Generating Natural Language Proofs with Verifier-Guided Search

Kaiyu Yang and **Jia Deng** and **Danqi Chen**
Department of Computer Science
Princeton University
{kaiyuy,jiadeng,danqic}@cs.princeton.edu

Abstract

Reasoning over natural language is a challenging problem in NLP. In this work, we focus on proof generation: Given a hypothesis and a set of supporting facts, the model generates a proof tree indicating how to derive the hypothesis from supporting facts. Compared to generating the entire proof in one shot, stepwise generation can better exploit the compositionality and generalize to longer proofs but has achieved limited success on real-world data. Existing stepwise methods struggle to generate proof steps that are both logically valid and relevant to the hypothesis. Instead, they tend to hallucinate invalid steps given the hypothesis. In this paper, we present a novel stepwise method, NLProofS (Natural Language Proof Search), which learns to generate relevant steps conditioning on the hypothesis. At the core of our approach, we train an independent *verifier* to check the validity of the proof steps to prevent hallucination. Instead of generating steps greedily, we search for proofs maximizing a global proof score judged by the verifier. NLProofS achieves state-of-the-art performance on EntailmentBank and RuleTaker. Specifically, it improves the correctness of predicted proofs from 27.7% to 33.3% in the distractor setting of EntailmentBank, demonstrating the effectiveness of NLProofS in generating challenging human-authored proofs.[1]

1 Introduction

A fundamental goal of AI since its inception is automated reasoning (McCarthy et al., 1960): given explicitly provided knowledge as assumptions, we want the system to draw logically valid conclusions. Research in automated reasoning has traditionally focused on structured domains such as formal logic (Robinson and Voronkov, 2001). On the other hand, recent work suggests that free-form natural language can be a suitable vehicle for reasoning (Clark et al., 2020; Dalvi et al., 2021), because natural language represents knowledge without requiring labour-intensive formalization. However, reasoning in natural language is challenging, as it requires compositional generalization to novel examples (Ruis et al., 2020)—a capability that state-of-the-art large language models struggle with (Rae et al., 2021).

In this work, we focus on proof generation in natural language (Fig. 1): given a hypothesis and a set of supporting facts in natural language, the model generates a proof tree indicating how the hypothesis is derived from a subset of the supporting facts. The proof tree may contain intermediate conclusions, which need to be *generated* by the model. Existing methods generate the proof either in a single shot or step by step. Stepwise methods leverage the compositionality of proofs, making it easier for the model to learn and generalize to longer proofs (Tafjord et al., 2021).

However, existing stepwise methods suffer from a trade-off between generating *valid* steps and *relevant* steps. Prior works (Sanyal et al., 2022; Bostrom et al., 2022) have observed that, given the hypothesis, the model often learns to hallucinate invalid proof steps leading to the hypothesis, instead of performing valid logical inference (see examples in Table 3). To mitigate this issue, previous attempts have restricted the model from accessing the hypothesis, forcing it to generate conclusions based solely on known premises. However, without the hypothesis, the model tends to generate many valid but irrelevant steps. This problem is especially prominent for real-world natural language proofs. Due to the inherent ambiguity of natural language, the search space for each proof step is much larger than that of simple synthetic tasks. That may explain why stepwise methods have demonstrated superior performance on the simple, synthetic RuleTaker dataset (Clark et al., 2020) but not on the more realistic, human-authored EntailmentBank dataset (Dalvi et al., 2021), which is the gap we aim to bridge.

We introduce NLProofS, a novel method for stepwise proof generation. It generates proof steps *conditioning on the hypothesis*, enabling the model to learn to generate only relevant steps. To prevent hallucination, it trains an independent *verifier* based on RoBERTa (Liu et al., 2019), which takes a single step (including multiple premises and one conclusion) as input and produces a score indicating its validity. During inference, instead of generating steps greedily, NLProofS searches for proofs

[1]The code is available at https://github.com/princeton-nlp/NLProofS.

Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 89 - 105
December 7-11, 2022

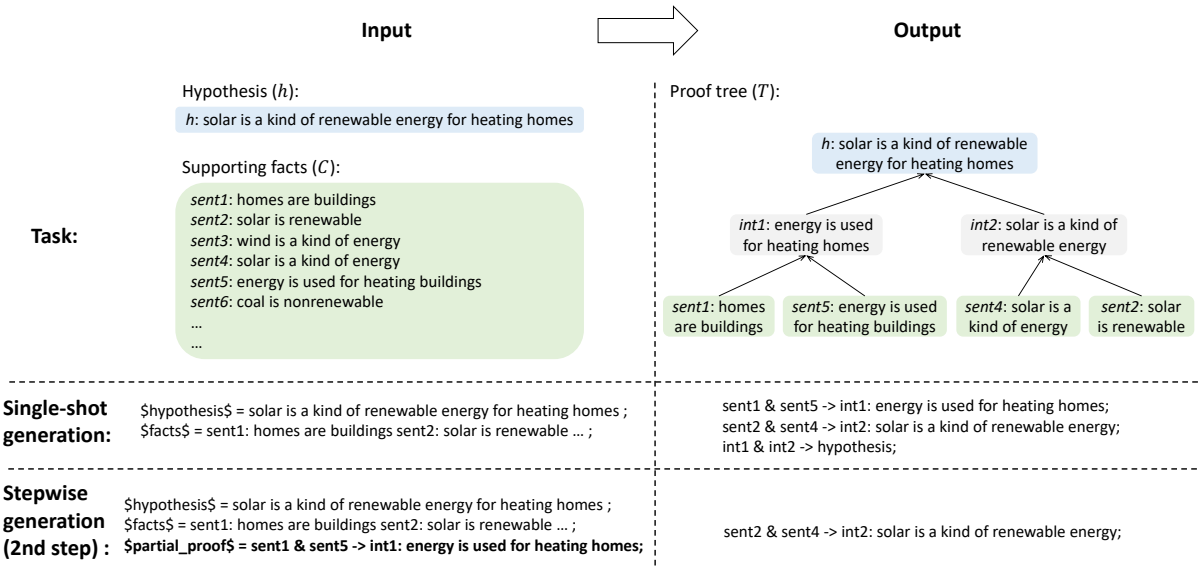

Figure 1: *Top*: In proof generation, given a hypothesis and multiple supporting facts (potentially with redundant facts), the model generates a proof tree, including both the tree structure and the intermediate conclusions (*int1* and *int2*). A common approach encodes the input/output as text sequences and generates the proof in a single-shot (*Middle*) or generates the proof step by step (*Bottom*, showing only one of the three steps) using text-to-text models.

that maximize a proof score aggregating the validity scores of all steps.

We evaluate NLProofS on two benchmarks: RuleTaker (Clark et al., 2020) and EntailmentBank (Dalvi et al., 2021). RuleTaker consists of simple, synthetic English sentences generated from templates. In contrast, proofs in EntailmentBank are authored by human experts and are more challenging. They are in unconstrained natural language and exhibit considerable fuzziness and ambiguity, as is typical for reasoning in natural language. NLProofS achieves state-of-the-art performance on both datasets. On EntailmentBank, it outperforms previous best results by a large margin. For example, in the distractor task setting, it improves the accuracy of generating complete proofs from 27.7% to 33.3% and the accuracy of identifying relevant supporting facts from 46.1% to 58.8%, which demonstrates the effectiveness of our method in generating challenging human-authored proofs.

In addition, we conduct extensive ablations to gain insights. First, we show that the verifier plays a crucial role in generating proofs by providing well-calibrated validity scores. Without the verifier, our method performs worse on EntailmentBank and fails completely on RuleTaker. Second, while generating long proofs remains a major challenge, NLProofS leads to large improvements for long proofs. Third, there is still a large room for future improvement, e.g., by generating more accurate and diverse proof steps as candidates for the search algorithm to explore.

Contributions. In summary, our contributions are two-fold. First, we introduce NLProofS, a stepwise proof generation method that searches for proofs whose validity is scored by a verifier. It substantially advances state-of-the-art performance on the challenging EntailmentBank dataset. Second, through extensive analyses and ablations, we shed light on the performance improvement and reveal the current bottleneck. Our work is a first step exploring the interplay between verifiers and proof search in generating natural language proofs, and we expect further advancements down the road.

2 Related Work

Proof generation in natural language. Table 1 summarizes existing methods for generating natural language proofs, including single-shot and stepwise methods. Single-shot methods generate the entire proof tree in one shot, enforcing structural constraints explicitly via linear integer programming (Saha et al., 2020; Sun et al., 2021) or implicitly via pretrained text-to-text transformers (Gontier et al., 2020; Dalvi et al., 2021) (Fig. 1 *Middle*). In contrast, stepwise methods generate the proof as individual proof steps, forward (Tafjord et al., 2021; Sanyal et al., 2022; Bostrom et al., 2022), backward (Liang et al., 2021; Qu et al., 2022; Dalvi et al., 2022), or both (Hong et al., 2022). Our method generates proofs stepwise, in the forward direction.

When generating a proof step, prior work has observed that if the hypothesis is available, the model often uses it to hallucinate the intermediate conclusion instead of drawing valid logical inferences (Table 3). Therefore, ProofWriter (Tafjord et al., 2021), FaiRR (Sanyal et al., 2022), and SCSearch (Bostrom et al., 2022) explicitly ban the model from accessing the hypothesis when generating intermediate conclusions[2], forcing it to draw

[2]The hypothesis may be used for premise selection.

Method	Stepwise	Proof direction	Generate intermediates w/ hypothesis	Verifier	Non-local search	Evaluated on human-authored proofs	No external data
PRover	✗	N/A	No intermediates	✗	N/A	✗	✓
EntailmentWriter	✗	N/A	✓	✗	N/A	✓	✓
ProofWriter	✓	→	✗	✗	✗	✗	✓
FaiRR	✓	→	✗	✗	✗	✗	✓
SCSearch	✓	→	✗	✗	✗	✗	✗
MetGen	✓	Both	✓	✗	✗	✓	✗
Dalvi et al. (2022)†	✓	←	✓	✓	✗	✗	✗
NLProofS (ours)	✓	→	✓	✓	✓	✓	✓

Table 1: A comparison of NLProofS with existing methods for proof generation: PRover (Saha et al., 2020), EntailmentWriter (Dalvi et al., 2021), ProofWriter (Tafjord et al., 2021), FaiRR (Sanyal et al., 2022), SCSearch (Bostrom et al., 2022), MetGen (Hong et al., 2022), and a concurrent work (Dalvi et al., 2022) marked with †. → and ← denote forward/backward stepwise proof generation.

inference from known premises only. However, without the hypothesis, the model may generate many valid proof steps irrelevant to the hypothesis. Unlike other forward stepwise methods, our model has access to the hypothesis but relies on a verifier to check the validity of proof steps and prevent hallucination.

Dalvi et al. (2022) is a concurrent work that also uses a verifier to score multiple candidate proof steps generated by the model. However, they use the scores to make a greedy local decision, selecting the best step and discarding others, whereas we search for proofs with the maximum aggregated scores. Besides, they train the verifier on additionally annotated negative examples, whereas we train on pseudo-negative examples generated automatically without additional annotation efforts (Sec. 4.2). Other stepwise methods in Table 1 do not have verifiers, and they make local decisions.

PRover (Saha et al., 2020), ProofWriter, and FaiRR have only evaluated on the simple RuleTaker dataset (Clark et al., 2020). And it is nontrivial to extend them to real-world data. For example, FaiRR assumes sentences fall into two categories: rules and facts, which are tailored for RuleTaker. Dalvi et al. (2021) introduce EntailmentBank, a challenging benchmark of proofs authored by human experts, which is used to evaluate EntailmentWriter, their method for single-shot proof generation. SCSearch and Dalvi et al. (2022) also use EntailmentBank but focus on different task settings that do not quantitatively evaluate the generated proofs.

Reasoning in other NLP tasks. Multi-hop reasoning can also be found in open-domain QA (Yang et al., 2018), fact verification (Jiang et al., 2020), and reading comprehension (Min et al., 2019; Sinha et al., 2019; Jiang et al., 2019). Compared to proof generation, reasoning chains in these tasks are much simpler, often consisting of only 2–3 supporting facts. Also, they are more coarse-grained, involving large chunks of texts such as passages instead of simple, short sentences.

Bostrom et al. (2021) generate conclusions from premises. Their method can potentially be a component in proof generation but does not consider whether the generated conclusions are relevant. In math word problems, Cobbe et al. (2021) demonstrate the benefits of using a verifier to re-rank the model's predicted solutions. However, these solutions are unconstrained texts, whereas proofs in our task are structured trees/graphs. Further, we use the verifier during proof generation rather than merely to rank the solutions post hoc. Our verifier is also related to natural language inference (Bowman et al., 2015), especially the multi-premises setting in Lai et al. (2017). Recently, large language models have shown the ability to solve multi-step reasoning through chain-of-thought prompting (Wei et al., 2022; Kojima et al., 2022) on arithmetic, symbolic and commonsense reasoning tasks.

Symbolic reasoning. Classical AI has invested significant efforts in reasoning in symbolic domains, e.g., automated theorem proving (ATP) (Kovács and Voronkov, 2013; Yang and Deng, 2019; Polu and Sutskever, 2020). Researchers have attempted to apply ATP to natural language through semantic parsing (Mineshima et al., 2015; Saparov and Mitchell, 2022). However, it is challenging (if not impossible) for semantic parsers to cover the full complexity of natural language. Therefore, researchers have developed reasoning approaches bypassing semantic parsing (Angeli et al., 2016; Kalyanpur et al., 2022; Yang and Deng, 2021).

One promising example is neurosymbolic reasoning. It uses neural networks to handle the complexity of natural language but incorporates inductive biases inspired by symbolic reasoning (Weber et al., 2019; Smolensky, 1990; Kathryn and Mazaitis, 2018; Lee et al., 2016). Our method also falls into this broad category. It uses large language models to generate individual reasoning steps but chains the steps together into a coherent, tree-structured proof using symbolic search algorithms.

3 Generating Natural Language Proofs

Task definition. Now we define the proof generation task. As in Fig. 1 (*Top*), the input consists of a hypothesis h and a set of supporting facts $C = \{\text{sent}_1, \text{sent}_2, \ldots, \text{sent}_n\}$. Both h and sent_i are natural language sentences. h can be derived from a subset of C through reasoning of one or multiple steps.

The output is a proof tree T specifying how h is derived from C. The tree has h as its root and sent_i as leaf nodes. The intermediate nodes are intermediate conclusions *generated* by the model. Each non-leaf node u corresponds to a reasoning step with u as the conclusion and its children as premises. To successfully perform the task, the model must select relevant sentences from C, use them as leaf nodes to compose a valid proof tree leading to h, and fill in all the intermediate conclusions.

Singles-shot vs. stepwise generation. A simple and effective method for proof generation, popularized by ProofWriter (Tafjord et al., 2021), is to finetune a pretrained T5 model (Raffel et al., 2020) to map the input (h and C) to the output (T), either in a single shot or stepwise. To that end, the input/output must be encoded as text sequences, e.g., encoding the input by concatenating h and C as illustrated in Fig. 1.[3]

The output proof tree can be encoded by post-order traversal. As in Fig. 1, nodes are labeled with identifiers: `sent*` for leaf nodes, `int*` for intermediate nodes, and `hypothesis` for the root. The output sequence is produced by traversing the tree in post-order, generating one proof step at each non-leaf node, using `&` for "and" and `->` for "entails". The tree may correspond to multiple valid sequences due to different ordering between proof steps and between premises within a step. Nevertheless, the evaluation metric can be calculated from the reconstructed trees instead of the raw text sequences.

In single-shot generation, the model generates the output sequence of the entire proof (Fig. 1 *Middle*), whereas in stepwise generation, each time the model takes the current partial proof as input (besides h and C) and generates only the next step (Fig. 1 *Bottom*).

4 Our Method: NLProofS

Now we present NLProofS, our method for generating natural language proofs. It has three main components: (1) a stepwise prover for generating candidate proof steps; (2) a verifier for scoring the validity of proofs; (3) an algorithm for searching for proofs that have high aggregated proof scores.

4.1 Stepwise Prover

Like prior work (Tafjord et al., 2021), we implement the stepwise prover by finetuning a pretrained T5 model. The training data is extracted from the steps in ground truth proofs. Let T be a proof tree and $u \in T$ be a non-leaf node corresponding to a step we want to extract. Take node `int1` in Fig. 1 as an example of u. Non-leaf nodes in T can be categorized into (1) u's descendants, e.g., none in Fig. 1; (2) u itself and its ancestors, e.g., `int1` and `h` in Fig. 1; (3) neither, e.g., `int2` in Fig. 1. The partial proof must include all of (1) but none of (2). It may or may not include nodes in (3). Therefore, for this particular example, the partial proof cannot include `int1` or `h` but has a free choice about whether to include `int2`. When preprocessing the training data, we make these choices randomly as a form of data augmentation.

During inference, the prover may generate syntactically ill-formed proof steps. For the example in Fig. 1 (*Bottom*), "`int1 & int2 -> hypothesis;`" is ill-formed, since the premise `int2` is not available in the current partial proof. We mitigate the issue by generating multiple proof steps from the model via beam search and using heuristics to filter out ill-formed ones, e.g., those with syntactical errors or unavailable premises.

4.2 Verifier

Scoring a proof step. We introduce an independent verifier, which is trained to check the validity of proof steps and prevent the prover from hallucinating invalid steps based on the hypothesis. A proof step has multiple premises and one conclusion. The verifier takes them as input and produces a continuous validity score in $[0, 1]$.

We implement the verifier by finetuning a pretrained RoBERTa model (Liu et al., 2019) to classify proof steps as valid or invalid. The input premises are shuffled randomly and concatenated with the conclusion. For training data, positive examples (valid steps) can be extracted from ground-truth proofs; however, there are no negative examples readily available. Instead of annotating additional negative examples as in Dalvi et al. (2022), we generate pseudo-negative examples automatically. Please refer to Appendix C for details.

Aggregating scores for the entire proof. Step scores are aggregated to produce the score of the entire proof tree. We associate scores with all nodes in the tree recursively. All leaves have a score of 1.0, as they are explicitly provided assumptions that always hold. Each non-leaf node u corresponds to a proof step s from its children $v_1, v_2, \ldots, v_l$ and has a score defined as

$$\texttt{scr}_n(u) = \min\big(\texttt{scr}_s(s), \texttt{scr}_n(v_1), \ldots, \texttt{scr}_n(v_l)\big), \tag{1}$$

where $\texttt{scr}_s(s)$ is the step score, e.g., produced by a verifier. Intuitively, $\texttt{scr}_n(u)$ reflects our confidence in u, and it is monotonically non-increasing w.r.t. the step score and the scores of its children. Eqn. 1 is just one simple way of defining $\texttt{scr}_n(u)$, and we leave a more thorough exploration of scoring options for future work. Finally, the proof score is $\texttt{scr}_n(h)$: the root's score.

4.3 Proof Search

Now we combine the prover and the verifier in our proof search algorithm, which looks for proofs with optimal proof scores. Our method is inspired by automated reasoning in formal logic (Russell and Norvig, 2002), where proofs are found by searching in a large space efficiently. Instead of greedy stepwise proof generation, we search for proof trees in a large proof graph (Fig. 2), allowing the model to explore different paths, recover from errors, and ultimately find better proofs.

[3] Our input encoding scheme has minor differences from EntailmentWriter (Dalvi et al., 2021) (details in Appendix B).

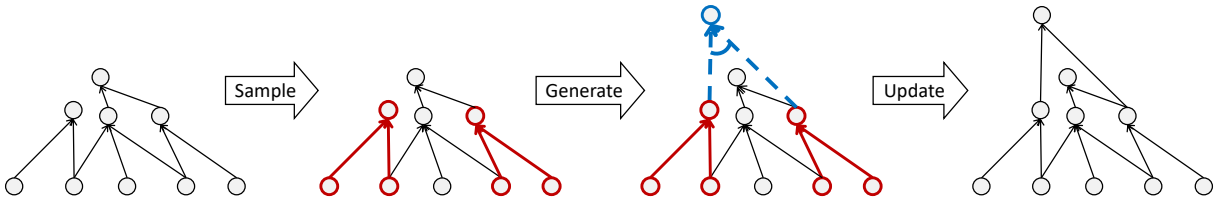

Figure 2: An iteration in the proof search. Nodes for proof step (S) are omitted for simplicity of illustration. (1) Sample a partial proof (red) from the proof graph. (2) Use the stepwise prover to generate potential steps (blue, only showing one but generating multiple) and score them using the verifier. (3) Execute the steps to update the graph.

Definition 4.1 (Proof graph)**.** A proof graph is a directed acyclic graph with the following properties:

- *Nodes*: It has four types of nodes (C, I, S, h), where C corresponds to supporting facts, I corresponds to intermediate conclusions, S consists of proof steps, and h is the hypothesis. Nodes in $\{h\} \bigcup I \bigcup C$ are associated with unique sentences.
- *Edges*: For any proof step node $s \in S$, it has one or more inbound edges, all of which originate from $I \bigcup C$. It has exactly one outbound edge, which points to a node in $I \bigcup \{h\}$. Besides these edges, the graph contains no additional edges. Any node $u \in I \bigcup \{h\}$ has at most one inbound edge.
- *Scores*: All nodes are associated with scores in $[0, 1]$. For any sentence $\text{sent}_i \in C$, $\texttt{scr}_n(\text{sent}_i) = 1$. For any node $u \in I \bigcup \{h\}$, $\texttt{scr}_n(u) = 0$ if it has no inbound edge. Otherwise, it must have exactly one inbound edge from $s \in S$, and s has inbound edges from $\{v_1, \ldots, v_l\} \subseteq I \bigcup C$. $\texttt{scr}_n(u)$ is defined by Eqn. 1. Scores of proof step nodes in S are provided externally by the algorithm that operates on the proof graph.

Proof trees correspond to paths in proof graphs (treating S as "and" nodes in and-or graphs). Therefore, our task is to search for a path from C to h that maximizes $\texttt{scr}_n(h)$. The search algorithm is outlined in Fig. 2 and Algorithm 1. Proof search takes place only in inference. In training, we train a stepwise prover $\mathcal{P}$ and a verifier $\mathcal{V}$. In inference, we use them to iteratively expand the proof graph and update the node scores until the graph can no longer be updated. At that point, we extract the best proof of h found so far.

Initialization (line 1–3 in Algorithm 1). We initialize the proof graph using the greedy proof generated by $\mathcal{P}$. We could also start from scratch, i.e., $I = S = \varnothing$ and $\texttt{scr}(h) = 0$, but the initialization accelerates proof search by providing a non-zero initial score for h, which can be used to prune unpromising paths during search.

Iteration (line 5–13 in Algorithm 1). We use $\mathcal{P}$ to generate proof steps for updating the graph. $\mathcal{P}$ is trained on partial proof trees rather than graphs. So in each iteration, we first sample a new partial proof tree from the graph as the candidate for expansion (details in Appendix E). Then, we use $\mathcal{P}$ to generate multiple proof steps $s_1, s_2, \ldots, s_k$ through beam search followed by filtering as discussed in Sec. 4.1. We calculate step scores $\texttt{scr}_s(s_1), \texttt{scr}_s(s_2), \ldots, \texttt{scr}_s(s_k)$ by averaging verifier scores v_scrs from $\mathcal{V}$ (Sec. 4.2) and prover scores p_scrs from $\mathcal{P}$, which are the likelihood scores in beam search.

Then we try to update the proof graph by executing these steps. Assume a step s_i has premises $v_1, \ldots, v_l$ and a conclusion u. First, we use Eqn. 1 to calculate a tentative score $\widehat{\texttt{scr}}_n(u)$. If u is an existing node in the graph with $\texttt{scr}_n(u) \geq \widehat{\texttt{scr}}_n(u)$, the step becomes a no-op, and we do not perform any update. Otherwise, there are two cases: (1) If u is not in the graph (Fig. 2), we just create a new node for it with $\texttt{scr}_n(u) = \widehat{\texttt{scr}}_n(u)$; (2) If u is in the graph and $\texttt{scr}_n(u) < \widehat{\texttt{scr}}_n(u)$, we update u by replacing the existing proof step leading to it with the new step s_i with $\texttt{scr}_n(u) = \widehat{\texttt{scr}}_n(u)$. According to Eqn. 1, the score change may affect u's successors, so we propagate the change to all of them.

Proof extraction (line 14 in Algorithm 1). When all proof steps in an iteration are no-op, we stop and extract the best proof of h found so far, which simply consists of all predecessors of h. The result is guaranteed to be a tree, as we prove in Appendix D.

Algorithm 1: Proof search.

Input : Hypothesis h, supporting facts C, stepwise prover $\mathcal{P}$, verifier $\mathcal{V}$
Output : Proof tree T

```
1  G ← generate_greedy(P, h, C)
2  PG ← initialize_graph(G)
3  explored ← ∅
4  while true do
5      partial_proof ← sample_new(PG, explored)
6      explored ← explored ∪ {partial_proof}
7      steps, p_scrs ← generate(P, partial_proof)
8      v_scrs ← verify(V, steps)
9      scrs ← (p_scrs + v_scrs)/2
10     PG' ← update(PG, steps, scrs)
11     if PG' = PG then
12         break
13     PG ← PG'
14 return extract_proof(PG)
```

5 Main Results

5.1 Experimental Setup

We evaluate NLProofS on proof generation using two benchmarks: a real-world benchmark EntailmentBank (Dalvi et al., 2021) and a synthetic benchmark RuleTaker (Clark et al., 2020). Training and inference details are in Appendix F. Bostrom et al. (2022) also evaluated on EntailmentBank but deviated from the original setting, instead formulating the task as distinguishing verifiable hypotheses from unverifiable ones. In order to have a fair comparison with their work, we also evaluate under their setting in Appendix G.

EntailmentBank. EntailmentBank consists of 1,840 proof trees constructed by expert annotators (1,313 for training, 187 for validation, and 340 for testing). It comes with three variants of the proof generation task (Sec. 3) with varying numbers of distractors in supporting facts C. *Task 1* does not have any distractor, i.e., C consists of exactly the leaf nodes of the ground truth proof tree. In *Task 2*, C always has 25 sentences, including ground truth supporting facts as well as distractors. In *Task 3*, C is a large corpus of 12K sentences derived from WorldTree V2 (Xie et al., 2020), requiring the model to retrieve relevant supporting facts from the corpus. We evaluate on all three tasks. Our method is directly applicable to Task 1 and Task 2. For Task 3, Dalvi et al. (2021) retrieve 25 supporting facts for each hypothesis. We use the same retrieved supporting facts and focus solely on proof generation.[4] And following their practice, we train the model on Task 2 and evaluate its zero-shot performance on Task 3.

A generated proof tree $\hat{T}$ is compared against the ground truth T using official metrics developed by EntailmentBank. In summary, the leaves, proof steps, and intermediate conclusions in $\hat{T}$ are compared against those in T to produce two metrics: the F1 score, and the AllCorrect score which evaluates exact matches.[5] In addition, the Overall-AllCorrect metric measures whether $\hat{T}$ is identical to T. As a caveat, these metrics do not account for the existence of multiple valid proof trees. Metrics for evaluating leaves are less impacted by this issue, as multiple valid trees often have the same set of leaves. Please refer to Appendix A and EntailmentBank for additional details. We report results produced by their official evaluation code.[6]

RuleTaker. To demonstrate the broad applicability of NLProofS to different reasoning datasets, we also evaluate on RuleTaker. In RuleTaker, h can be either proved, disproved, or neither. The model has to do two things: (1) predict the answer as one of those three categories and (2) generate a proof when h can be proved or disproved. Unlike EntailmentBank, examples in RuleTaker are made of simple, synthetic English sentences generated by templates. We use the OWA (open-world assumption) version of the dataset introduced by Tafjord et al. (2021). Following the setup in Sanyal et al. (2022), we train and test on the D0–D3 subset, which consists of proofs of depth ≤ 3. It has 129K examples—90K for training, 13K for validation, and 26K for testing.

The predicted answer is evaluated using accuracy, whereas proofs are evaluated using Overall-AllCorrect but ignoring the intermediate conclusions.[7]

5.2 Proof Generation on EntailmentBank

Table 2 shows test results on EntailmentBank. We compare with EntailmentWriter (Dalvi et al., 2021) and MetGen (Hong et al., 2022): two prior state-of-the-art methods that also finetune a T5 model to generate proofs. EntailmentWriter generates the entire proof in a single shot, whereas MetGen generates the proof stepwise. EntailmentWriter has two versions, one with T5-large (737 million parameters) and the other with T5-11B (11 billion parameters). All other methods, including ours, use only T5-large due to computational constraints.

NLProofS significantly outperforms EntailmentWriter across the board. Take Task 2 as an example. First, it generates more correct proofs overall, improving the Overall-AllCorrect metric from 20.9% to 33.3%. Second, it identifies relevant supporting facts more effectively, improving the Leaves-AllCorrect from 35.6% to 58.8%. Third, it generates more accurate proof steps and intermediate conclusions, as demonstrated by the Steps and Intermediates metrics. Moreover, our method with T5-large even outperforms EntailmentWriter with T5-11B by a large margin.

Compared to MetGen, we perform competitively on Task 1 and Task 3 but much better on Task 2, improving the Overall-AllCorrect metric from 27.7% to 33.3%. Note that our model is trained only on EntailmentBank, whereas MetGen requires much more data annotation efforts (Sec. 4.1.2 in Hong et al. (2022)). First, the MetGen authors manually design templates of different reasoning types and use them to collect additional training data from Wikipedia. Second, they manually annotate the reasoning types of 400 training proof steps in EntailmentBank. MetGen needs these annotations since the model takes the reasoning type as input.

We also examine whether the proof can be generated in a single shot by very large language models such as GPT-3 (Brown et al., 2020) or Codex (Chen et al., 2021), through prompting with in-context examples. Results in Appendix H show that in-context prompting performs substantially worse than our method.

Table 3 shows two examples of invalid steps generated by EntailmentWriter but avoided by NLProofS , likely due to its verifier. In the first example, "June" in EntailmentWriter's conclusion is hallucinated based on the hypothesis, as the word does not appear in the

[4] Appendix H includes additional discussion on prior work focusing on improving the retriever (Ribeiro et al., 2022).

[5] Intermediates are compared using BLEURT (Sellam et al., 2020): a learning-based sentence similarity measure.

[6] https://github.com/allenai/entailment_bank

[7] The metric was introduced by PRover (Saha et al., 2020).

Task	Method	Leaves		Steps		Intermediates		Overall
		F1	AllCorrect	F1	AllCorrect	F1	AllCorrect	AllCorrect
Task 1 (no-distractor)	EntailmentWriter	98.7	86.2	50.5	37.7	67.6	36.2	33.5
	EntailmentWriter (T5-11B)	99.0	89.4	51.5	38.2	71.2	38.5	35.3
	MetGen†	**100.0**	**100.0**	**57.7**	41.9	70.8	39.2	36.5
	NLProofS (ours)	97.8 ± 0.2	90.1 ± 1.2	55.6 ± 0.6	**42.3 ± 0.4**	**72.4 ± 0.5**	**40.6 ± 0.7**	**38.9 ± 0.7**
Task 2 (distractor)	EntailmentWriter	84.3	35.6	35.5	22.9	61.8	28.5	20.9
	EntailmentWriter (T5-11B)	89.1	48.8	41.4	27.7	66.2	31.5	25.6
	MetGen†	82.7	46.1	41.3	29.6	61.4	32.4	27.7
	NLProofS (ours)	**90.3 ± 0.4**	**58.8 ± 1.8**	**47.2 ± 1.7**	**34.4 ± 1.7**	**70.2 ± 0.5**	**37.8 ± 1.6**	**33.3 ± 1.5**
Task 3 (full-corpus)	EntailmentWriter	35.7	2.9	6.1	2.4	33.4	7.7	2.4
	EntailmentWriter (T5-11B)	39.9	3.8	7.4	2.9	35.9	7.1	2.9
	MetGen†	34.8	**8.7**	9.8	**8.6**	36.6	**20.4**	**8.6**
	NLProofS (ours)	**43.2 ± 0.6**	8.2 ± 0.7	**11.2 ± 0.6**	6.9 ± 0.7	**42.9 ± 1.0**	17.3 ± 0.5	6.9 ± 0.7

Table 2: Test results of proof generation on EntailmentBank (Dalvi et al., 2021). †: MetGen (Hong et al., 2022) is trained on *additional data* collected from Wikipedia, whereas other methods are trained only on EntailmentBank. Here we report the results of the MetGen-prefixed model, as the other MetGen-separated model performs slightly better but is 5 times larger. All methods are based on T5-large (Raffel et al., 2020) unless otherwise noted. For our method, we report the average performance and the standard deviation for 5 independent runs. Bold and underlined texts highlight the best method and the runner-up.

Hypothesis	Premises	Conclusions generated by models
The next new moon will occur on June 30.	1. A new moon is a kind of phase of the moon. 2. A moon phase occurs 28 days after the last time it occurs.	**EntailmentWriter**: The next new moon will occur 28 days after June 2.
		NLProofS (ours): The next new moon will occur 28 days after the last new moon.
Planting trees prevents soil from washing away.	1. Planting trees increases the amount of trees in an environment. 2. Tree roots decrease / reduce soil erosion.	**EntailmentWriter**: Plants trees increases the amount of trees in an environment.
		NLProofS (ours): Planting trees decreases soil erosion.

Table 3: Examples of invalid proof steps generated by EntailmentWriter (Dalvi et al., 2021) but not our method. In the first example, "June" in the conclusion is hallucinated rather than derived from the premises. In the second example, EntailmentWriter simply copies one of the premises without performing any meaningful reasoning.

premises. The second example is a typical undesirable behavior also observed by Bostrom et al. (2022). When the model has difficulties in generating a conclusion, it falls back into copying one of the premises. Our method generates reasonable conclusions in these two examples.

5.3 Generating Answers and Proofs on RuleTaker

Hypotheses in RuleTaker can be provable, disprovable, or neither. To benchmark on RuleTaker, we use a similar scheme to Bostrom et al. (2022) to adapt any proof generation system capable of producing proof scores. In training, we (1) discard hypotheses that are neither provable nor disprovable and (2) convert disprovable hypotheses into provable ones by negating them. We negate sentences by adding an "I don't think" prefix.

In testing, given a hypothesis h, we try to generate proofs and the associated scores for both h and its negation $\neg h$. Then we train a linear classifier on top of the two scores to predict the answer. Depending on the predicted answer, we take the generated proof to be the proof of h, $\neg h$, or neither.

Results are in Table 4. ProofWriter is the iterative model in Tafjord et al. (2021). It generates proofs stepwise based on T5. Our method performs competitively with ProofWriter and FaiRR (Sanyal et al., 2022).

6 Analyses

6.1 Ablation Studies

EntailmentBank. Our full model searches for stepwise proofs, relying on both the verifier and the prover for producing scores. We conduct ablation studies on Task 2 of EntailmentBank to better understand the empirical gains coming from each of these components.

First, we compare the full model with the stepwise prover without search (Sec. 4.1). Results in Table 5 show that the full model significantly improves upon this stepwise baseline across the board, demonstrating the benefits of searching for proofs at inference time.

Note that the stepwise baseline without search also performs significantly better than EntailmentWriter (31.8% vs. 20.9% in Overall-AllCorrect). We ask how much of the improvement is due to stepwise proof generation as opposed to implementation details and hyperparameters.[8]. In Table 5, we replicate EntailmentWriter

[8]We do not have access to the exact details of Entailmen-

Method	Answer accuracy						Proof accuracy					
	N/A	0	1	2	3	All	N/A	0	1	2	3	All
FaiRR†	99.6	**100.0**	99.7	98.9	96.6	99.2	99.6	**100.0**	99.5	97.2	95.3	98.8
ProofWriter†	**99.7**	**100.0**	99.9	**99.7**	**99.7**	**99.8**	**99.7**	**100.0**	99.9	**99.4**	**99.1**	**99.7**
NLProofS (ours)	99.5	**100.0**	**100.0**	99.4	96.4	99.3	99.5	**100.0**	**100.0**	**99.4**	95.1	99.2

Table 4: Test results on RuleTaker (OWA) (Tafjord et al., 2021). Models are trained and tested on the D0–D3 subset. Methods with † are reported by Sanyal et al. (2022). All methods are based on T5-large. The 'All' columns are accuracies on the entire testing set. '0', '1', '2', and '3' are accuracies broken down by the length of testing proofs. 'N/A' includes testing examples without ground truth proofs since they can be neither proved nor disproved.

Method	Leaves		Steps		Intermediates		Overall	Time
	F1	AllCorrect	F1	AllCorrect	F1	AllCorrect	AllCorrect	
NLProofS (full model)	**90.3 ± 0.4**	**58.8 ± 1.8**	**47.2 ± 1.7**	**34.4 ± 1.7**	**70.2 ± 0.5**	**37.8 ± 1.6**	**33.3 ± 1.5**	4.4
w/o search	89.7 ± 0.6	56.5 ± 1.7	45.9 ± 1.3	33.7 ± 1.4	67.4 ± 2.3	36.4 ± 1.5	31.8 ± 1.4	2.2
w/o search w/o stepwise	86.9 ± 0.6	45.6 ± 1.5	42.6 ± 1.6	29.7 ± 1.3	64.6 ± 1.4	32.2 ± 2.1	27.1 ± 1.5	**1.9**
w/o prover score	**90.3 ± 0.7**	57.8 ± 1.9	43.9 ± 0.9	30.4 ± 0.5	68.9 ± 0.5	35.3 ± 1.2	29.7 ± 0.8	4.6
w/o verifier score	89.7 ± 0.6	55.8 ± 2.2	45.8 ± 1.4	33.8 ± 1.5	68.5 ± 0.4	36.1 ± 1.4	31.9 ± 1.3	3.3

Table 5: Ablation results on Task 2 of the test set of EntailmentBank. The last column shows the average inference time (in seconds) per test example when running with a batch size of 1 and a beam width of 10.

(*Ours w/o search w/o stepwise*) in a unified implementation with our other methods. It outperforms EntailmentWriter (27.1% vs. 20.9% in Overall-AllCorrect) but still falls behind the stepwise baseline, demonstrating the effectiveness of stepwise proof generation.

Instead of averaging the scores from the prover and the verifier, what if we use only one of them? Can we still produce accurate and well-calibrated scores that are useful in proof search? In Table 5, we experiment with two additional versions of NLProofS: one without the prover score and the other without the verifier score. Results show that they fail to improve upon the stepwise baseline without search, demonstrating the necessity of combining the verifier and the prover.

Table 5 also includes different methods' average inference time per test example, measured with batch size 1 and beam width 10. NLProofS takes 4.4 seconds to process a test example, which is 2x slower than the stepwise baseline. Longer run time is a natural consequence of proof search, and 2x is a modest slow down.

RuleTaker. We perform similar ablation experiments also on RuleTaker. Results in Table 6 show similar patterns as the ablations on EntailmentBank. However, the main difference is that proof search leads to a much larger improvement on RuleTaker, and the two baselines without search perform much lower than prior methods (ProofWriter in Table 4). This is due to how we adapt proof generation systems to the task of RuleTaker.

As described in Sec. 5.3, the answer is produced by a linear classifier over proof scores of the hypothesis h and its negation $\neg h$. To perform well, we need the proof generation system to (1) assign high scores to valid hypotheses and (2) assign low scores to invalid hypotheses. However, proof generation systems without verifiers—such as the two baselines—have never seen invalid proof steps in training. They are good at (1) but not necessarily (2); this is sufficient for EntailmentBank (with only valid hypotheses) but not RuleTaker. In contrast, proof generation systems with verifiers—such as our full model—are good at both (1) and (2). In other words, NLProofS can generate more accurate proof scores for both valid and invalid hypotheses.

In addition, Table 6 shows that the verifier score alone is sufficient for RuleTaker; adding the prover score does not make much difference. This is because verifiers trained on RuleTaker are highly accurate, and they do not need to be supplemented by the prover.

6.2 NLProofS with Oracles

The stepwise prover and the verifier are two major components in NLProofS. To analyze which one is the bottleneck, we construct "oracle" versions of them, both of which have access to ground-truth information for better predictions. Given a partial proof, the oracle prover generates multiple potential proof steps just like a regular prover. But it additionally includes all ground truth steps that are valid for the partial proof, i.e., steps whose premises have been satisfied by the partial proof. The oracle verifier also builds on a regular verifier but always assigns the highest score (1.0) to proof steps in the ground truth. Note that we call them "oracles", but neither of them is perfect. For example, the oracle verifier cannot reliably tell whether a proof step is valid if the step deviates even slightly from the ground truth.

Table 7 shows the validation results on Task 2 of EntailmentBank. The oracle prover alone improves the performance significantly (e.g., a boost of 29.2 in Overall-AllCorrect), demonstrating that the prover is a major bottleneck. In contrast, the oracle verifier alone does not help much, improving only 0.8 in Overall-AllCorrect.

tWriter since the training code has not been released.

Method	Answer accuracy						Proof accuracy					
	N/A	0	1	2	3	All	N/A	0	1	2	3	All
NLProofS (full model)	99.5	**100.0**	**100.0**	**99.4**	**96.4**	**99.3**	99.5	**100.0**	**100.0**	**99.4**	**95.1**	**99.2**
w/o search	89.2	55.9	41.7	33.7	36.8	64.3	89.2	99.6	81.2	62.4	65.0	84.2
w/o search w/o stepwise	77.5	49.9	27.1	30.4	30.4	54.8	77.5	97.0	49.2	59.8	56.9	72.7
w/o prover score	**99.6**	**100.0**	99.9	98.9	96.2	**99.3**	**99.6**	99.8	99.8	98.8	94.3	99.0
w/o verifier score	86.5	55.6	47.2	47.2	52.2	67.0	86.5	98.9	93.3	95.5	92.5	91.4

Table 6: Ablation results on the D0–D3 test set of RuleTaker (OWA) (Tafjord et al., 2021).

Method	Leaves		Steps		Intermediates		Overall
	F1	AllCorrect	F1	AllCorrect	F1	AllCorrect	AllCorrect
NLProofS (no oracle)	89.4 ± 0.8	56.0 ± 0.7	50.4 ± 1.9	38.4 ± 1.3	71.9 ± 1.4	41.3 ± 1.4	37.1 ± 1.5
oracle verifier	90.0 ± 0.5	56.9 ± 1.6	51.3 ± 2.2	39.1 ± 2.1	72.9 ± 1.5	42.0 ± 2.1	37.9 ± 2.2
oracle prover	94.6 ± 0.3	76.2 ± 1.9	75.8 ± 0.9	67.0 ± 1.8	85.3 ± 0.5	67.5 ± 1.6	66.3 ± 1.9
oracle prover + verifier	**95.7 ± 0.7**	**82.7 ± 2.7**	**83.0 ± 3.1**	**75.4 ± 3.9**	**88.1 ± 1.8**	**75.7 ± 4.1**	**75.2 ± 3.8**

Table 7: Validation results on EntailmentBank (Task 2) of replacing the prover/verifier with oracles.

However, 0.8 might underestimate the importance of the verifier, as the oracle verifier is not useful if the prover fails to generate the ground truth proof step in the first place. Actually, adding the oracle verifier to the oracle prover improves Overall-AllCorrect by 8.9, demonstrating that the verifier also bears room for improvement.

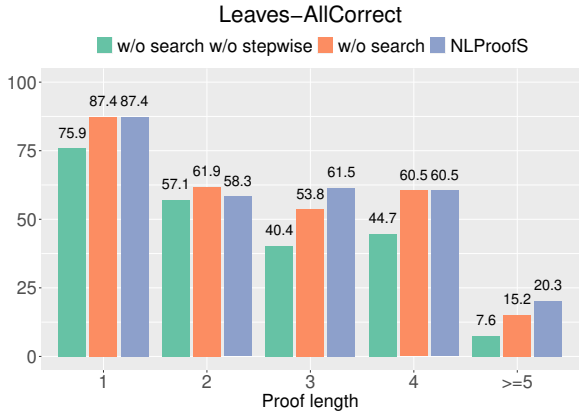

Figure 3: Test results on Task 2 (distractor) broken down by the length of the ground truth proof. Here we show the Leaves-AllCorrect metric.

6.3 Impact of Proof Length

Prior work has demonstrated that proof generation models struggle with long proofs (Dalvi et al., 2021). In Fig. 3, we break down the test performance on Task 2 of EntailmentBank by proof length, i.e., the number of steps in the ground truth proof. Here we show only the Leaves-AllCorrect metric. Leaves metrics are relatively robust against the issue of multiple valid proof trees per example. Appendix H include figures of other metrics. But they may exaggerate the difficulty with long proofs, as the issue of multiple valid proofs is particularly prominent for long proofs. Nevertheless, we still see a significant performance drop in Fig. 3 when the proof length exceeds 1–2, suggesting that generating long proofs remains a challenge. However, we also see the benefits of proof search since its improvements over the stepwise baseline are more evident for long proofs.

In addition, NLProofS tends to generate longer proofs compared to the baselines. On the validation set of Task 2, the ground truth proofs have an average length of 3.2 steps, whereas the average lengths of the generated proofs are 2.6, 2.7, and 2.9 for the single-shot baseline, the stepwise baseline, and NLProofS.

6.4 Reduced Hallucination

The verifier in NLProofS aims to prevent the model from hallucinating invalid proof steps. However, it is difficult to evaluate hallucination automatically: when the model generation deviates from ground truth, it is difficult to evaluate whether it is a valid proof step. Therefore, besides qualitative examples in Table 3, we also perform a human evaluation similar to Bostrom et al. (2022).

We compare three models: EntailmentWriter, NLProofS w/o search (our model without the verifier-guided search), and NLProofS (our full model). For each model, we sample 100 generated proof steps and manually annotate them as valid/invalid. The percentage of valid steps is 43%, 65%, and 77%, demonstrating the effectiveness of NLProofS in mitigating hallucination.

7 Conclusion

We have introduced NLProofS for stepwise proof generation in natural language. It learns to generate relevant proof steps conditioning on the hypothesis. To prevent hallucination, NLProofS searches for proofs that maximize a validity score judged by a verifier. Our method has achieved state-of-the-art performance on EntailmentBank and RuleTaker, demonstrating the promise of stepwise proof generation for human-authored proofs. In the future, we hope to see increasing applications of verifiers and proof search in various reasoning tasks.

Limitations

Despite the strong performance on two benchmarks, our method still has substantial room for future improvement. Currently, the prover (Sec. 4.1) uses beam search as the decoding algorithm, which has two problems: First, it generates equivalent proof steps such as "`sent1 & sent2 -> hypothesis`" and "`sent2 & sent1 -> hypothesis`". It would be more efficient if we make the decoding invariant to the permutation of premises. Second, the generated proof steps lack diversity. Since the verifier can filter out invalid proof steps, it is more important for the prover to have coverage and diversity than precision. It would be interesting to try more advanced decoding algorithms such as Diverse Beam Search (Vijayakumar et al., 2018). Like prior work (Tafjord et al., 2021; Dalvi et al., 2021), our prover concatenates all supporting facts into a long text sequence and applies a Transformer encoder to it. This could be an inefficient use of computation and may have problems scaling to longer sentences or a larger number of supporting facts. Solutions like Fusion-in-Decoder (Izacard and Grave, 2021) may help solve this problem.

Ethical Considerations

Machine learning and NLP are moving from lab curiosity into real-world systems that make critical decisions in areas such as hiring, loan approval, and college admission. It is imperative that these decisions are interpretable to humans. Proof generation enhances interpretability by requiring the model to produce not only the final decision but also an explicit proof. However, the interpretability is jeopardized if the model learns to hallucinate invalid proof steps, like a person trying to find unfaithful excuses to justify a decision. Our method uses an independently trained verifier to check the validity of proof steps, which effectively reduces hallucination and enables the generated proof to explain the decision more faithfully.

Acknowledgements

This work is partially supported by the Office of Naval Research under Grant N00014-20-1-2634. We gratefully acknowledge financial support from the Schmidt DataX Fund at Princeton University made possible through a major gift from the Schmidt Futures Foundation. We also thank Darby Haller, Jane Pan, Shunyu Yao, and the members of the Princeton NLP group for helpful discussion and valuable feedback.

References

Gabor Angeli, Neha Nayak, and Christopher D Manning. 2016. Combining natural logic and shallow reasoning for question answering. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Kaj Bostrom, Zayne Sprague, Swarat Chaudhuri, and Greg Durrett. 2022. Natural language deduction through search over statement compositions. *arXiv preprint arXiv:2201.06028*.

Kaj Bostrom, Xinyu Zhao, Swarat Chaudhuri, and Greg Durrett. 2021. Flexible generation of natural language deductions. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Samuel Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. 2015. A large annotated corpus for learning natural language inference. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. 2020. Transformers as soft reasoners over language. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Bhavana Dalvi, Peter Jansen, Oyvind Tafjord, Zhengnan Xie, Hannah Smith, Leighanna Pipatanangkura, and Peter Clark. 2021. Explaining answers with entailment trees. *arXiv preprint arXiv:2104.08661*.

Bhavana Dalvi, Oyvind Tafjord, and Peter Clark. 2022. Towards teachable reasoning systems. *arXiv preprint arXiv:2204.13074*.

Nicolas Gontier, Koustuv Sinha, Siva Reddy, and Chris Pal. 2020. Measuring systematic generalization in neural proof generation with transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Ruixin Hong, Hongming Zhang, Xintong Yu, and Changshui Zhang. 2022. MetGen: A module-based entailment tree generation framework for answer explanation. In *Findings of the North American Chapter of the Association for Computational Linguistics: NAACL*.

Gautier Izacard and Édouard Grave. 2021. Leveraging passage retrieval with generative models for open domain question answering. In *European Chapter of the Association for Computational Linguistics (EACL)*, pages 874–880.

Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. 2020. HoVer: A dataset for many-hop fact extraction and claim verification. In *Findings of the Association for Computational Linguistics: EMNLP*.

Yichen Jiang, Nitish Joshi, Yen-Chun Chen, and Mohit Bansal. 2019. Explore, propose, and assemble: An interpretable model for multi-hop reading comprehension. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Aditya Kalyanpur, Tom Breloff, David Ferrucci, Adam Lally, and John Jantos. 2022. Braid: Weaving symbolic and neural knowledge into coherent logical explanations. In *AAAI Conference on Artificial Intelligence*.

William W Cohen Fan Yang Kathryn and Rivard Mazaitis. 2018. TensorLog: Deep learning meets probabilistic databases. *Journal of Artificial Intelligence Research*, 1:1–15.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Laura Kovács and Andrei Voronkov. 2013. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification (CAV)*.

Alice Lai, Yonatan Bisk, and Julia Hockenmaier. 2017. Natural language inference from multiple premises. In *International Joint Conference on Natural Language Processing (IJCNLP)*.

Moontae Lee, Xiaodong He, Wen-tau Yih, Jianfeng Gao, Li Deng, and Paul Smolensky. 2016. Reasoning in vector space: An exploratory study of question answering. In *International Conference on Learning Representations (ICLR)*.

Zhengzhong Liang, Steven Bethard, and Mihai Surdeanu. 2021. Explainable multi-hop verbal reasoning through internal monologue. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

John McCarthy et al. 1960. *Programs with common sense*. RLE and MIT Computation Center.

Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2019. Multi-hop reading comprehension through question decomposition and rescoring. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Koji Mineshima, Pascual Martínez-Gómez, Yusuke Miyao, and Daisuke Bekki. 2015. Higher-order logical inference with compositional semantics. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*.

Hanhao Qu, Yu Cao, Jun Gao, Liang Ding, and Ruifeng Xu. 2022. Interpretable proof generation via iterative backward reasoning. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. 2021. Scaling language models: Methods, analysis & insights from training Gopher. *arXiv preprint arXiv:2112.11446*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research (JMLR)*, 21:1–67.

Danilo Ribeiro, Shen Wang, Xiaofei Ma, Rui Dong, Xiaokai Wei, Henry Zhu, Xinchi Chen, Zhiheng Huang, Peng Xu, Andrew Arnold, et al. 2022. Entailment tree explanations via iterative retrieval-generation reasoner. In *Findings of the North American Chapter of the Association for Computational Linguistics: NAACL*.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Alan JA Robinson and Andrei Voronkov. 2001. *Handbook of automated reasoning*, volume 1. Elsevier.

Laura Ruis, Jacob Andreas, Marco Baroni, Diane Bouchacourt, and Brenden M Lake. 2020. A benchmark for systematic generalization in grounded language understanding. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Stuart Russell and Peter Norvig. 2002. *Artificial intelligence: a modern approach*.

Swarnadeep Saha, Sayan Ghosh, Shashank Srivastava, and Mohit Bansal. 2020. PRover: Proof generation for interpretable reasoning over rules. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Soumya Sanyal, Harman Singh, and Xiang Ren. 2022. FaiRR: Faithful and robust deductive reasoning over natural language. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Abulhair Saparov and Tom M Mitchell. 2022. Towards general natural language understanding with probabilistic worldbuilding. *Transactions of the Association for Computational Linguistics (TACL)*, 10:325–342.

Thibault Sellam, Dipanjan Das, and Ankur Parikh. 2020. BLEURT: Learning robust metrics for text generation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L Hamilton. 2019. CLUTRR: A diagnostic benchmark for inductive reasoning from text. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Paul Smolensky. 1990. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46:159–216.

Changzhi Sun, Xinbo Zhang, Jiangjie Chen, Chun Gan, Yuanbin Wu, Jiaze Chen, Hao Zhou, and Lei Li. 2021. Probabilistic graph reasoning for natural proof generation. In *Findings of the Association for Computational Linguistics: ACL*.

Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. 2021. ProofWriter: Generating implications, proofs, and abductive statements over natural language. In *Findings of the Association for Computational Linguistics: ACL*.

Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. 2018. Diverse Beam Search: Decoding diverse solutions from neural sequence models. In *AAAI Conference on Artificial Intelligence*.

Leon Weber, Pasquale Minervini, Jannes Münchmeyer, Ulf Leser, and Tim Rocktäschel. 2019. NLProlog: Reasoning with weak unification for question answering in natural language. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Zhengnan Xie, Sebastian Thiem, Jaycie Martin, Elizabeth Wainwright, Steven Marmorstein, and Peter Jansen. 2020. WorldTree V2: A corpus of science-domain structured explanations and inference patterns supporting multi-hop inference. In *International Conference on Language Resources and Evaluation (LREC)*.

Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*.

Kaiyu Yang and Jia Deng. 2021. Learning symbolic rules for reasoning in quasi-natural language. *arXiv preprint arXiv:2111.12038*.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

A Evaluation Metrics on EntailmentBank

We evaluate on EntailmentBank using their official evaluation metrics calculated by their evaluation code. Below is a summary; please refer to the EntailmentBank paper for further details.

Let $\hat{T}$ be a generated proof tree, with T being the ground truth. First, nodes in $\hat{T}$ are aligned with nodes in T using a tree alignment algorithm based on the "sent*" labels. Once aligned, it is scored using 4 types of metrics—Leaves, Steps, Intermediates, and Overall.

- *Leaves (F1, AllCorrect)*: The Leaves metrics compare the leaf nodes of $\hat{T}$ and T to calculate an F1 score and an "AllCorrect" score, which means all predicted nodes are correct. In other words, AllCorrect = 1 if F1 = 1, and AllCorrect = 0 if F1 < 1.
- *Steps (F1, AllCorrect)*: The Steps metrics measure whether predicted proof steps are structurally correct. A predicted step corresponds to an internal node $u \in \hat{T}$ (aligned to $v \in T$). It is structurally correct if the children of u and v are also perfectly aligned. Since there are multiple steps in $\hat{T}$ and T, we can calculate F1 and AllCorrect.
- *Intermediates (F1, AllCorrect)*: An intermediate conclusion $u \in \hat{T}$ (aligned to $v \in T$) is correct if the BLEURT (Sellam et al., 2020)[9] score between u and v is greater than 0.28. We calculate F1 and AllCorrect from all intermediate conclusions in $\hat{T}$ and T.
- *Overall (AllCorrect)*: The Overall metric evaluates whether the leaves, steps, and intermediates are all correct, i.e., AllCorrect = 1 if and only if $\hat{T}$ matches completely with T.

B Different Input Formats

We use a slightly different input format from EntailmentWriter (Dalvi et al., 2021), as their format had not been released when we developed our method.

Consider the example in Fig. 1. The input to our single-shot baseline (*NLProofS w/o search w/o stepwise* in Table 5) is "`$hypothesis$ = solar is a kind of renewable energy for heating homes ; $context$ = sent1: homes are buildings sent2: solar is renewable . . . ;`", whereas the their input is "`$proof$ ; $question$ = As a kind of renewable energy, what can solar be used for? ; $answer$ = heating homes ; $hypothesis$ = solar is a kind of renewable energy for heating homes ; $context$ = sent1: homes are buildings sent2: solar is renewable . . . ;`", which includes more information (`$question$` and `$answer$`) than ours.

We experiment with single-shot methods implemented in our codebase using their input format. Results in Table A indicate no significant difference.

[9] We use the `bleurt-large-512` model following Dalvi et al. (2021).

C Pseudo-negative Examples for Training the Verifier

As mentioned in Sec. 4.2, the negative examples used for training the verifier are constructed automatically using the procedure below:

- As in Fig. A, for each positive example consisting of premises and a conclusion, we either remove some premises or replacing one premise with a distractor retrieved from C using BM25 (Robertson et al., 2009).
- For EntailmentBank, as in Fig. B, we generate additional pseudo-negatives by copying one of the premises as the conclusion.
- For RuleTaker, as in Fig. C, we generate additional pseudo-negatives by negating the conclusion.

D Proof Graphs are Loopless

We prove that the proof graph in Algorithm 1 is loopless. Intuitively, for a proof graph (C, I, S, h), as we traverse along any path, the node scores in $C \cup I \cup \{h\}$ are non-increasing due to Eqn. 1, which prevents loops.

Lemma D.1. *Let $\mathcal{G}$ be a proof graph with nodes (C, I, S, h). For any $s \in S$ and $v, u \in C \cup I \cup \{h\}$ s.t. edges (v, s) and (s, u) exist, we have $\texttt{scr}_n(u) \leq \texttt{scr}_n(v)$.*

Proof. In this case, s must be a proof step with u as the conclusion and v as one of its premises. According to Definition 4.1 and Eqn. 1, we have

$$\texttt{scr}_n(u) = \min\big(\texttt{scr}_s(s), \texttt{scr}_n(v), \dots\big).$$

Therefore, $\texttt{scr}_n(u) \leq \texttt{scr}_n(v)$. □

Lemma D.2. *Let $\mathcal{G}$ be a proof graph with nodes (C, I, S, h). For any $v, u \in C \cup I \cup \{h\}$ s.t. v is a predecessor of u, we have $\texttt{scr}_n(u) \leq \texttt{scr}_n(v)$.*

Proof. According to Definition 4.1, there exists a path $v \to s_1 \to w_1 \to s_2 \to w_2 \to, \dots, \to s_k \to u$, where $\forall i, s_i \in S, w_i \in C \cup I \cup \{h\}$. The lemma can be proved by performing induction on the path and applying Lemma D.1. □

Theorem D.3. *In Algorithm 1, if the proof graph is loopless after initialization, then it will remain loopless.*

Proof. We just need to prove that it is impossible to introduce a loop during any iteration in Algorithm 1. We prove it by contradiction, assuming we could introduce a loop in an iteration, as in Fig. D. We have two nodes $v, u \in C \cup I \cup \{h\}$, and v is a predecessor of u before introducing the loop. Further assume that the loop is introduced as a result of executing a proof step s, which created the edges (u, s) and (s, v) (the blue arrow in Fig. D; s is omitted). In this hypothetical scenario, the loop would be $v \to \dots \to u \to s \to v$.

Apply Lemma D.2 to the path from v to u, and we have $\texttt{scr}_n(u) \leq \texttt{scr}_n(v)$ before introducing the loop.

Method	Leaves		Steps		Intermediates		Overall
	F1	AllCorrect	F1	AllCorrect	F1	AllCorrect	AllCorrect
Our format	86.9 ± 0.6	45.6 ± 1.5	**42.6 ± 1.6**	**29.7 ± 1.3**	**64.6 ± 1.4**	32.2 ± 2.1	27.1 ± 1.5
EntailmentWriter format	**87.6 ± 0.5**	**47.1 ± 2.1**	42.2 ± 1.1	**29.7 ± 1.6**	64.5 ± 0.6	**32.3 ± 1.7**	**27.5 ± 1.8**

Table A: Test results of single-shot models on EntailmentBank (Dalvi et al., 2021) (Task 2) with different input formats. All methods are based on T5-large (Raffel et al., 2020).

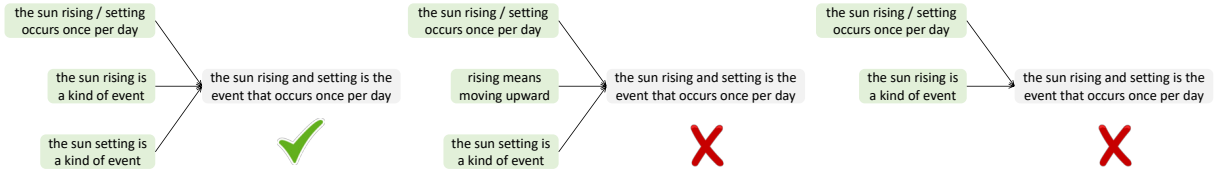

Figure A: Pseudo-negative examples constructed by perturbing positive examples.

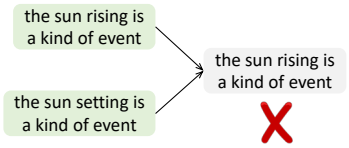

Figure B: EntailmentBank pseudo-negative examples constructed by copying premises. The step is technically valid but not useful.

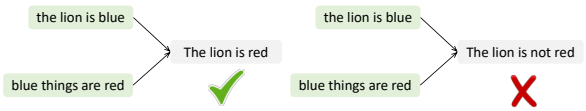

Figure C: RuleTaker pseudo-negative examples constructed by negating the conclusion.

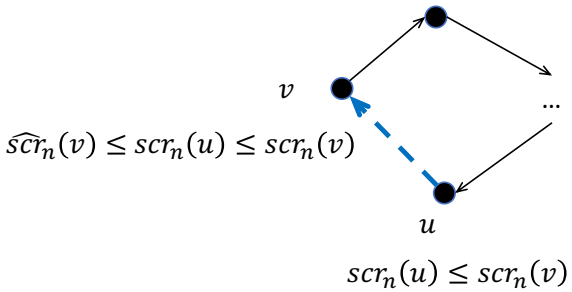

Figure D: A hypothetical loop in the proof graph. Nodes for proof step (S) are omitted for simplicity of illustration. The blue arrow is a hypothetical proof step that introduces the loop, which should actually be a no-op because the tentative score $\widehat{\texttt{scr}}_n(v)$ not greater than the existing score $\texttt{scr}_n(v)$ (Sec. 4.3).

Remember how the step s is executed (Sec. 4.3), the tentative score $\widehat{\texttt{scr}}_n(v) = \min\big(\texttt{scr}_s(s), \texttt{scr}_n(u), \dots\big) \leq \texttt{scr}_n(u) \leq \texttt{scr}_n(v)$. The tentative score is not greater than the original score of v. So the step is a no-op that should not be executed. Therefore, it is impossible to introduce loops. □

E Procedure for Sampling Partial Proofs

The `sample_new` function in Algorithm 1 samples a partial proof tree from the proof graph. First, the graph is a DAG, so we can visit nodes in the order of a topological sort—successors before predecessors. Second, when visiting a node, if it is not already a part of the partial proof, we add it with a probability of 0.5. Third, whenever we add a node, we also add all of its predecessors. This ensures the result is a valid partial proof.

F Training and inference details

We use T5-large (Raffel et al., 2020) for the prover and RoBERTa-large (Liu et al., 2019) for the verifier. All experiments are run on machines with 2 CPUs, 16GB memory, and one NVIDIA A6000 GPU. Models are optimized using AdamW (Loshchilov and Hutter, 2019). The learning rate warms up linearly from 0 to a maximum value and then decays following the cosine schedule. Hyperparameters are tuned on the validation data separately for each task/method. We report test results of models trained on the training set alone, excluding the validation set. We report the average performance and the standard deviation for 5 independent runs.

Our results on EntailmentBank are produced by the official evaluation code.[10] The code had a bug fix in May 2022 that impacted the Intermediate-AllCorrect metric of methods evaluated earlier, including IRGR (Ribeiro et al., 2022) and arXiv versions v1, v2 of EntailmentWriter (Dalvi et al., 2021). We evaluate NLProofS using the evaluation code after the bug fix. And we report the EntailmentBank numbers based on their fixed arXiv version v3 that was released on May 28, 2022. The numbers in the IRGR paper have not been updated yet, so we report its Intermediates-AllCorrect metric based on private correspondence with the authors.

[10] https://github.com/allenai/entailment_bank

Method	Leaves		Steps		Intermediates		Overall
	F1	AllCorrect	F1	AllCorrect	F1	AllCorrect	AllCorrect
EntailmentWriter	86.2	43.9	40.6	28.3	67.1	34.8	27.3
EntailmentWriter (T5-11B)	**89.4**	52.9	46.6	35.3	69.1	36.9	32.1
NLProofS (ours)	**89.4 ± 0.8**	**56.0 ± 0.7**	**50.4 ± 1.9**	**38.4 ± 1.3**	**71.9 ± 1.4**	**41.3 ± 1.4**	**37.1 ± 1.5**
GPT-3 (Brown et al., 2020)	64.2 ± 2.3	15.3 ± 1.9	17.6 ± 0.6	12.3 ± 1.4	53.6 ± 1.4	22.3 ± 1.1	12.3 ± 1.4
Codex (Chen et al., 2021)	68.9 ± 3.7	19.8 ± 3.2	21.4 ± 3.0	14.6 ± 1.7	55.6 ± 2.2	23.2 ± 1.9	14.4 ± 1.4

Table B: Validation results of proof generation on EntailmentBank (Dalvi et al., 2021). Results of GPT-3 and Codex are based on prompting with 7 in-context examples randomly sampled from the training data.

Method	Answer accuracy						Proof accuracy					
	N/A	0	1	2	3	All	N/A	0	1	2	3	All
NLProofS	99.7	100.0	100.0	99.4	97.0	99.4	99.7	100	100	99.4	96.1	99.3

Table C: Validation results on the D0–D3 subset of RuleTaker (OWA) (Tafjord et al., 2021).

G Distinguishing Valid and Invalid Hypotheses

Method	Task 1	Task 2
Learned (Goal) + PPM†	82.0 ± 1.0	86.0 ± 1.0
EntailmentWriter†	53.0 ± 2.0	65.0 ± 2.0
Ours	**82.4 ± 0.8**	**90.9 ± 1.2**

Table D: Area under the ROC curve (AUROC) of distinguishing valid/invalid hypotheses on EntailmentBank (Dalvi et al., 2021) validation set. Methods with † are reported by Bostrom et al. (2022). All methods are based on T5-large (Raffel et al., 2020).

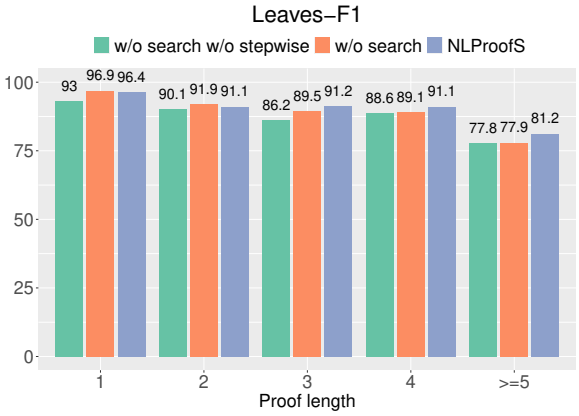

Figure E: The Leaves-F1 metric of test results on task 2 (distractor) broken down by the length of the ground truth proof.

We also evaluate on distinguishing valid/invalid hypotheses introduced by Bostrom et al. (2022). In this task, the model is given a hypothesis h and supporting facts C. But unlike in proof generation, here h can be either valid or invalid w.r.t. C. And the model has to classify h as valid/invalid. We use the dataset Bostrom et al. (2022) constructed from EntailmentBank: Examples with valid hypotheses come directly from EntailmentBank. Examples with invalid hypotheses are

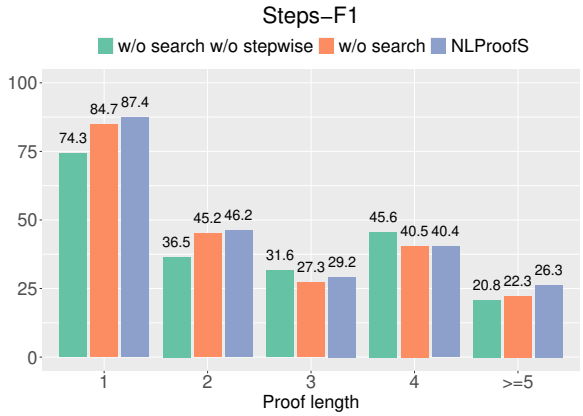

Figure F: The Steps-F1 metric of test results on task 2 (distractor) broken down by the length of the ground truth proof.

constructed by pairing the supporting facts in one example with the hypothesis in another random example.

NLProofS is developed for proof generation, and it has seen only valid hypotheses in training. So we follow Bostrom et al. (2022) to adapt proof generation systems to this new task: (1) Train the system to generate proofs for valid hypotheses. (2) Apply the system to generate proof scores for both valid and valid hypotheses. (3) Train a linear classifier on top of the scores to predict the validity of hypotheses. It requires the system to be able to produce proof scores. For our method, we use $\texttt{scr}_n(h)$ defined in Eqn 1 as the proof score.

Results in Table D show that our method compares favorably with SCSearch, whereas EntailmentWriter falls behind. The results suggest that proof scores generated by us are more well-calibrated: they are high for valid hypotheses and low for invalid ones. This is largely attributed to our verifier, which prevents the model from hallucinating invalid proofs with confidence.

However, results on this task should be interpreted

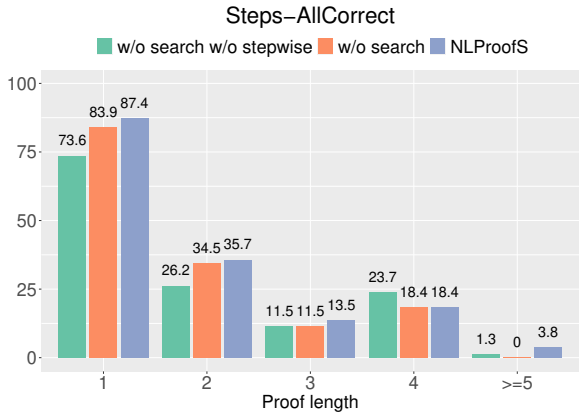

Figure G: The Steps-AllCorrect metric of test results on task 2 (distractor) broken down by the length of the ground truth proof.

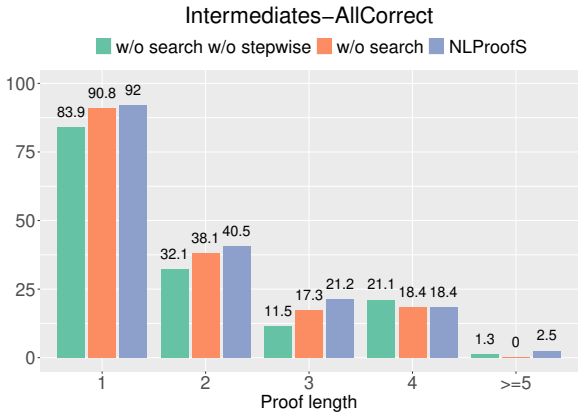

Figure I: The Intermediates-AllCorrect metric of test results on task 2 (distractor) broken down by the length of the ground truth proof.

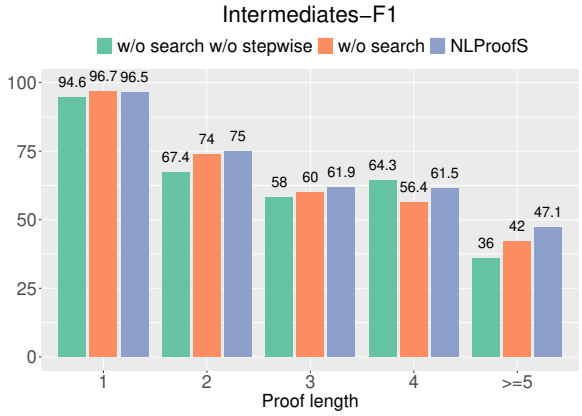

Figure H: The Intermediates-F1 metric of test results on task 2 (distractor) broken down by the length of the ground truth proof.

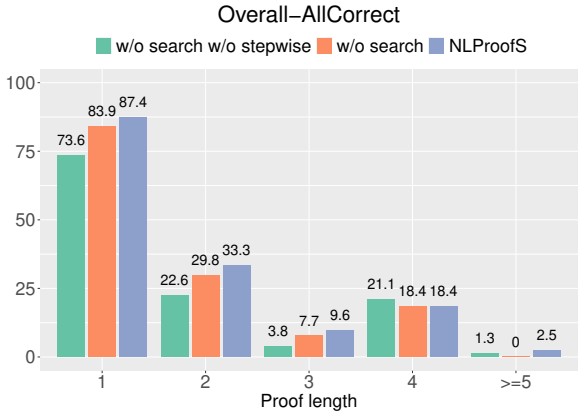

Figure J: The Overall-AllCorrect metric of test results on task 2 (distractor) broken down by the length of the ground truth proof.

with caution. First, they do not reflect the performance on proof generation, and SCSearch has not been evaluated on proof generation. Second, none of the methods are explicitly optimized for this task. They see only valid hypotheses during training but are asked to distinguish valid/invalid hypotheses during inference. Third, the particular dataset constructed by Bostrom et al. (2022) is too easy. An invalid hypothesis has very little lexical overlap with the supporting facts, which can be used as a cue for classifying hypotheses accurately. As a result, a simple RoBERTa baseline directly optimized for classifying the hypothesis can solve this task to almost 100%.

H Additional Experimental Results

Validation results. Table B shows our proof generation results on the validation set of EntailmentBank (Task 2) (Dalvi et al., 2021), corresponding to Table 2. Table C shows the validation results on RuleTaker (OWA) (Tafjord et al., 2021), corresponding to Table 4.

Few-shot prompting with GPT-3 or Codex. We investigate whether proof generation can be solved out of the box by prompting GPT-3 (Brown et al., 2020) or Codex (Chen et al., 2021) with few-shot examples. Fig. K shows an example prompt consisting of 7 in-context examples randomly sampled from the training set of EntailmentBank (Task 2), as well as a validation example for which we want to make predictions.

Table B includes the results on the full validation set. They were obtained on October 20, 2022 using the model `text-davinci-002` for GPT-3 and `code-davinci-002` for Codex. We report the mean and standard deviation from 3 independent runes with different in-context examples in the prompt. GPT-3 and Codex perform substantially worse than other methods, demonstrating that we cannot easily solve proof generation through few-shot prompting. In addition, Codex performs better than GPT-3, which is consistent with the observations in Madaan et al. (2022) though we do not format the output as Python programs.

7 in-context training examples (w/ proofs)

```
Hypothesis: if a fossil of a bird cannot be identified then that kind of bird is probably extinct
Context:
sent1: identifying is similar to determining
sent2: if a fossil is of an organism that cannot be identified then that organism is probably extinct
...
sent25: fossils can be used to study the history of organisms and environments on earth
Proof: sent13 & sent24 -> int1: a bird is a kind of organism; int1 & sent2 -> hypothesis;

Hypothesis: an animal requires water and air and food for survival
Context:
sent1: breathing in is when animals inhale air into their lungs
sent2: animals / living things require water for survival
...
sent25: the amount of something is similar to the availability of something
Proof: sent12 & sent8 -> int1: an animal requires air for survival; int1 & sent2 -> int2: an animal requires water and
air for suvival; sent13 & sent21 -> int3: an animal requires food for survival; int2 & int3 -> hypothesis;

Hypothesis: stars that are blue in color are hottest in temperature
Context:
sent1: a hot substance is a source of heat
sent2: the surface of the sun is extremely hot in temperature with values as high as 20 000 000 c
...
sent25: surface type is a kind of characteristic
Proof: sent19 & sent5 -> hypothesis;

Hypothesis: as mileage per gallon of oil increases, the amount of time that oil is available will be extended
Context:
sent1: performing a task in less time / more quickly / faster has a positive impact on a person 's life
sent2: a measure of time is a length of time
...
sent25: to provide means to supply
Proof: sent14 & sent6 -> int1: gasoline is a kind of resource; int1 & sent12 -> int2: as the use of gasoline decreases,
the length of time that gasoline is available will increase; int2 & sent21 -> int3: as mileage per gallon of gasoline
increases, the length of time that gasoline is available will increase; int3 & sent8 -> int4: as mileage per gallon of
oil increases, the amount of time that oil is available will increase; int4 & sent3 -> hypothesis;

Hypothesis: the firecracker stores chemical energy as its original energy
Context:
sent1: if something emits something else then that something increases the amount of that something else
sent2: phase means state
...
sent25: heat energy is synonymous with thermal energy
Proof: sent11 & sent21 -> hypothesis;

Hypothesis: humans throwing garbage into a stream causes harm to the stream
Context:
sent1: absorbing something harmful has a negative impact on a thing
sent2: objects in an environment are a part of that environment
...
sent25: waste must be removed
Proof: sent12 & sent3 & sent8 -> int1: humans throwing garbage in an environment causes harm to that environment;
sent14 & sent7 -> int2: a stream is a kind of environment; int1 & int2 -> hypothesis;

Hypothesis: the sun will appear larger than other stars because it is the closest star to earth
Context:
sent1: to move away means to increase distance
sent2: size is a property of objects and includes ordered values of microscopic / tiny / small / medium / large
...
sent25: distance is a property of space and includes ordered values of close / far
Proof: sent22 & sent23 -> int1: earth is a kind of celestial object; int1 & sent19 & sent3 -> int2: as the distance
from a star to earth decreases, the star will appear larger; int2 & sent15 -> hypothesis;
```

1 validation example (w/o proof)

```
Hypothesis: the sun rising and setting is the event that occurs once per day
Context:
sent1: increasing is a kind of order
sent2: the sunlight occurs during the day
...
sent25: all the time means at day and at night
Proof:
```

Figure K: A prompt for GPT-3 and Codex. Each example has 25 supporting facts. We only show 3 for simplicity.

Test results by different proof length. Fig. E, F, G, H, I, and J are EntailmentBank (Task 2) test results broken down by proof length (also Fig. 3).

Improving the retriever. For Task 3 of EntailmentBank, all methods in Table 2 use the same retrieved supporting facts in Dalvi et al. (2021) and focus solely on proof generation. An orthogonal direction is improving the retriever. IRGR (Ribeiro et al., 2022) designs a multi-step retriever, which obtains significant improvements on Task 3 (11.8% on the Overall-AllCorrect metric) but worse results on Task 1 and Task 2 compared to the EntailmentWriter baseline. We do not compare with IRGR, since improving the retriever is orthogonal to our contributions.