Theory Acquisition as Constraint-Based Program Synthesis

Haoliang Wang
Dept. of Psychology
UC San Diego
haw027@ucsd.edu

Edward Vul
Dept. of Psychology
UC San Diego
evul@ucsd.edu

Nadia Polikarpova
Dept. of Computer Science & Engineering
UC San Diego
nadia.polikarpova@ucsd.edu

Judith E. Fan
Dept. of Psychology
UC San Diego
jefan@ucsd.edu

Abstract

What computations enable humans to leap from mere observations to rich explanatory theories? Prior work has focused on stochastic algorithms that rely on random, local perturbations to model the search for satisfactory theories. Here we introduce a new approach inspired by the practice of 'debugging' from computer programming, whereby learners use past experience to constrain future proposals, and are thus able to consider large leaps in their current theory to fix specific deficiencies. We apply our 'debugging' algorithm to the magnetism domain introduced by (Ullman, Goodman, & Tenenbaum, 2010) and compare its efficiency and accuracy to their stochastic-search algorithm. We find that our algorithm not only requires fewer iterations to find a solution, but that the solutions it finds more reliably recover the correct latent theories, and are more robust to sparse data. Our findings suggest the promise of such constraint-based approaches to emulate the way humans efficiently navigate large, discrete hypothesis spaces.

Keywords: debugging; program synthesis; theory learning

Introduction

A fundamental aspect of human reasoning is the drive to go beyond individual observations toward more general explanations (Lombrozo, 2006). This drive to explain more of the world we see in terms of more unified theories is perhaps most obviously manifest in the process of scientific discovery. For example, at the turn of the 20th century many physicists were convinced that they were on the cusp of confirming the existence of the *luminiferous aether*, an invisible medium permeating all of space that supposedly carried light waves. The decisive blow to the aether came as a result of the Michelson-Morley experiment, which found no evidence for such a medium, and its failure eventually led to the formulation of special relativity, a more successful unifying physical theory (Staley, 2009). The importance of using such evidence to strongly constrain the development of scientific theories has long been of a subject of discussion in philosophy of science (Kuhn, 1962; Platt, 1964). The key component of such a theory-development process is that the addition of some constraining evidence may require large, discontinuous jumps in the space of all possible theories. How might similar principles be applied to explain the theories that ordinary people derive to explain their everyday observations (Gopnik & Meltzoff, 1998; Schulz, 2012; Carey, 1985)?

A large body of work in cognitive science investigating the principles governing which theories people acquire across a variety of different domains has argued that it conforms to the predictions of Bayesian inference, whereby learners reliably select hypotheses according to their posterior probability (Tenenbaum, 1999; Ullman et al., 2010; Ullman & Tenenbaum, 2020). However, efficiently navigating the posterior probability landscapes poses severe algorithmic challenges, especially for hypotheses defined over large combinatoric spaces of discrete alternatives. In such domains it is computationally intractable to evaluate every hypothesis, and conventional search or sampling strategies are often stymied by their non-differentiable, disjointed character. A prominent proposal for how humans might nevertheless navigate such hypothesis spaces, known as stochastic search, employs Markov Chain Monte Carlo (MCMC) to explore the space of possible theories, which operates by randomly sampling local changes to the currently held theory (Ullman et al., 2010).

Although this algorithm achieved some success, it is not clear whether it can serve as a reasonable process model of how humans update their theories. In particular, such stochastic sampling approaches cannot propose directed modifications to the current theory that make effective use of previously encountered constraints, and the proposed modifications are usually local. Here we present a computational model that updates its currently held theory by asking where and why its current proposal went wrong, and accumulates these constraints to guide its future proposals. We call our algorithm the "debugging" algorithm, reflecting the similarity between the way it operates and the way that computer programmers diagnose and repair specific deficiencies in software (Shapiro, 1982; Zeller, 1999; Rule, 2020). Our experiments show that our algorithm is not only substantially more efficient than stochastic search, but that it also achieves higher accuracy in recovering the latent theories, even under sparse data.

Theories as Logic Programs

What is a theory? Consider the example domain of magnetism introduced in (Ullman et al., 2010): the learner is confronted with a collection of objects $a, b, \ldots, j$, and gets to observe that some pairs of these objects interact, and other pairs do not (see Fig. 1 "data"). The learner must come up with an explanation for these patterns of interactions. It might postulate that there are three types of objects: plastic, magnets, and metals. Further, it might specify laws that allow it to deduce

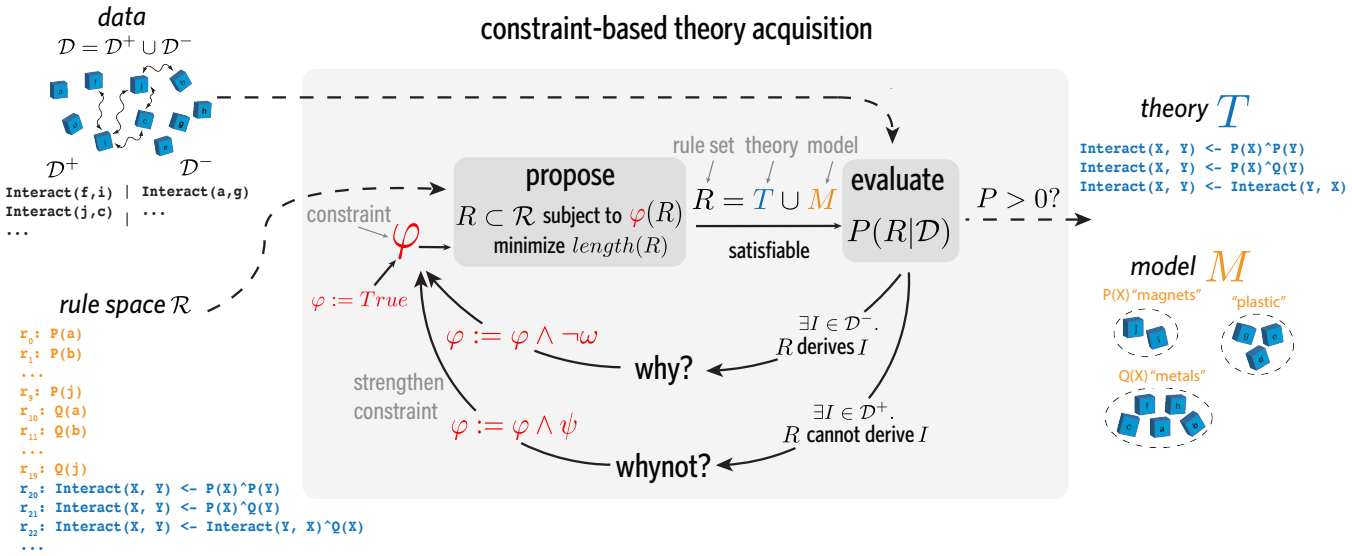

Figure 1: The "debug" algorithm applied to the domain of magnetism. The data D includes all pairs which we know to either interact, or not. To explain these data, we postulate a logical program (R) comprised of a theory (domain laws) and a model (mapping predicates of the theory onto objects). The algorithm finds a suitable program by repeatedly proposing candidates, evaluating where they deviated from the data, and growing constraints to avoid similar errors in the future (see text).

whether a given pair of objects interacts based on their types (*e.g.* magnets interact with magnets). Such laws capture the abstract, reusable knowledge, and hence may be considered the "theory". Finally, to connect the theory with the observations, the learner must also classify the objects $a, b, \ldots, j$ into the three types; this concrete, non-reusable knowledge is referred to as the "model".

Following (Ullman et al., 2010), we formalize knowledge as *logic programs* (Cropper & Dumančić, 2020; Evans & Grefenstette, 2018). A logic program is a set of *rules*. For example, we can represent the theory of magnetism using the following three rules:

$$\begin{array}{lllll} \text{Rule 1:} & \mathsf{Interact}(X,Y) & \leftarrow & \mathsf{P}(X) \wedge \mathsf{P}(Y) & \\ \text{Rule 2:} & \mathsf{Interact}(X,Y) & \leftarrow & \mathsf{P}(X) \wedge \mathsf{Q}(Y) & (1) \\ \text{Rule 3:} & \mathsf{Interact}(X,Y) & \leftarrow & \mathsf{Interact}(Y,X) & \end{array}$$

The meaning of a rule is that its left-hand side is true if its right-hand side is true. Hence, this theory states that objects of class P can interact with each other and with objects of class Q, and that interactions are symmetric. Here P, Q, and Interact are *predicates*, which represent set and relations over objects; X and Y are *variables* that can be substituted with *atoms*, *i.e.* concrete objects. A rule with an empty right-hand side is called a *fact*; for example $\mathsf{P}(f)$ is a fact that simply states that atom f belongs to type P. Thus, using a combination of general rules and facts, we can represent both the theory and the model in a unified format as a logic program.

A logic program can *derive* new facts: for example, given $\mathsf{P}(f)$ and $\mathsf{P}(j)$ and the rules in Eq. 1, we can derive the fact $\mathsf{Interact}(j,f)$ using the rules 1 and 3. Under the logic-program formulation, if a particular interaction cannot be derived from known rules, then it is assumed to be false.

Theory acquisition as Bayesian Program Induction

A learner is confronted with an unlabeled collection of objects, and notes that some pairs of objects interact (D^+), while other pairs fail to interact (D^-); see Fig. 1 "data". Given the the observed data $D = \{D^+, D^-\}$, the learner must find a set of rules R (*i.e.* a logic program) that derives the facts in D^+ and cannot derive the facts in D^-. The rule set $R = T \cup M$ contains both theory rules T and model rules M. We assume that the space of all possible rules $\mathcal{R}$ is given, so $R \subset \mathcal{R}$.

Candidate programs can be evaluated in a Bayesian framework: by using the posterior probability of the rules given the observed data D:

$$P(R|D) \propto P(D|R)P(R) \qquad (2)$$

where $P(D|R)$ represents the likelihood of the data given the rule set. Here we adopt a simple error likelihood, wherein observed data have a probability of ε of deviating from the predictions of the program R. $P(R)$ represents the prior probability of the rule set that discourages large rule sets and complex rules.

While we can define the *goal* of the learner in these simple terms, the actual challenge lies in finding suitable programs under this posterior. The set of candidate programs is a huge discrete space: even with only 55 possible rules—the size of $\mathcal{R}$ used in our experiments—the size of the search space is on the order of 10^{16}. What kind of algorithm might support search in such a seemingly intractable space? Prior work on theory acquisition (Ullman et al., 2010) used *stochastic search* to explore this space, which we argue is inefficient; instead we propose a new approach based on *constraint-based program synthesis*. Next we give an overview of the prior approach, and then present our approach.

Stochastic Search

Ullman et al. (2010) suggest that the intractable search program inherent in theory acquisition may be characterized as stochastic search via Markov Chain Monte Carlo (MCMC). MCMC maintains the current hypothesis R, and in each search iteration is proposes a new hypothesis R' from a distribution based on R. When learning logic programs, the proposal distribution $Q(R'|R)$ is obtained by syntactically local changes to the current program, such as randomly adding, deleting, or substituting a predicate or a rule. The proposed theories are then scored according to the posterior, and accepted in accordance with the Metropolis-Hastings rule. Ullman et al. (2010) specifies a particular nested search process, wherein an outer MCMC loop samples theory rules, and then an inner loop attempts to find the best model predicate assignments for that theory using Gibbs sampling (Geman & Geman, 1984).

The key feature of such MCMC-based stochastic search is that proposal distributions are, by definition, Markovian: they only depend on the current theory, regardless of which theories had been previously considered. This means that new theories are proposed with no regard for how successful previous hypotheses were or where they went wrong. Consequently, the search process has no memory, and does not itself *learn* during search.

Constraint-Based Synthesis

To make the search more efficient, we need to equip it with a memory, so the algorithm can keep track of its past mistakes and learn from them. To this end we take inspiration from an area of computer science called *program synthesis*; in particular, we build upon an existing synthesizer for logic programs called ProSynth (Raghothaman, Mendelson, Zhao, Naik, & Scholz, 2019). Fig. 1 illustrates our full algorithm. In what follows, we first summarize the core ProSynth algorithm and then highlight the changes we made to adapt it to the theory acquisition domain.

Like stochastic search, ProSynth has a proposal stage and an evaluation stage, but, critically, it maintains a *constraint* to accumulate knowledge about where prior hypotheses went wrong. The constraint φ is a logical formula that describes the set of acceptable logic programs. For example, the constraint $\varphi = (r_0 \vee r_2) \wedge \neg(r_0 \wedge r_1)$ means that a candidate program must include one of the rules r_0 or r_2 and exclude one of the rules r_0 or r_1. At the proposal stage, ProSynth proposes a new theory subject to the current constraint φ; it uses a constraint solver to do so efficiently.

At the start of the search, φ starts out empty ($True$), but throughout the course of the algorithm it accumulates knowledge from failures of previously entertained theories. At the evaluation stage, ProSynth considers counter-examples—predictions that do not match the data—and grows the constraints to avoid the causes of errors in the future. There are two kinds of counter-examples. When the current program derives a fact $I \in D^-$ known to be false, ProSynth asks **why**

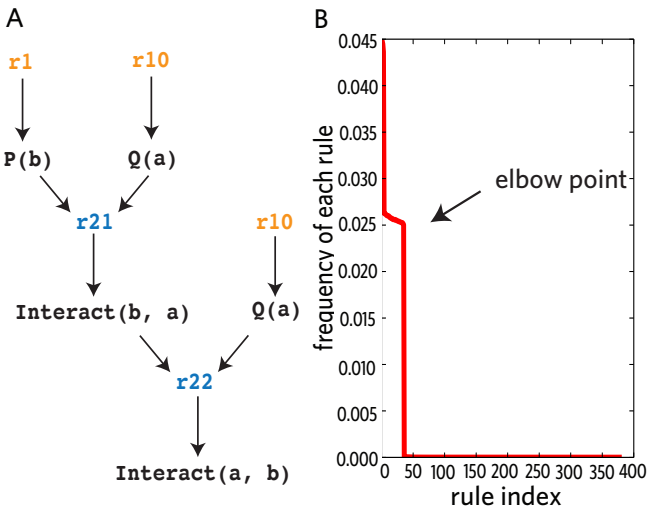

Figure 2: (A) Derivation tree for $\mathsf{Interact}(a,b)$ in our running example. Model rules colored in orange, theory rules in blue. (B) There are 376 unique rules found during the 500 runs of MCMC. We take the rules to the left of the elbow point to be our initial candidate rule set.

that erroneous derivation was made (which rules were used to derive I), and modifies the constraint to exclude this combination of rules from all future programs. Likewise, when the current theory fails to derive a fact $I \in D^+$ known to be true, ProSynth asks **why-not?** (which rule omissions are preventing I from being derived) and modifies the constraints to require that one of these rules be included.

Why? Suppose our current proposed theory is $T = \{r_{20}, r_{21}, r_{22}\}$ and our current model is $M = \{r_0, r_1, r_{10}\}$; these rules are shown in Fig. 1 "rule space". Fig. 2A shows how this rule set can derive the fact $\mathsf{Interact}(a,b) \in D^-$ that is known to be false. Since this derivation is using the rules $\{r_1, r_{10}, r_{21}, r_{22}\}$, ProSynth updates the current constraint φ to $\varphi \wedge \neg(r_1 \wedge r_{10} \wedge r_{21} \wedge r_{22})$ to require that one of the rules from the problematic combination be excluded.

Why not? Suppose in the next iteration the constraint solver proposes the same model but removes r_{22} from the theory in order to satisfy the new constraint. However, the new program fails to derive $\mathsf{Interact}(f,i) \in D^+$: the constraint solver has clearly excluded too many rules. At this point ProSynth attempts to find a small set of excluded rules that are absolutely necessary to derive $\mathsf{Interact}(f,i)$. To this end, it uses a technique called *delta debugging* (Zeller, 1999), which systematically explores smaller and smaller subsets of the excluded rules and asks "what if I included *all* rules except this small subset"? In this sense, delta debugging can be thought of as *counterfactual reasoning*. In our example, ProSynth finds that a program with all rules except $\{r_5, r_{18}\}$ (*i.e.* $\mathsf{P}(f)$ and $\mathsf{Q}(f)$) still fails to derive $\mathsf{Interact}(f,i)$: in other words, even with the most permissive theory, f cannot interact unless it is either a magnet or a metal. Hence one

of these two rules must be included in all future programs; it updates the current constraint φ accordingly to $\varphi \wedge (r_5 \vee r_{18})$.

Jointly learning theory and model

The typical problem formulation in logic program synthesis is: given the input facts (the model) and the output facts (the observations), find the rules that can derive the observations from the model. The theory acquisition domain differs from this typical formulation in that the model is not given, but needs to be learned. Stochastic search (Ullman et al., 2010) uses a separate Gibbs sampling loop to learn the best model for the current theory. However, this approach is fundamentally inconsistent with learning from failures, since it is impossible to impose a constraint on a theory irrespective of the model. Instead, unlike stochastic search and traditional program synthesis, our algorithm learns the model and theory jointly by postulating a place-holder truth predicate that allows PROSYNTH to treat the model rules as part of the program rather than the input to the program.

Bayesian Constraint-Based Synthesis

Core PROSYNTH is not suited for Bayesian program induction. The reason is that its proposal stage does not perform optimization, and instead simply queries a SAT (*i.e.* Boolean satisfiability) solver for any program that satisfies the constraint φ. As a result, although PROSYNTH is able to find a program that perfectly fits the data, it does not guarantee that this program is parsimonious according to the prior $P(R)$.

To overcome this issue, we made an important modification to the PROSYNTH's proposal stage. Instead of using a SAT solver, we use a MaxSAT (*i.e.* maximal satisfiability) solver (Bjorner, Phan, & Fleckenstein, 2015), to find proposals that not only satisfy the constraints, but also maximize the prior.

$$P(R) \propto \exp(-\text{length}(R)) \tag{3}$$

For our algorithm we use $\varepsilon = 0$ in the definition of the likelihood, which means that any program that makes mispredictions has a likelihood (and posterior) of zero, and all programs that make no errors have a likelihood of 1.

$$P(D|R) = \mathbb{1}[\text{match}(\textit{prediction}, D)] \tag{4}$$

Consequently, a program that maximizes the prior and has a non-zero likelihood also maximizes the posterior. As a result, the first program we evaluate that has makes no prediction errors is guaranteed to maximize the posterior (Eq. 2).

Stochastic Rule-Space Prioritization

The final modification we made to the core PROSYNTH algorithm has to do with the rule space $\mathcal{R}$. As explained above, PROSYNTH takes as input a finite space of rules, from which the constraint solver picks the current proposal. In the domain of theory acquisition, however, the space of all possible rules is unbounded: one can always create longer and longer rules by adding more predicates to the right-hand side. To overcome this discrepancy, we can *stratify* the infinite space of rules into finite layers $\mathcal{R} = \{\mathcal{R}_0 \cup \mathcal{R}_1 \cup \ldots\}$. With this stratification in place, we can first invoke PROSYNTH with the space $\mathcal{R}_0$; if there is no solution in this space (*i.e.* the constraint φ becomes unsatisfiable), we can invoke PROSYNTH again with $\mathcal{R}_0 \cup \mathcal{R}_1$, and so on. The challenge is to split the rules into layers in such a way that rules that are more likely a priori to be a part of the optimal program appear in earlier strata.

A naive way to split the rules is by their length, however not all rules of the same length are equally useful in theories. Ullman et al. (2010) used this observation to speed up stochastic search by equipping the MCMC algorithm with templates, or canonical forms of rules that capture structure likely to be used. These templates were encouraged to be used during sampling an initial theory and proposing a new theory. To provide PROSYNTH with the same beneficial bias, we randomly sampled 500 times from their given templates as seeds and ran MCMC chain from each seed for 1600 iterations, always accepting the proposed change. We collected every rule in every theory explored during these runs and calculated the frequency of each rule (see Fig. 2B); these frequencies essentially provide an estimate of the true prior used by the stochastic search in (Ullman et al., 2010). We then stratify the rules based on fixed cut-offs on their frequency, setting the initial layer $\mathcal{R}_0$ to include all the rules left of the elbow point as an approximation of how templates were used in MCMC. This stratification yields 35 theory rules in $\mathcal{R}_0$ (capturing 99.87% of the theory rules sampled by MCMC).

Results

In the preceding section, we introduced our algorithm for theory acquisition. Here, we compare our algorithm with MCMC introduced in (Ullman et al., 2010) and test the claim that our algorithm improves upon its stochastic search counterpart.

Learning more efficiently and accurately

Following Ullman et al., we consider a concrete system with 3 magnets, 5 magnetic objects and 2 non-magnetic objects. The learner was given full observations (i.e. interactions between every pair of objects) and none of the laws or core predicate structure to begin with. We run both models 10 times. For MCMC, each run comprises 1600 iterations of the outer Metropolis-Hastings loop sampling over candidate theories and each outer iteration sprouts an inner loop that learns the correct model assignment. For the "debugging" algorithm, each outer iteration corresponds to one call to the SAT solver, and each inner iteration corresponds to one hypothetical theory and model proposed in the why-not step. For fair comparison, we unroll both outer and inner loops in both algorithms and record every iteration at the finest grain, this amounts to considering one iteration to be an evaluation of a single candidate ruleset. The results are shown in Figure 3.

We first measure the posterior probability of the theories learned by both algorithms, which is the standard established

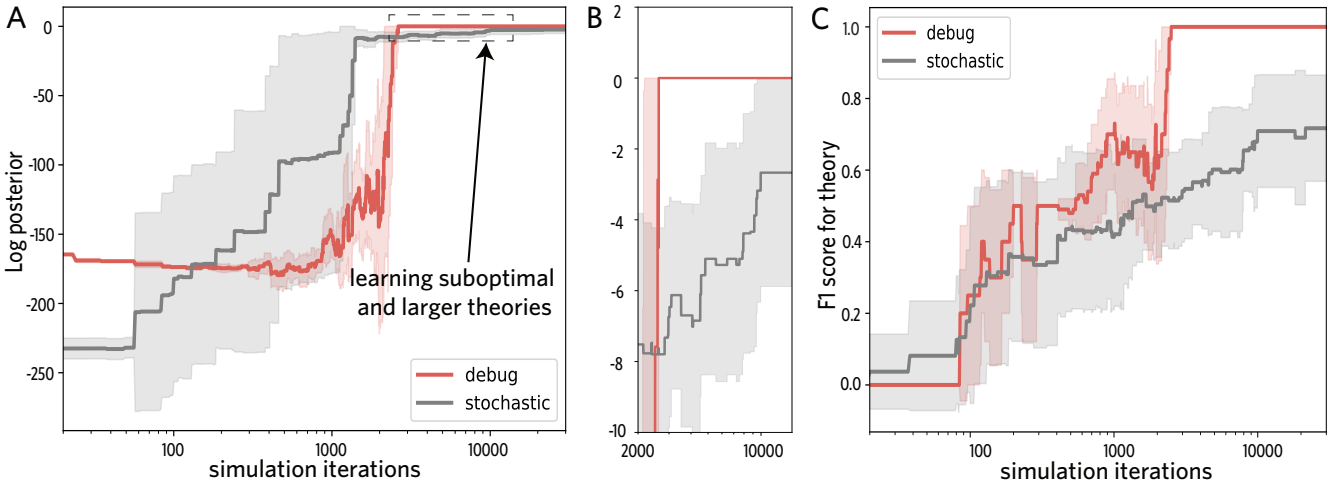

Figure 3: (A) Log posterior for both algorithms across all runs. (B) Zooming into the dashed box in (A), when both algorithms' log posterior score went flat. (C) F_1 score for theories learned by both algorithms. Note that all x axes are in logarithmic scale.

by Ullman et al. (2010). To favor MCMC, we use the posterior that it uses to search (rather than the all-or-nothing posterior implicit in the debugging algorithm). Although for MCMC the posterior rises quickly, as the algorithm aims to fit the data, the MCMC algorithm usually gets stuck in local optima and ends up learning sub-optimal, larger theories. In contrast, the "debugging" algorithm always learns the globally optimal theory, yielding an eventual posterior 14.88 ($e^{2.70}$) times higher than the answer found by MCMC (see Figure 3A).

We further evaluate both algorithms on another two metrics:

(a) Accuracy – we used the F_1 score as our measure of theory-learning accuracy, which reflects the degree to which the theory learned by the algorithm coincided with the ground truth theory. F_1 lies in the range [0,1] with higher scores indicating greater accuracy. It is computed by taking the harmonic mean of the precision (i.e., the proportion of learned theory rules that coincided with the ground truth theory) and recall (i.e., the proportion of the ground truth theory that coincided with the learned theory):

$$F_1 = \frac{2}{(precision^{-1} + recall^{-1})} \tag{5}$$

(b) Efficiency -– how quickly the algorithm converges on a satisfactory solution.

Figure 3C shows the theory F_1 score for both algorithms as a function of iteration, revealing three effects. First, the MCMC algorithm exhibits greater variance because the proposed new theories are random, unguided, perturbations of the current theory, whereas the "debugging" algorithm produces a relatively more consistent result. Second, the "debugging" algorithm performs better in terms of asymptotic F_1 score: of all 10 runs, the "debugging" algorithm always converges to the groundtruth theory, whereas for MCMC, only 2 out of 10 learned the exact correct theory. In addition, the number of iterations the "debugging" algorithm requires to converge is one order of magnitude smaller than that of MCMC.

More robust learning under sparse data

To test the robustness of learning when only partial evidence is observable, we evaluated the algorithms on random subsets of the data. We considered subsets ranging from 5% to 100% (in 5% increments) of the full set of observations. For each subset fraction, we sampled 10 times and run both algorithms on each sample. Both algorithms can learn programs from subsets of data that not only capture the observed data, but also correctly generalize to new data (Figure 4A). However, the "debugging" algorithm exhibits much better accuracy for theories at nearly all subsampling ratios (see Figure 4B).

Furthermore, even at a low percentage of sampled data (e.g. 20%), the "debugging" algorithm can still learn a reasonable theory (F_1 = 0.89), even though at that subsampling ratio it cannot accurately generalize to unobserved data (F_1 = 0.64). This result is consistent with the "Blessing of Abstraction" (Goodman, Ullman, & Tenenbaum, 2011): abstract theory knowledge is acquired relatively easily, even when concrete details of the model (such as which objects get which predicates) are still ambiguous.

How does "debugging" learn?

We claimed that MCMC can only propose local changes to its current theory while the "debugging" algorithm can make large, non-local changes, leading to better performance. To verify this, for both algorithms, we measured the *similarity* between the proposed theory and current theory, defined as $2X/N$, where X represents the number of matched laws between the current theory and the proposed theory, and N represents the total number of laws in both theories.

The "debugging" algorithms shows a lower similarity score, suggesting that it's able to make larger, more variable

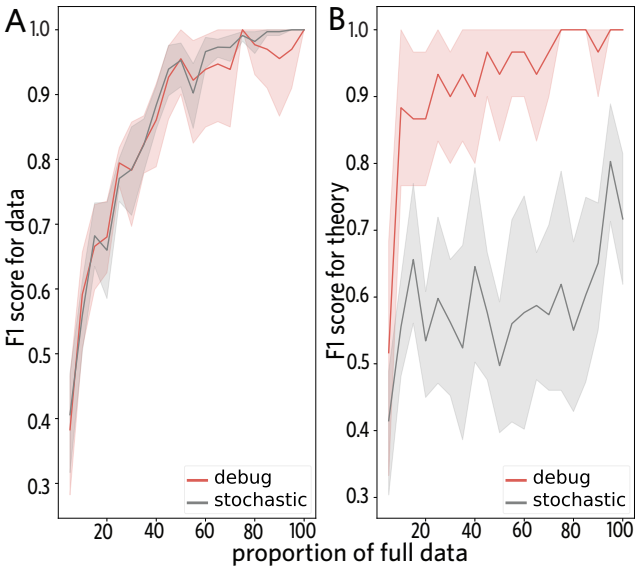

Figure 4: The F_1 score for data and theory as a function of proportion of randomly sampled data from the full observation. MCMC learns suboptimal and larger theories.

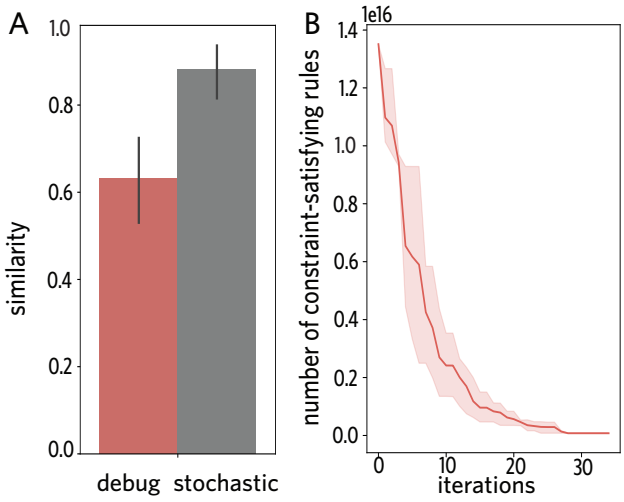

Figure 5: (A) The similarity between the proposed and current theory. Score is averaged across 10 runs. (B) The size of hypothesis space where the "debugging" algorithm can propose new theories from as a function of iterations. Score is averaged across 10 runs.

jumps as needed in syntactic space. The lower mean similarity coincides with greater variance, indicating that the mean is lower due to its willingness to undertake the occasional large jumps (see Figure 5A).

We next test the claim that accumulating evidence from prior iterations of the search process helps the "debugging" algorithm find answers efficiently. We calculated the number of theories that satisfy all constraints after each call to the SAT solver. As shown in Figure 5B, the space of possible theories shrinks by a factor of 4 per outer loop iteration as the constraints builds up, indicating that accumulating constraints are critical to the efficiency of our search algorithm.

Discussion

In this paper, we presented a computational model that can learn from its past failures when searching for a theory to explain its observations. We demonstrated that our algorithm acquires theories more efficiently and accurately than its stochastic counterparts, and exhibits more robust learning under sparse data. Our "debugging" algorithm relies on the accumulation of errors to make its search more efficient. Such efficient use of errors is reminiscent of the way that backpropagation is used to guide learning of weight parameters in artificial neural networks. However, despite the efficiency of such learning algorithms in continuous parameter spaces, they fundamentally rely on calculating gradients, and thus are not well suited to discrete, combinatoric hypothesis spaces. By contrast, the "debugging" algorithm employs an error-driven learning strategy that can efficiently search over disjointed, non-differentiable domains, including the space of all programs.

Capturing graded patterns in human theory acquisition will require several refinements to our algorithm. First, the "debugging" algorithm currently assumes that observations have no noise, and thus is designed to accumulate constraints to fit all observations. Human learners, by contrast, cannot avoid such noise, so in future work we aim to alter the way our "debugging" algorithm handles constraints in order to accommodate noisy observations. A second concern about the real-world plausibility of the "debugging" algorithm is that it relies on a state-of-the-art SAT solver to find a set of rules to satisfy the set of constraints learned incrementally during the search process. Nonetheless, the SAT solver exhibits some key features of human reasoning that are conspicuously absent from stochastic search. Specifically, it learns during search by accumulating errors and thus becomes markedly more efficient after the first few iterations. Overall, we believe that further development of such constraint-based algorithms that incorporate error-driven learning may lead to better algorithmic theories of how humans efficiently navigate large, discrete hypothesis spaces.

Acknowledgments

We thank Tomer Ullman for providing the code for the stochastic search model. We also appreciate Yuyao Wang, Jiaying Xu and members of the Cognitive Tools Lab at UC San Diego for very helpful discussion. This work was supported by NSF CAREER Award #2047191 to J.E.F.

References

Bjorner, N., Phan, A.-D., & Fleckenstein, L. (2015). vZ - An Optimizing SMT Solver. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 9035* (pp. 194–

199). Berlin, Heidelberg: Springer-Verlag. Retrieved from doi:10.1007/978-3-662-46681-0_14

Carey, S. (1985). *Conceptual change in childhood*. MIT press.

Cropper, A., & Dumančić, S. (2020). Inductive logic programming at 30: a new introduction. *arXiv preprint arXiv:2008.07912*.

Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, *61*, 1–64.

Geman, S., & Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*(6), 721–741.

Goodman, N. D., Ullman, T. D., & Tenenbaum, J. B. (2011). Learning a theory of causality. *Psychological review*, *118*(1), 110.

Gopnik, A., & Meltzoff, A. N. (1998). *Words, thoughts, and theories*. Mit Press.

Kuhn, T. S. (1962). The structure of scientific revolutions.

Lombrozo, T. (2006). The structure and function of explanations. *Trends in cognitive sciences*, *10*(10), 464–470.

Platt, J. R. (1964). Strong inference. *SCIENCE*, *146*(3642).

Raghothaman, M., Mendelson, J., Zhao, D., Naik, M., & Scholz, B. (2019). Provenance-guided synthesis of datalog programs. *Proceedings of the ACM on Programming Languages*, *4*(POPL), 1–27.

Rule, J. S. (2020). *The child as hacker: building more human-like models of learning*. Unpublished doctoral dissertation, Massachusetts Institute of Technology.

Schulz, L. (2012). The origins of inquiry: Inductive inference and exploration in early childhood. *Trends in cognitive sciences*, *16*(7), 382–389.

Shapiro, E. Y. (1982). Algorithmic program debugging. *Ph. D. Thesis*.

Staley, R. (2009). Albert michelson, the velocity of light, and the ether drift. *Einstein's generation. The origins of the relativity revolution*.

Tenenbaum, J. B. (1999). *A bayesian framework for concept learning*. Unpublished doctoral dissertation, Massachusetts Institute of Technology.

Ullman, T. D., Goodman, N. D., & Tenenbaum, J. B. (2010). Theory acquisition as stochastic search.

Ullman, T. D., & Tenenbaum, J. B. (2020). Bayesian models of conceptual development: Learning as building models of the world. *Annual Review of Developmental Psychology*, *2*, 533–558.

Zeller, A. (1999). Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, *24*(6), 253–267.