Simulating Learning From Language and Examples

No Author Given

No Institute Given

Abstract. Simulations of human learning have the potential to revolutionize several active areas of educational technology research including student modeling and the authoring of intelligent tutoring systems. Current simulated students learn predominantly from worked examples and correctness feedback, but not natural language instruction. We present a method for simulating how students learn individual mathematical skills from narrated tutorial instruction—a multi-modal combination of natural language and worked examples. We simulate the process of a student imprecisely interpreting an instructor's verbal instructions, and then using an accompanying worked example to resolve ambiguities in the instructor's language. We find that our system and an alternative approach using Github Copilot are more accurate at synthesizing mathematical formulae from from crowd-worker generated instructional utterances using a combination of both hints and examples, than from either form of instruction taken alone. For the purposes of authoring and student modeling our system has practical benefits over Copilot and slightly better accuracy.

Keywords: Learning Simulations · Computational Models of Learning · Natural Language Processing · Code Synthesis · Authoring Tools

1 Introduction

This work details a method for simulating learning from narrated tutorial instruction and has applications for adding natural language understanding to computational models of learning and for building non-programmer friendly Intelligent Tutoring System (ITS) authoring tools. The presented system utilizes a multi-modal combination of natural language instruction and worked examples to synthesize formulae for mathematical concepts. For applications of modeling student learning this capability extends the scope of instructional interactions from which simulated students can learn [7]. For the sake of aiding ITS authoring, this capability enables an input modality whereby non-programmers can verbally instruct a simulated learner to automatically synthesize functions that produce next-step hints and grading behavior in an ITS.

1.1 Student Modeling

Modeling student learning is an important subject in the learning sciences. The vast majority of student models are built by fitting to student data, modeling learning as a change in student performance as a function of time, practice opportunities, or other features [12]. These statistical modeling approaches can be useful for choosing ideal next problems for student practice [12], and for indicating places for improvement in instructional technologies [2]. They however, cannot directly reveal precise information about what instructional changes may improve learning materials and, moreover, describe very little on their own about the nuances of student cognition.

Computational models of learning are an avenue of research with a great deal of potential for pursuing a much deeper understanding of human learning. These methods use simulated students that directly experience the same instructional opportunities as human students, and induce executable skills from the available hints and correctness feedback. Instead of modeling performance as a change in scalar probability, simulated learners have underlying reasons for the actions they take. At any point in learning their knowledge has a precise representation that can produce both correct next actions and errors. Prior work has suggested that computational models of learning can be used to test the efficacy of instructional technology a priori, and to test competing cognitive theories of learning against human data [19], in addition to many other use cases [17].

Many simulated students implement rather complex bottom-up AI [8][9][15], yet have been limited in the forms of instruction they can utilize [7]. Current systems learn predominantly from worked examples and correctness feedback. This work expands the set of instructional modalities that simulated students can learn from to include natural language explanations that accompany worked examples.

1.2 ITS Authoring with Simulated Learners

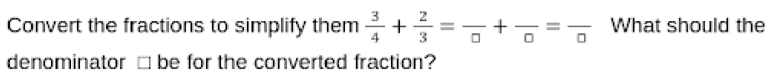
Prior work has also demonstrated that simulated learners hold the potential to dramatically speed up the process of building Intelligent Tutoring Systems (ITSs). ITSs are characterized by fine-grained step-by-step feedback, next-step hints, and adaptive problem selection [16], and have in some cases shown greater learning gains than human-to-human tutoring [5]. The power of ITSs lies in the comprehensiveness of their adaptivity, making for costly development times—about 200-300 hours of development time per hour of instruction. While non-programmer friendly tools like example-tracing cut this ratio down to 100:1 or less [1], preliminary studies of simulated learner based tools have been shown to be 7x faster than example-tracing for authoring of at least one domain [18]. Even popular GUI-based tools like example-tracing require programming based interactions to utilize some features. By contrast, simulated student based authoring tools strive to operate purely by natural tutoring interactions [7], which may allow authors to build more elaborate and adaptive tutoring systems efficiently without programming, opening up ITS authoring to a wider set of users.

A major barrier toward building a general purpose ITS authoring tool with simulated learners is the use of an often intractable form of brute-force search used to drive a learning mechanism called *how-learning*. In the context of using simulated students as ITS authoring tools, *how-learning* essentially automates the process of writing formulae that specify how next-step values are computed at each step. *How-learning* composes domain general functions to explain an author's demonstrated actions from values visible in a tutoring interface. The trouble is that the set of functions made available to *how-learning* must be made very large for a general purpose authoring tool. And with this large corpus of functions—which might include many kinds of functions beyond just arithmetic operations—*how-learning* may search through an intractably large space of function compositions and can be prone to stopping short on incorrect formulae that reproduce the demonstrated action but are incorrect in general.

In this work we offer a method for processing authors' verbal instructions along with demonstrated worked examples. These verbal instructions clarify the particular composition of functions used to produce their worked example. By interpreting the natural language and example together, our system overcomes the intractability of performing *how-learning* on demonstrated examples alone. Beyond the scope of this work, other challenges within the simulated learner authoring tool space include inducing complex decision making processes from examples to program the kinds of dynamic control structures for ITSs, typically only implementable by programming production rules. In this work we are concerned only with generating next step formulae for problem individual steps—just the "then" part in the "if-then" structure of production rules.

1.3 Prior How-learning Mechanisms in Simulated Learners

In many ITSs, such as cognitive tutors [13], human students experience worked examples as bottom-out-hints, the final hints in a sequence of available hints. Hint sequences often begin with high-level conceptual suggestions (e.g. "Convert the fractions so they have the same denominator"), then progressively become more specific and operation-oriented (e.g. "You can use the butterfly method, multiply the denominators to find a common denominator."). ITS hints are an on-demand source of the sorts of verbal instruction that would typically be provided by a human tutor. While human students make use of all sorts of hints, current simulated students are generally only able to utilize the final worked-example bottom-out hints. For instance, a bottom-out hint for the following problem step



Convert the fractions to simplify them $\frac{3}{4} + \frac{2}{3} = \frac{}{\square} + \frac{}{\square} = \frac{}{\square}$ What should the denominator $\square$ be for the converted fraction?

might be "put '12' in the denominator". For lack of natural language processing capabilities simulated learners typically experience the worked-examples from bottom-out hints as Selection-ActionType-Value triples: e.g. ("left_converted_denominator", "UpdateTextField", "12"), that specify the interface element acted upon, the kind of action taken, and the value inserted.

Typically a simulated learner's *how-learning* mechanism tries to explain the "value" field of these triples using the functions available to it and the values visible in the interface. In the process of searching for an explanation it might use the visible '4' and '3' plus a prior knowledge function 'multiply' to find that "4*3 = 12", which yields the correct formula "f(A,B) = A*B". *How-learning* via this guess-and-check search process to reproduce worked examples typically only succeeds in producing correct formulae in simple cases. Consider a problem with a slightly more complex formulae:

```
"Find the slope of the line that passes through (5, 4) and (7, 8)"
```

The worked example solution is "2", and the correct composition for this case is "(8-4)/(7-5) = 2", however a *how-learning* mechanism would almost certainly first consider more parsimonious explanations like "8/4 = 2" or "7-5 = 2" in their search process. Additionally if the target formula is sufficiently complex, having a form consisting of many compositions of compositions of functions and so on, then the space of possible compositions that must be checked to arrive at the target formulae can be intractably large—exceeding available time and/or memory limitations.

2 Verbal Instruction and Examples Together Reduce Ambiguity

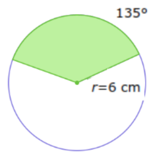
Prior *how-learning* approaches that exclusively induce explanations (i.e., function compositions) from worked example steps can often be cognitively infeasible. In the worst cases, the explanation search may consider tens of thousands of function compositions. Additionally, within the set of compositions considered, a large subset may reproduce the worked example. In other words, traditional *how-learning* approaches have issues of intractability and ambiguity. There is a glaring missing piece to this purely example-based approach to learning—that a great deal of human learning relies on written or verbal instruction articulated in natural language.

Natural language instruction on its own can often be quite clear and direct, though it too can produce sources of ambiguity. For instance, the sentence "Add 3 and 4 times 5" can be reasonably interpreted in multiple ways: as (3+4)*5 or 3+(4*5). Together, worked examples and narrated tutorial instruction can resolve each other's ambiguities. For instance, accompanying the previous utterance with a demonstration of "23", verifies that the latter formula, "3+(4*5)", was the intended one.

The power of narrated tutorial instruction is that it grounds verbal descriptions of concepts within an example. Teaching a human is not like programming a computer. Learning is an imprecise constructive process that involves interpreting information observed from various forms of instruction, then applying and refining the knowledge induced from those instructional experiences. Multimedia learning principles dictate that a combination of language-based and visual instruction is often a boon to learning [10]. This work models this multi-model interpretive process computationally as a process of using two sources of instructional information that each disambiguate the other.

3 Grounded Operational Hints

The system we present in this work utilizes a particular kind of instructional utterance, which we refer to as grounded operational hints. A grounded operational hint is, **1) Grounded**: It specifies all values utilized in the execution of a demonstrated worked example. **2) Operational**: It explicitly specifies every operation utilized in the execution of the demonstration.



The radius of a circle is 6 centimeters. What is the area of a sector bounded by a 135° arc?

What value should be placed in □ if the area is expressed as □π cm^2?

Hint to participants: For a circle with radius r the area $A = \pi r^2$

Fig. 2: (P10) Finding the area of the arc.

An appropriate grounded operational hint for the above problem might be: "Divide 135 degrees by 360 degrees to get the proportion of area in the arc and multiply this by the square of the radius 6." In this example "135", "360", and "6" are the values necessary to compute the solution, while "divide", "multiple", and "square" are the explicit operations. Grounded operational hints might include additional conceptual details, like "to get the proportion of area in the arc" or extra descriptive words like "degrees" or "radius". Extra non-operational information is admissible so long as every value and operation is present. In this work, we also assume that grounded operational hints are articulated as if they were spoken, and thus shouldn't include mathematical notation. For instance, "divide 135 by 360" as opposed to "135 / 360".

A limitation of this work is that our systems learns only from the grounded operational portions of hints. There are perhaps many ways of characterizing, and simulating learning from instructional utterances beyond this. Other considerations include utterances pertaining to named objects, or to preconditions (i.e. "if" parts) of skills, as opposed to only the action parts (i.e. "then" parts) of skills. Li et. al. [6] offer one compelling approach in these directions.

4 Guiding Formula Search with Grounded Operational Hints

Our system utilizes both grounded operational hints and worked examples together, by operating in two stages. In stage 1 the grounded operational hint is parsed into a search policy. In stage 2 a modified version of one of the Apprentice Learner framework's [18] *how-learning* planners uses this policy to search for an explanation for the worked example value. This two phase approach simulates a processes of *mutual disambiguation* [11], where a student interprets an instructor's natural language to get a rough idea, and clarify its meaning with a worked example. For instance, for the following problem:

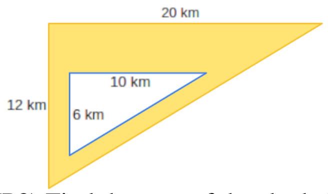


Fig. 3: (P8) Find the area of the shaded region.

A user could provide the demonstration "90" accompanied by the hint "Subtract half of 20 times 12 from half of 10 times 6". The parser outputs a policy that suggests an interpretation for this sentence:

```
1: [(multiply, ['20','12']), (multiply ['10','6'])]
2: [(half, [])]
3: [(subtract, [])]
```

Each of the lines of this policy specify precisely the operations to attempt at successive recursion depths of how-search. At depth 1), 20 and 12, and 10 and 6 are multiplied, and the results added to the total set of available intermediate values. At depth 2) the halves of the available values are introduced to the set of available values as well. At depth 3) every pair of values is subtracted from one another.

The empty brackets "[]" at depths 2 and 3 indicate an operation in the policy for which the operands are not specified, meaning every combination of available values are tried as arguments. A current limitation of our parser is that it cannot produce policies that specify operands for operations in stages beyond the first. If all of the operands at each depth of search could be uniquely specified then the policy would be isomorphic with a single formula, otherwise the policies are loose interpretations of hints that circumfscribe a space of possible formulae.

The policy in the above example encloses a search space of many possible function compositions—albeit far fewer than a search space unguided by a policy. Within this space of formulae, only the target composition (20*12)/2 - (10*6)/2, yields the worked example value “90”. And so the target formula f(a,b,c,d)=(a*b)/2 - (c*d)/2 is the only formula recovered by searching with this policy.

Our program which translates grounded operational hints into policies utilizes a neural network-based grammar parser [4] and coreference resolver[1]. As with all neural network systems, these components of our parser do not produce the correct output 100% of the time. Nonetheless, by loosening and re-executing policies in phases, bad policies can be generalized, and often still recover correct formulae. This functionality reduces the sensitivity of our system to the unpredictable errors of neural components.

5 Translating Natural Language to Policies

Our system utilizes only a combination of pretrained neural components from the Spacy NLP toolset [4] and our own hard-coded rules. We do not fit our system to any domain-specific data. Our system operates in five processing steps. Steps 1 and 2 prepare the hint text for grammatical parsing, and steps 3-5 produce the resulting policy.

Step 1 Apply Special Rules: First a small set of regular expressions are matched against the text string to handle special cases. Each match is replaced with a placeholder word “xcat”. These special regular expressions are needed for capturing patterns common in mathematical language like “7 times 8” and “9 divided by 12”, which mimic the grammar of mathematical notation (e.g. “7*8” and “9/12”), but are otherwise grammatical anomalies compared to most English utterances. For instance, someone might instruct someone to “mix red paint and blue paint”, but they wouldn’t tell them to “red paint mix blue paint”. We only use these special rules in cases where mathematical language diverges from normal English grammar conventions, and to match multi-word operations like “one’s digit”. Cases with typical grammatical structures, like “subtract 2 from 3”, are processed by the grammar parser as normal.

Step 2 Replace Numbers and Operations: Mathematical language lends itself to a few additional quirks that can confuse a grammar parser trained primarily for parsing non-mathematical English text. For instance in the parse of the sentence below on the left, the operation “divide” is interpreted incorrectly as a noun, and the number “8” is parsed as a numerical modifier of it—grammatically more similar to a sentence like “we witches three” than the intended interpretation where “divide” is a verb, and “8” and “4” are its direct objects.

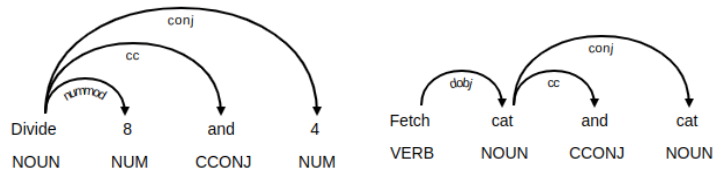


Fig. 4: (left) A hint parsed before steps 1 and 2. (right) A hint after steps 1 and 2 with operations replaced by “fetch” and operands replaced by “cat”.

To avoid incorrect parses like these we do a preliminary parse and replace operations not tagged as verbs or as nouns in prepositional phrases with “fetch”, and every

[1] https://spacy.io/universe/project/coreferee

number is replaced with “cat”. During development we found that these placeholder words were the most likely to produce the correct parts-of-speech among several that we tried. Operations are identified by hard-coded sets of keywords. For instance, division can be identified by “divide”, “quotient”, “ratio”, or “proportion”, in addition to morphological variants of these like “division”, or “divided”. Spacy identifies these morphological variants automatically.

Step 3 Grammatical Parsing & Coreference Resolution: After making any replacements a final parse is performed. Figure 4 shows simple examples of how the Spacy parser adds part-of-speech tags to words and structures them into dependency trees. This hierarchical structure can be used to determine the operands of an operation, and the stage of how-search at which it should be applied. In cases where pronouns like “it” or “them” refer to earlier utterances, a coreference resolver is used to help establish what part of the sentence the pronoun is referring to. This too helps determine the order of each operation and their operands.

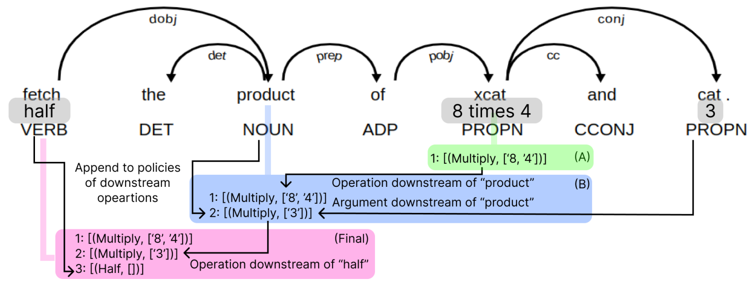

dobj
det
prep
pobj
cc
conj
fetch
half
VERB
the
DET
product
NOUN
of
ADP
xcat
8 times 4
PROPN
and
CCONJ
cat .
3
PROPN
1: [(Multiply, ['8', '4'])]
(A)
Append to policies of downstream opeartions
Operation downstream of “product”
(B)
1: [(Multiply, ['8', '4'])]
2: [(Multiply, ['3'])]
Argument downstream of “product”
1: [(Multiply, ['8', '4'])]
2: [(Multiply, ['3'])]
3: [(Half, [])]
(Final)
Operation downstream of “half”

Fig. 5: Steps 1-4 applied to hint “half the product of 8 times 4 and 3”. The root verb “half” is replaced by “fetch”. The special pattern “8 times 4” is replaced with “xcat”, and the value “3” with “cat”. Colored boxes (A), (B), and (Final) demonstrate the process of a policy being constructed by extending the policies of downstream operations.

Step 4 Building Policies by Recursion: Grammatical parse trees are typically rooted in the main verb of each sentence. To build a policy from this tree the dependency links are traversed recursively, and each recursion returns the working policy of its downstream dependencies. If an operation is identified at any step, it is added to the working policy produced from downstream. If the operation token has any of several direct object-like dependencies with these downstream tokens, then the operation is added to a new depth-level in the policy—indicating that it should be applied after the dependent operations—instead of at the current deepest depth of the working policy. When operands are encountered they are added to a working list of operands and are associated with the first upstream operation. If any of the dependencies of an operation have a coreference link then the operation is added to the working policy at one depth deeper than the longest working policy of the dependencies. Future implementations may use these links from coreferences and downstream operations to specify operands of operations that are calculated from evaluating other operations.

Step 5 Post-Processing: Finally the resulting policy is cleaned of logical inconsistencies. A policy’s first depth should contain only operations applied on operands explicitly stated in the grounded operational hint. Thus, any operation-argument pair like this found in a later depth is moved to the first depth. Any pair of operations and arguments at the first depth with too few arguments is copied into depth two with its arguments removed. Finally any redundant pairs in each depth are removed.

6 Applying Policies in Stages

Because the grammar parser and coreference resolver can produce errors, the hint-to-policy translation process doesn't always produce perfect policies. Our system will nonetheless often produce correct formulae by incrementally generalizing the found policy in phases. If any phase produces no formulae that explain the worked example or produce more than 100 candidate formulae, then the next phase is engaged.

In phase 1 the policy is executed as normal. If multiple formulae are returned then they are scored and sorted to favor solutions that use the pairs of operations and arguments stated in the policy. In phase 2 the policy is generalized by removing any specified operands, forcing the search process to try all permutations of visible or stated values with each operation. In phase 3 the operands are kept but the policy is spread out so that each operation at each depth is also tried at the next depth. This makes the policy tried at phase 3 one depth deeper than the original policy. In phase 4 all operations mentioned in the hint are tried at all depths out to a fixed depth of 3. In phase 5 all operations in the available function set are tried out to a fixed depth of 3. Phase 5 is the same as applying *how-learning* search with only the worked example and no hint.

7 Related Work

Much prior work on natural language processing of mathematical language pertains to the translation of word problems, or other text describing mathematical relationships, directly into equations [20] [14]—essentially the skill of translating word-problems to algebraic equations. By contrast, our system interprets the operational language of natural language instruction, and outputs executable knowledge structures that can perform steps in mathematical procedures.

Recently large language models (LLMs) like OpenAI's Codex [3] have been shown to be able to translate text descriptions to synthesized code. These recent works are closer to ours, in the sense that, like our system, they can synthesize executable formulae. We compare our system's performance with Codex via the Github CoPilot plugin for Visual Studio.

8 Large Language Models are Poor Models of Novices

When simulating learning it is important to appropriately model the prior knowledge of target learners. Unlike an LLM our simulation only assumes prior knowledge of basic natural language parsing capabilities (by use of a pre-trained grammar parser and coreference resolver). LLMS by contrast often have broad generative capabilities trained from many domain-specific experiences greater in number and diversity than any human would ever encounter in a lifetime (like tens of millions of github repositories for Codex).

Toward the goal of building computational models of learning, an LLM's massive scale of prior experience completely contravenes the objective of modeling a novices' learning trajectory from first experiences to mastery. Our system by contrast only uses prior training experiences for grammatical parsing, making it a suitable model of a reasonably articulate but not necessarily mathematically knowledgeable novice.

Toward the goal of using simulated students as authoring tools, however, the cognitive fidelity of various learning mechanisms is not a constraining factor. Thus, if an LLM like Codex proves better than our approach at producing target formulae—even if only because somewhere in its vast training set there is an example analogous to the target task—then it may ultimately be the preferable tool. Yet, if Codex succeeds largely because it is directly mimicking instances from its training set—not because it is suitably powerful as a general text to code translation tool—then it may fail systematically when tasked with aiding authors at building tutoring systems for one-of-a-kind materials. We evaluate this possibility by comparing our system with Codex on descriptions of made-up formulae.

9 Methods

We recruited 10 crowd workers through Prolific to generate grounded operational hints. Participants solved 14 unique math problem steps, and provided both conceptual hints (i.e. "describe the concept in broad terms") and grounded operational hints for that problem step. We requested conceptual hints simply so participants had an alternative place to write any conceptual instructions that came to mind, and to encourage consideration of the distinction between the two kinds of hints. The conceptual hints are not used in our evaluations. We additionally asked participants provide grounded operational hints "as if spoken aloud". They were asked to not used mathematical notation such as "+*/-". Participants were also given a short five question "check your understanding" survey to ensure they understood the difference between conceptual and grounded-operational-hints, and each section of the form began with a short reference sheet.

Two of the authors independently coded each participants' grounded operational hints to mark if they were indeed both grounded (i.e. "mentions all required arguments and constants"), and operational (i.e. "mentions all required operations"). An inter-rater reliability of 97.2% was achieved, and the discrepancies were resolved through discussion. Hints marked as both grounded and operational, we refer to as "good" hints.

For each response where participants produced the correct step answer, we ran the accompanying participant-generated grounded operational hints through our system with and without hints. We did the same with Codex via Github Copilot with and without worked examples, which were used to eliminate functions based on return value. Our system had the 7 functions "Add, Multiply, Subtract, Divide, Half, OnesDigit, and Square" available to it, which was sufficient for building function compositions for all of the target formulae, plus 8 additional functions not needed for any of the target tasks "TensDigit, Power, Double, Increment, Decrement, Log2, Sin, and Cos".

Typically Github Copilot takes a function header and doc string and automatically produces a function implementation. We filled our participants' grounded operational hints into Github Copilot as the doc-string of an empty Python function with the header "foo():". We recorded the extended set of suggestions Copilot produced, not just the first suggested implementation. As with our system, this extended set of suggestions produces a small variable number of candidate solutions. Typically Copilot uses arguments specified in the function header. Since this information is not present in the participant's hints, we omit arguments and just evaluate whether CoPilot suggests function implementations that are functionally equivalent to the target formulae, expressed with constants instead of arguments. When examples were included, we also executed each of these functions to see if they reproduce the worked example.

For both systems we measured whether or not a correct formula was produced for each grounded operational hint, and counted the number of incorrect formulae produced. We considered any algebraic rearrangements (e.g. A*B = B*A) of the target formula to be correct. Thus, we use the average number of incorrect formulae as the principle measure of error magnitude, instead of proportion correct, which would be sensitive to returning several isomorphic formulae. The principal measures of the rate of correctness are 1) the percentage of the participant provided hints where the system produces at least one correct formula (i.e. "has correct"), and 2) the percentage where only the correct formula is produced (i.e. "only correct").

To evaluate Codex's potential performance on one-of-a-kind formulae unlikely to be present in its training set, we repeated these evaluations for a set of 10 made-up formulae with accompanying grounded operational hints that we wrote ourselves.

10 Results

Each of our 10 participants finished 14 problems for a total of 140 responses. We removed 26 responses where participants produced incorrect answers and used the remaining 114 grounded operational hints for evaluation. Our system produced sets of formulae containing at least one correct formula 82.4% of the time, and 69.2% of the time only correct formulae were returned. Of the 114 grounded operational hints 87 were coded as "good". On average our system performed better for the "good" hints, 86.2% had correct formulae and 73.6% had only correct formulae. For the set of all 114 responses our system produced an average of 1.54 unique incorrect formulae, whereas for the "good" hints it produced an average of 0.54 incorrect formulae.

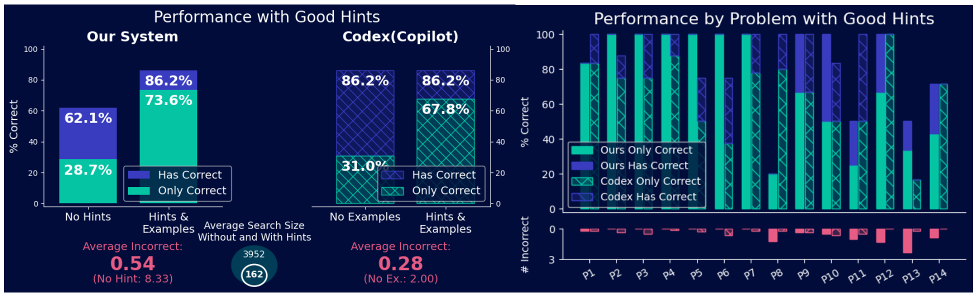


Fig. 6: Overall performance and performance by problem on "good" hints for our system and Github Copilot. Reduction in how-search size from hints is shown.

Overall our system's performance (Figure 6) was improved considerably by parsing hints in addition to worked examples. When using only worked examples without hints, as in prior work, our system produced only correct solutions 28.7% of the time. Succeeding on just 4 of the 14 problems. Hints also reduced the average number of incorrect formulae from 8.33 to 0.54, and the average search space size from 3952 function compositions to 162.

Github Copilot achieved the same "has correct" performance as our system on "good" hints with a slightly worse "only correct" performance of 67.4%. Our system performed nearly perfectly on problems 1-7 which had shorter formulae and participant hints—an average of 16 words, versus 24 words in problems 8-14. Most of our systems' errors came from difficulties parsing the verbose hints from problems 8-14. Copilot's performance was more varied, but was much higher than our system for some problems (e.g. P8, P11). This is likely partially due to memoization of similar problems in Codex's training set, since in some cases it produced code with suspiciously domain-specific inline comments and variable names. However, Copilot also performed perfectly on our corpus of 10 made-up hints and worked examples, verifying that it is indeed strong at novel code generation. By contrast, our system also produced correct formulae for all 10 made-up hints, and only a few incorrect formulae for 4 of them.

11 Discussion

Both systems showed improvements consistent with the assertion that the combination of langauage and worked examples benefits instruction comprehension over either taken alone. Our system generally performed best on more concise hints—a pattern of performance that bodes well for computational modeling purposes. Humans often learn better from concise directed learning experiences. Problems like problem P8 (Figure 3), which our system performed poorly on, are often scaffolded into multiple steps in ITSs. Copilot by contrast had no consistent performance pattern, and showed signs of leveraging prior domain-specific knowledge.

12 Conclusion

For ITS authoring purposes these systems could be used interactively with speech-to-text. Authors could build ITSs with these systems through narrated tutorial instruction, explaining worked examples verbally as they write them. Trial-and-error may refine authors' verbal instructions and produce a higher rate of success than our crowd-worker data suggests. In principle, our system could also help authors write their own formulae, by revealing the syntax and many available functions of an authoring tools' formula language. Both systems produced a fairly small average number of incorrect formulae, meaning correct formulae can be easily selected from among a small set of candidates. Even if no candidate formulae are correct, it may still be easier for authors to fix incorrect formulae than to write new ones. In this work we've only composed mathematical functions, but more varied function compositions including string manipulations and other special operations could be used by our system, with little modification. The same is probably not true for Copilot, but is worth testing.

Future computational modeling work may investigate how these simulated language comprehension abilities compare to various human learners. In this work we've demonstrated two means of generate knowledge from grounded operational instruction. This is, however, only a first step. Our system takes a relatively structured approach, compared to the method using Copilot, making it conducive to many future refinements and investigations. This opens opportunities to investigate questions of how learners interpret tutorial instruction, and how instruction and ITS hints may be improved as a result. For now we've achieved our main objective, to expand what simulated students can feasibly learn, and the set of instructional experience they can learn from.

References

1. 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)
2. Cen, H., Koedinger, K., Junker, B.: Learning factors analysis–a general method for cognitive model evaluation and improvement. In: International Conference on Intelligent Tutoring Systems. pp. 164–175. Springer (2006)
3. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
4. Honnibal, M., Montani, I.: spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing (2017), to appear
5. Kulik, J.A., Fletcher, J.: Effectiveness of intelligent tutoring systems: a meta-analytic review. Review of educational research **86**(1), 42–78 (2016)
6. Li, T.J.J., Radensky, M., Jia, J., Singarajah, K., Mitchell, T.M., Myers, B.A.: Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In: Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology. p. 577–589. Association for Computing Machinery (2019)
7. MacLellan, C.J., Harpstead, E., Marinier III, R.P., Koedinger, K.R.: A framework for natural cognitive system training interactions. Advances in Cognitive Systems **6**, 1–16 (2018)
8. 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)
9. 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)
10. Mayer, R.E.: Using multimedia for e-learning. Journal of computer assisted learning **33**(5), 403–423 (2017)
11. Oviatt, S.: Mutual disambiguation of recognition errors in a multimodel architecture. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. p. 576–583. Association for Computing Machinery (1999)
12. Pavlik Jr, P.I., Cen, H., Koedinger, K.R.: Performance factors analysis–a new alternative to knowledge tracing. Online Submission (2009)
13. Ritter, S., Anderson, J.R., Koedinger, K.R., Corbett, A.: Cognitive tutor: Applied research in mathematics education. Psychonomic Bulletin & Review **14**(2), 249–255 (2007)
14. Roy, S., Roth, D.: Mapping to declarative knowledge for word problem solving. Transactions of the Association for Computational Linguistics **6**, 159–172 (2018)
15. VanLehn, K.: Learning one subprocedure per lesson. Artificial Intelligence **31**(1), 1–40 (1987)
16. VanLehn, K.: The behavior of tutoring systems. International Journal of Artificial Intelligence in Education **16**(3), 227–265 (2006)
17. VanLehn, K., Ohlsson, S., Nason, R.: Applications of simulated students: An exploration. Journal of artificial intelligence in education **5**, 135–135 (1994)
18. Weitekamp, D., Harpstead, E., Koedinger, K.: An interaction design for machine teaching to develop ai tutors. CHI (2020)
19. Weitekamp, D., Harpstead, E., MacLellan, C.J., Rachatasumrit, N., Koedinger, K.R.: Toward near zero-parameter prediction using a computational model of student learning. International Educational Data Mining Society (2019)
20. Zou, Y., Lu, W.: Text2math: End-to-end parsing text into math expressions. arXiv preprint arXiv:1910.06571 (2019)