LLMs cannot *find* reasoning errors, but can *correct* them!

Anonymous ACL submission

Abstract

While self-correction has shown promise in improving LLM outputs in terms of style and quality (e.g. Chen et al., 2023; Madaan et al., 2023), recent attempts to self-correct logical or reasoning errors often cause correct answers to become incorrect, resulting in worse performances overall (Huang et al., 2023). In this paper, we break down the self-correction process into two core components: **mistake finding** and **output correction**. For mistake finding, we release BIG-Bench Mistake, a dataset of logical mistakes in Chain-of-Thought reasoning traces. We provide benchmark numbers for several state-of-the-art LLMs, and demonstrate that LLMs generally struggle with finding logical mistakes. For output correction, we propose a backtracking method which provides large improvements when given information on mistake location. We construe backtracking as a lightweight alternative to reinforcement learning methods, and show that it remains effective with a reward model at 60-70% accuracy.

1 Introduction

Large Language Models (LLMs) have dominated the field of NLP in recent years, achieving state-of-the-art performance in a large variety of applications. In particular, LLMs have demonstrated the ability to solve tasks with zero- or few-shot prompting, giving rise to prompting methods such as Chain-of-Thought (CoT) (Wei et al., 2022), Self-Consistency (SC) (Wang et al., 2023), ReAct (Yao et al., 2022), etc.

Recent literature on few- or zero-shot prompting has focused on the concept of *self-correction*, i.e. having an LLM correct its own outputs (Shinn et al., 2023; Miao et al., 2023; Madaan et al., 2023; Chen et al., 2023; Saunders et al., 2022). (See Pan et al. (2023) for a review of the literature.)

However, Huang et al. (2023) note that while self-correction may prove effective for improving model outputs in terms of style and quality, when it comes to *reasoning* tasks, LLMs struggle to identify and fix errors without external feedback: for example, Reflexion (Shinn et al., 2023) and RCI (Kim et al., 2023) both use ground truth correctness as a signal to halt the self-correction loop. Initially observed by Madaan et al. (2023) on a math dataset, Huang et al. (2023) further demonstrate this shortcoming of self-correction in 2 additional datasets.

While previous work typically present self-correction as a single process, we divide it into **mistake finding** and **output correction**.

Mistake finding is a fundamental reasoning skill that has been studied and utilised extensively in philosophy, psychology, and mathematics, spawning concepts such as critical thinking, and logical and mathematical fallacies. One might expect that the ability to find mistakes should also be an important requirement for LLMs. However, our results show that state-of-the-art LLMs currently *cannot* find mistakes reliably.

Output correction involves partially or completely changing previously generated outputs. With self-correction, this is typically done with outputs generated by the same model (see Pan et al. (2023)). Despite LLMs' inability to find mistakes, our results show that they can *correct* outputs using our backtracking method, if given information about the mistakes, such as via a small, supervised reward model.

Our contributions for this paper are as follows:

1. With Chain-of-Thought prompting, any task can be turned into a mistake-finding task. We collect and release[1] to the research community **BIG-Bench Mistake**, a dataset of CoT-style traces[2] generated using PaLM 2 (Anil et al., 2023), and annotated according to where the first logical mistake is. To our knowledge, BIG-Bench Mistake is the first dataset of its kind that goes be-

[1]Available at [redacted].

[2]We refer to a set of CoT reasoning steps as a *trace*.

yond problems in mathematics.

2. We produce benchmark results for our dataset to test the reasoning capabilities of five state-of-the-art LLMs. We demonstrate that these LLMs **struggle with mistake finding, even for objective, unambiguous cases**. We hypothesise that this is a main contributing factor to LLMs' inability to self-correct reasoning errors, and call on the research community to pursue further improvements on the mistake finding task.

3. We propose **backtracking** as an output correction technique that uses mistake location information to improve performance on the original task. We demonstrate that this method corrects outputs that are originally incorrect, with minimal effect on outputs that are originally correct.

4. We construe backtracking as a form of "verbal reinforcement learning" (Shinn et al., 2023), allowing iterative improvement on CoT outputs without requiring any weight updates. We propose that backtracking can be used with a trained classifier as a reward model, and demonstrate the effectiveness of backtracking at various reward model accuracies.

2 BIG-Bench Mistake

BIG-Bench Mistake contains 2186 sets of CoT-style traces. Each trace is generated with PaLM 2 Unicorn (Anil et al., 2023), and annotated with the location of the first logical error. Table 1 shows an example trace where the mistake location[3] is the 4^{th} step.

Our traces span across a set of 5 tasks from the BIG-bench dataset (Srivastava et al., 2023): word sorting, tracking shuffled objects, logical deduction, multi-step arithmetic, and Dyck languages[4]. CoT prompting is used to prompt PaLM 2 to answer questions from each task. As we wanted to separate our CoT traces into distinct steps, we follow Yao et al. (2022) and generate each step separately, using the newline as a stop token.

Sort the following words alphabetically: List: hypochlorite ponderosa phone credulity	
Thought 1:	I should start by looking at the first letter of the words in the list. The first letter: "hypochlorite": "h" (8). "ponderosa": "p" (16). "phone": "p" (16). "credulity": "c" (3).
Thought 2:	We now have: (3) "credulity" < (8) "hypochlorite" < (16) ["ponderosa" ? "phone"].
Thought 3:	Now let's sort this subpart ["ponderosa" ? "phone"] by looking at their second letters. The second letter: "ponderosa": "o" (15). "phone": "h" (8).
Thought 4: **(MISTAKE)**	We now have: (8) "phone" < (15) "ponderosa" for the subpart. Hence, we have "credulity" < "phone" < "ponderosa".
Thought 5:	I have now sorted all the words. The answer is credulity hypochlorite phone ponderosa

Table 1: Example of a CoT trace for the word sorting task. There is a mistake in Thought 4: the ordering "credulity" < "phone" < "ponderosa" is missing the word *hypochlorite*.

All traces are generated with temperature $= 0$. The correctness of answers are determined by exact match. Prompts can be found at [redacted].

2.1 Annotation

Each generated trace is annotated with the first logical error. We ignore any subsequent errors as they may be dependent on the original error.

Note that traces can contain a logical mistake yet arrive at the correct answer. To disambiguate the two types of correctness, we will use the terms **correct**$_{ans}$ and **incorrect**$_{ans}$ to refer to whether the final **ans**wer of the trace is correct. **Accuracy**$_{ans}$ would therefore refer to the overall accuracy for the task, based on how many final answers are correct. To refer to whether the trace contains a logical **mis**take (rather than the correctness of the final answer), we will use **correct**$_{mis}$ and **incorrect**$_{mis}$.

2.1.1 Human annotation

For 4 of the 5 tasks, we recruit human annotators to go through each trace and identify any errors. Annotators have no domain expertise but are given guidelines[5] to complete the task.

Before annotation, we sample a set of 300 traces for each task, where 255 (85%) are incorrect$_{ans}$, and 45 (15%) are correct$_{ans}$. Since human annotation is a limited and expensive resource, we chose this distribution to maximise the number of steps

[3] As some traces may not contain mistakes, we use the term *mistake location* as a multi-class label that can refer to either the integer N where the N^{th} step contains the first mistake, or that there are no mistakes.

[4] These 5 tasks are selected because 1) Anil et al. (2023) demonstrate that PaLM 2 performs poorly on these tasks, so it is likely to generate mistakes in CoT traces; 2) mistakes in these tasks are likely to be unambiguous, therefore minimising subjectivity during annotation; and 3) identifying mistakes for these tasks does not require expertise knowledge.

[5] See [redacted] for further details.

Task	# of **correct**$_{ans}$ traces	# of **incorrect**$_{ans}$ traces	# of **incorrect**$_{mis}$ traces	**Total**
Word sorting	45	255	266	300
Tracking shuffled objects	45	255	260	300
Logical deduction	45	255	294	300
Multistep arithmetic	45	255	238	300
Dyck languages	482	504	650	986
Dyck languages (sampled)	88	504	545	592

Table 2: Number of traces in our dataset that are correct and incorrect. Dyck languages (sampled) is a set of traces sampled so that the ratio of correct$_{ans}$ to incorrect$_{ans}$ traces matches other tasks.

containing mistakes and to prevent over-saturation of correct steps. We also include some correct$_{ans}$ traces because some may contain logical errors despite the correct answer, and to ensure that the dataset included examples of correct steps that are near the end of the trace. To account for this skewed distribution, results in section 4 are split according to whether the original trace is correct$_{ans}$ or not.

Following Lightman et al. (2023), annotators are instructed to go through each step in the trace and indicate whether the step is correct or not (binary choice). Annotators only submit their answers when all steps are annotated, or there is one incorrect step. If an incorrect step is identified, the remaining steps are skipped. This is to avoid ambiguities where a logically correct deduction is dependent on a previous mistake. Our annotation guidelines can be found at [redacted], and we include a screenshot of the user interface in Appendix D.

Each trace is annotated by at least 3 annotators. If there are any disagreements, we take the majority label. We calculate Krippendorff's alpha (Hayes and Krippendorff, 2007) to measure inter-rater reliability (see Table 3).

Task	Krippendorff's α
Word sorting	0.979
Tracking shuffled objects	0.998
Logical deduction	0.996
Multistep arithmetic	0.984

Table 3: Inter-rater reliability for the human-annotated tasks, measured by Krippendorff's alpha.

2.1.2 Automatic annotation

For Dyck languages, we use mostly automatic instead of human annotation, as the traces show limited variation in phrasing and solution paths.

For each trace, we algorithmically generate a set of steps based on the format used in the prompt examples. Using pattern matching, we identify whether each model-generated step conforms to the same format. If so, we compare the two and assume that the trace is incorrect if the symbols do not match. Additionally, we account for edge cases such as where the final two steps are merged into one, or variations in presentation where symbols may or may not be placed in quotes. We release the code at [redacted] along with our dataset.

3 Benchmark results

Table 4 shows the accuracy of GPT-4-Turbo, GPT-4, GPT-3.5-Turbo, Gemini Pro, and PaLM 2 Unicorn on our mistake-finding dataset. For each question, the possible answers are either: that there are no mistakes, or; if there is a mistake, the number N indicating the step in which the first mistake occurs. A model's output is only considered correct if the location matches exactly, or the output correctly indicates that there are no mistakes.

All models are given the same 3-shot prompts[5]. We use three different prompting methods:

- **Direct trace-level prompting** involves using the whole trace as input to the model and directly prompting for the mistake location. The model must output either the number representing the step, or "No".
- **Direct step-level prompting** prompts for a binary Yes/No output for every step, indicating whether or not the step is correct. In each generation call, the input contains the partial trace up to (and including) the target step, but does not contain results for previous steps. The final answer is inferred from where the first "No" output occurs (subsequent steps are ignored).
- **CoT step-level prompting** is an extension of direct, step-level prompting. Instead of a binary Yes/No response, we prompt the model to check the (partial) trace through a series of reasoning steps. This method is the most resource intensive of all three methods as it involves generating a whole CoT sequence for every step. As with direct step-level prompting, the final answer is inferred from where the first "No" output occurs (subsequent steps are ignored).

3.1 Discussion

All five models appear to struggle with our mistake finding dataset. GPT-4 attains the best results but only reaches an overall accuracy of 52.87 with direct step-level prompting. While exact parameter counts are undisclosed, GPT-4 is likely one of the largest models, along with PaLM 2 Unicorn[6], while Gemini Pro and GPT-3.5-Turbo are among the smaller models.

Our findings are in line with and builds upon results from Huang et al. (2023), who show that existing self-correction strategies are ineffective on reasoning errors. In our experiments, we specifically target the models' *mistake finding* ability and provide results for additional tasks. We show that state-of-the-art LLMs clearly struggle with mistake finding, even in the most simple and unambiguous cases. (For comparison, humans can identify mistakes without specific expertise, and have a high degree of agreement, as shown in Table 3.)

We hypothesise that LLMs' inability to find mistakes is a main contributing factor to why LLMs are unable to self-correct reasoning errors. If LLMs are unable to *identify* mistakes, it should be no surprise that they are unable to self-correct either.

3.2 Comparison of prompting methods

As we compare results across the three methods, we find that the accuracy on traces with no mistakes goes down[7] considerably from direct, trace-level prompting to CoT, step-level prompting. Figure 1 demonstrates this trade-off.

We hypothesise that this is due to the number of outputs generated by the model. Our three methods involve generating increasingly complex outputs, starting with direct, trace-level prompting requiring a single token, then direct, step-level prompting requiring one token per step, and finally CoT step-level prompting requiring several sentences per step. If each generation call has some probability of identifying a mistake, then the more calls made on each trace, the more likely the model will identify at least one mistake.

Model	Direct (trace)	Direct (step)	CoT (step)
Word sorting (11.7)			
GPT-4-Turbo	36.33	33.00	–
GPT-4	35.00	44.33	34.00
GPT-3.5-Turbo	11.33	15.00	15.67
Gemini Pro	10.67	–	–
PaLM 2 Unicorn	11.67	16.33	14.00
Tracking shuffled objects (5.4)			
GPT-4-Turbo	39.33	61.67	–
GPT-4	62.29	65.33	90.67
GPT-3.5-Turbo	10.10	1.67	19.00
Gemini Pro	37.67	–	–
PaLM 2 Unicorn	18.00	28.00	55.67
Logical deduction (8.3)			
GPT-4-Turbo	21.33	75.00	–
GPT-4	40.67	67.67	10.33
GPT-3.5-Turbo	2.00	25.33	9.67
Gemini Pro	8.67	–	–
PaLM 2 Unicorn	6.67	38.00	12.00
Multistep arithmetic (5.0)			
GPT-4-Turbo	38.33	43.33	–
GPT-4	44.00	42.67	41.00
GPT-3.5-Turbo	20.00	26.00	25.33
Gemini Pro	21.67	–	–
PaLM 2 Unicorn	22.00	21.67	23.67
Dyck languages† (24.5)			
GPT-4-Turbo	15.33*	28.67*	–
GPT-4	17.06	44.33*	41.00*
GPT-3.5-Turbo	8.78	5.91	1.86
Gemini Pro	2.00	–	–
PaLM 2 Unicorn	10.98	14.36	17.91
Overall			
GPT-4-Turbo	30.13	48.33	–
GPT-4	39.80	52.87	43.40
GPT-3.5-Turbo	10.44	14.78	14.31
Gemini Pro	16.14	–	–
PaLM 2 Unicorn	17.09	23.67	24.65

Table 4: Mistake finding accuracy across 5 tasks. The average number of steps in CoT reasoning traces in each task is in brackets. Unless otherwise indicated, the number of traces is in Table 2. We provide scores split by correctness$_{ans}$ of the original trace in Appendix E. Due to cost and usage limits, we are unable to provide results indicated by –.
† indicates that traces were sampled to contain 15% correct$_{ans}$ and 85% incorrect$_{ans}$ traces (see Table 2).
* indicates that traces were sampled to contain 45 correct$_{ans}$ and 255 incorrect$_{ans}$ traces to reduce costs.

[6] Note that the traces in our dataset are generated using PaLM 2 Unicorn and are sampled according to whether the final answer was correct or not. Therefore, we expect that using PaLM 2 itself to do mistake finding will produce different and likely biased results. Further work is needed to elucidate the difference between cross-model evaluation and self-evaluation.

[7] Note that the traces in BIG-Bench Mistake are sampled to contain more incorrect$_{ans}$ traces than correct$_{ans}$ traces (and therefore more incorrect$_{mis}$ traces than correct$_{mis}$ traces), so the overall mistake location accuracy appears higher for per-step prompting in Table 4, despite the poor accuracy for correct$_{mis}$ traces. For a full set of scores split by correctness$_{mis}$, see Appendix E.

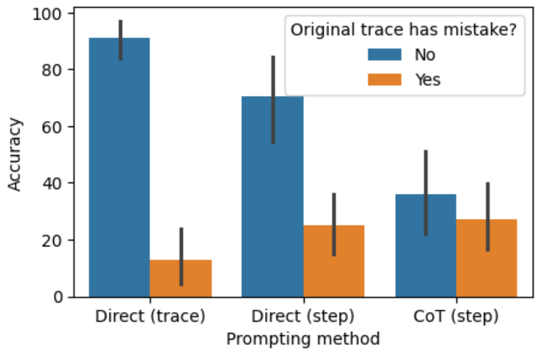

Figure 1: Graph of mistake location accuracies for each prompting method (excluding GPT-4-Turbo and Gemini Pro which we do not have all results for). Blue bars show accuracies on traces with no mistakes, so the model must predict that the trace has no mistake to be considered correct; orange bars show accuracies on traces with a mistake, so the model must predict the precise location of the mistake to be considered correct.

3.3 Few-shot prompting for mistake location as a proxy for correctness

In this section, we investigate whether our prompting methods can reliably determine the correctness$_{ans}$ of a trace rather than the mistake location. Our motivation was that even humans use mistake finding as a strategy for determining whether an answer is correct or not, like when going through mathematical proofs or argumentation. Additionally, it may be the case that directly predicting the correctness$_{ans}$ of a trace is easier than pinpointing the precise location of an error.

We calculate averaged F1 scores based on whether the model predicts there is a mistake in the trace. If there is a mistake, we assume the model prediction is that the trace is incorrect$_{ans}$. Otherwise, we assume the model prediction is that the trace is correct$_{ans}$. In Table 5, we average the F1s calculated with correct$_{ans}$ and incorrect$_{ans}$ as positive labels, weighted according to the number of times each label occurs. Note that the naive baseline of predicting all traces as incorrect achieves a weighted F1 average of 78.

The weighted F1 scores show that prompting for mistakes is likely a poor strategy for determining the correctness of the final answer. This is in line with our previous finding that LLMs struggle to identify mistake locations, and also builds upon results from Huang et al. (2023), who demonstrate that improvements from Reflexion (Shinn et al., 2023) and RCI (Kim et al., 2023) are only from using oracle correctness$_{ans}$ information.

Model	Direct (trace)	Direct (step)	CoT (step)
Word sorting			
GPT-4-Turbo	87.73	86.68	–
GPT-4	81.50	85.12	81.19
GPT-3.5-Turbo	6.58	35.07	77.79
Gemini Pro	69.93	–	–
PaLM 2 Unicorn	21.08	56.66	62.92
Tracking shuffled objects			
GPT-4-Turbo	52.23	74.31	–
GPT-4	76.38	75.69	95.03
GPT-3.5-Turbo	32.04	77.61	78.11
Gemini Pro	79.66	–	–
PaLM 2 Unicorn	22.18	48.77	78.29
Logical deduction			
GPT-4-Turbo	86.46	81.79	–
GPT-4	84.54	83.38	23.96
GPT-3.5-Turbo	10.34	67.62	61.31
Gemini Pro	48.57	–	–
PaLM 2 Unicorn	31.67	37.93	21.21
Multistep arithmetic			
GPT-4-Turbo	71.17	86.24	–
GPT-4	72.97	78.67	79.67
GPT-3.5-Turbo	3.76	53.18	64.08
Gemini Pro	32.21	–	–
PaLM 2 Unicorn	33.69	13.42	70.94
Dyck languages			
GPT-4-Turbo	51.96	85.87	–
GPT-4	62.33	85.73	79.60
GPT-3.5-Turbo	46.57	79.31	77.79
Gemini Pro	61.24	–	–
PaLM 2 Unicorn	31.17	31.63	25.20

Table 5: Weighted average F1 scores for predicted correctness$_{ans}$ of traces across 5 tasks. Baseline is 78 if we only select the incorrect$_{ans}$ label. As in Table 4, traces for the Dyck languages task has been sampled to match the ratio of correct$_{ans}$ to incorrect$_{ans}$ traces of the other tasks. See Table 2 for a full breakdown.

4 Backtracking

Madaan et al. (2023) and Huang et al. (2023) both demonstrate that self-correction is only effective with external feedback, e.g. both Shinn et al. (2023) and Kim et al. (2023) rely on oracle labels for improvements. However, there is often no external feedback available in many real-world applications.

As an alternative, we explore the possibility of replacing external feedback with a lightweight classifier trained on a small amount of data. Analogous to reward models in conventional reinforcement learning, this classifier detects any logical errors in a CoT trace, which is then fed back to the generator model to improve on the output. This can be done over multiple iterations to maximise improvements.

We propose a simple backtracking method to improve model outputs based on the location of logical errors:

1. First, the model generates an initial CoT trace. In our experiments, we use temperature $= 0$.
2. We then determine the mistake location in this trace using a reward model.
3. If there are no mistakes, we move onto the next trace. If there is a mistake (e.g. at Thought 4 in the example trace in Table 1), we prompt the model again for the same step but at temperature $= 1$, generating 8 outputs[8]. We use same prompt and the partial trace containing all steps up to but not including the mistake step (e.g. up to Thought 3, prompting for Thought 4).
4. From the 8 outputs, we filter out any options that match what was previously identified as a mistake. From the remaining outputs, we select one with the highest log-probability.
5. Finally, with the new, regenerated step in place of the previous one, we generate the remaining steps of the trace again at temperature $= 0$.

Our backtracking method provides several benefits over existing self-correction methods:

- Unlike Shinn et al. (2023), Kim et al. (2023), etc., our approach does not depend on oracle knowledge of the answer. Instead, it relies on information (e.g. from trained a reward model) about logical errors, which can be determined on a step-by-step basis using a reward model. Logical errors can occur in correct$_{ans}$ traces, or not occur in incorrect$_{ans}$ traces[9].
- Unlike Miao et al. (2023), Shinn et al. (2023), and many others, backtracking does not rely on any specific prompt text or phrasing, thereby reducing associated idiosyncrasies.
- Compared to approaches that require regenerating the entire trace, backtracking reduces computational cost by reusing previous steps that are known to be logically sound.
- Backtracking improves on the quality of the intermediate steps directly, which can be useful in scenarios that require correct steps (e.g. generating solutions to math questions), and also generally improves interpretability.

Backtracking with mistake location information from a reward model can be construed as a lightweight RL method. However, unlike conventional deep reinforcement learning:

- Backtracking with a reward model does not does not require any training of the original generator model. Once the reward model is trained, it can be used for backtracking with any LLM as the generator, and can also be updated independently of the generator LM. This can be especially helpful when LLMs are frequently updated to new checkpoints.
- Backtracking only requires training of a small reward model. Compared to methods that require training of the generator model, backtracking is far more efficient in terms of computing resources and available data.
- The process of backtracking is more interpretable than updating the weights of the generator model directly, as is required for many deep RL methods. It clearly pinpoints the location at which an error occurs, which can help the debugging process and allow faster development and iterations of models.

4.1 Backtracking with gold mistake location

As an initial experiment, we use labels from BIG-Bench Mistake to test if an LLM is able to correct logical errors using backtracking, independent of its inherent ability to identify these errors or any other reward model.

For example, if the mistake location is in step 4, we use backtracking to regenerate that step and continue the rest of the chain. If the mistake location is that there are no logical mistakes, we do not backtrack and use the original result.

4.1.1 Results

The results are shown in Table 6. To show that performance increases are not due to randomly resampling outputs, we compare our results to a random baseline, where a mistake location[10] is randomly selected for each trace and we perform backtracking based on the random location.

[8] For this paper, we only report results where 8 outputs are used, and leave for future investigation the effects of varying this number.

[9] Having no logical errors in incorrect$_{ans}$ traces is much rarer but does exist, for example when the answer is correct but is not captured by exact match, or if the original question is faulty and has multiple possible answers.

[10] As described above, the mistake location can be either the number representing the step, or that there are no mistakes. If there are no mistakes, we do not use backtracking and simply use the original trace.

Task	With **mistake** location		With **random** location		Avg. num. of steps
	Δ accuracy$_{\checkmark}$	**Δaccuracy$_{\times}$**	**Δ accuracy$_{\checkmark}$**	**Δaccuracy$_{\times}$**	
Word sorting	-11.11	+23.53	-15.56	+11.76	11.7
Tracking shuffled objects	-6.67	+43.92	-6.67	+20.39	5.4
Logical deduction	-11.43	+36.86	-13.33	+21.57	8.3
Multistep arithmetic	-0.00	+18.04	-8.89	+10.59	5.0
Dyck languages	-6.82	+18.06	-15.91	+5.16	24.5

Table 6: Absolute differences in accuracy$_{ans}$ before and after backtracking. "With mistake location" indicates that backtracking was done using oracle mistake locations from the dataset; "With random location" indicates that backtracking was done based on randomly selected locations. **Δaccuracy$_{\checkmark}$** refers to differences in accuracy$_{ans}$ on the set of traces whose *original* answer was correct$_{ans}$; **Δaccuracy$_{\times}$** for traces whose original answer was incorrect$_{ans}$. The average number of steps in a trace is shown to demonstrate the likelihood of randomly selecting the correct mistake location in the random baseline condition.

Note that Table 6 separates results into numbers for the correct set and the incorrect set, referring to whether the *original* trace was correct$_{ans}$ or not. This gives a clearer picture than the overall accuracy$_{ans}$, which would be skewed by the proportion of traces that were originally correct$_{ans}$ (15%) and incorrect$_{ans}$ (85%).

Scores represent the absolute differences in accuracy$_{ans}$. We perform backtracking on both correct$_{ans}$ and incorrect$_{ans}$ traces, as long as there is a mistake in one of the steps.

Δaccuracy$_{\checkmark}$ refers to differences in accuracy$_{ans}$ on the set of traces whose *original* answer was correct$_{ans}$. Note that we take losses here because, despite the correct answer, there is a logical mistake in one of the steps. Therefore, the answer may change to an incorrect one when we backtrack.

Δaccuracy$_{\times}$ is the same but for incorrect$_{ans}$ traces, so the answers may have been corrected, hence increasing accuracy$_{ans}$.

For example, for the word sorting task, 11.11% of traces that were originally correct$_{ans}$ became incorrect$_{ans}$, while 23.53% of traces that were originally incorrect$_{ans}$ became correct$_{ans}$.

4.1.2 Discussion

The scores show that the gains from correcting incorrect$_{ans}$ traces are larger than losses from changing originally correct answers. Additionally, while the random baseline also obtained improvements, they are considerably smaller than if the true mistake location was used. Note that tasks involving fewer steps are more likely to improve performance in the random baseline, as the true mistake location is more likely to be identified.

While our numbers do show that our gains are higher than our losses, it should be noted that changes in the overall accuracy depends on the original accuracy achieved on the task. For example, if the original accuracy on the tracking shuffled objects task was 50%, the new accuracy would be 68.6%. On the other hand, if the accuracy was 99%, the new accuracy would drop to 92.8%. As our dataset is highly skewed and only contains 45 correct$_{ans}$ traces per task, we leave to future work to assess the effectiveness of backtracking in a more comprehensive way.

4.2 Backtracking with a simulated reward model

We show in subsection 4.1 that backtracking can be used to correct CoT traces using gold mistake location labels. To explore what level of accuracy reward model is needed when gold labels are not available, we use backtracking with *simulated* reward models, designed to produce labels at different levels of accuracy. We use accuracy$_{RM}$ to refer to the accuracy of the simulated reward model at identifying mistake locations.

For a given reward model at $X\%$ accuracy$_{RM}$, we use the mistake location from BIG-Bench Mistake $X\%$ of the time. For the remaining $(100 - X)\%$, we sample a mistake location randomly. To mimic the behaviour of a typical classifier, mistake locations are sampled to match the distribution found in the dataset. We also ensure that the sampled location does not match the correct location.

4.2.1 Results

Results are shown in Figure 2. We can see that the losses in Δaccuracy$_{\checkmark}$ begins to plateau at 65%. In fact, for most tasks, Δaccuracy$_{\checkmark}$ is already larger than Δaccuracy$_{\times}$ at around 60-70% accuracy$_{RM}$. This demonstrates that while higher accuracies produce better results, backtracking is still effective even without gold standard mistake location labels.

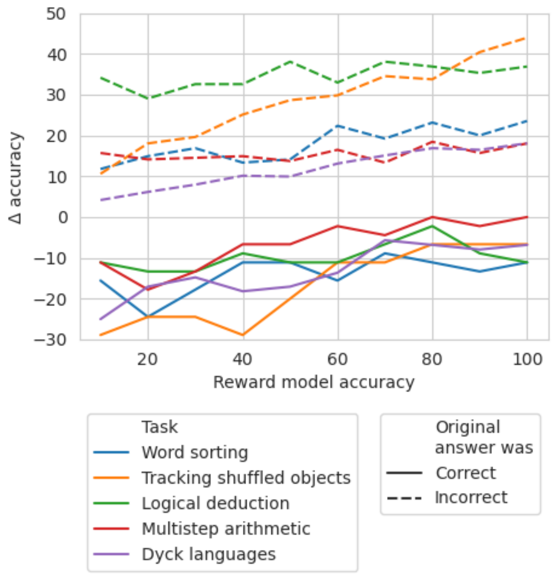

Figure 2: Δaccuracy$_{\checkmark}$ and Δaccuracy$_{\times}$ on each dataset as accuracy$_{RM}$ increases.

4.3 Reward Modeling

We investigate whether mistake-finding can benefit from a dedicated reward model, and if learning to find mistakes can transfer to out-of-distribution tasks. We fine-tune a PaLM 2-XS-Otter model with our available data for 20k steps and choose the checkpoint with the best validation results. We hold out one task for evaluation while training the reward model on the other 4 tasks.

Held-out task	Difference
Word sorting	+11.66
Tracking shuffled objects	+19.67
Logical deduction	-0.67
Multi-step arithmetic	+4.00
Dyck languages	+22.59

Table 7: Absolute difference in mistake finding accuracy between PaLM 2 Unicorn and a small, trained reward model.

Note the reward model we train is significantly smaller than our inference model. We show the relative improvements and losses in Table 7 vs. a zero-shot baseline on PaLM 2 Unicorn. We see gains for 4 out of 5 of the tasks. This suggests that it may be possible to train reward models to assist in backtracking, and that these reward models do not have to be large. Further, such a reward model can work on out-of-distribution mistakes. However, we believe more data may be necessary to improve results across the board on all tasks. We leave to future work the collection of this larger dataset and a more rigorous investigation of the trade-offs of model size vs. performance of the reward model.

We also leave for future investigation the effect of backtracking iteratively with a reward model: for example, the generator model may make another mistake after backtracking for the first time, which can then be identified and corrected again.

5 Related work

Datasets To our knowledge, the only publicly available dataset containing mistake annotations in LLM outputs is PRM800K (Lightman et al., 2023), which is a dataset of solutions to Olympiad-level math questions. Our dataset BIG-Bench Mistake covers a wider range of tasks to explore the reasoning capabilities of LLMs more thoroughly. Additionally, the generator LLM used in PRM800K has been fine-tuned on 1.5B math tokens and a dataset of step-by-step math solutions, which is not always possible for other use cases. In our paper, we want to explore few-shot in-context learning methods, which is more typical with API-based LLMs.

Self-correction Pan et al. (2023) present a plethora of self-correction methods in recent literature. While their list includes training-time correction strategies such as RLHF (Ouyang et al., 2022) and self-improve (Huang et al., 2022), our backtracking method falls into the category of post-hoc correction, where the correction process is applied to outputs that have already been generated.

Our paper focuses on correction of reasoning errors, rather than stylistic or qualitative improvements. Previous post-hoc correction methods that are applied to reasoning errors include Reflexion (Shinn et al., 2023) and RCI (Kim et al., 2023), both of which cause performance deterioration when the oracle labels are not used (Huang et al., 2023). Other methods like Self-Refine (Madaan et al., 2023) and iterative refinement (Chen et al., 2023) focus on qualitative or stylistic improvements rather than correcting logical errors.

6 Conclusion

In this paper, we describe and release our dataset BIG-Bench Mistake for mistake finding, and propose a backtracking method to correct logical errors in CoT style traces. We show that LLMs generally struggle with finding logical errors without external feedback, but argue that this feedback can come from a reward model instead. Finally, we demonstrate the effectiveness of backtracking, both with gold standard labels as well as with simulated reward models at lower levels of accuracy.

Limitations

One main limitation of our dataset is that it features tasks that are artificial and unrealistic for real-world applications. We made this choice to minimise ambiguity and subjectivity during the mistake finding process, but further work needs to be done to determine the effectiveness of backtracking in a more realistic setting.

Another limitation is that our paper does not evaluate backtracking on the original datasets on BIG-Bench, only showing results on the limited set that we sampled in a skewed manner, in order to maximise the value of the human annotators' time. We leave the full evaluation to future work.

References

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

Pinzhen Chen, Zhicheng Guo, Barry Haddow, and Kenneth Heafield. 2023. Iterative translation refinement with large language models. *arXiv preprint arXiv:2306.03856*.

Andrew F Hayes and Klaus Krippendorff. 2007. Answering the call for a standard reliability measure for coding data. *Communication methods and measures*, 1(1):77–89.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491*.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Ning Miao, Yee Whye Teh, and Tom Rainforth. 2023. Selfcheck: Using llms to zero-shot check their own step-by-step reasoning. *arXiv preprint arXiv:2308.00436*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188*.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. 2022. Self-critiquing models for assisting human evaluators. *arXiv preprint arXiv:2206.05802*.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. 2022. Big-bench-hard/cot-prompts. `https://github.com/suzgunmirac/BIG-Bench-Hard/tree/main/cot-prompts`. Accessed: 2023-10-31.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *ICLR 2023*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

A Dataset details

Our dataset, BIG-Bench Mistake, is available at [redacted] under the Apache License 2.0. The five tasks used in our dataset are based on BIG-Bench (Srivastava et al., 2022), also released under the Apache License 2.0. All five tasks are in the English language.

A.1 3-shot CoT prompting to generate traces for BIG-Bench Mistake

We use PaLM 2 (Unicorn) to generate the traces used in BIG-Bench Mistake. All traces are generated at temperature $= 0$.

Our prompts and examples can be found at [redacted]. Our prompts are based on chain-of-thought prompts in the BIG-Bench Hard dataset (Suzgun et al., 2022), with four main changes:

1. Example CoT traces in the prompt is broken up into smaller steps (typically one sentence per step). This is done so that mistake location information is more precise.

2. Following Yao et al. (2022), each step in the prompt is signposted with "Thought 1", "Thought 2:", etc. This allows us to refer to the number of the step when prompting for mistake location.

3. For the logical deduction task, we find that the notation used in the original prompt with question marks is often inconsistent. It becomes difficult for annotators to determine whether a question mark is a mistake or not, because the correctness of the question mark is dependent on its interpretation. To minimise such ambiguity, the question mark notation is rewritten into text descriptions of the objects.

4. For the multistep arithmetic task, one of the prompt examples is altered to increase the length of the equation. This is because the BIG-Bench Hard dataset (where the prompts are taken from) only used equations of a specific length, but our dataset contains equations of averaged a variety of lengths, in accordance with the original BIG-Bench dataset (Srivastava et al., 2022).

Following Yao et al. (2022), we use the newline as the stop token, which generates one step with every generation call. We algorithmically append "Thought N:" before each step. This allows us to split up steps in a clear and systematic way. We stop generating once an answer is reached, which is detected using the following regex:
`(?<=[Tt]he answer is).*$`

A.2 3-shot prompting to identify mistakes in BIG-Bench Mistake

As described in section 3, we explore three different methods of prompting for mistake location: direct trace-level prompting, direct step-level prompting, and CoT step-level prompting. We use 3-shot prompting for all methods, and our prompts and examples can be found at [redacted].

Our prompts follow OpenAI's chat completion format. All results were obtained with temperature $= 0$ and no stop tokens.

B Annotation

We release our annotation guidelines at [redacted]. Our annotators are recruited via our institution and contracted at the market rate in their country of residence.

During annotation of the multistep arithmetic task, we found that the first CoT step given in the original BIG-Bench Hard prompt examples (Suzgun et al., 2022) was incorrect. Since all generated traces contained the same first step, we removed that step before showing traces to the annotators.

Figure 3 contains an example screenshot of the user interface. For every trace, we provide the input question as well as the target answer, with a note to be aware of errors that may occur in correct$_{ans}$ traces.

Annotators can click on words to highlight the same word across the trace and the question text, which we found was particularly helpful for some tasks such as word sorting and tracking shuffled objects. Buttons on the right automatically become inactive if a previous step has been labelled as negative.

C Reward model training

To train our reward models (see subsection 4.3), we fine-tune PaLM 2 XS Otter on 4 of our 5 tasks, holding out one task for evaluation. This is done for each of our 5 tasks.

All 5 models are fine-tuned for 20k steps with a batch size of 32. The learning rate is $1e^{-5}$ with a linear ramp and cosine decay. After 20k steps, we select the checkpoint with the best validation results. The number of steps trained for each model are shown in Table 8.

Held-out task	**Training steps**
Word sorting	6800
Tracking shuffled objects	8000
Logical deduction	9000
Multi-step arithmetic	10000
Dyck languages	10000

Table 8: Number of training steps to fine-tune each reward model.

All models are trained as a binary classifier on whether a CoT step is correct, given the task and previous steps. Due to the limited data, we include in training the CoT steps that occur after the first mistake step. These steps are considered incorrect for the purposes of training (despite not being human-annotated as such).

D User interface

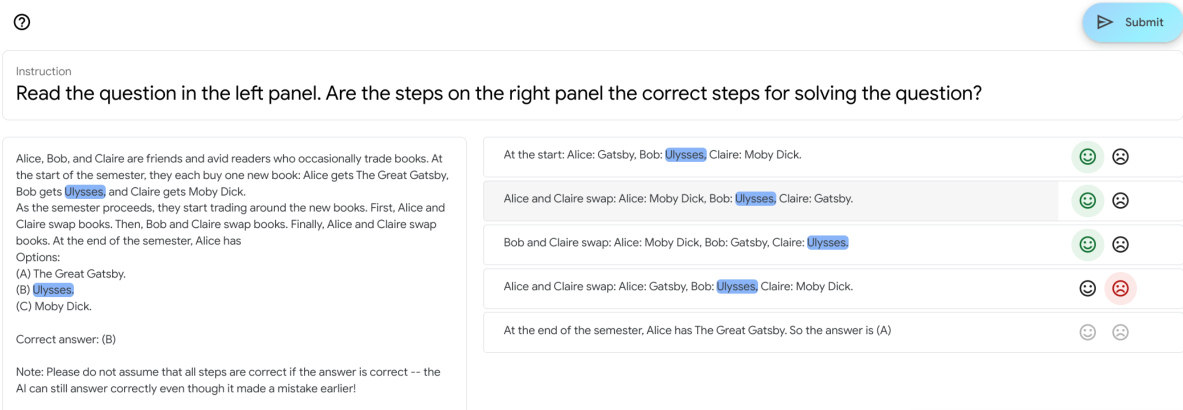

Figure 3: Screenshot of the user interface for a question from the tracking shuffled objects task.

E Benchmark scores

Model	Direct (trace)	Direct (step)	CoT (step)
Word sorting			
GPT-4-Turbo	67.74	38.24	–
GPT-4	88.24	82.35	58.82
GPT-3.5-Turbo	100.00	97.06	20.59
Gemini Pro	44.12	–	–
PaLM 2 Unicorn	100.00	73.53	35.29
Tracking shuffled objects			
GPT-4-Turbo	90.00	77.50	–
GPT-4	82.50	82.50	80.00
GPT-3.5-Turbo	67.50	0.00	0.00
Gemini Pro	12.50	–	–
PaLM 2 Unicorn	100.00	85.00	47.50
Logical deduction			
GPT-4-Turbo	100.00	83.33	–
GPT-4	100.00	100.00	0.00
GPT-3.5-Turbo	100.00	50.00	100.00
Gemini Pro	33.33	–	–
PaLM 2 Unicorn	100.00	100.00	50.00
Multistep arithmetic			
GPT-4-Turbo	57.69	40.32	–
GPT-4	53.23	46.77	27.42
GPT-3.5-Turbo	96.77	79.03	58.06
Gemini Pro	83.87	–	–
PaLM 2 Unicorn	83.87	93.55	29.03
Dyck languages			
GPT-4-Turbo	96.42	30.00	–
GPT-4	98.41	78.57	13.79
GPT-3.5-Turbo	95.74	4.76	0.00
Gemini Pro	0.00	–	–
PaLM 2 Unicorn	100.00	80.95	19.05

(a) Mistake finding accuracy for traces that do not contain mistakes (correct$_{mis}$).

Model	Direct (trace)	Direct (step)	CoT (step)
Word sorting			
GPT-4-Turbo	32.71	32.33	–
GPT-4	28.20	39.47	30.83
GPT-3.5-Turbo	0.00	4.51	15.04
Gemini Pro	6.39	–	–
PaLM 2 Unicorn	0.38	9.02	11.28
Tracking shuffled objects			
GPT-4-Turbo	31.54	59.23	–
GPT-4	59.14	62.69	92.31
GPT-3.5-Turbo	1.17	1.92	21.92
Gemini Pro	41.54	–	–
PaLM 2 Unicorn	5.38	19.23	56.92
Logical deduction			
GPT-4-Turbo	20.81	74.83	–
GPT-4	39.46	67.01	10.54
GPT-3.5-Turbo	0.00	24.83	7.82
Gemini Pro	8.16	–	–
PaLM 2 Unicorn	4.76	36.73	11.22
Multistep arithmetic			
GPT-4-Turbo	34.27	44.12	–
GPT-4	41.60	41.60	44.54
GPT-3.5-Turbo	0.00	12.18	16.81
Gemini Pro	5.46	–	–
PaLM 2 Unicorn	5.88	2.94	22.27
Dyck languages			
GPT-4-Turbo	6.99	28.46	–
GPT-4	7.37	40.81	43.91
GPT-3.5-Turbo	1.28	6.05	2.08
Gemini Pro	2.25	–	–
PaLM 2 Unicorn	0.38	6.43	17.77

(b) Mistake finding accuracy for traces that contain mistakes (incorrect$_{mis}$).

Table 9: Mistake finding accuracy across 5 tasks for correct$_{mis}$ and incorrect$_{mis}$ traces. The combined scores of Table 9a and Table 9b make up Table 4.