Explainable Multi-hop Verbal Reasoning Through Internal Monologue

Zhengzhong Liang[1], **Steven Bethard**[2] **and Mihai Surdeanu**[1]
[1] Computer Science Department, The University of Arizona, Tucson, AZ
[2]School of Information, The University of Arizona, Tucson, AZ
`{zhengzhongliang, bethard, msurdeanu}@email.arizona.edu`

Abstract

Many state-of-the-art (SOTA) language models have achieved high accuracy on several multi-hop reasoning problems. However, these approaches tend to not be interpretable because they do not make the intermediate reasoning steps explicit. Moreover, models trained on simpler tasks tend to fail when directly tested on more complex problems. We propose the Explainable multi-hop Verbal Reasoner (EVR) to solve these limitations by (a) decomposing multi-hop reasoning problems into several simple ones, and (b) using natural language to guide the intermediate reasoning hops. We implement EVR by extending the classic reasoning paradigm General Problem Solver (GPS) with a SOTA generative language model to generate subgoals and perform inference in natural language at each reasoning step. Evaluation of EVR on Clark et al. (2020)'s synthetic question answering (QA) dataset shows that EVR achieves SOTA performance while being able to generate all reasoning steps in natural language. Furthermore, EVR generalizes better than other strong methods when trained on simpler tasks or less training data (up to 35.7% and 7.7% absolute improvement respectively).[1]

1 Introduction

Large pretrained language models such as BERT (Devlin et al., 2019) and RoBERTa (Liu et al., 2019) have been successfully used in multi-hop reasoning problems (Banerjee et al., 2020; Asai et al., 2019; Yadav et al., 2019). Usually, these pretrained language models solve multi-hop reasoning problems in a discriminative end-to-end manner: these models take the question and *all* the relevant evidence as the input, and produce the final answer to the question. This raises two problems. First, this direction lacks interpretability, i.e., it is hard to know which individual reasoning steps are taken in each iteration and why. Second, the trained models usually suffer from the compositionality generalization problem, meaning that they tend to fail when the number of reasoning steps are much larger in the evaluation set than in the training set (Hupkes et al., 2020; Hahn, 2020; Clark et al., 2020).

(**Input Facts**:) Alan is blue. Alan is rough. Alan is young. Bob is big. Bob is round. Charlie is big. Charlie is blue. Charlie is green. Dave is green. Dave is rough.
(**Input Rules**:) Big people are rough. If someone is young and round then they are kind. If someone is round and big then they are blue. All rough people are green.
Q1: Bob is green. True/false? [**Answer**: T]
Q2: Bob is kind. True/false? [**Answer**: F]

Figure 1: An example taken from (Clark et al., 2020). The context includes two types of statements: facts and rules. Multiple facts and rules are usually needed to answer the questions. For example, to prove "Bob is green," the model needs to construct the reasoning chain "All rough people are green ← Big people are rough ← Bob is big".

Newell (1994) categorized cognitive processes based on their time scales: unconscious activities take around 50 ms, whereas conscious actions can vary from 100 ms to hours. Importantly, Newell (1994) argued that conscious actions are sequences of simple conscious/unconscious actions. Extrapolating from cognitive science to natural language processing (NLP), in this paper we ask the question: can we design an interpretable multi-hop reasoning system that sequentially applies neural networks trained on simpler tasks? Further, motivated by the finding from cognitive science that people might use internal monologues to guide their reasoning, we want to explore whether it is possible to use natural language to guide this sequential process.

In this paper, we propose a solution for these important questions. We provide a neural implementation for a classic planning/reasoning paradigm that is designed to mimic the human reasoning process:

---

[1] The code is available at: `https://github.com/clulab/releases/tree/master/naacl2021-evr`

Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 1225–1250
June 6–11, 2021. ©2021 Association for Computational Linguistics

the General Problem Solver (GPS) (Newell et al., 1959). We augment GPS with a SOTA sequence-to-sequence (Seq2Seq) model and apply this model recursively to achieve high interpretability and better generalization of compositionality for a synthetic QA task (Clark et al., 2020).

The contributions of our paper are the following:

(1) We extend (Clark et al., 2020)'s dataset with natural language intermediate goals/statements necessary to answer each question.

(2) We propose a neural GPS to address this QA task while generating all intermediate goals/statements in natural language.

(3) Evaluation on the above task shows that our proposed method achieves SOTA performance. Importantly, our method generalizes better when trained only on simpler tasks (26.5% to 35.7% absolute improvement), and less training data (7.7% absolute improvement) compared with other strong reasoning methods.

2 Task Description and Baseline Models

We build our approach from the multi-hop reasoning problem proposed in (Clark et al., 2020), which we summarize first. Figure 1 shows an example from this dataset. Each reasoning problem consists of the context C, the question Q and the answer $A = \{True, False\}$. C includes facts F and rules R. To answer a question, multiple statements in C need to be combined.

2.1 Proof Strategy

The proofs of the questions are provided by the creators of the dataset, and each question is proved by one of three available strategies: "proof", "inv-proof" and "fail-to-prove". "Proof" directly proves a statement is true using the facts and the rules; "inv-proof" proves a statement is false using the facts and rules; and, lastly, "fail-to-prove" means the statement could not be explicitly proved to be true or false given the rules and facts. In the latter case, a positive statement is considered to be **False**, and a negative statement is considered to be **True**.

For example, assuming we are given the facts and rules in Figure 1, the reasoning chain provided to prove "Bob is green" is using the "proof" strategy. Conversely, "Alan is not green" is false by "inv-proof", because "Alan is green" can be proved by "All rough people are green $\leftarrow$ Alan is rough". Finally, "Alan is nice" is false and "Alan is not nice" is true due to "fail-to-prove".

2.2 Dataset Details

The dataset is synthesized using hand-crafted rules and formal language, then translated to natural language. Some language variation is inserted (e.g., in Figure 1 the rules are expressed differently). Depending on the number of rules and facts needed, there are 5 partitions in the dataset: **DU0**, **DU1**, **DU2**, **DU3** and **DU5**, where "DU" stands for "Depth Upto". "DU0" means the reasoning depth of the questions is 0, i.e., the questions can be answered by just looking at the facts without applying any rules. DU5 means the questions may require applying the rules for upto 5 times (but DU5 also has questions that require applying the rules for 0 to 4 times).

Additionally, a "birds-electricity" dataset is also provided to test the model's generalization ability. The F, R, Q are generated by similar templates of DU0 to DU5, but with different entities/predicates/attributes that do not appear in the DU0 to DU5 partitions.

Summing up all partitions, the dataset has approximately 500K questions, and the train/dev/test ratio is 70/10/20. More details can be found in (Clark et al., 2020).

2.3 Baseline Models

We will compare our approach against two strong baselines. The first baseline is a RoBERTa classifier from the original paper of Clark et al. (2020). In this approach, the questions are solved in a text classification manner. That is, for each question, the model takes C and Q as the input, and calculates the probability of A being true or false. We abbreviate this baseline as "RT" in our paper.

The second baseline is PROVER (Saha et al., 2020), which handles the reasoning problem as a graph problem. This approach takes the input C and Q to produce both the final answer $\{True, False\}$, and a graph that indicates the reasoning path. We abbreviate this baseline is as "PR".

2.4 Shortcomings and Opportunities

We see two shortcomings for these research directions, which motivate our proposed idea.

Shortcoming #1: Limited interpretability RT models this reasoning problems as a text classification task over a "bag of evidences". That is, RT takes *all* context and produces an answer in a *single* forward inference process. Although its predictions can often achieve high accuracy, there are several

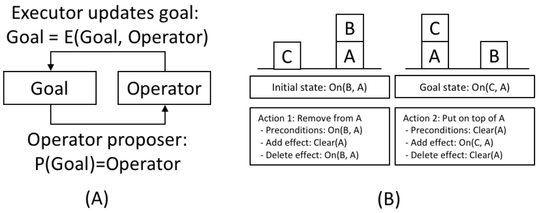


Figure 2: (A) GPS's working cycle. (B) A toy example showing the goal and operators (action 1 and 2). This problem can be solved in two cycles: in the first cycle, the proposer reads the goal state $On(C, A)$ and search for available actions. In this case action 2 is proposed because its effect matches the goal state. Next, the goal is updated as $Clear(A)$ (i.e., preconditions of action 2). In the second cycle, the proposer reads the new goal $Clear(A)$ and action 1 is proposed. By applying action 1 the goal is further updated as $On(B, A)$. Since $On(B, A)$ is the initial state, the proposer then terminates the reasoning process.

problems with this and similar directions, which limit their interpretability. First, it is unclear which part of the context was used by the answering engine. Second, although some methods such as PR improve over RT by producing an explanation consisting of multiple supporting facts at the end, it is still hard to explain the underlying reasoning process in human understandable terms. Finally, it has been shown theoretically and empirically that neural networks suffer from the compositionality generalization problem (Hupkes et al., 2020; Hahn, 2020). That is, neural networks have limited ability to learn recursive patterns, and they fail to generalize to recursive patterns that are much deeper than the ones seen in training.

Shortcoming #2: Differences from the human cognitive process People do not usually solve very complex problems at once. Instead, people constantly generate subgoals and solve complex problems incrementally (Newell, 1966). Second, verbal strategies are sometimes used to guide one's reasoning (Bacon et al., 2003). This is different from the approaches taken by both RT and PR.

A desired explainable multi-hop verbal reasoner: Motivated by these shortcomings, we propose several desired characteristics for an ideal problem solver. First, the method should be able to decompose complex problems into simple ones that are easy to answer. This should not only increase the interpretability of the reasoning process, but also help reduce the compositionality generalization problem, because the unseen distributions (the complex problem) can be reduced to a series of seen distributions (simple problems). Second, each reasoning step should be guided by natural language, so that each step is easily explainable to the human end user.

3 Neural General Problem Solver for Multi-hop Verbal Reasoning

In this section, we first review a classic planning/reasoning paradigm that is designed to mimic the human reasoning process, the General Problem Solver (GPS) (Newell et al., 1959), then propose our neural implementation of it.

3.1 The General Problem Solver

GPS works in cycles (Figure 2 (A)): in each cycle, the **operator proposer** P reads the current goal G to propose an operator $O = P(G)$. Then the proposed operator is used by the **executor** E to update the goal $G = E(G, O)$. The cycle stops when the goal is satisfied, or no new operators are proposed. Figure 2 (B) shows a toy example from the block world, where the agent starts from the goal state and searches for a sequence of actions to reach the initial state.

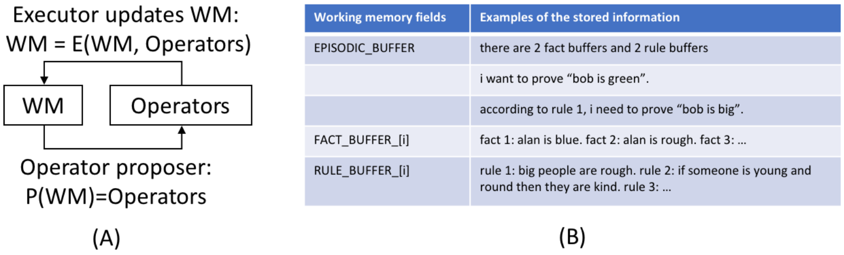
3.2 The Neural General Problem Solver

Although GPS has been widely used to mimic the human reasoning process, it has shortcomings. First, the representations of the goals are usually in a formal language, which has limited expressiveness and readability compared to natural language. Second, the proposer uses human-crafted rules to match and propose operators, which may not be flexible enough to handle situations that diverge from the training examples.

Due to these drawbacks, we propose to add neural components to GPS (a working cycle is shown in Figure 3). More specifically, the neural GPS has the following characteristics:

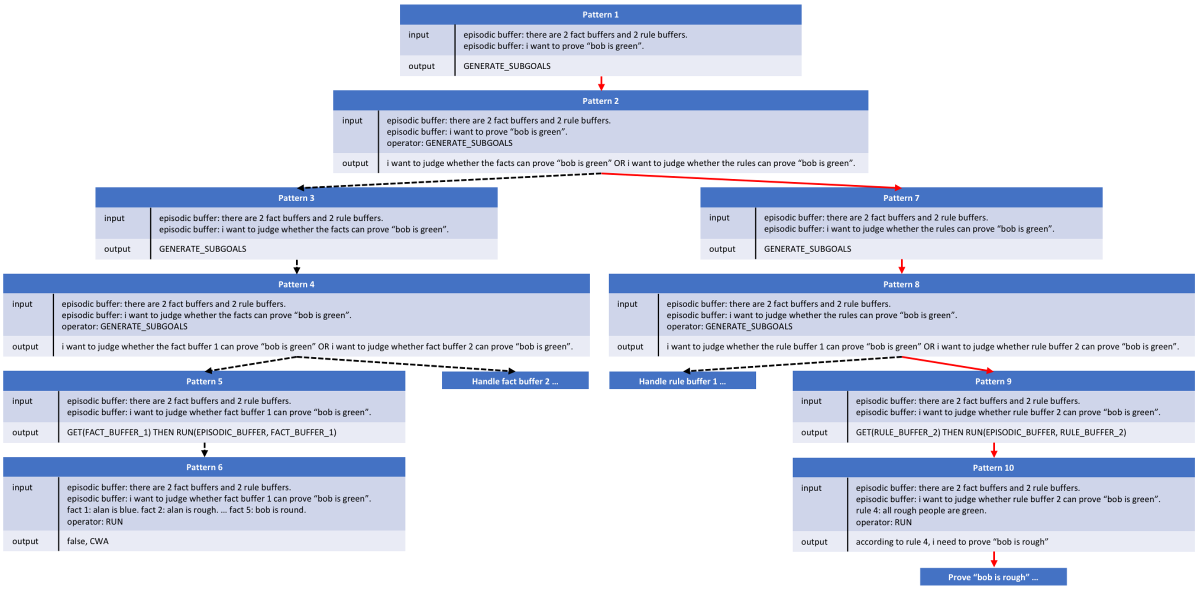
Goal (Extended to Working Memory): First, the goal is represented in natural language instead of formal language to enable better readability and expressiveness. Second, the goal is extended to a working memory buffer, which contains not only the goal but also other information that might be useful to the reasoning process.

Operator Proposer and Executor: The operator proposer is no longer using explicit rules. Instead, we use a Seq2Seq neural network to directly map



Working memory fields	Examples of the stored information
EPISODIC_BUFFER	there are 2 fact buffers and 2 rule buffers
	i want to prove "bob is green".
	according to rule 1, i need to prove "bob is big".
FACT_BUFFER_[i]	fact 1: alan is blue. fact 2: alan is rough. fact 3: ...
RULE_BUFFER_[i]	rule 1: big people are rough. rule 2: if someone is young and round then they are kind. rule 3: ...

Figure 3: (A) A working cycle for the proposed neural GPS. The goal in GPS is extended to a working memory buffer (WM), and multiple operators might be generated in a cycle. (B) The composition of the working memory and possible values stored in each field.



Pattern 1	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to prove "bob is green".
output	GENERATE_SUBGOALS

Pattern 2	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to prove "bob is green". operator: GENERATE_SUBGOALS
output	i want to judge whether the facts can prove "bob is green" OR i want to judge whether the rules can prove "bob is green".

Pattern 3	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to judge whether the facts can prove "bob is green".
output	GENERATE_SUBGOALS

Pattern 7	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to judge whether the rules can prove "bob is green".
output	GENERATE_SUBGOALS

Pattern 4	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to judge whether the facts can prove "bob is green". operator: GENERATE_SUBGOALS
output	i want to judge whether the fact buffer 1 can prove "bob is green" OR i want to judge whether fact buffer 2 can prove "bob is green".

Pattern 8	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to judge whether the rules can prove "bob is green". operator: GENERATE_SUBGOALS
output	i want to judge whether the rule buffer 1 can prove "bob is green" OR i want to judge whether rule buffer 2 can prove "bob is green".

Pattern 5	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to judge whether fact buffer 1 can prove "bob is green".
output	GET(FACT_BUFFER_1) THEN RUN(EPISODIC_BUFFER, FACT_BUFFER_1)

Handle fact buffer 2 ...

Handle rule buffer 1 ...

Pattern 9	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to judge whether rule buffer 2 can prove "bob is green".
output	GET(RULE_BUFFER_2) THEN RUN(EPISODIC_BUFFER, RULE_BUFFER_2)

Pattern 6	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to judge whether fact buffer 1 can prove "bob is green". fact 1: alan is blue. fact 2: alan is rough. ... fact 5: bob is round. operator: RUN
output	false, CWA

Pattern 10	
input	episodic buffer: there are 2 fact buffers and 2 rule buffers. episodic buffer: i want to judge whether rule buffer 2 can prove "bob is green". rule 4: all rough people are green. operator: RUN
output	according to rule 4, i need to prove "bob is rough"

Prove "bob is rough" ...

Figure 4: A walk through example of EVR solving the problem mentioned in Figure 1. The format in the example is the same as the actual output of our method. The red solid lines indicate the path that proves the goal statement. An example of the actual output of our model is shown in Appendix A.5.

the text in the working memory to a sequence of operators. The operators are later used by the executor, which also has neural components, to update the working memory buffer.

3.3 Adapting the Neural GPS to Multi-hop Verbal Reasoning

In this section we explain in detail the working flow of the neural GPS (referred as Explainable Verbal Reasoner or EVR later) and the design of each of its components for multi-hop verbal reasoning.

A Walk-Through Example: Figure 4 shows an example of how our method solves the problem in Figure 1. Every two consecutive blocks form a working cycle (e.g., patterns 1&2 or 3&4). In each cycle, the odd pattern is the operator proposing stage, and the even pattern is the executing stage.

For example, pattern 1 shows an operator proposing stage in a cycle. There are two episodic buffers available for the operator proposer, with the first buffer storing some general knowledge about the problem, and the second describing the goal. The operator proposer, a Seq2Seq neural model, first concatenates the two episodic buffers, then proposes "GENERATE_SUBGOALS" as the operator. At the executing stage, the executor (another Seq2Seq neural model), takes the two episodic buffers in the working memory and the "GENERATE_SUBGOALS" operator to produce the subgoals: judge whether the facts can prove the goal or the rules can prove the goal. Finally, the newly generated subgoals replace the old goal in the episodic buffer (i.e., the goals in pattern 3 and 7 are different from pattern 1, because the goal in pattern 1 is replaced) and one working cycle is finished. At pattern 10, the EVR discovers that the new goal is to prove "bob is rough", so another recursive search process starts (largely repeating the process of pattern 1 to 10).

Working Memory: Working memory is a global memory space with several storage fields, where each field is indexed by a textual key (Figure 3 (B)). In this verbal reasoning problem, three types of information can be stored in the working memory: episodic buffer (indexed by the key "EPISODIC_BUFFER"), fact buffer (indexed by FACT_BUFFER_[i], since there are probably more than one fact buffers), and rule buffer (indexed by RULE_BUFFER_[i]).

The episodic buffer stores three types of information: (a) general statements about the reasoning task that are useful throughout the reasoning process (e.g., row 1 in Figure 3 (B)); (b) goal (row 2, Figure 3 (B)). Similar to GPS, a description of goal should be included in the working memory. We use natural language to describe the goals, and goals are updated periodically; (c) inferred knowledge during the reasoning process (row 3, Figure 3 (B)).

Note that the working memory is not equivalent to the input text in the patterns in Figure 4. The input text in Figure 4 is obtained from the working memory, but some information in the working memory might not be needed by many patterns. For example, fact buffers and rule buffers are only the input for pattern 6 and 10, whereas other patterns do not use them.

Ideally, at each cycle, both the proposer and the executor need to determine what information to use from the working memory, and the executor also needs to determine what information to modify in the working memory. This is a hard problem: assume there are n pieces of information in the working memory, there will be 2^n ways to read from/modify the memory. Therefore we make the following simplifications for this verbal reasoning problem. First, the size of episodic buffer is fixed to 2, and the first episodic buffer slot (i.e., "there are X fact buffers and Y rule buffers") can not be modified; the second episodic buffer slot is constantly modified as the new subgoals are generated. Second, the fact buffers and the rule buffers could not be modified. Third, the proposer and the executor only use the two episodic buffers as the input by default. The fact buffer and rule buffer can be used as input, but only when explicit commands are generated (e.g., pattern 5 and 6 in Figure 4).

Operator Proposer: We use a SOTA Seq2Seq language model, Google T5 (Raffel et al., 2020), as the operator proposer. The proposer concatenates the two episodic buffer slots as a single piece of text, and uses this text as the input to produce a sequence of operators.

Executor: The executor has three functions: (1) parse the operators, (2) call the correct neural module given the operator, to get the answer/get the subgoal, and (3) update the working memory. Since the operators are fairly simple, we just use several if-else conditions to determine what actions need to be taken. Some examples are given in Table 1. As discussed above, the changing of working memory is restricted to changing the second episodic buffer slot. The major part of the executor is the neural module, which is responsible for generating subgoals (pattern 2 ,4 ,8 in Figure 4), answering questions (pattern 6 in Figure 4) and deriving statements (pattern 10 in Figure 4). Again we use Google T5 as this neural module to read from the working memory and produce textual output. More details can be found in Appendix A.3.

Operators: We design a simple domain specific language (DSL) as the operators. The meaning of some DSL commands are shown in Table 1.

3.4 Training Data Generation

Finally, to train the working flow we proposed in Section 3.3, 12 patterns of training data need to be generated (only 10 are shown in Figure 4). The generation strategies for some critical patterns are shown in Table 2. In summary, we write rules to generate the input text and the output text for each pattern. Around 1M training samples are generated for the 12 patterns in total.[2]

We implemented three variants of EVR to learn these 12 tasks:

EVR1: This is the EVR baseline. For this baseline we use three distinct T5 models: one to learn pattern 6 data, one to learn pattern 10 data, and one to learn the rest of the patterns. The fact buffer size is set to 5 and the rule buffer size is set to 3 (5 facts per fact buffer and 3 rules per rule buffer).

EVR2: The fact buffer size and rule buffer size are the same as the EVR1. However, we use a *single* T5 model to learn all patterns of data. This is to test whether multi-task learning helps or harms the performance of EVR.

EVR3: We use three T5 models like EVR1, but the fact buffer size is set to 20 and the rule buffer

---

[2] The data generation code can be found at: `https://github.com/clulab/releases/tree/master/naacl2021-evr`

Operator	Example	Description
AND/OR	Pattern 2, 4, 8 in Figure 4	Conjunction/disjunction operator to connect two branches.
GENERATE_SUBGOALS	Pattern 2, 4, 8 in Figure 4	When the neural model takes this operator it generates subgoals
NAF/CWA	Pattern 6 in Figure 4	NAF is the abbrevation for "Negation as Failure". It is generated when a negative statement (e.g., "bob is not nice") is not contradicted by any facts. "CWA" is the abbrevation for "Close World Assumption". It is output when a positive statement (e.g., "Bob is nice") is not confirmed by any facts.
GET(FACT_BUFFER_[i]) THEN RUN(EPISODIC_BUFFER, FACT_BUFFER_[i])	Pattern 5, 6 in Figure 4	"GET(FACT_BUFFER_[i])" is to get the text of the facts in FACT_BUFFER_[i] (e.g., in pattern 6 Figure 4, FACT_BUFFER_1 contains the text from fact 1 to fact 5). "THEN" connects two commands that need to be executed sequentially. Finally, the neural module in the executor takes the episodic buffer and fact buffer as the input to produce the output.

Table 1: The major defined operators and their meanings.

Ptn	Generation Specifications
1	The input always has two episodic buffers: "there are [X] fact buffers and [Y] rule buffers"; "i want to prove [statement]". The output is always "GENERATE_SUBGOALS".
2	The input consists of two parts: the episodic buffer copied from pattern 1's input, and the added operator "operator: GENERATE_SUBGOALS". The output is "i want to judge whether the facts can prove [statement] OR i want to judge whether the rules can prove [statement]" if [statement] is a positive statement. For the negative statement, the output is "i want to judge whether the facts do not contradict [statement] AND i want to judge whether the rules do not contradict [statement]".
3	The input has two episodic buffers: "there are [X] fact buffers and [Y] rule buffers", and "i want to judge whether the facts can prove/do not contradict [statement]", depending on the generated goals in pattern 2. The output is always "GENERATE_SUBGOALS".
4	The input consists of two parts: the episodic buffer copied from pattern 3's input, and the added operator "operator: GENERATE_SUBGOALS". The output is "i want to judge whether fact buffer 1 can prove [statement] OR i want to judge whether fact buffer 2 can prove [statement] OR ..." if [statement] is a positive statement. For negative statement, the output is "i want to judge whether fact buffer 1 does not contradict [statement] AND i want to judge whether fact buffer 2 does not contradict [statement] AND ...".
5	The input has two episodic buffers: "there are [X] fact buffers and [Y] rule buffers", and "i want to judge whether fact buffer [i] can prove/does not contradict [statement]", depending on the generated goals in pattern 4. The output is always "GET(FACT_BUFFER_[i]) THEN RUN(EPISODIC_BUFFER, FACT_BUFFER_[i])".
6	The input has three parts. (1) the episodic buffer copied from the input of pattern 5; (2) the facts from the fact buffer indicated by the output of pattern 5; (3) the operator "operator: RUN". There are four possible outputs: when [statement] is a positive statement, the output is "true, this is confirmed by fact [i]" if there is a fact in the fact buffer to prove it, and is "false, CWA" if the [statement] is not proved by any facts in the buffer. When [statement] is a negative statement, the output is "true, NAF" if no facts in the fact buffer contradict it, and is "false, this is contradicted by fact [i]" if a fact in the fact buffer contradicts it.
10	The input has two parts: the episodic buffer and the rule buffer determined by pattern 9, where episodic buffer is "there are [X] fact buffers and [Y] rule buffers" plus "i want to judge whether rule buffer [j] can prove/does not contradict [statement]", and the rule buffer is all the rules in RULE_BUFFER_[j] as determined by pattern 9 (please check Figure 4 as an example). The output is the statements derived from the matched rules. For example, "all rough people are green" can be used to prove "bob is green", and in order to prove "bob is green" using this rule, one needs to prove "bob is rough". In this case the output is "according to rule [i], i need to prove bob is rough". There are other edge cases that need to be handled such as negative query. For the handling of other edge cases, please check Appendix A.4.

Table 2: The strategy to generate the training data for some of the patterns. Each pattern has an input and an output to be generated. [statement] is the query to prove, e.g., "bob is green". Patterns 7, 8, 9, 11 and 12 are not included because they are generated in the similar manner as the patterns shown here. Please see Appendix A.2 for the generation strategies of all patterns.

size to 10. Therefore for all problems there will be only one fact buffer and one rule buffer. Since there are more facts and rules in the buffer, the input text to the T5 will be longer. And since there are more rules in the rule buffer, the number of matched rules could be more, so the target text could be potentially longer. We conjecture the longer input and output will make the model harder to train.

4 Experiments

4.1 Training and Evaluation

We use T5 small for all experiments, with the learning rate set to 1e-4. In each epoch, the models are trained on 24,000 training examples, and evaluated on the first 2,000 dev samples. Edit distance between the generated text and target text is used to

D	**0**	**1**	**2**	**3**	**4**	**5**	**all**
Cnt	6299	4434	2915	2396	2134	2003	20192
DU1							
RT	**100.0**	99.0	36.8	23.1	11.4	12.3	63.5
PR	—	—	—	—	—	—	73.7
EVR1	99.7	98.8	97.8	95.5	91.7	90.7	97.0
EVR2	99.5	98.3	96.8	93.1	89.3	88.1	95.9
EVR3	99.8	**99.5**	**99.0**	**98.9**	**98.2**	**97.9**	**99.2**
DU5							
RT	**100.0**	98.4	98.4	98.8	**99.2**	**99.8**	99.2
PR	**100.0**	**99.0**	**98.8**	**99.1**	98.8	99.3	**99.3**
EVR1	99.5	98.2	96.5	92.8	88.3	86.2	95.5

Table 3: Answer accuracy of EVR variants trained on DU1 (top) and DU5 (bottom), and evaluated on all data depths (DU5).

evaluate the models performance on the dev set. The training is stopped when the edit distance on the dev set starts to increase.

In addition to the accuracy of the model's prediction of the final answer (T/F), we also report the quality of the generated proofs. We extracted the critical facts/rules from EVR's reasoning process and reconstructed the reasoning chain in the same format as the provided proofs in the dataset. The proof is considered correct as long as the generated reasoning chain matches one of the provided reasoning chains. We use depth-$\{0, 1, 2, 3, 4, 5\}$ data (i.e., all depths) from DU5 to test all methods.

4.2 EVR Compared with Other Baselines

Table 3 lists the performance of the three EVR variants for QA. We compare EVR against RT and PR trained on the same DU1 data (top part of the table), and all the data (DU5) (bottom part). EVR outperforms the other two baseline methods trained on DU1 on nearly all splits. The best performing EVR is EVR3, which successfully maintains a 97.9 accuracy on depth-5 testing data, and a 99.2 accuracy on all testing splits. EVR3 trained on DU1 approaches the performance of the other methods trained on DU5. This indicates that when the training data are abundant, longer input or output does not harm the performance of our method.

Table 4 shows the quality of the generated proofs. Only the samples that can be proved (either by "proof" or "inv-proof") are compared with our method's proofs. The number of samples subject to this comparison is indicated by "Cnt". The results in the table demonstrate that EVR obtains high-quality proofs most of the time, regardless of the proof depth.

D	**0**	**1**	**2**	**3**	**4**	**5**	**all**
Cnt	1934	1934	1934	1934	1934	1934	11604
DU1							
EVR1	**96.4**	91.9	86.3	75.7	73.1	64.5	81.3
EVR2	96.1	91.2	86.7	77.9	76.8	67.4	82.7
EVR3	**96.4**	**93.0**	**89.8**	**83.2**	**84.1**	**75.1**	**86.9**
DU5							
EVR1	96.0	90.6	86.6	76.8	74.6	66.0	81.8

Table 4: Justification quality of our approach. "Cnt" indicates the number of questions eligible for proof comparison.

Subset	**B1**	**B2**	**E1**	**E2**	**E3**	**E4**	**all**
Cnt	40	40	162	180	624	4224	5270
DU1							
RT	100.0	100.0	88.9	80.0	93.9	97.5	96.2
EVR1	72.5	72.5	100.0	100.0	98.1	98.4	**98.1**
EVR2	77.5	72.5	100.0	100.0	98.1	93.6	94.3
EVR3	87.5	77.5	100.0	100.0	99.0	95.7	96.2
DU5							
RT	97.5	100.0	96.9	98.3	91.8	76.7	80.2
PR	95.0	95.0	100.0	100.0	89.7	84.8	86.5
EVR1	70.0	75.0	64.8	60.0	63.1	61.1	61.6
EVR1^{c1}	75.0	72.5	100.0	75.0	98.1	93.8	**93.6**

Table 5: Evaluation of EVR answers, under the zero-shot learning scenario on the birds-electricity dataset. There are 6 subsets in the dataset, denoted from **B1** to **E4**.

Subset	**B1**	**B2**	**E1**	**E2**	**E3**	**E4**	**all**
Cnt	28	28	72	90	312	1206	1736
DU1							
EVR1	28.6	32.1	100.0	100.0	73.1	90.5	86.3
EVR2	35.7	46.4	100.0	100.0	73.1	87.6	84.6
EVR3	50.0	67.9	100.0	100.0	75.0	91.0	**88.0**
DU5							
EVR1	28.6	28.6	58.3	40.0	45.2	68.4	61.1
EVR1^{c1}	28.6	35.7	100.0	50.0	70.8	81.6	**77.2**

Table 6: Evaluation of EVR-generated proofs, under the zero-shot learning scenario on the birds-electricity dataset.

4.3 Zero-Shot Results on the Birds-Electricity Dataset

Table 5 shows the performance of EVR when tested under a zero-shot learning scenario on the birds-electricity dataset. The results show that EVR yields good generalization ability and outperforms the other two baseline methods in general. Notably, EVR1 trained on DU1 considerably outperforms the baseline methods trained on DU5. A surprising result is that RT trained on DU1 yields a better results than that trained on DU5. The RT creators explained that it is because some extremely rare cases in the training data are not well learned by their DU5 model.

Surprisingly, EVR1 trained on DU5 yields a low performance in the evaluation (e.g., 61.6 on all datasets). We inspected several outputs of the reasoning steps generated by EVR1, and observed

D	**0**	**1**	**2**	**3**	**4**	**5**	**all**
Cnt	6299	4434	2915	2396	2134	2003	20192
10k							
PR	—	—	—	—	—	—	87.1
EVR1	99.9	99.1	96.8	90.4	84.3	82.9	**94.8**
30k							
PR	—	—	—	—	—	—	**97.8**
EVR1	99.6	98.5	96.8	93.4	89.6	88.5	96.0
70k							
PR	100.0	99.0	98.8	99.1	98.8	99.3	**99.3**
EVR1	99.7	98.8	97.8	95.5	91.7	90.7	97.0

Table 7: Evaluation of the predicted answers of EVR1, when trained on less data. 70k is the number of original training examples. PR is trained on DU5 whereas EVR1 is trained on DU1.

that the major reason for this failure is because pattern 4 is not successfully learned due to the bias in DU5 data. In DU5 training data, there are at least 7 facts for each question (i.e., at least 2 fact buffers when the fact buffer size is set to 5). In this case, the target output for pattern 4 would be "I want to judge whether fact buffer 1 . . . OR I want to judge whether fact buffer 2 . . . ". In contrast, in the birds-electricity dataset, some questions have only 3 supporting facts, so there is only 1 fact buffer. In this case, T5 should generate "I want to judge whether fact buffer 1 can prove [query]." without further disjunctions. However, since such examples never appear in the DU5 training data, the T5 still generates instructions to loop over multiple fact buffers, which causes the reasoning program to fail (due to the attempt to access non-existent fact buffers).

To verify the above observation, we trained pattern 6 and 10 using DU5, and all other patterns on DU1. We report this model's performance (indicated by EVR^{c1}) in Table 5 and 6. These results indicate that our intuition was correct, as EVR^{c1} yields good generalization performance on the birds-electricity dataset.

Table 6 shows EVR generates high-quality proofs overall, but the proofs on B1 and B2 are poor. The reason is that the B1 and B2 datasets contain some examples that have proofs unsupported by our model (e.g., to directly prove a positive statement is **False** by contradiction).

4.4 Results Using Less Training Data

Tables 7 and 8 show that EVR1 yields stable performance and proofs when trained with considerably less data (10k and 30k examples). In the lowest data configuration (10k), EVR outperforms PR considerably, even when PR is trained on DU5. This experiment supports our claim that EVR suffers less from the compositionality generalization problem by recursively decomposing complex problems and reasoning over simple ones.

D	**0**	**1**	**2**	**3**	**4**	**5**	**all**
Cnt	1934	1934	1934	1934	1934	1934	11604
10k	96.4	92.1	86.0	73.6	70.4	62.5	80.2
30k	96.4	91.5	84.9	75.0	72.6	64.8	80.9
70k	96.4	91.9	86.3	75.7	73.1	64.5	81.3

Table 8: Evaluation of the generated proofs of EVR1, when trained on less data.

5 Related Work

Neural Symbolic Methods: One branch of neural-symbolic reasoning methods is to design different components in the network, but keeping the whole network differentiable. Typical works include the Differential Neural Computer (DNC) (Graves et al., 2016), End-to-end Memory Networks (Sukhbaatar et al., 2015), Dynamic Memory Networks (DMN) (Kumar et al., 2016) and Compositional Attention Networks (MAC) (Hudson and Manning, 2018). Another direction are the neural modular networks, where what components to use are determined dynamically for each question (Gupta et al., 2019; Jiang and Bansal, 2019). However, it is hard to prove the components are actually fulfilling the designed functionality after training due to the distributed nature of the intermediate representations. In contrast, we explicitly evaluate the performance of each component of EVR after training, achieving better faithfulness (Subramanian et al., 2020).

Formal Theorem Prover: Neural components have been used to augment formal theorem proving in several ways. Polu and Sutskever (2020) apply a Seq2Seq neural network for mathematical theorem proving by training the neural network to generate the proof at each step. Some works seek to use distributed representations to augment the rule-based backward chaining (Weber et al., 2019; Dong et al., 2018). However, these works still highly rely on the formal representations and they do not generate the natural language subgoals at each step.

Problem Solver and Cognitive Architectures: Our work is also largely inspired by cognitive architectures such as ACT-R (Anderson et al., 1997) and SOAR (Laird, 2012), which originate from Newell's GPS. These cognitive architectures employ symbolic systems to simulate the human general cognitive processes, but have not been used on complex reasoning problems in NLP.

Internal Monologue: Internal monologue is the subjective experience of language without overt articulation, and it plays important roles in cognition (Alderson-Day and Fernyhough, 2015). The role of internal monologue in problem solving/reasoning is mixed. Studies have been shown that internal monologue might not be crucial to visual reasoning (Phillips, 1999), whereas in verbal reasoning tasks, some subjects indeed rely more on the internal monologue than the visual imagery (Bacon et al., 2003). However, there is still not a wide concensus on the form/grammar of the internal monologue.

Question Decomposition: Our work is also different from several existing works about question decomposition, where the strategy of decomposition is largely reflected by the question itself (Min et al., 2019; Wolfson et al., 2020). In contrast, the expressions of our questions are already simple, and don't reflect the decomposition strategies.

6 Discussion and Future Work

6.1 Discussion of the Current Method

Does EVR solve the problems raised in Section 2.4? We believe our neural GPS at least partially solves the issues mentioned in Section 2.4. First, EVR decomposes a hard problem into several simple ones, thus resembling the human thinking process more. In addition, this modular strategy also enables EVR to suffer less from the compositionality generalization problem (as shown in Section 4). Second, during each step of reasoning, all the subgoals and the derived statements are expressed in natural language, thus making the reasoning proofs interpretable.

Could this task be addressed with a pure rule-based approach? Due to the language variations introduced (by the authors) to this dataset, a pure rule-based method is probably not easily applicable. For example, during the generation of pattern 10 training data, we use the provided meta data (formal language, no variation, not available at test time) to help us produce the necessary supervision. We found it otherwise hard to compose the rules directly from the natural language representation.

Can the operator proposer be replaced by a rule-based one? Due to the synthetic natural of the dataset, it is possible to replace the operator proposer in Section 3.3 with some rule-based algorithm (e.g., patterns 1, 3, 5, 7, 9, 11 can be processed with pure rules). However, this is not the goal of our work. In this paper, we study how well a Seq2Seq model can learn under more realistic conditions, i.e., in real-world scenarios, the agent might only have access to input-output training pairs (rather than rules).

6.2 Challenges of Applying EVR on Real World Problems

While EVR achieved the goals set in this work, several important questions remain for future work:

Limited language variation: Although the data used here contains some variations in language, it is still considerably simpler than real-world natural language. Thus, it remains unanswered whether the Seq2Seq components can achieve a robust performance under actual natural language.

Can other complex problems be reduced to simple ones? It is unclear whether most of the real-world multi-hop reasoning problems can be reduced to a (not too large) set of basic cognitive processes that can be learned.

Non-recursive problems and memory management: Due to the recursive nature of the problem we solve in this paper, the memory access/modification can be simplified. However, it is unknown whether recursive and context-free patterns are the only way in which human think. If not, the access and modification of working memory will become a challenging problem.

Acquisition of training data: Finally, the acquisition of high-quality training data is not always easy in real world. It is possible that low-quality training data introduce dangerous cascading errors.

7 Conclusion

In this paper we propose the Explainable multi-hop Verbal Reasoner (EVR) to solve a synthetic question answering problem that requires multi-hop reasoning (Clark et al., 2020). EVR answers a question by reducing a complex one to several simple ones, and guiding all reasoning steps with natural language for better interpretability. Evaluation of EVR shows it achieves high accuracy, suffer less from the compositionality generalization problem, and generalizes well when training data are not abundant.

References

Ben Alderson-Day and Charles Fernyhough. 2015. Inner speech: development, cognitive functions, phenomenology, and neurobiology. *Psychological bulletin*, 141(5):931.

John R Anderson, Michael Matessa, and Christian Lebiere. 1997. Act-r: A theory of higher level cognition and its relation to visual attention. *Human–Computer Interaction*, 12(4):439–462.

Akari Asai, Kazuma Hashimoto, Hannaneh Hajishirzi, Richard Socher, and Caiming Xiong. 2019. Learning to retrieve reasoning paths over wikipedia graph for question answering. In *International Conference on Learning Representations*.

Alison Bacon, Simon Handley, and Stephen Newstead. 2003. Individual differences in strategies for syllogistic reasoning. *Thinking & reasoning*, 9(2):133–168.

Pratyay Banerjee, Kuntal Kumar Pal, Arindam Mitra, and Chitta Baral. 2020. Careful selection of knowledge to solve open book question answering. In *57th Annual Meeting of the Association for Computational Linguistics, ACL 2019*, pages 6120–6129. Association for Computational Linguistics (ACL).

Peter Clark, Oyvind Tafjord, and Kyle Richardson. 2020. Transformers as soft reasoners over language. *arXiv preprint arXiv:2002.05867*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. 2018. Neural logic machines. In *International Conference on Learning Representations*.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476.

Nitish Gupta, Kevin Lin, Dan Roth, Sameer Singh, and Matt Gardner. 2019. Neural module networks for reasoning over text. In *International Conference on Learning Representations*.

Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171.

Drew A Hudson and Christopher D Manning. 2018. Compositional attention networks for machine reasoning. In *International Conference on Learning Representations*.

Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. 2020. Compositionality decomposed: How do neural networks generalise? *Journal of Artificial Intelligence Research*, 67:757–795.

Yichen Jiang and Mohit Bansal. 2019. Self-assembling modular networks for interpretable multi-hop reasoning. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4464–4474.

Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. 2016. Ask me anything: Dynamic memory networks for natural language processing. In *International conference on machine learning*, pages 1378–1387.

John E Laird. 2012. *The Soar cognitive architecture*. MIT press.

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*.

Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2019. Multi-hop reading comprehension through question decomposition and rescoring. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6097–6109.

Allen Newell. 1966. On the analysis of human problem solving protocols.

Allen Newell. 1994. *Unified theories of cognition*. Harvard University Press.

Allen Newell, John C Shaw, and Herbert A Simon. 1959. Report on a general problem solving program. In *IFIP congress*, volume 256, page 64. Pittsburgh, PA.

Louise H Phillips. 1999. The role of memory in the tower of london task. *Memory*, 7(2):209–231.

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

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*, 21:1–67.

Swarnadeep Saha, Sayan Ghosh, Shashank Srivastava, and Mohit Bansal. 2020. Prover: Proof generation for interpretable reasoning over rules. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 122–136.

Sanjay Subramanian, Ben Bogin, Nitish Gupta, Tomer Wolfson, Sameer Singh, Jonathan Berant, and Matt Gardner. 2020. Obtaining faithful interpretations from compositional neural networks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5594–5608.

Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. 2015. End-to-end memory networks. In *Advances in neural information processing systems*, pages 2440–2448.

Leon Weber, Pasquale Minervini, Jannes Münchmeyer, Ulf Leser, and Tim Rocktäschel. 2019. Nlprolog: Reasoning with weak unification for question answering in natural language. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6151–6161.

Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Yoav Goldberg, Daniel Deutch, and Jonathan Berant. 2020. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 8:183–198.

Vikas Yadav, Steven Bethard, and Mihai Surdeanu. 2019. Quick and (not so) dirty: Unsupervised selection of justification sentences for multi-hop question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2578–2589.

A Appendix

A.1 Reproducibility Checklist

Source code: https://github.com/clulab/releases/tree/master/naacl2021-evr

Computing infrastructure: Experiments are done with Nvidia RTX2070 and Nvidia V100.

Average run time: The running time of EVR1: it takes about 4 to 6 hours to train the three T5 models on RTX2070. On V100 this time is reduced by about 40%. Evaluation costs a lot of time. EVR1's evaluation on about 20000 test data (depth from 0 to 5) takes about 84 hours. Time using V100 is reduced by about 40%. This is because for EVR1, every question requires running T5 for many times, and the time grows with depth.

Number of parameters: T5 small:60 million; RT (RoBERTa-large): 355 million; PR: uses RoBERTa (355 million).

Validation performance: Not provided.

Evaluation Measure: Provided in the result section.

Bounds for hyper-parameter Only 3 hyperparameters. The learning rate is set to 1e-4, according to recommendations of other tutorials of using T5. The fact buffer size and rule buffer size are manually tuned.

Number of training and evaluation runs: Only run for one time, because evaluation consumes a lot of time. But random seed is set to 0 to ensure reproducibility.

Hyperparameter Configuration: Provided in paper. Fact buffer size = 20 and rule buffer size = 10. This is manually tuned.

Statistics of results: Not applicable because models are evaluated for only once.

Number of training samples: Provided.

Data processing: discussed in paper and appendix. For the detailed process please look at our code: https://github.com/clulab/releases/tree/master/naacl2021-evr

Train/Dev/Test spits: provided.

Downloadable link: https://github.com/clulab/releases/tree/master/naacl2021-evr

Name of language: English

Data collection process: Discussed in paper and appendix.

A.2 Full Table of All Data Pattern Generations

Table 9 shows the generation strategies of all the 12 patterns of training data. Note that the pattern 12's generated goal (i.e., "i want to prove [statement]") has exactly the same format as the pattern 1's input goal (i.e., "i want to prove [statement]").

Ptn	Generation Specifications
1	The input always has two episodic buffers: "there are $[X]$ fact buffers and $[Y]$ rule buffers"; "i want to prove [statement]". The output is always "GENERATE_SUBGOALS".
2	The input consists of two parts: the episodic buffer copied from pattern 1's input, and the added operator "operator: GENERATE_SUBGOALS". The output is "i want to judge whether the facts can prove [statement] OR i want to judge whether the rules can prove [statement]" if [statement] is a positive statement. For the negative statement, the output is "i want to judge whether the facts do not contradict [statement] AND i want to judge whether the rules do not contradict [statement]".
3	The input has two episodic buffers: "there are $[X]$ fact buffers and $[Y]$ rule buffers", and "i want to judge whether the facts can prove/do not contradict [statement]", depending on the generated goals in pattern 2. The output is always "GENERATE_SUBGOALS".
4	The input consists of two parts: the episodic buffer copied from pattern 3's input, and the added operator "operator: GENERATE_SUBGOALS". The output is "i want to judge whether fact buffer 1 can prove [statement] OR i want to judge whether fact buffer 2 can prove [statement] OR ..." if [statement] is a positive statement. For negative statement, the output is "i want to judge whether fact buffer 1 does not contradict [statement] AND i want to judge whether fact buffer 2 does not contradict [statement] AND ...".
5	The input has two episodic buffers: "there are $[X]$ fact buffers and $[Y]$ rule buffers", and "i want to judge whether fact buffer $[i]$ can prove/does not contradict [statement]", depending on the generated goals in pattern 4. The output is always "GET(FACT_BUFFER_$[i]$) THEN RUN(EPISODIC_BUFFER, FACT_BUFFER_$[i]$)".
6	The input has three parts. (1) the episodic buffer copied from the input of pattern 5; (2) the facts from the fact buffer indicated by the output of pattern 5; (3) the operator "operator: RUN". There are four possible outputs: when [statement] is a positive statement, the output is "true, this is confirmed by fact $[i]$" if there is a fact in the fact buffer to prove it, and is "false, CWA" if the [statement] is not proved by any facts in the buffer. When [statement] is a negative statement, the output is "true, NAF" if no facts in the fact buffer contradict it, and is "false, this is contradicted by fact $[i]$" if a fact in the fact buffer contradicts it.
7	The input has two episodic buffers: "there are $[X]$ fact buffers and $[Y]$ rule buffers", and "i want to judge whether the rules can prove/do not contradict [statement]", depending on the generated goals in pattern 2. The output is always "GENERATE_SUBGOALS".
8	The input consists of two parts: the episodic buffer copied from pattern 7's input, and the added operator "operator: GENERATE_SUBGOALS". The output is "i want to judge whether rule buffer 1 can prove [statement] OR i want to judge whether rule buffer 2 can prove [statement] OR ..." if [statement] is a positive statement. For negative statement, the output is "i want to judge whether rule buffer 1 does not contradict [statement] AND i want to judge whether rule buffer 2 does not contradict [statement] AND ...".
9	The input has two episodic buffers: "there are $[X]$ fact buffers and $[Y]$ rule buffers", and "i want to judge whether rule buffer $[i]$ can prove/does not contradict [statement]", depending on the generated goals in pattern 8. The output is always "GET(RULE_BUFFER_$[i]$) THEN RUN(EPISODIC_BUFFER, RULE_BUFFER_$[i]$)".
10	The input has two parts: the episodic buffer and the rule buffer determined by pattern 9, where episodic buffer is "there are $[X]$ fact buffers and $[Y]$ rule buffers" plus "i want to judge whether rule buffer $[j]$ can prove/does not contradict [statement]", and the rule buffer is all the rules in RULE_BUFFER_$[j]$ as determined by pattern 9 (please check Figure 4 as an example). The output are the statements derived from the matched rules. For example, "all rough people are green" can be used to prove "bob is green", and in order to prove "bob is green" using this rule, one needs to prove "bob is rough". In this case the output is "according to rule $[i]$, i need to prove bob is rough". There are other edge cases that need to be handled, such as multiple matched rules and negative query. For the handling of other edge cases, please check Appendix A.4.
11	The input always has two episodic buffers: "there are $[X]$ fact buffers and $[Y]$ rule buffers"; "according to rule $[i]$, i need to prove [statement]" depending on the generated text of pattern 10. The output is always "GENERATE_SUBGOALS".
12	The input consists of two parts: the episodic buffer copied from pattern 11's input, and the added operator "operator: GENERATE_SUBGOALS". The output is "i want to prove [statement]" (the [statement] is what comes from the episodic buffer).

Table 9: The generation strategy for all the 12 patterns of training data. In this verbal reasoning problem, any question with any depth reasoning can be solved by solving these 12 patterns of small problems.

A.3 Details of the Implementation of the Executor

As discussed in section 3.3, the executor has the three functions: (1) it first parses the instructions generated by the operator proposer; (2) it calls the corresponding components according to the parsed instruction; (3) it updates the working memory.

Instruction Parsing: In this paper, the executor needs to determine whether the instruction is "GENERATE_SUBGOALS", or "GET(FACT_BUFFER_[i]) THEN RUN(EPISODIC_BUFFER, FACT_BUFFER_[i])" or "GET(RULE_BUFFER_[i]) THEN RUN(EPISODIC_BUFFER, RULE_BUFFER_[i])". So we simply use "if-elif-else" to judge which instruction is generated.

Handling the Parsed Instruction: The executor calls the corresponding module according to the parsed instruction.

If the instruction is "GENERATE_SUBGOALS", then the executor runs T5 with this instruction (the input-output examples are pattern 2, 4, 8 in Figure 4).

If the instruction is "GET(FACT_BUFFER_[i]) THEN RUN(EPISODIC_BUFFER, FACT_BUFFER_[i])", the executor first gets the text of the facts of FACT_BUFFER_[i], then it runs T5 using the text of the current episodic buffer and the text of the retrieved fact buffer (an input-output example is pattern 6 in Figure 4).

If the instruction is "GET(RULE_BUFFER_[i]) THEN RUN(EPISODIC_BUFFER, RULE_BUFFER_[i])", the executor first gets the text of the rules of RULE_BUFFER_[i], then it runs T5 using the text of the current episodic buffer and the text of the retrieved rule buffer (an input-output example is pattern 10 in Figure 4).

Updating Working Memory: After the executor calls the correct modules according to the parsed instruction, new textual information will be generated. If the operator is "GENERATE_SUBGOALS", the generated textual information is a new subgoal, and this new subgoal will be used to replace the old goal in the working memory. If the operator is "GET(RULE_BUFFER_[i]) THEN RUN(EPISODIC_BUFFER, RULE_BUFFER_[i])" and there are matched rules after running this operator, a textual subgoal will also be generated and be used to replace the old subgoal in the working memory.

A.4 Elaborated Description of Pattern 10 Generation

During the generation of pattern 10, we use the formal representations of the facts and rules provided by the data to generate the input-output pairs. Note that this formal representation is only used in the generation of training data and not used at testing time.

All the facts have the formal representation “subject-verb-object”. This “S-V-O” combination is called a triple. And all the rules have the formal representation “triple1 (and optionally triple 2) -> triple”. For example, the statement “Bob is green” has the formal representation “(\" bob\" \" is\" \" green\" \"+\")” and “All green things are rough” has the formal representation “(((\" something\" \" is\" \" green\" \"+\")) -> (\" something\" \" is\" \" rough\" \"+\"))”. The “+” at the end means the polarity, i.e., whether there is negation in the triple.

Note that there are some variations when this formal representation is converted to natural language. For example, “(((\" something\" \" is\" \" green\" \"+\")) -> (\" something\" \" is\" \" rough\" \"+\"))” can be translated to “all green things are rough” or “if someone is green then it is rough” (or other expressions).

We use the formal representations of facts and rules to determine the formal representation to be generated, and then translate the generated formal representation to natural language. Assume a rule “tripleA and tripleB -> tripleC”, we call tripleA and tripleB “preconditions”, call tripleC “Effect”. Table 10 shows how to generate the preconditions when a rule’s effect matches a query. Assume the query has the form “E_q-V_q-O_q” (query subject entity, query verb, query object). Similarly, “E_p-V_p-O_p” means a **precondition** triple and “E_e-V_e-O_e” means an **effect** triple. “S” means “something” or “someone”. Note that in order for a rule to be matched with a query, V_q should be the same as V_e and O_q should be the same as O_e.

Positive Query	**Matched Effect**	**Matched Preconditions**	**Preconditions to Generate**
E_q-V_q-O_q	S-V_q-O_q	S-V_p-O_p	E_q-V_p-O_p
		E_p-V_p-O_p	E_p-V_p-O_p
	E_q-V_q-O_q	S-V_p-O_p	S-V_p-O_p
		E_p-V_p-O_p	E_p-V_p-O_p
S-V_q-O_q	S-V_q-O_q	S-V_p-O_p	S-V_p-O_p
		E_p-V_p-O_p	E_p-V_p-O_p
	E_p-V_q-O_q	S-V_p-O_p	S-V_p-O_p
		E_p-V_p-O_p	E_p-V_p-O_p
No matched rule	directly return false		
Negative Query	**Matched Effect**	**Matched Preconditions**	**Preconditions to Generate**
E_q-not-V_q-O_q	S-V_q-O_q	S-V_p-O_p	E_q-V_p-O_p
		E_p-V_p-O_p	E_p-V_p-O_p
	E_q-V_q-O_q	S-V_p-O_p	S-V_p-O_p
		E_p-V_p-O_p	E_p-V_p-O_p
S-not-V_q-O_q	directly return true		
No matched rule	directly return true		

Table 10: Matching rules for pattern 10 data to generate preconditions that are matched. Note that for negative query, the rule with positive effect is matched. This is used for inverse proof of a negative query (i.e., prove a negative statement is false).

For example, given the query “bob is green” and “all rough things are green”, the generated precondition should be “bob is rough”. Using formal representations, the representation for “bob is green” is “bob-is-green”, and the rule is “(something-is-rough)->(something-is-green)”. This situation is the row 1 in Table 10. The generated precondition should be “E_q-V_p-O_p”, i.e., the subject is the subject from the query, and the verb and the object are from the preconditions of the rule. So the representation of the generated precondition is “bob-is-rough”

A.5 An Actual Output of EVR

The following shows an actual output of our model on the first test problem of depth-4 and Table 11 shows the final result and the evaluation.

Evaluation Metric	**Value**
Question Label	True
Model’s Prediction	True
Model’s Proof	((((((((triple8 NAF) -> rule8)) -> rule9)) -> rule4) NAF) -> rule8)
All Candidate Proofs Provided by Dataset	[’((((((((triple8 NAF) -> rule8)) -> rule9)) -> rule4) NAF) -> rule8)’]

Table 11: Final result and the evaluation of our model on the first question of depth-4 test set.

====================

procssing instance 0
facts:
fact 1: the cat chases the rabbit.
fact 2: the cat is red.
fact 3: the cat sees the rabbit.
fact 4: the cat visits the mouse.
fact 5: the lion is green.
fact 6: the lion visits the rabbit.
fact 7: the mouse does not see the cat.
fact 8: the mouse sees the lion.
fact 9: the mouse visits the cat.
fact 10: the mouse does not visit the lion.
fact 11: the rabbit sees the cat.
rules:
rule 1: if something is red and kind then it does not visit the cat.
rule 2: if something sees the cat then it is not kind.
rule 3: if something does not visit the rabbit then it is big.
rule 4: if something chases the cat then the cat sees the lion.
rule 5: if the rabbit visits the mouse and the rabbit is big then the mouse visits the lion.
rule 6: if something is green then it sees the rabbit.
rule 7: if something chases the rabbit and it sees the mouse then the mouse sees the lion.
rule 8: if something sees the lion and it is not blue then it is kind.
rule 9: if something is kind then it chases the cat.
question and answer:
the cat is kind. True

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: i want to prove "the cat is kind".’]
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether the facts can prove "the cat is kind". OR i want to judge whether the rules can prove "the cat is kind".

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: i want to judge whether the facts can prove "the cat is kind".’]
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether fact buffer 1 can prove "the cat is kind". OR i want to judge whether fact buffer 2 can prove "the cat is kind". OR i want to judge whether fact buffer 3 can prove "the cat is kind".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 1 can prove "the cat is kind".']

generated operation: GET(FACT_BUFFER_1) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_1)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 1 can prove "the cat is kind". fact 1: the cat chases the rabbit. fact 2: the cat is red. fact 3: the cat sees the rabbit. fact 4: the cat visits the mouse. fact 5: the lion is green. operator: RUN </s>

generated answer:false, CWA.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 2 can prove "the cat is kind".']

generated operation: GET(FACT_BUFFER_2) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_2)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 2 can prove "the cat is kind". fact 6: the lion visits the rabbit. fact 7: the mouse does not see the cat. fact 8: the mouse sees the lion. fact 9: the mouse visits the cat. fact 10: the mouse does not visit the lion. operator: RUN </s>

generated answer:false, CWA.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 3 can prove "the cat is kind".']

generated operation: GET(FACT_BUFFER_3) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_3)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 3 can prove "the cat is kind". fact 11: the rabbit sees the cat. operator: RUN </s>

generated answer:false, CWA.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the rules can prove "the cat is kind".']

generated operation: GENERATE_SUBGOALS

generated subgoal:i want to judge whether rule buffer 1 can prove "the cat is kind". OR i want to judge whether rule buffer 2 can prove "the cat is kind". OR i want to judge whether rule buffer 3 can prove "the cat is kind".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 1 can prove "the cat is kind".']

generated operation: GET(RULE_BUFFER_1) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_1)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 1 can prove "the cat is kind". rule 1: if something is red and kind then it does not visit the cat. rule 2: if something sees the cat then it is not kind. rule 3: if something does not visit the rabbit then it is big. operator: RUN </s>

generated answer:false

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 2 can prove "the cat is kind".']

generated operation: GET(RULE_BUFFER_2) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_2)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 2 can prove "the cat is kind". rule 4: if something chases the cat then the cat sees the

lion. rule 5: if the rabbit visits the mouse and the rabbit is big then the mouse visits the lion. rule 6: if something is green then it sees the rabbit. operator: RUN </s>
generated answer:false

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 3 can prove "the cat is kind".']
generated operation: GET(RULE_BUFFER_3) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_3)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 3 can prove "the cat is kind". rule 7: if something chases the rabbit and it sees the mouse then the mouse sees the lion. rule 8: if something sees the lion and it is not blue then it is kind. rule 9: if something is kind then it chases the cat. operator: RUN </s>
generated answer:according to rule 8, i need to prove the cat sees the lion and the cat is not blue.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: according to rule 8, i need to prove the cat sees the lion and the cat is not blue.'] generated operation: GENERATE_SUBGOALS
generated subgoal:i want to prove "the cat sees the lion". AND i want to prove "the cat is not blue".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to prove "the cat sees the lion".']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether the facts can prove "the cat sees the lion". OR i want to judge whether the rules can prove "the cat sees the lion".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the facts can prove "the cat sees the lion".']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether fact buffer 1 can prove "the cat sees the lion". OR i want to judge whether fact buffer 2 can prove "the cat sees the lion". OR i want to judge whether fact buffer 3 can prove "the cat sees the lion".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 1 can prove "the cat sees the lion".']
generated operation: GET(FACT_BUFFER_1) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 1 can prove "the cat sees the lion". fact 1: the cat chases the rabbit. fact 2: the cat is red. fact 3: the cat sees the rabbit. fact 4: the cat visits the mouse. fact 5: the lion is green. operator: RUN </s>
generated answer:false, CWA.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 2 can prove "the cat sees the lion".']
generated operation: GET(FACT_BUFFER_2) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 2 can prove "the cat sees the lion". fact 6: the lion visits the rabbit. fact 7: the mouse does not see the cat. fact 8: the mouse sees the lion. fact 9: the mouse visits the cat. fact 10: the mouse does not visit the lion. operator: RUN </s>
generated answer:false, CWA.

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: i want to judge whether fact buffer 3 can prove "the cat sees the lion".’]
generated operation: GET(FACT_BUFFER_3) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_3)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 3 can prove "the cat sees the lion". fact 11: the rabbit sees the cat. operator: RUN </s>
generated answer:false, CWA.

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: i want to judge whether the rules can prove "the cat sees the lion".’]
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether rule buffer 1 can prove "the cat sees the lion". OR i want to judge whether rule buffer 2 can prove "the cat sees the lion". OR i want to judge whether rule buffer 3 can prove "the cat sees the lion".

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: i want to judge whether rule buffer 1 can prove "the cat sees the lion".’]
generated operation: GET(RULE_BUFFER_1) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 1 can prove "the cat sees the lion". rule 1: if something is red and kind then it does not visit the cat. rule 2: if something sees the cat then it is not kind. rule 3: if something does not visit the rabbit then it is big. operator: RUN </s>
generated answer:false

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: i want to judge whether rule buffer 2 can prove "the cat sees the lion".’]
generated operation: GET(RULE_BUFFER_2) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 2 can prove "the cat sees the lion". rule 4: if something chases the cat then the cat sees the lion. rule 5: if the rabbit visits the mouse and the rabbit is big then the mouse visits the lion. rule 6: if something is green then it sees the rabbit. operator: RUN </s>
generated answer:according to rule 4, i need to prove something chases the cat.

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: according to rule 4, i need to prove something chases the cat.’]
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to prove "something chases the cat".

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: i want to prove "something chases the cat".’]
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether the facts can prove "something chases the cat". OR i want to judge whether the rules can prove "something chases the cat".

---

episodic buffer [’episodic buffer: there are 3 fact buffers and 3 rule buffers.’, ’episodic buffer: i want to judge whether the facts can prove "something chases the cat".’]
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether fact buffer 1 can prove "something chases the cat". OR i want

to judge whether fact buffer 2 can prove "something chases the cat". OR i want to judge whether fact buffer 3 can prove "something chases the cat".

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 1 can prove "something chases the cat".']
generated operation: GET(FACT_BUFFER_1) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 1 can prove "something chases the cat". fact 1: the cat chases the rabbit. fact 2: the cat is red. fact 3: the cat sees the rabbit. fact 4: the cat visits the mouse. fact 5: the lion is green. operator: RUN </s>
generated answer:false, CWA.

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 2 can prove "something chases the cat".']
generated operation: GET(FACT_BUFFER_2) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 2 can prove "something chases the cat". fact 6: the lion visits the rabbit. fact 7: the mouse does not see the cat. fact 8: the mouse sees the lion. fact 9: the mouse visits the cat. fact 10: the mouse does not visit the lion. operator: RUN </s>
generated answer:false, CWA.

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 3 can prove "something chases the cat".']
generated operation: GET(FACT_BUFFER_3) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_3)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 3 can prove "something chases the cat". fact 11: the rabbit sees the cat. operator: RUN </s>
generated answer:false, CWA.

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the rules can prove "something chases the cat".']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether rule buffer 1 can prove "something chases the cat". OR i want to judge whether rule buffer 2 can prove "something chases the cat". OR i want to judge whether rule buffer 3 can prove "something chases the cat".

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 1 can prove "something chases the cat".']
generated operation: GET(RULE_BUFFER_1) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 1 can prove "something chases the cat". rule 1: if something is red and kind then it does not visit the cat. rule 2: if something sees the cat then it is not kind. rule 3: if something does not visit the rabbit then it is big. operator: RUN </s>
generated answer:false

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 2 can prove "something chases the cat".']

generated operation: GET(RULE_BUFFER_2) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 2 can prove "something chases the cat". rule 4: if something chases the cat then the cat sees the lion. rule 5: if the rabbit visits the mouse and the rabbit is big then the mouse visits the lion. rule 6: if something is green then it sees the rabbit. operator: RUN </s>
generated answer:false

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 3 can prove "something chases the cat".']
generated operation: GET(RULE_BUFFER_3) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_3)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 3 can prove "something chases the cat". rule 7: if something chases the rabbit and it sees the mouse then the mouse sees the lion. rule 8: if something sees the lion and it is not blue then it is kind. rule 9: if something is kind then it chases the cat. operator: RUN </s>
generated answer:according to rule 9, i need to prove something is kind.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: according to rule 9, i need to prove something is kind.']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to prove "something is kind".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to prove "something is kind".']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether the facts can prove "something is kind". OR i want to judge whether the rules can prove "something is kind".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the facts can prove "something is kind".']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether fact buffer 1 can prove "something is kind". OR i want to judge whether fact buffer 2 can prove "something is kind". OR i want to judge whether fact buffer 3 can prove "something is kind".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 1 can prove "something is kind".']
generated operation: GET(FACT_BUFFER_1) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 1 can prove "something is kind". fact 1: the cat chases the rabbit. fact 2: the cat is red. fact 3: the cat sees the rabbit. fact 4: the cat visits the mouse. fact 5: the lion is green. operator: RUN </s>
generated answer:false, CWA.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 2 can prove "something is kind".']
generated operation: GET(FACT_BUFFER_2) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 2 can prove "something is kind". fact 6: the lion visits the rabbit. fact 7: the mouse

does not see the cat. fact 8: the mouse sees the lion. fact 9: the mouse visits the cat. fact 10: the mouse does not visit the lion. operator: RUN </s>
generated answer:false, CWA.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 3 can prove "something is kind".']
generated operation: GET(FACT_BUFFER_3) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_3)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 3 can prove "something is kind". fact 11: the rabbit sees the cat. operator: RUN </s>
generated answer:false, CWA.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the rules can prove "something is kind".']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether rule buffer 1 can prove "something is kind". OR i want to judge whether rule buffer 2 can prove "something is kind". OR i want to judge whether rule buffer 3 can prove "something is kind".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 1 can prove "something is kind".']
generated operation: GET(RULE_BUFFER_1) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 1 can prove "something is kind". rule 1: if something is red and kind then it does not visit the cat. rule 2: if something sees the cat then it is not kind. rule 3: if something does not visit the rabbit then it is big. operator: RUN </s>
generated answer:false

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 2 can prove "something is kind".']
generated operation: GET(RULE_BUFFER_2) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 2 can prove "something is kind". rule 4: if something chases the cat then the cat sees the lion. rule 5: if the rabbit visits the mouse and the rabbit is big then the mouse visits the lion. rule 6: if something is green then it sees the rabbit. operator: RUN </s>
generated answer:false

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 3 can prove "something is kind".']
generated operation: GET(RULE_BUFFER_3) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_3)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 3 can prove "something is kind". rule 7: if something chases the rabbit and it sees the mouse then the mouse sees the lion. rule 8: if something sees the lion and it is not blue then it is kind. rule 9: if something is kind then it chases the cat. operator: RUN </s>
generated answer:according to rule 8, i need to prove something sees the lion and something is not blue.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: according to rule 8, i need to prove something sees the lion and something is not blue.']

generated operation: GENERATE_SUBGOALS

generated subgoal:i want to prove "something sees the lion". AND i want to prove "something is not blue".

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to prove "something sees the lion".']

generated operation: GENERATE_SUBGOALS

generated subgoal:i want to judge whether the facts can prove "something sees the lion". OR i want to judge whether the rules can prove "something sees the lion".

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the facts can prove "something sees the lion".']

generated operation: GENERATE_SUBGOALS

generated subgoal:i want to judge whether fact buffer 1 can prove "something sees the lion". OR i want to judge whether fact buffer 2 can prove "something sees the lion". OR i want to judge whether fact buffer 3 can prove "something sees the lion".

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 1 can prove "something sees the lion".']

generated operation: GET(FACT_BUFFER_1) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_1)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 1 can prove "something sees the lion". fact 1: the cat chases the rabbit. fact 2: the cat is red. fact 3: the cat sees the rabbit. fact 4: the cat visits the mouse. fact 5: the lion is green. operator: RUN </s>

generated answer:false, CWA.

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 2 can prove "something sees the lion".']

generated operation: GET(FACT_BUFFER_2) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_2)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 2 can prove "something sees the lion". fact 6: the lion visits the rabbit. fact 7: the mouse does not see the cat. fact 8: the mouse sees the lion. fact 9: the mouse visits the cat. fact 10: the mouse does not visit the lion. operator: RUN </s>

generated answer:true, this is confirmed by fact 8.

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to prove "something is not blue".']

generated operation: GENERATE_SUBGOALS

generated subgoal:i want to judge whether the facts do not contradict "something is not blue". AND i want to judge whether the rules do not contradict "something is not blue".

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the facts do not contradict "something is not blue".']

generated operation: GENERATE_SUBGOALS

generated subgoal:i want to judge whether fact buffer 1 does not contradict "something is not blue". AND i want to judge whether fact buffer 2 does not contradict "something is not blue". AND i want to judge whether fact buffer 3 does not contradict "something is not blue".

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to

judge whether fact buffer 1 does not contradict "something is not blue".']
generated operation: GET(FACT_BUFFER_1) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 1 does not contradict "something is not blue". fact 1: the cat chases the rabbit. fact 2: the cat is red. fact 3: the cat sees the rabbit. fact 4: the cat visits the mouse. fact 5: the lion is green. operator: RUN </s>
generated answer:true, NAF.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 2 does not contradict "something is not blue".']
generated operation: GET(FACT_BUFFER_2) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 2 does not contradict "something is not blue". fact 6: the lion visits the rabbit. fact 7: the mouse does not see the cat. fact 8: the mouse sees the lion. fact 9: the mouse visits the cat. fact 10: the mouse does not visit the lion. operator: RUN </s>
generated answer:true, NAF.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 3 does not contradict "something is not blue".']
generated operation: GET(FACT_BUFFER_3) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_3)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 3 does not contradict "something is not blue". fact 11: the rabbit sees the cat. operator: RUN </s>
generated answer:true, NAF.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the rules do not contradict "something is not blue".']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether rule buffer 1 does not contradict "something is not blue". AND i want to judge whether rule buffer 2 does not contradict "something is not blue". AND i want to judge whether rule buffer 3 does not contradict "something is not blue".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 1 does not contradict "something is not blue".']
generated operation: GET(RULE_BUFFER_1) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 1 does not contradict "something is not blue". rule 1: if something is red and kind then it does not visit the cat. rule 2: if something sees the cat then it is not kind. rule 3: if something does not visit the rabbit then it is big. operator: RUN </s>
generated answer:true

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 2 does not contradict "something is not blue".']
generated operation: GET(RULE_BUFFER_2) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 2 does not contradict "something is not blue". rule 4: if something chases the cat then

the cat sees the lion. rule 5: if the rabbit visits the mouse and the rabbit is big then the mouse visits the lion. rule 6: if something is green then it sees the rabbit. operator: RUN </s>

generated answer:true

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 3 does not contradict "something is not blue".']

generated operation: GET(RULE_BUFFER_3) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_3)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 3 does not contradict "something is not blue". rule 7: if something chases the rabbit and it sees the mouse then the mouse sees the lion. rule 8: if something sees the lion and it is not blue then it is kind. rule 9: if something is kind then it chases the cat. operator: RUN </s>

generated answer:true

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to prove "the cat is not blue".']

generated operation: GENERATE_SUBGOALS

generated subgoal:i want to judge whether the facts do not contradict "the cat is not blue". AND i want to judge whether the rules do not contradict "the cat is not blue".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the facts do not contradict "the cat is not blue".']

generated operation: GENERATE_SUBGOALS

generated subgoal:i want to judge whether fact buffer 1 does not contradict "the cat is not blue". AND i want to judge whether fact buffer 2 does not contradict "the cat is not blue". AND i want to judge whether fact buffer 3 does not contradict "the cat is not blue".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 1 does not contradict "the cat is not blue".']

generated operation: GET(FACT_BUFFER_1) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_1)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 1 does not contradict "the cat is not blue". fact 1: the cat chases the rabbit. fact 2: the cat is red. fact 3: the cat sees the rabbit. fact 4: the cat visits the mouse. fact 5: the lion is green. operator: RUN </s>

generated answer:true, NAF.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 2 does not contradict "the cat is not blue".']

generated operation: GET(FACT_BUFFER_2) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_2)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 2 does not contradict "the cat is not blue". fact 6: the lion visits the rabbit. fact 7: the mouse does not see the cat. fact 8: the mouse sees the lion. fact 9: the mouse visits the cat. fact 10: the mouse does not visit the lion. operator: RUN </s>

generated answer:true, NAF.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether fact buffer 3 does not contradict "the cat is not blue".']

generated operation: GET(FACT_BUFFER_3) THEN RUN (EPISODIC_BUFFER, FACT_BUFFER_3)

buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether fact buffer 3 does not contradict "the cat is not blue". fact 11: the rabbit sees the cat. operator: RUN </s>
generated answer:true, NAF.

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether the rules do not contradict "the cat is not blue".']
generated operation: GENERATE_SUBGOALS
generated subgoal:i want to judge whether rule buffer 1 does not contradict "the cat is not blue". AND i want to judge whether rule buffer 2 does not contradict "the cat is not blue". AND i want to judge whether rule buffer 3 does not contradict "the cat is not blue".

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 1 does not contradict "the cat is not blue".']
generated operation: GET(RULE_BUFFER_1) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_1)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 1 does not contradict "the cat is not blue". rule 1: if something is red and kind then it does not visit the cat. rule 2: if something sees the cat then it is not kind. rule 3: if something does not visit the rabbit then it is big. operator: RUN </s>
generated answer:true

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 2 does not contradict "the cat is not blue".']
generated operation: GET(RULE_BUFFER_2) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_2)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 2 does not contradict "the cat is not blue". rule 4: if something chases the cat then the cat sees the lion. rule 5: if the rabbit visits the mouse and the rabbit is big then the mouse visits the lion. rule 6: if something is green then it sees the rabbit. operator: RUN </s>
generated answer:true

---

episodic buffer ['episodic buffer: there are 3 fact buffers and 3 rule buffers.', 'episodic buffer: i want to judge whether rule buffer 3 does not contradict "the cat is not blue".']
generated operation: GET(RULE_BUFFER_3) THEN RUN (EPISODIC_BUFFER, RULE_BUFFER_3)
buffer input:episodic buffer: there are 3 fact buffers and 3 rule buffers. episodic buffer: i want to judge whether rule buffer 3 does not contradict "the cat is not blue". rule 7: if something chases the rabbit and it sees the mouse then the mouse sees the lion. rule 8: if something sees the lion and it is not blue then it is kind. rule 9: if something is kind then it chases the cat. operator: RUN </s>
generated answer:true
