When is Tree Search Useful for LLM Planning? It Depends on the Discriminator

Anonymous ACL submission

Abstract

In this paper, we examine how large language models (LLMs) solve multi-step problems under a language agent framework with three components: a generator, a discriminator, and a planning method. We investigate the practical utility of two advanced planning methods, iterative correction and tree search. We present a comprehensive analysis of how discrimination accuracy affects the overall performance of agents when using these two methods or a simpler method, re-ranking. Experiments on two tasks, text-to-SQL parsing and mathematical reasoning, show that: (1) advanced planning methods demand discriminators with at least 90% accuracy to achieve significant improvements over re-ranking; (2) current LLMs' discrimination abilities have not met the needs of advanced planning methods to achieve such improvements; (3) with LLM-based discriminators, advanced planning methods may not adequately balance accuracy and efficiency. For example, compared to the other two methods, tree search is at least 10–20 times slower but leads to negligible performance gains, which hinders its real-world applications.[1]

1 Introduction

Planning plays a crucial role in intelligent behaviors of human and AI agents. Since the early stage of AI research, various methods have been proposed to build agents that can plan efficiently and accurately (Newell and Simon, 1956; Russell and Norvig, 2010). The problem-solving procedure in these AI agents usually involves three steps: searching for possible action sequences, predicting their expected outcomes with an internal world model, and finding an action sequence to achieve the best expected outcome (Russell and Norvig, 2010; Mattar and Lengyel, 2022). This procedure shares common traits with how large language models (LLMs) solve multi-step tasks, including mathematical reasoning (Wei et al., 2022), multi-hop question answering (Yao et al., 2023b), and code generation (Yang et al., 2023). At each step, an LLM searches for possible next actions and generates their language representations (*generation*). To evaluate the actions, the LLM utilizes itself or another LLM to predict the outcomes of actions, in the form of rewards or correctness (*discrimination*). Afterwards, it incorporates the outcomes into its problem-solving process with some strategy to find the best action sequence (*planning*).

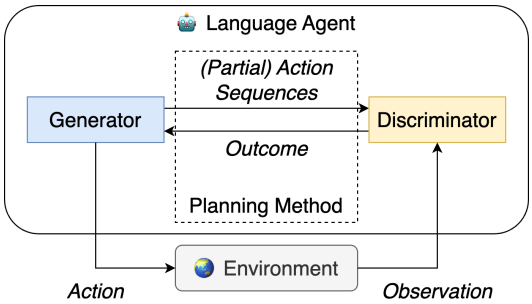


Figure 1: A generator-discriminator framework of language agents, where planning methods control the interaction between a generator and a discriminator, both of which are usually instantiated by some LLM.

Motivated by the similarity, we critically examine how LLMs solve multi-step tasks from a language-agent view. We unify different problem-solving procedures of LLMs into an agent framework (Figure 1) consisting of a generator, a discriminator, and a planning method. Under this framework, we investigate the practical utility of more advanced planning methods, such as tree search, in comparison with simpler methods (e.g. re-ranking). We hypothesize that the discriminator may be a deciding factor and systematically investigate two research questions: **(RQ1)** *How does discrimination accuracy affect the performance of language agents using different planning methods?* **(RQ2)** *Can LLM-based discriminators correctly assess language agents' actions in practical settings?*

[1] Code and data will be released online.

To this end, we analyze LLMs' discrimination abilities and their impact on three categories of planning methods: re-ranking, iterative correction, and tree search. We comprehensively evaluate these methods on two real-world tasks, text-to-SQL parsing and mathematical reasoning, with open-source, closed-source, and fine-tuned LLM discriminators. First, we use oracle environmental information to simulate discriminators with different levels of accuracy. The simulation experiments exhibit a strong correlation between discrimination accuracy and overall task performance among all three types of planning methods. Then, in a non-oracle setting, we closely investigate the LLM-based discriminators and show how environmental observations can effectively improve them. Finally, we conduct end-to-end evaluations of the discriminators and planning methods to verify and strengthen our findings. In summary, our experiments show that:

(1) Advanced planning methods, i.e., iterative correction and tree search, demand highly accurate discriminators ($\geq 90\%$ accuracy) to achieve decent improvements over the simpler method, re-ranking.

(2) Using environmental feedback, we improve the discrimination accuracy of LLMs by up to 30.2 and 8.4 absolute points on text-to-SQL parsing and mathematical reasoning, respectively. Yet, our end-to-end evaluations suggest they have barely met the need for advanced planning methods to show significant improvements over re-ranking.

(3) Meanwhile, advanced planning methods may not adequately balance accuracy and efficiency when using LLM-based discriminators. In our experiments, compared to the other two methods, tree search is at least 10–20 times slower but leads to negligible performance gains. This accuracy-efficiency trade-off can impede the deployment of tree search in real-world applications.

2 Related Work

A lot of recent research efforts have focused on advanced planning methods for improving the multi-step problem-solving abilities of LLMs (Li et al. 2023b; Madaan et al. 2023; Wang et al. 2023b; Yao et al. 2023a,b; Zhou et al. 2023, *inter alia*). Despite different designs, all these methods use a discriminator to evaluate the agents' actions, or planning steps. In fact, instead of planning methods, an agent's discriminator could be the more critical component. Since incorrect outcome predictions could lead to suboptimal plans, discriminators may decide the performance of an agent, regardless of its planning method (Mattar and Lengyel, 2022).

While it is commonly believed that discrimination is easier than generation for human and AI agents (Gu et al., 2023), West et al. (2024) pose the hypothesis that state-of-the-art generative AI models, including LLMs, may not have discrimination abilities matching their generation abilities. This hypothesis coincides with the findings of Huang et al. (2024) and Wang et al. (2023a) that, without any external feedback or with obviously absurd feedback, LLMs may recognize some of their self-generated correct plans as wrong. Huang et al. (2024) also note that the performance gains of self-correction, a kind of iterative correction method, may rely on some high-quality external feedback, such as checking ground-truth labels or test sets for planning loop termination. However, such external feedback usually does not exist in practical applications because solutions to new problems are unknown, and annotating comprehensive test cases can be nontrivial and costly.

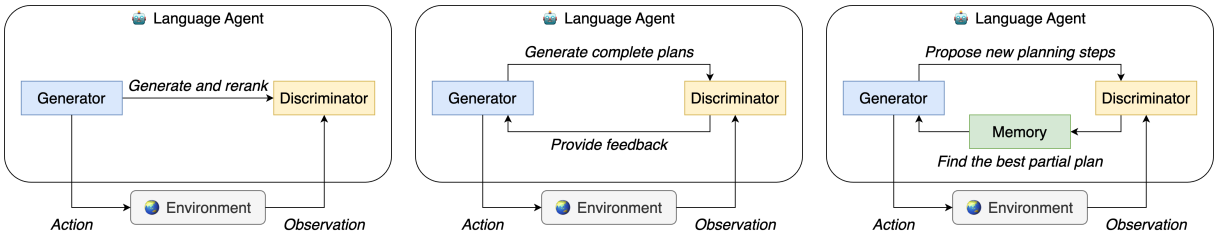
Distinct from these existing studies, our work focuses on studying the relationship between discriminators and planning methods, including but not limited to self-correction, and attempts to improve LLMs' discrimination capability. Our findings can provide useful guidelines for choosing planning methods and implementing language agents in practice. In light of our findings, we encourage future research to thoroughly evaluate language agents with various practical, non-oracle discriminators. We also advocate that improving LLM-based discriminators is an important future direction to enhance agents' accuracy and efficiency when using advanced planning methods.

3 Our Framework

As shown in Figure 1, we systematically analyze different planning methods in a unified generator-discriminator framework. Our framework consists of a generator that proposes (partial) action sequences, a discriminator that evaluates the outcomes of these actions, and a planning method that ranks the actions according to their outcomes and manages the interaction between the two models. In this section, we describe each of the three components and how they are instantiated on text-to-SQL parsing and mathematical reasoning (Section 4.1).

3.1 Generator

For each planning step, we prompt the generator to sample action sequences (SQL queries or Python



(a) Re-ranking. (b) Iterative Correction. (c) Tree Search.

Figure 2: Illustration of three categories of planning methods examined in our unified generator-evaluator framework.

programs for math reasoning). For text-to-SQL parsing, we use 1-shot prompting, where the example is retrieved from the training sets using BM25 (Robertson and Zaragoza, 2009). For math reasoning, we use a fixed 2-shot prompt adapted from Ni et al. (2023b). See prompts in Appendix C.

3.2 Discriminator

Given some (partial) action sequences, we formulate the discrimination task as binary question answering (Kadavath et al., 2022; Ke et al., 2023). The discrimination score of each tested example is the probability of "Yes" being generated as the next token. Specifically, we prompt the LLMs with the question "Is the SQL/python program correct given the utterance/problem?" to generate one single token with its probability as the score. With this formulation, we evaluate three types of LLMs in our experiments (Section 4.2). Similar to the generator, we use 1-shot prompting with BM25 retrieval for text-to-SQL parsing and a fixed 2-shot prompt for math reasoning. Details are in Appendix A.

3.3 Planning Methods

Re-ranking. Re-ranking is a straightforward planning method. After sampling a few complete action sequences from the generator, it uses the discriminator to score them and return the highest-scoring plan (Figure 2a). Although simple, it is commonly used for code generation (Ni et al., 2023a) and mathematical reasoning tasks (Wang et al., 2023b; Li et al., 2023b). We consider re-ranking as a baseline planning method for more advanced ones.

Iterative correction. Like re-ranking, iterative correction starts with the generator proposing a complete action sequence. Then it leverages multiple rounds of revision to improve the initial plan based on the discriminator's feedback (Figure 2b). When the generator and the discriminator are the same LLM, it becomes a prevalent planning method, self-correction (Madaan et al., 2023; Shinn et al., 2023; Yao et al., 2023b; Chen et al., 2024).

While some work uses greedy generation, our implementation samples the same number of action sequences as other planning methods for fair comparison. Then, it uses the discriminator to select the best-scoring one for the next round's revision. We allow up to 10 rounds of corrections, with early exiting when the best plan meets a threshold of discrimination score (> 0.99), or the score is not improved for 3 consecutive iterations. For fair comparison, we prompt the generator to revise plans with 0-shot instruction following (Appendix C) instead of few-shot, since in-context examples may introduce additional information.

Tree Search. Tree search is another popular planning method for language agents, such as Monte-Carlo Tree Search (Chaffin et al., 2022), Pangu (Gu et al., 2023), Tree of Thoughts (Yao et al., 2023a), and Language Agent Tree Search (Zhou et al., 2023). It uses a memory structure (e.g., a heap) to store observed partial action sequences and their scores. For each iteration, it prompts the generator for possible next steps for the current best partial plan, calls the discriminator to evaluate the steps, and updates the memory with new plans and scores (Figure 2c). Our tree search implementation is a kind of MCTS (Zhang et al., 2023):

(1) *Selection*: Find the highest scoring partial plan in the memory, implemented as a heap structure.

(2) *Expansion*: Prompt the generator for the next step of this partial plan. We follow recent work to define a step to be a SQL clause (Chen et al., 2023c) or one line of Python code (Bui et al., 2022), which is semantically more meaningful.

(3) *Simulation*: Reuse the generator to complete the partial plans as Monte-Carlo simulations.

(4) *Evaluation*: Evaluate the simulations with the discriminator. The score for each new step is the maximum score of all simulations starting from it.

(5) *Backpropagation*: Update the partial plan with the new step and score (if higher) and insert them into the heap memory. After the update, if there is a complete plan in the heap memory, we terminate the tree search and return this plan.

4 Experimental Setup

4.1 Tasks and Datasets

Text-to-SQL Parsing. Text-to-SQL parsing is a code generation task of mapping natural language utterances to SQL queries. It requires agents to ground utterances to database environment and generate multi-step plans as SQL queries, making it an appropriate testbed in our study. To evaluate language agents' potential for text-to-SQL parsing, we adapt two widely used datasets, Spider (Yu et al., 2018) and Bird (Li et al., 2023a).

We use the entire training split in each dataset to prompt or fine-tune LLMs.[2] For evaluation, due to resource and budget constraints, we randomly select 400 and 300 development set examples in Spider and Bird, respectively. We also note that model performance may be lower on our evaluation sets because we uniformly sampled examples from each difficulty level, while the original development sets have skewed distributions towards easier examples (Appendix A.1).

Mathematical Reasoning. Mathematical reasoning is a common task for evaluating language agents' multi-step reasoning and planning capabilities. With 500 random examples from GSM8K's development set (Cobbe et al., 2021), we follow program of thoughts (Chen et al., 2023b) to test the agents' ability to plan in Python programs and solve these grade school math word problems.

4.2 Models

In all experiments, we use CodeLlama-13B-Instruct as the generator in our framework. We also evaluate three kinds of LLMs as the discriminator: **(1)** *open-source LLMs*: CodeLlama-7B-Instruct and CodeLlama-13B-Instruct (Rozière et al., 2024), **(2)** *closed-source LLMs*: GPT-3.5-Turbo (OpenAI, 2022) and GPT-4-Turbo (OpenAI, 2023), and **(3)** *fine-tuned LLMs*: CodeLlama-7B-Instruct-FT and CodeLlama-13B-Instruct-FT. Their implementation details are in Appendix A.3. For brevity, we will omit "Instruct" in model names.

4.3 Evaluation

Intrinsic Evaluation. We measure the discrimination abilities of LLMs with four intrinsic metrics. **(1)** Discrimination accuracy (**Acc**): Given a pair of correct and wrong programs, we calculate the percentage where the correct program obtains a higher discrimination score than the wrong one (Bai et al., 2022; Touvron et al., 2023). **(2)** Classification macro F1 (**F1**): We treat "correct" and "wrong" as two classes and compute the macro average of F1 scores on these two labels. **(3)** Hit@1 (**H@1**): Given a batch of candidate programs, we calculate the percentage where the highest scoring candidate is correct. **(4)** Mean reciprocal rank (**MRR**): We compute the standard MRR score by the highest-ranking correct program in the batches.

End-to-End Evaluation. To show the impact of discriminators, we evaluate language agents' end-to-end performance using our three planning methods, with execution accuracy for text-to-SQL parsing and answer accuracy for math reasoning.

5 Simulation Experiments with Oracle

5.1 Oracle-Based Discriminator

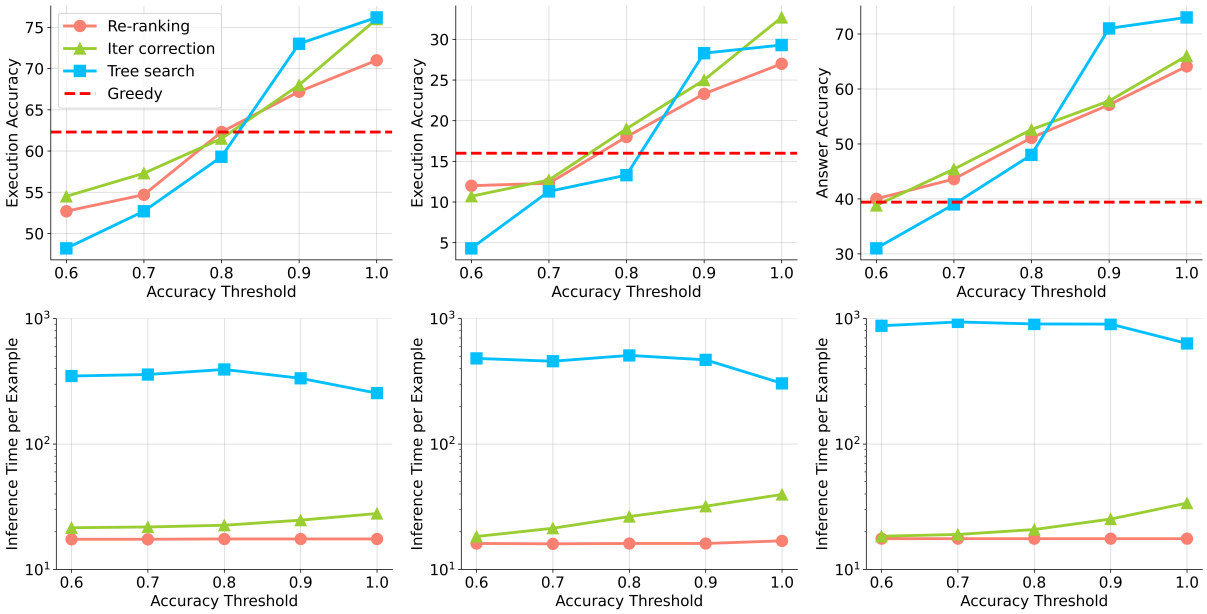
To investigate how discrimination accuracy affects the overall performance of language agents using different planning methods (*RQ1*), we utilize oracle environmental feedback to simulate a discriminator with controllable accuracy. For text-to-SQL parsing, we compare the first five rows in the execution results of predicted and gold SQL queries and calculate their table cell overlaps (Appendix A.4). For mathematical reasoning, we compare the predicted Python programs' answers with the ground truth.

We use a probability-based threshold τ to control the accuracy of each simulated discriminator (Gao et al., 2022). When evaluating each plan, the discriminator first computes a score s with oracle information. Then, it uses a random function to generate a number $p \in [0, 1)$. If $p < \tau$, the discriminator returns the score s. Otherwise, it returns an inverted score $1 - s$. In this way, we ensure that the discriminator's accuracy is at most τ.

5.2 Results and Analysis

As shown in Figure 3, discrimination accuracy closely correlates with the performance of agents on all three datasets, no matter which planning method is used. For instance, the performance of re-ranking agents improves linearly as we increase the discrimination accuracy threshold, setting up a strong baseline for agents using other planning methods. We also note that it takes around 80% discrimination accuracy for all agents to outperform greedy generation on text-to-SQL parsing, demonstrating the task's difficulty. To answer *RQ1*, we further analyze the performance of agents using

[2] In Bird, we exclude training examples for one database, `retail_world`, due to annotation errors.



(a) Spider. (b) Bird. (c) GSM8K.

Figure 3: End-to-end evaluation results (the first row) and average inference time in log scale (the second row) of our simulation experiments with oracle.

iterative correction and tree search as follows:

Advanced planning methods demand highly accurate discriminators. For iterative correction agents, their performance usually cannot distinguish from the re-ranking baselines until we maximize the threshold $\tau = 1.0$ (Figure 3). This finding resonates with Huang et al. (2024) that high-quality feedback may be the key to the success of iterative correction. More interestingly, tree search agents consistently underperform the other two when the discrimination accuracy threshold $\tau \leq 0.8$. Moreover, when raising the threshold to 0.9, we observe a sharp increase of their performance, with which they start to beat other kinds of agents.

Advanced planning methods may not adequately balance accuracy and efficiency. By calculating the average inference time per example (Figure 3), we find that our implementation of tree search is at least 10–20 times slower than the other two planning methods, mainly due to frequent generation of Monte-Carlo simulations (Zhang et al., 2023). While we can remove the simulations to be more efficient and evaluate partial plans, in our preliminary study, we find LLMs would struggle in this setting. This accuracy-efficiency trade-off may hinder real-world applications of tree search methods. Meanwhile, the inference time for iterative correction increases as the accuracy threshold is raised, suggesting more iterations are required to derive a correct answer. This indicates that developing efficient and accurate planning methods remains a key problem for AI agents.

Monte-Carlo tree search can be unstable, especially in the early stages. We observe that iterative correction outperforms tree search on Bird (Figure 3b) when the accuracy threshold is 1.0. This observation may be caused by the instability of Monte-Carlo tree search. We first note that McNemar's test finds no difference between iterative correction and tree search ($p > 0.05$), despite their performance gap (29.3 vs 32.7). The rationales are discussed in Appendix B. Furthermore, we analyze all 25 examples of which iterative correction derives the correct answer but tree search fails. In 12 out of the 25 examples (48%), tree search fails to select the correct partial plan when the discrimination scores are the same. Especially, this can happen in the early stages of tree search, where a correct program has not yet been discovered and all the steps receive a score of 0 from the oracle discriminator. Thus, we consider this underperformance a consequence of search instability.

6 LLM-Based Discriminators

While we have shown that iterative correction and tree search work well with oracle discriminators, it remains unclear whether LLM-based discriminators can correctly assess language agents' actions (*RQ2*). To answer this question, we leverage generator outputs in the simulation experiments and re-label them with ground-truths to evaluate the LLMs' discrimination accuracy (Appendix A.2).

Models	Spider				Bird				GSM8K‡			
	Acc	F1	H@1	MRR	Acc	F1	H@1	MRR	Acc	F1	H@1	MRR
CodeLlama-7B	54.0	37.1	56.0	62.3	44.6	**46.7**	13.0	18.0	48.6	**38.7**	36.2	46.9
CodeLlama-13B	58.2	37.1	57.0	63.1	49.4	**46.7**	12.7	18.3	**62.2**	**38.7**	**41.8**	**51.0**
CodeLlama-7B-FT	62.4	60.3	59.5	64.6	52.4	**46.7**	14.3	19.1	-	-	-	-
CodeLlama-13B-FT	**69.7**	**67.2**	**61.3**	**65.7**	**62.1**	**46.7**	**16.0**	**20.5**	-	-	-	-
GPT-3.5-Turbo	67.0	47.3	59.0	64.3	64.3	35.7	16.0	20.5	72.1	49.1	46.6	54.0
GPT-4-Turbo	**76.5**	**54.9**	**63.0**	**66.7**	**76.2**	**50.1**	**20.3**	**23.0**	**93.8**	**91.1**	**59.8**	**61.6**

Table 1: Intrinsic evaluation results of naive LLMs' discrimination abilities. The **best performance** is in bold for open-source and closed-source LLMs. ‡Since GSM8K's training set does not have program of thoughts annotated for fine-tuning, we have only evaluated the models with in-context learning.

	CodeLlama-13B			GPT-3.5-Turbo			CodeLlama-13B-FT	
	Spider	Bird	GSM8K	Spider	Bird	GSM8K	Spider	Bird
Naive Discriminator	58.2	49.4	62.2	67.0	64.3	72.1	69.7	62.1
+ Executability Check	78.7	78.8	64.5	84.8	86.3	73.2	83.6	82.2
++ Execution Result	**83.6**	**79.6**	**70.6**	**90.0**	**89.2**	**76.5**	**88.5**	**85.1**
Improvement	<u>25.4</u>	<u>30.2</u>	<u>8.4</u>	23.0	24.9	4.4	18.8	23.0

Table 2: Discrimination accuracy of observation-enhanced LLMs. The **best performance** (in bold) is achieved using both kinds of environmental observations. We also underline the <u>largest improvement</u> for each dataset.

6.1 Naive Discriminators

As Table 1 shows, most open-source LLMs have mediocre discrimination abilities. After fine-tuning, CodeLlama-13B-FT could reach the same level of performance as GPT-3.5. In comparison, closed-source LLMs exhibit stronger discrimination abilities, with GPT-4 achieving the best performance across all three datasets. Although GPT-4 has 93.8 discrimination accuracy on GSM8K and is also better than GPT-3.5 on text-to-SQL parsing, due to its high cost, we will use GPT-3.5 in our experiments.

6.2 Observation-Enhanced Discriminators

To improve LLMs' discrimination abilities, we conduct an error analysis for CodeLlama-13B on its worst-performing intrinsic evaluation set, Bird. We sample 50 pairs of SQL queries from the Bird intrinsic evaluation set with incorrect predictions. In 25 of the 50 pairs (50%), CodeLlama-13B assigns a higher score to non-executable SQL queries. Consequently, no matter using which planning method, language agents could hardly perform well with such discriminators.

Motivated by our error analysis, we first propose to add a program executability check as a safeguard for LLMs. If a program is non-executable, our discriminator would discard LLMs' score and return 0. Otherwise, it returns the original LLM score. Besides executability check, we incorporate the execution results of predicted programs (first 5 table rows of SQL queries or answer of Python program) into the in-context examples and fine-tuning data (Ni et al., 2023a). If a program is non-executable, we use `ERROR` to represent its execution result.

Evaluation results (Table 2) show that these two non-oracle environmental observations can effectively improve LLMs' discrimination accuracy. Enhanced with environmental observations, CodeLlama-13B can obtain up to 25.4, 30.2, and 8.4 points absolute accuracy gain on Spider, Bird, and GSM8K, respectively. For the other two models, we also observe significant gains compared to the naive discriminator baseline. Such notable improvements also highlight the importance of filtering out non-executable programs, or invalid plans, during planning.

7 End-to-End Evaluation

While we have evaluated their discrimination abilities with a fixed test set, to answer *RQ2*, we wonder if LLMs can correctly assess constantly changing sets of programs in actual planning processes. To this end, we evaluate the end-to-end performance of language agents with LLM-based discriminators and the three planning methods.

7.1 Text-to-SQL Parsing

As shown in Table 3, agents using naive LLM-based discriminators do not perform well on text-to-SQL parsing. On Spider, the re-ranking agent using CodeLlama-13B-FT has the best accuracy (61.5), which is still lower than greedy generation (62.3) that requires no planning and is more efficient. On Bird, GPT-3.5-Turbo and re-ranking

Discriminators	Spider (Greedy Gen = 62.3)			Bird (Greedy Gen = 16.0)		
	Re-ranking	Iter. Correct.	Tree Search	Re-ranking	Iter. Correct.	Tree Search
CodeLlama-13B	**57.5**	51.7	55.5	**13.3**	**13.3**	**13.3**
GPT-3.5-Turbo	**58.3**	52.7	56.2	**18.0**	17.3	14.0
CodeLlama-13B-FT	**61.5**	51.7	56.0	**14.3**	13.0	13.0
CodeLlama-13B^E	**65.5**	62.0	62.5	21.0	**24.3**	22.7
GPT-3.5-Turbo^E	67.0	**67.5**	66.0	22.3	**25.0**	22.7
CodeLlama-13B-FT^E	**70.3**	68.0	67.5	23.7	**26.3**	21.7
Oracle Simulation ($\tau = 1.0$)	71.0	76.0*	76.2*	27.0	32.7*	29.3

Table 3: End-to-end execution accuracy on text-to-SQL parsing. The **best performance** for each discriminator is in bold. The overall best performance for naive and enhanced discriminators on each dataset is underlined. ^{E}Observation-enhanced discriminators. *Statistically significant ($p < 0.05$; McNemar's) compared to re-ranking with the same discriminator on each dataset. We only observe such improvement with the oracle discriminator.

show an accuracy of 18.0, which is slightly higher than greedy generation (16.0). In addition to the mediocre performance, we find that when using naive discriminators, iterative correction and tree search consistently show worse or the same performance as re-ranking. These results mostly agree with our findings in previous experiments that **(1)** advanced planning methods need strong discriminators, and **(2)** naive LLM-based discriminators are not accurate enough.

After enhancing the discriminators with two environmental observations (Section 6.2), we effectively improve the agents' performance without any modifications to the generator or the planning methods. In 5 of the 6 experiments, CodeLlama-13B-FT^E results in the best execution accuracy among all discriminators. It also leads to the overall best performance on Spider with re-ranking (70.3) and on Bird with iterative correction (26.3), showing the effectiveness of fine-tuning LLMs for discrimination and using environmental observations.

7.2 Mathematical Reasoning

The most interesting result in mathematical reasoning evaluation (Table 4) is the failure of iterative correction with naive discriminators. When prompting the generator CodeLlama-13B for 0-shot correction, it would disregard the instruction to "generate a fixed python program" (Appendix C), copy the program to be modified, and generate explanations and correction steps in natural language. Such natural language steps, usually having some lexical overlap with the math problem, would increase the discrimination score of LLMs while being non-executable. As a result, our iterative correction agent only has 10.2 answer accuracy when using CodeLlama-13B to evaluate its own generation. While this issue also exists when using GPT-3.5-Turbo as the discriminator, it is less severe because GPT would sometimes assign a high score (> 0.99) to the initial Python program. These scores trigger an early exit condition in iterative correction (Section 3.3) and stop the agent from calling the generator to add any natural language, thus avoiding the issue. These findings echo related analysis on self-correction (Stechly et al., 2023; Valmeekam et al., 2023; Huang et al., 2024).

Discriminators	Re-ranking	Iter. Correct.	Tree Search‡
CodeLlama-13B	39.7	10.2	**41.0**
GPT-3.5-Turbo	47.0	37.0	**50.0**
CodeLlama-13B^E	42.8	42.2	**46.0**
GPT-3.5-Turbo^E	47.6	48.4	**51.0**
Oracle Simulation ($\tau = 1.0$)	64.1	66.0	73.0

Table 4: End-to-end answer accuracy on GSM8K (Greedy Gen = 39.4). Notations have the same meaning as in Table 3. McNemar's does not find difference between methods on GSM8K. ‡Tree search is evaluated on 100 randomly selected examples from the 500 evaluation examples due to slow inference speed (Figure 3c). For McNemar's, we compare tree search results with those of re-ranking on the same 100 examples.

With an executability check, enhanced discriminators help mitigate this issue in iterative correction, which now achieves better performance (42.2 and 48.4) than greedy generation (39.4). Overall, the tree search agent using GPT-3.5-Turbo^E achieves the best answer accuracy. Nevertheless, McNemar's test finds no difference ($p > 0.05$) between the performance of re-ranking (47.6) and that of iterative correction (48.4) or tree search (51.0).

7.3 Analysis

To better understand the end-to-end evaluation results, we conduct an in-depth analysis of examples where re-ranking returns the correct program, but

Error Type	Spider		Bird		GSM8K	
	Iter. Correct.	Tree Search	Iter. Correct.	Tree Search	Iter. Correct.	Tree Search
Discrimination	29 (78.4%)	17 (60.7%)	9 (52.9%)	12 (50.0%)	30 (62.5%)	6 (66.7%)
Exploration	8 (21.6%)	11 (39.3%)	8 (47.1%)	12 (50.0%)	18 (37.5%)	3 (33.3%)
Total	37	28	17	24	48	9

Table 5: Error analysis of examples where re-ranking outperforms advanced planning methods. We list the actual number of error cases and their percentages in parenthesis for each dataset and planning method.

iterative correction or tree search does not (Table 5). Specifically, we analyze cases of the strongest discriminators, CodeLlama-13B-FT^E for text-to-SQL parsing and GPT-3.5-Turbo^E for mathematical reasoning, and divide them into two kinds of errors. **(1)** *Discrimination error*: The discriminator assigns a higher score for wrong programs than correct ones, which is not recoverable by any planning method. **(2)** *Exploration error*: The planning method has not found the correct program before termination. Our analysis suggests that:

LLM-based discriminators have not yet met the needs of advanced planning methods. Across all datasets, 50% or more discrimination errors are observed in each planning method. On Spider, the number of such errors in iterative correction is as large as 29 out of 37 (78.4%). In fact, among the 29 errors, iterative correction has already found the correct SQL queries for 15 (40.5% of the total 37 errors) of them. However, not only does the discriminator fail to trigger early exits, but it also assigns a higher score for wrong SQL queries in new iterations. Consequently, these erroneous SQL queries override the originally correct ones, leading to an overall performance drop. The same issue is also serious in tree search. When an incorrect partial program receives a high discrimination score, tree search will commit to it and hardly explore other possibilities, including the correct partial programs. Such discrimination errors usually cannot be recovered by the planning methods themselves, unless they find another correct program with even higher scores. This finding also demonstrates that determining early exits using oracle information in iterative correction may introduce a larger benefit than previously thought (Huang et al., 2024).

Advanced planning methods need more thorough exploration. For the remaining cases, we observe that advanced planning methods have not found a correct program before terminating, which we call exploration errors. This kind of error circles our discussion back to the accuracy-efficiency trade-off mentioned in our simulation experiments with oracle (Section 5.2). Indeed, we can extend the exploration of planning methods in various ways, such as loosening termination conditions, increasing the number of generation samples for each step, and adjusting some hyperparameters for more diverse program samples. Yet, all these adjustments can slow down the planning methods and reduce the language agents' efficiency. Additionally, we note that these strategies may not always result in better performance, as the discriminators may give unseen wrong programs a higher score.

For these reasons, **iterative correction and tree search cannot gain decent improvement** over re-ranking with the same LLM-based discriminator. On text-to-SQL parsing, tree search even shows worse performance than re-ranking when using CodeLlama-13B-FT^E (Table 3: 67.5 vs 70.3 on Spider; 21.7 vs 23.7 on Bird). More surprisingly, on GSM8K, advanced planning methods may not perform much better than re-ranking even with the oracle discriminator ($p > 0.05$; McNemar's). Admittedly, some of the performance gains appear considerable, but McNemar's tells us there are still decent chances of the simpler agent outperforming a more complex one (Appendix B).

8 Conclusions

This paper presents a thorough investigation into the relationship between discrimination accuracy and performance of planning methods in language agents. Through comprehensive experiments on text-to-SQL parsing and mathematical reasoning, we find that: Discrimination accuracy strongly correlates with the overall performance of language agents using different planning methods and also affects their efficiency (*answer to RQ1*). LLM-based discriminators can correctly assess a decent number of language agents' actions with their environmental observations, but they are still not accurate enough for advanced planning methods (*answer to RQ2*). Future research should investigate the development of more accurate discrimination models for language agents, e.g. by improving their grounded understanding of execution results beyond error signals.

Limitations

Experiments with Other Models. In this study, we focus on studying the generation and discrimination of instruction-tuned LLMs that have seen code data during pre-training. This consideration is because: **(a)** They may have better in-context learning performance on our two tasks, text-to-SQL parsing and mathematical reasoning with program-of-thought (Ni et al., 2023b); **(b)** We want to leverage their 0-shot instruction following capabilities in iterative correction for fair comparisons with other planning methods; **(c)** For GSM8K problems, LLMs tend to generate natural language plans instead of programs with 2-shot prompting, and some instructions other than in-context examples help to mitigate this issue. Future research may extend our study to other LLMs of code and conduct an ablation study of instruction-tuning's impact on models' discrimination accuracy.

Experiments with Natural Language Plans. Our study focuses on the generation and discrimination of formal language plans, i.e., programs, as they can directly interact with the environment. Although feasible for mathematical reasoning (Wei et al., 2022), natural language plans require another semantic parsing step to convert them into actions defined in the corresponding environment, which may introduce intermediate errors and add noise to our analysis. Therefore, we conduct the experiments with formal language plans using LLMs trained on code data. As a future direction, it would be interesting to extend our study to natural language plans and see how the intermediate semantic parsing step would affect the overall performance of agents for mathematical reasoning.

Impact of Generators on Planning Methods. While our work focuses on studying the relationship between different discriminators and planning methods, we acknowledge that the generator can also actively affect different planning methods. For example, we can transform the generator's perplexity into a probability and multiply it by the discriminator's score. We exclude such uses of the generator because in our preliminary experiments, we find that incorporating its perplexity leads to mixed results. These results make it even harder to analyze how language agents behave when using different planning methods. Thus, we exclude the generator to have a clear picture of how discriminators can affect planning methods. Nevertheless, it is worth studying the generator's impact on planning methods in future work.

References

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El-Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, Scott Johnston, Shauna Kravec, Liane Lovitt, Neel Nanda, Catherine Olsson, Dario Amodei, Tom Brown, Jack Clark, Sam McCandlish, Chris Olah, Ben Mann, and Jared Kaplan. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback.

Nghi Bui, Yue Wang, and Steven C.H. Hoi. 2022. Detect-localize-repair: A unified framework for learning to debug with CodeT5. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 812–823, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Antoine Chaffin, Vincent Claveau, and Ewa Kijak. 2022. PPL-MCTS: Constrained textual generation through discriminator-guided MCTS decoding. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2953–2967, Seattle, United States. Association for Computational Linguistics.

Shijie Chen, Ziru Chen, Huan Sun, and Yu Su. 2023a. Error detection for text-to-SQL semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 11730–11743, Singapore. Association for Computational Linguistics.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023b. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Ziru Chen, Shijie Chen, Michael White, Raymond Mooney, Ali Payani, Jayanth Srinivasa, Yu Su, and Huan Sun. 2023c. Text-to-SQL error correction with language models of code. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1359–1372, Toronto, Canada. Association for Computational Linguistics.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems.

Allen L. Edwards. 1948. Note on the "correction for continuity" in testing the significance of the difference between correlated proportions. In *Psychometrika*, volume 13, page 185–187.

Ge Gao, Eunsol Choi, and Yoav Artzi. 2022. Simulating bandit learning from user feedback for extractive question answering. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5167–5179, Dublin, Ireland. Association for Computational Linguistics.

Yu Gu, Xiang Deng, and Yu Su. 2023. Don't generate, discriminate: A proposal for grounding language models to real-world environments. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4928–4949, Toronto, Canada. Association for Computational Linguistics.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, Scott Johnston, Sheer El-Showk, Andy Jones, Nelson Elhage, Tristan Hume, Anna Chen, Yuntao Bai, Sam Bowman, Stanislav Fort, Deep Ganguli, Danny Hernandez, Josh Jacobson, Jackson Kernion, Shauna Kravec, Liane Lovitt, Kamal Ndousse, Catherine Olsson, Sam Ringer, Dario Amodei, Tom Brown, Jack Clark, Nicholas Joseph, Ben Mann, Sam McCandlish, Chris Olah, and Jared Kaplan. 2022. Language models (mostly) know what they know.

Pei Ke, Fei Huang, Fei Mi, Yasheng Wang, Qun Liu, Xiaoyan Zhu, and Minlie Huang. 2023. DecompEval: Evaluating generated texts as unsupervised decomposed question answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9676–9691, Toronto, Canada. Association for Computational Linguistics.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023a. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. 2023b. Making language models better reasoners with step-aware verifier. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5315–5333, Toronto, Canada. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Marcelo G. Mattar and Máté Lengyel. 2022. Planning in the brain. *Neuron*, 110(6):914–934.

Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. In *Psychometrika*, volume 12, page 153–157.

A. Newell and H. Simon. 1956. The logic theory machine–a complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. 2023a. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR.

Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, Shafiq Joty, Yingbo Zhou, Dragomir Radev, and Arman Cohan. 2023b. L2ceval: Evaluating language-to-code generation capabilities of large language models.

OpenAI. 2022. Chatgpt.

OpenAI. 2023. Gpt-4 technical report.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

Stuart Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach*, 3 edition. Prentice Hall.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Kaya Stechly, Matthew Marquez, and Subbarao Kambhampati. 2023. GPT-4 doesn't know it's wrong: An analysis of iterative prompting for reasoning problems. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. 2023. Investigating the effectiveness of self-critiquing in LLMs solving planning tasks. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Boshi Wang, Xiang Yue, and Huan Sun. 2023a. Can ChatGPT defend its belief in truth? evaluating LLM reasoning via debate. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 11865–11881, Singapore. Association for Computational Linguistics.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023b. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

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.

Peter West, Ximing Lu, Nouha Dziri, Faeze Brahman, Linjie Li, Jena D. Hwang, Liwei Jiang, Jillian Fisher, Abhilasha Ravichander, Khyathi Chandu, Benjamin Newman, Pang Wei Koh, Allyson Ettinger, and Yejin Choi. 2024. The generative AI paradox: "what it can create, it may not understand". In *The Twelfth International Conference on Learning Representations*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

John Yang, Akshara Prabhakar, Karthik R Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models.

A Implementation Details

A.1 Text-to-SQL Parsing Evaluation Sets

For text-to-SQL parsing, we sub-sample the development splits of each dataset, Spider and Bird, following three steps: **(1)** categorize development set examples by difficulty levels defined in each dataset, **(2)** randomly select a database and choose one associated example, and **(3)** repeat step 2 until we have 100 samples for each difficulty level. In this way, we ensure a uniform distribution across different difficulty levels and database. Since there are four and three difficulty levels in Spider and Bird, respectively, our evaluation sets have 400 and 300 examples for each dataset.

Text-to-SQL parsing models, including LLMs, may show lower performance on our evaluation sets because of their uniformly distributed difficulty (100 examples per level). In comparison, the original datasets have skewed distributions towards easier examples. Spider's development set has 248 (24.0%) examples at easy level and 446 (43.1%) examples at medium level, while the hard and extra hard examples only sum up to 32.9 % of the 1,034 examples. In Bird, 925 out of the 1,534 (60.3%) development set examples are at simple level, 465 examples (30.3%) are at moderate level, and only 144 examples (9.4%) are at challenging level. Our evaluation sets normalize these skewed distributions and make the macro averages of model performance less biased (Section 4.1).

A.2 Intrinsic Evaluation Data

To evaluate LLMs' discrimination performance, we reuse the generation results from our oracle-simulation experiments (Section 6). Specifically, we use the evaluation scripts to re-label the generated programs in simulated re-ranking experiments (accuracy threshold $\tau = 1.0$). Then, we construct our intrinsic evaluation sets based on the relabeled programs (Table A.1). Intuitively, the number of program batches for each dataset is the same as the end-to-end evaluation examples we have, and the the number of programs is all unique programs we can get from the batches. To pair the programs and calculate discrimination accuracy, we iterate through each batch and enumerate combinations of correct and wrong programs within the batch. We do not include cross-batch pairs, as those do not align with our end-to-end evaluation settings.

For discrimination accuracy, we enumerate pairs of correct and wrong programs and ask LLMs to select the better one. For classification F1, we let LLMs predict the correctness of each individual program. For Hit@1 and MRR, we use LLMs to score the batches of programs in simulation experiments.

	Spider	**Bird**	**GSM8K**
Number of Programs	1,221	1,291	2,453
Number of Program Pairs	409	269	1,238
Number of Program Batches	400	300	500

Table A.1: Statistics of our intrinsic evaluation sets.

A.3 Prompting and Training LLMs

Prompting the Generator LM. We prompt our generator LM, CodeLlama-13B-Instruct, with temperature-based sampling for different program suggestions (Section 3.1). We use the model checkpoint and generation function implemented by Hugging Face (Wolf et al., 2020). We set the maximum generation length `max_length` $= 300$, temperature `temperature` $= 0.6$ and number of samples `num_return_sequences` $= 5$.

Data for Discriminator LMs. For text-to-SQL parsing, we perform 2-fold cross-validation on the training sets to synthesize incorrect SQL queries for each example (Chen et al., 2023a). We prompt the LM using one pair of correct and wrong SQL queries (labeled with "Yes" and "No"), also retrieved by BM25 (Section 3.2). Alternatively, we fine-tune the LM on the entire training set with ground-truth and synthesized SQL queries to generate "Yes" or "No." For mathematical reasoning, we annotate two incorrect python programs for the two examples used in generator. Similar to text-to-SQL parsing, we use the two program pairs to prompt LMs for binary question answering. Since the training set of GSM8K is not annotated with program of thoughts, we are not able to fine-tune LMs on this dataset.

Prompting Discriminator LMs. For CodeLlama-7B-Instruct and CodeLlama-13B-Instruct (Section 3.2), we simply feed the input prompt to the models to get the last logit's values, which give us the token-level probability after applying the softmax function.

For GPT-3.5-Turbo and GPT-4-Turbo, we access them through the API of (OpenAI, 2022, 2023). The specific model versions are `gpt-3.5-turbo-1106` and `gpt-4-1106-preview`, respectively. We prompt the LLMs to generate one token and leverage the `top_logprobs` request to

check the top-5 tokens and their probabilities[3]. If "Yes" appears as one of the top-5 tokens, we take its probability p without any modifications. If "Yes" is missing and "No" appears as one of the top-5 tokens, we inverse its probability $1 - p$ as the score. Otherwise, our implementation returns 0 if both tokens are missing, though this case should be rare or even does not happen in our experiments.

Training Discriminator LMs. To get CodeLlama-7B-Instruct-FT and CodeLlama-13B-Instruct-FT (Section 3.2), we again use the checkpoints and trainer implemented by Hugging Face. We fine-tune the models to generate the next token ("Yes" or "No") base on the input prompts using the following hyperparameters:

- Number of epochs: 1
- Batch size: 128
- Learning rate: 1e-5
- Warmup ratio: 3%
- Scheduler: cosine

The inference procedure of fine-tuned models is the same as how we prompt the original LLMs, but without using any in-context example.

Computing Resources. All of our experiments on Spider and GSM8K use up to four NVIDIA RTX A6000 GPU (48GB). Experiments on Bird use up to four NVIDIA A100 Tensor Core GPU (80GB).

A.4 Implemendation of Oracle Discriminator

For text-to-SQL parsing, our oracle uses the first five rows in execution results of the predicted and gold SQL query and calculate the table cell overlap. More specifically, the calculation is similar to span F1 in machine reading comprehension. Our oracle function first compares each row in the execution results head-to-head under a strong assumption that the rows are ordered. Although strict, this assumption is helpful for evaluating the correctness of SQL queries with an `ORDER BY` clause. Then, the function count how many table cells overlap with each other in an unordered manner. We divide the number of overlapping cells by the total number of cells in execution results of the gold SQL query (precision) and the predicted one (recall). Finally, we compute the harmonic mean of these two numbers to get the oracle score (F1).

For instance, given "-- countryid: 1, 2, 4, 5 -- countryname: usa, germany, japan, italy" as the gold execution result and "-- countryid: 1, 4, 6 -- countryname: usa, japan, japan" as the result of predicted SQL query. We compare (1, usa), (4, japan), and (6, japan) the first, second, and third row in the gold result, respectively. They have 2, 0, and 1 overlapping table cells, respectively. Thus, we have our precision to be $3/8 = 0.375$ and recall to be $3/6 = 0.5$. The oracle's score would be:

$$\frac{2 \cdot 0.375 \cdot 0.5}{0.375 + 0.5} = 0.43$$

For mathematical reasoning, our oracle directly checks if the predicted answer equals to the ground-truth. If the answer is `None` (non-executable program) or does not equal to the ground-truth, it returns 0. Otherwise, it returns 1.

B McNemar's Test for Statistical Significance

We measure the statistical significance of performance gains using the exact McNemar's Test[4] (McNemar, 1947). We choose the test's exact binomial version because our sample sizes are relatively small (Edwards, 1948), and the first two significant digits of p-values are the same for this binomial version and the original chi-square version in our tests. Intuitively, this test measures how likely the weaker method can still outperform the stronger one.

For example, we consider the comparison between tree search and iterative correction on Bird when using CodeLlama-13B-FT^E as the discriminator (Section 5.2). By computing a 2×2 contingency table (Table B.1), McNemar's Test focuses on the 40 examples where only one of the two method have predicted correctly. Specifically, there are 25 examples that iterative correction finds the correct answer, but tree search does not, which is the source of performance gain. Also, there are 15 examples that iterative correction fails, but tree search succeeds. According to McNemar's Test, these 15 (37.5% of the total 40) examples result in a p-value of 0.15, meaning there is still some chance for tree search to outperform iterative correction.

In contrast, suppose there are only 10 examples that iterative correction finds the correct answer, but tree search does not. Meanwhile, there are no examples that iterative correction fails, but tree search succeeds. Then, we can still observe the

[3] https://platform.openai.com/docs/api-reference/chat/create#chat-create-top_logprobs

[4] https://www.statsmodels.org/dev/generated/statsmodels.stats.contingency_tables.mcnemar.html

	IC Correct	IC Wrong
TS Correct	73	15
TS Wrong	25	187

Table B.1: Contingency table for tree search (TS) and iterative correction (IC) on Bird, using CodeLlama-13B-FT^E as the discriminator (Section 5.2).

same number of accuracy gain, but it is now statistically different because it is almost impossible for tree search to outperform iterative correction (0 out of 10). The same rationale also applies to the results of other tests in Section 7.

C Prompt Examples

```
Given database schema and a question in natural language, generate the corresponding SQL query.

-- Database climbing:
-- Table mountain: mountain_id, name, height, prominence, range, country
-- Table climber: climber_id, name, country, time, points, mountain_id
-- Question: How many distinct countries are the climbers from?
-- SQL:
SELECT COUNT(DISTINCT country) FROM climber;

-- Database concert_singer:
-- Table stadium: stadium_id, location, name, capacity, highest, lowest, average
-- Table singer: singer_id, name, country, song_name, song_release_year, age, is_male
-- Table concert: concert_id, concert_name, theme, stadium_id, year
-- Table singer_in_concert: concert_id, singer_id
-- Question: What are all distinct countries where singers above age 20 are from?
-- SQL:
SELECT
```

Table C.1: An example prompt for 1-shot generation (text-to-SQL parsing).

```
Given database schema and a question in natural language, correct the buggy SQL query and
generate a fixed SQL query.

-- Database concert_singer:
-- Table stadium: stadium_id, location, name, capacity, highest, lowest, average
-- Table singer: singer_id, name, country, song_name, song_release_year, age, is_male
-- Table concert: concert_id, concert_name, theme, stadium_id, year
-- Table singer_in_concert: concert_id, singer_id
-- Question: What are all distinct countries where singers above age 20 are from?
-- Buggy SQL:
SELECT DISTINCT country FROM singer WHERE age > 20;
-- Fixed SQL:
SELECT
```

Table C.2: An example prompt for 0-shot iterative correction (text-to-SQL parsing).

```
Answer the following Yes/No question: Is the SQL correct given the utterance?

-- Utterance: How many different countries are all the swimmers from?
-- SQL:
SELECT COUNT(DISTINCT nationality) FROM swimmer;
-- Answer: Yes

-- Utterance: How many different countries are all the swimmers from?
-- SQL:
SELECT DISTINCT country FROM swimmer;
-- Answer: No

-- Utterance: What are all distinct countries where singers above age 20 are from?
-- SQL:
SELECT DISTINCT country FROM singer WHERE age > 20;
-- Answer:
```

Table C.3: An example prompt for 1-shot discrimination (text-to-SQL parsing). For discrimination, each in-context example has a pair of correct and wrong programs.

```
Answer the following Yes/No question: Is the SQL correct given the utterance and its result?

-- Utterance: How many different countries are all the swimmers from?
-- SQL:
SELECT COUNT(DISTINCT nationality) FROM swimmer;
-- Result:
-- count(distinct nationality): 7
-- Answer: Yes

-- Utterance: How many different countries are all the swimmers from?
-- SQL:
SELECT DISTINCT country FROM swimmer;
-- Result:
ERROR
-- Answer: No

-- Utterance: What are all distinct countries where singers above age 20 are from?
-- SQL:
SELECT DISTINCT country FROM singer WHERE age > 20;
-- Result:
-- country: Netherlands, United States, France
-- Answer:
```

Table C.4: An example prompt for 1-shot discrimination with execution results (text-to-SQL parsing). For discrimination, each in-context example has a pair of correct and wrong programs.

```
## Given questions in the comment, use python programs to produce the correct answers with
the 'answer' variable.

## James takes 2 Tylenol tablets that are 375 mg each, every 6 hours. How many mg
does he take a day?
## Python Program:
mg_tylenol_per_tablet = 375
mg_tylenol_taken_each_time = 2 * mg_tylenol_per_tablet
hours_per_day = 24
times_per_day = hours_per_day / 6
mg_each_day = mg_tylenol_taken_each_time * times_per_day
answer = mg_each_day

## There were 63 Easter eggs in the yard. Hannah found twice as many as Helen. How
many Easter eggs did Hannah find?
## Python Program:
n_easter_eggs = 63
unit_times = 2
total_units = unit_times + 1
n_easter_eggs_per_unit = n_easter_eggs / total_units
n_easter_eggs_helen = n_easter_eggs_per_unit * 1
n_easter_eggs_hannah = n_easter_eggs_per_unit * 2
answer = n_easter_eggs_hannah

## Gloria is shoe shopping when she comes across a pair of boots that fit her shoe
budget. However, she has to choose between the boots and two pairs of high heels that
together cost five dollars less than the boots. If one pair of heels costs $33 and the other
costs twice as much, how many dollars are the boots?
## Python Program:
```

Table C.5: An example prompt for 2-shot generation (mathematical reasoning).

```
## Given the question in the comment, correct the buggy python program and generate a fixed
python program to produce the correct answer with the 'answer' variable.

## Gloria is shoe shopping when she comes across a pair of boots that fit her shoe
budget. However, she has to choose between the boots and two pairs of high heels that
together cost five dollars less than the boots. If one pair of heels costs $33 and the other
costs twice as much, how many dollars are the boots?
## Buggy Python Program:
price_boots = 50
price_heels = 33
price_heels_twice = 2 * price_heels
price_heels_total = price_heels + price_heels_twice
price_boots_difference = price_boots - price_heels_total
answer = price_boots_difference
## Fixed Python Program:
```

Table C.6: An example prompt for 0-shot iterative correction (mathematical reasoning).

```
## Answer the following Yes/No question: Is the python program correct given the problem in
the comment?

## James takes 2 Tylenol tablets that are 375 mg each, every 6 hours. How many mg
does he take a day?
## Python Program:
mg_tylenol_per_tablet = 375
mg_tylenol_taken_each_time = 2 * mg_tylenol_per_tablet
hours_per_day = 24
times_per_day = hours_per_day / 6
mg_each_day = mg_tylenol_taken_each_time * times_per_day
answer = mg_each_day
## Answer: Yes

## James takes 2 Tylenol tablets that are 375 mg each, every 6 hours. How many mg
does he take a day?
## Python Program:
mg_per_tablet = 375
n_tablets_per_day = 2
n_tablets_per_6hrs = n_tablets_per_day / 6
mg_per_6hrs = mg_per_tablet * n_tablets_per_6hrs
answer = mg_per_6hrs
## Answer: No

## There were 63 Easter eggs in the yard. Hannah found twice as many as Helen. How
many Easter eggs did Hannah find?
## Python Program:
n_easter_eggs = 63
unit_times = 2
total_units = unit_times + 1
n_easter_eggs_per_unit = n_easter_eggs / total_units
n_easter_eggs_helen = n_easter_eggs_per_unit * 1
n_easter_eggs_hannah = n_easter_eggs_per_unit * 2
answer = n_easter_eggs_hannah
## Answer: Yes

## There were 63 Easter eggs in the yard. Hannah found twice as many as Helen. How
many Easter eggs did Hannah find?
## Python Program:
eggs_in_yard = 63
eggs_found_by_hannah = 2 * eggs_in_yard
eggs_found_by_helen = eggs_found_by_hannah / 2
answer = eggs_found_by_hannah
## Answer: No

## Gloria is shoe shopping when she comes across a pair of boots that fit her shoe
budget. However, she has to choose between the boots and two pairs of high heels that
together cost five dollars less than the boots. If one pair of heels costs $33 and the other
costs twice as much, how many dollars are the boots?
## Python Program:
price_boots = 50
price_heels = 33
price_heels_twice = 2 * price_heels
price_heels_total = price_heels + price_heels_twice
price_boots_difference = price_boots - price_heels_total
answer = price_boots_difference
## Answer:
```

Table C.7: An example prompt for 2-shot discrimination (mathematical reasoning). For discrimination, each in-context example has a pair of correct and wrong programs.

```
## Answer the following Yes/No question: Is the python program correct given its result and
the problem in the comment?

## James takes 2 Tylenol tablets that are 375 mg each, every 6 hours. How many mg
does he take a day?
## Python Program:
mg_tylenol_per_tablet = 375
mg_tylenol_taken_each_time = 2 * mg_tylenol_per_tablet
hours_per_day = 24
times_per_day = hours_per_day / 6
mg_each_day = mg_tylenol_taken_each_time * times_per_day
answer = mg_each_day
## Result: 3000.0
## Answer: Yes

## James takes 2 Tylenol tablets that are 375 mg each, every 6 hours. How many mg
does he take a day?
## Python Program:
mg_per_tablet = 375
n_tablets_per_day = 2
n_tablets_per_6hrs = n_tablets_per_day / 6
mg_per_6hrs = mg_per_tablet * n_tablets_per_6hrs
answer = mg_per_6hrs
## Result: 125.0
## Answer: No

## There were 63 Easter eggs in the yard. Hannah found twice as many as Helen. How
many Easter eggs did Hannah find?
## Python Program:
n_easter_eggs = 63
unit_times = 2
total_units = unit_times + 1
n_easter_eggs_per_unit = n_easter_eggs / total_units
n_easter_eggs_helen = n_easter_eggs_per_unit * 1
n_easter_eggs_hannah = n_easter_eggs_per_unit * 2
answer = n_easter_eggs_hannah
## Result: 42
## Answer: Yes

## There were 63 Easter eggs in the yard. Hannah found twice as many as Helen. How
many Easter eggs did Hannah find?
## Python Program:
eggs_in_yard = 63
eggs_found_by_hannah = 2 * eggs_in_yard
eggs_found_by_helen = eggs_found_by_hannah / 2
answer = eggs_found_by_hannah
## Result: 126
## Answer: No

## Gloria is shoe shopping when she comes across a pair of boots that fit her shoe
budget. However, she has to choose between the boots and two pairs of high heels that
together cost five dollars less than the boots. If one pair of heels costs $33 and the other
costs twice as much, how many dollars are the boots?
## Python Program:
price_boots = 50
price_heels = 33
price_heels_twice = 2 * price_heels
price_heels_total = price_heels + price_heels_twice
price_boots_difference = price_boots - price_heels_total
answer = price_boots_difference
## Result: -49
## Answer:
```

Table C.8: An example prompt for 2-shot discrimination with execution results (mathematical reasoning). For discrimination, each in-context example has a pair of correct and wrong programs.