A sound and complete procedure for hierarchical task-network planning. In: Aips. vol. 94, pp. 249–254 (1994)
25. Gao, R., Merzdorf, H.E., Anwar, S., Hipwell, M.C., Srinivasa, A.: Automatic assessment of text-based responses in post-secondary education: A systematic review. Computers and Education: Artificial Intelligence p. 100206 (2024)
26. Graesser, A.C., Lu, S., Jackson, G.T., Mitchell, H.H., Ventura, M., Olney, A., Louwerse, M.M.: Autotutor: A tutor with dialogue in natural language. Behavior Research Methods, Instruments, & Computers **36**, 180–192 (2004)
27. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Communications of the ACM **55**(8), 97–105 (2012)
28. Hecht-Nielsen, R.: Theory of the backpropagation neural network. In: Neural networks for perception, pp. 65–93. Elsevier (1992)
29. Heffernan, N.T., Heffernan, C.L.: The assistments ecosystem: Building a platform that brings scientists and teachers together for minimally invasive research on human learning and teaching. International Journal of Artificial Intelligence in Education **24**(4), 470–497 (2014)
30. Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., Steinhardt, J.: Measuring mathematical problem solving with the math dataset. arXiv preprint arXiv:2103.03874 (2021)
31. Hirsh, H.: Polynomial-time learning with version spaces. In: AAAI. pp. 117–122 (1992)
32. Hogg, C., Munoz-Avila, H., Kuter, U.: Htn-maker: Learning htns with minimal additional knowledge engineering required. In: AAAI. pp. 950–956 (2008)
33. Huffman, S.B., Laird, J.E.: Flexibly instructable agents. Journal of Artificial Intelligence Research **3**, 271–324 (1995)
34. Ji, Z., Yu, T., Xu, Y., Lee, N., Ishii, E., Fung, P.: Towards mitigating llm hallucination via self reflection. In: Findings of the Association for Computational Linguistics: EMNLP 2023. pp. 1827–1843 (2023)
35. Kaliszyk, C., Urban, J., Michalewski, H., Olšák, M.: Reinforcement learning of theorem proving. Advances in Neural Information Processing Systems **31** (2018)
36. Klenk, M., Forbus, K.: Analogical model formulation for transfer learning in ap physics. Artificial intelligence **173**(18), 1615–1638 (2009)
37. Koedinger, K.R., Aleven, V.: Exploring the assistance dilemma in experiments with cognitive tutors. Educational Psychology Review **19**, 239–264 (2007)
38. Koedinger, K.R., Carvalho, P.F., Liu, R., McLaughlin, E.A.: An astonishing regularity in student learning rate. Proceedings of the National Academy of Sciences **120**(13), e2221311120 (2023)
39. Koedinger, K.R., Corbett, A., et al.: Cognitive tutors: Technology bringing learning sciences to the classroom. na (2006)
40. Koedinger, K.R., Corbett, A.T., Perfetti, C.: The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning. Cognitive science **36**(5), 757–798 (2012)
41. Koedinger, K.R., Kim, J., Jia, J.Z., McLaughlin, E.A., Bier, N.L.: Learning is not a spectator sport: Doing is better than watching for learning from a mooc. In: Proceedings of the second (2015) ACM conference on learning@ scale. pp. 111–120 (2015)

42. Kohavi, R., Kunz, C.: Option decision trees with majority votes. In: ICML. vol. 97, pp. 161–169 (1997)
43. Konda, V.R., Borkar, V.S.: Actor-critic–type learning algorithms for markov decision processes. SIAM Journal on control and Optimization **38**(1), 94–123 (1999)
44. Kulik, J.A., Fletcher, J.: Effectiveness of intelligent tutoring systems: a meta-analytic review. Review of educational research **86**(1), 42–78 (2016)
45. Laird, J.E.: The Soar cognitive architecture. MIT press (2019)
46. Laird, J.E., Gluck, K., Anderson, J., Forbus, K.D., Jenkins, O.C., Lebiere, C., Salvucci, D., Scheutz, M., Thomaz, A., Trafton, G., Wray, R.E., Mohan, S., Kirk, J.R.: Interactive Task Learning. IEEE Intelligent Systems **32**(4), 6–21 (2017). https://doi.org/10.1109/MIS.2017.3121552, http://ieeexplore.ieee.org/document/8012335/
47. Laird, J.E., Lebiere, C., Rosenbloom, P.S.: A standard model of the mind: Toward a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. Ai Magazine **38**(4), 13–26 (2017)
48. Lample, G., Charton, F.: Deep learning for symbolic mathematics. arXiv preprint arXiv:1912.01412 (2019)
49. Langley, P.: Learning hierarchical problem networks for knowledge-based planning. In: International Conference on Inductive Logic Programming. pp. 69–83. Springer (2022)
50. Larkin, J.H.: Display-based problem solving. Complex information processing: The impact of Herbert A. Simon pp. 319–341 (1989)
51. Lau, T., Wolfman, S.A., Domingos, P., Weld, D.S.: Programming by demonstration using version space algebra. Machine Learning **53**(1), 111–156 (2003)
52. Lawley, L., Maclellan, C.: Val: Interactive task learning with gpt dialog parsing. In: Proceedings of the CHI Conference on Human Factors in Computing Systems. pp. 1–18 (2024)
53. Lee, G.G., Latif, E., Wu, X., Liu, N., Zhai, X.: Applying large language models and chain-of-thought for automatic scoring. Computers and Education: Artificial Intelligence **6**, 100213 (2024)
54. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., et al.: Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems **33**, 9459–9474 (2020)
55. Li, N., Cohen, W.W., Koedinger, K.R., Matsuda, N.: A machine learning approach for automatic student model discovery. In: Edm. pp. 31–40. ERIC (2011)
56. Li, N., Cushing, W., Kambhampati, S., Yoon, S.: Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences. ACM Transactions on Intelligent Systems and Technology (TIST) **5**(2), 1–32 (2014)
57. Li, N., Matsuda, N., Cohen, W.W., Koedinger, K.R.: Integrating representation learning and skill learning in a human-like intelligent agent. Artificial Intelligence **219**, 67–91 (2015)
58. Li, N., Stampfer, E., Cohen, W., Koedinger, K.: General and efficient cognitive model discovery using a simulated student. In: Proceedings of the Annual Meeting of the Cognitive Science Society. vol. 35 (2013)
59. Li, T.J.J., Azaria, A., Myers, B.A.: SUGILITE: Creating multimodal smartphone automation by demonstration. In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. pp. 6038–6049. ACM (2017)
60. Li, T.J.J., Labutov, I., Li, X.N., Zhang, X., Shi, W., Ding, W., Mitchell, T.M., Myers, B.A.: Appinite: A multi-modal interface for specifying data descriptions in programming by demonstration using natural language instructions. In: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 105–114. IEEE (2018)
61. Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., Cobbe, K.: Let's verify step by step. arXiv preprint arXiv:2305.20050 (2023)
62. Lin, J., Chen, E., Han, Z., Gurung, A., Thomas, D.R., Tan, W., Nguyen, N.D., Koedinger, K.R.: How can i improve? using gpt to highlight the desired and undesired parts of open-ended responses. arXiv preprint arXiv:2405.00291 (2024)
63. Long, Y., Aleven, V.: Educational game and intelligent tutoring system: A classroom study and comparative design analysis. ACM Transactions on Computer-Human Interaction (TOCHI) **24**(3), 1–27 (2017)
64. MacLellan, C.J.: Computational Models of Human Learning: Applications for Tutor Development, Behavior Prediction, and Theory Testing. Ph.D. thesis, Carnegie Mellon University (2017)
65. MacLellan, C.J., Gupta, A.: Learning expert models for educationally relevant tasks using reinforcement learning. International Educational Data Mining Society (2021)
66. Maclellan, C.J., Harpstead, E., Patel, R., Koedinger, K.R.: The Apprentice Learner Architecture: Closing the loop between learning theory and educational data. International Educational Data Mining Society (2016)
67. MacLellan, C.J., Harpstead, E., Wiese, E.S., Zou, M., Matsuda, N., Aleven, V., Koedinger, K.R.: Authoring tutors with complex solutions: A comparative analysis of example tracing and simstudent. In: AIED workshops (2015)
68. MacLellan, C.J., Koedinger, K.R.: Domain-general tutor authoring with apprentice learner models. International Journal of Artificial Intelligence in Education pp. 1–42 (2020)

69. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., De Raedt, L.: Deepproblog: Neural probabilistic logic programming. Advances in neural information processing systems **31** (2018)
70. Matsuda, N., Cohen, W.W., Koedinger, K.R.: Teaching the teacher: Tutoring simstudent leads to more effective cognitive tutor authoring. International Journal of Artificial Intelligence in Education **25**(1), 1–34 (2015)
71. Mayer, R.E.: Should there be a three-strikes rule against pure discovery learning? American psychologist **59**(1), 14 (2004)
72. McDaniel, R.G., Myers, B.A.: Gamut: Demonstrating whole applications. In: Symposium on User Interface Software and Technology: Proceedings of the 10 th annual ACM symposium on User interface software and technology. vol. 14, pp. 81–82 (1997)
73. McNeill, K.L., Lizotte, D.J., Krajcik, J., Marx, R.W.: Supporting students' construction of scientific explanations by fading scaffolds in instructional materials. The journal of the Learning Sciences **15**(2), 153–191 (2006)
74. Meyersa, P., Hana, A., Grewala, R., Potnisa, M., Stampera, J.: Focal: A proposed method of leveraging llms for automating assessments
75. Mitchell, T.M.: Generalization as search. Artificial Intelligence **18**(2), 203–226 (1982)
76. Mitchell, T.M.: Version spaces: an approach to concept learning. Tech. rep., STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE (1978)
77. Mitrovic, A.: An intelligent sql tutor on the web. International Journal of Artificial Intelligence in Education **13**(2-4), 173–197 (2003)
78. Mitrovic, A.: Modeling domains and students with constraint-based modeling. In: Advances in intelligent tutoring systems, pp. 63–80. Springer (2010)
79. Mitrovic, A., Ohlsson, S., Barrow, D.K.: The effect of positive feedback in a constraint-based intelligent tutoring system. Computers & Education **60**(1), 264–272 (2013)
80. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
81. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. nature **518**(7540), 529–533 (2015)
82. Modran, H., Bogdan, I.C., Ursuțiu, D., Samoila, C., Modran, P.L.: Llm intelligent agent tutoring in higher education courses using a rag approach (2024)
83. Mohan, S., Laird, J.: Learning goal-oriented hierarchical tasks from situated interactive instruction. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 28 (2014)
84. Mohan, S., Mininger, A., Kirk, J., Laird, J.E.: Learning grounded language through situated interactive instruction. In: 2012 AAAI Fall Symposium Series (2012)
85. Mohseni-Kabir, A., Chernova, S., Rich, C.: Collaborative learning of hierarchical task networks from demonstration and instruction. In: 2014 AAAI Fall Symposium Series (2014)
86. Moore, S., Nguyen, H.A., Bier, N., Domadia, T., Stamper, J.: Assessing the quality of student-generated short answer questions using gpt-3. In: European conference on technology enhanced learning. pp. 243–257. Springer (2022)
87. Myers, B.A.: Creating user interfaces by demonstration. Academic Press Professional, Inc. (1988)
88. Nejati, N., Langley, P., Konik, T.: Learning hierarchical task networks by observation. In: Proceedings of the 23rd international conference on Machine learning. pp. 665–672 (2006)
89. Neumann, M., Rauschenberger, M., Schön, E.M.: "we need to talk about chatgpt": The future of ai and higher education. In: 2023 IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG). pp. 29–32. IEEE (2023)
90. Neves, D.M.: Learning procedures from examples and by doing. In: IJCAI. pp. 624–630. Citeseer (1985)
91. Nguyen, C., Reifsnyder, N., Gopalakrishnan, S., Munoz-Avila, H.: Automated learning of hierarchical task networks for controlling minecraft agents. 226–231 (2017)
92. Nguyen-Tuong, D., Peters, J.: Model learning for robot control: a survey. Cognitive processing **12**, 319–340 (2011)
93. Nwana, H.S.: Intelligent tutoring systems: an overview. Artificial Intelligence Review **4**(4), 251–277 (1990)
94. Pardos, Z.A., Bhandari, S.: Chatgpt-generated help produces learning gains equivalent to human tutor-authored help on mathematics skills. Plos one **19**(5), e0304013 (2024)
95. Pardos, Z.A., Tang, M., Anastasopoulos, I., Sheel, S.K., Zhang, E.: Oatutor: An open-source adaptive tutoring system and curated content library for learning sciences research. In: Proceedings of the 2023 chi conference on human factors in computing systems. pp. 1–17 (2023)
96. Patel, R., Liu, R., Koedinger, K.R.: When to block versus interleave practice? evidence against teaching fraction addition before fraction multiplication. In: CogSci (2016)
97. Poesia, G., Dong, W., Goodman, N.: Contrastive reinforcement learning of symbolic reasoning domains. Advances in neural information processing systems **34**, 15946–15956 (2021)

98. Puntambekar, S., Hubscher, R.: Tools for scaffolding students in a complex learning environment: What have we gained and what have we missed? Educational psychologist **40**(1), 1–12 (2005)
99. Quinlan, J.R.: Learning first-order definitions of functions. Journal of artificial intelligence research **5**, 139–161 (1996)
100. Rau, M.A., Aleven, V., Rummel, N.: Intelligent tutoring systems with multiple representations and self-explanation prompts support learning of fractions. In: Artificial intelligence in education. pp. 441–448. Ios Press (2009)
101. Reiser, B.J.: Scaffolding complex learning: The mechanisms of structuring and problematizing student work. In: Scaffolding, pp. 273–304. Psychology Press (2018)
102. Reynolds, L., McDonell, K.: Prompt programming for large language models: Beyond the few-shot paradigm. In: Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems. pp. 1–7 (2021)
103. Ritter, F.E., Tehranchi, F., Oury, J.D.: Act-r: A cognitive architecture for modeling cognition. Wiley Interdisciplinary Reviews: Cognitive Science **10**(3), e1488 (2019)
104. Schmucker, R., Xia, M., Azaria, A., Mitchell, T.: Ruffle &riley: Insights from designing and evaluating a large language model-based conversational tutoring system. In: International Conference on Artificial Intelligence in Education. pp. 75–90. Springer (2024)
105. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
106. Settles, B.: Active learning literature survey (2009)
107. Smith, G., Gupta, A., MacLellan, C.: Apprentice tutor builder: A platform for users to create and personalize intelligent tutors. arXiv preprint arXiv:2404.07883 (2024)
108. Tate, T., Doroudi, S., Ritchie, D., Xu, Y., Warschauer, M.: Educational research and ai-generated writing: Confronting the coming tsunami. EdArXiv. January **10** (2023)
109. Tu, K., Pavlovskaia, M., Zhu, S.C.: Unsupervised structure learning of stochastic and-or grammars. Advances in neural information processing systems **26** (2013)
110. VanLehn, K.: Learning one subprocedure per lesson. Artificial Intelligence **31**(1), 1–40 (1987)
111. VanLehn, K.: Mind bugs: The origins of procedural misconceptions. MIT press (1990)
112. VanLehn, K.: The behavior of tutoring systems. International Journal of Artificial Intelligence in Education **16**(3), 227–265 (2006)
113. VanLehn, K.: The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. Educational Psychologist **46**(4), 197–221 (2011)
114. VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., Wintersgill, M.: The andes physics tutoring system: Lessons learned. International Journal of Artificial Intelligence in Education **15**(3), 147–204 (2005)
115. VanLehn, K., Siler, S., Murray, C., Yamauchi, T., Baggett, W.B.: Why do only some events cause learning during human tutoring? Cognition and Instruction **21**(3), 209–249 (2003)
116. Waterman, D.A.: A guide to expert systems. Addison-Wesley Longman Publishing Co., Inc. (1985)
117. Weitekamp, D., Harpstead, E., Koedinger, K.: An interaction design for machine teaching to develop ai tutors. CHI (2020 in press)
118. Weitekamp, D., Harpstead, E., Koedinger, K.: Toward stable asymptotic learning with simulated learners. In: International Conference on Artificial Intelligence in Education. pp. 390–394. Springer (2021)
119. Weitekamp, D., Rachatasumrit, N., Wei, R., Harpstead, E., Koedinger, K.: Simulating learning from language and examples. In: International Conference on Artificial Intelligence in Education. pp. 580–586. Springer (2023)
120. Weitekamp, D., Ye, Z., Rachatasumrit, N., Harpstead, E., Koedinger, K.: Investigating differential error types between human and simulated learners. In: International Conference on Artificial Intelligence in Education. pp. 586–597. Springer (2020)
121. Welleck, S., West, P., Cao, J., Choi, Y.: Symbolic brittleness in sequence models: on systematic generalization in symbolic mathematics. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 8629–8637 (2022)
122. White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C.: A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382 (2023)
123. Xiao, Z., Zhang, D.: A deep reinforcement learning agent for geometry online tutoring. Knowledge and Information Systems **65**(4), 1611–1625 (2023)
124. Yates, K., Sullivan, M., Clark, R.: Integrated studies on the use of cognitive task analysis to capture surgical expertise for central venous catheter placement and open cricothyrotomy. The American journal of surgery **203**(1), 76–80 (2012)
125. Yuan, X., Wang, T., Wang, Y.H., Fine, E., Abdelghani, R., Lucas, P., Sauzéon, H., Oudeyer, P.Y.: Selecting better samples from pre-trained llms: A case study on question generation. arXiv preprint arXiv:2209.11000 (2022)

126. Zhu, X.: Machine teaching: An inverse problem to machine learning and an approach toward optimal education. In: Proceedings of the AAAI conference on artificial intelligence. vol. 29 (2015)