DrPlanner: Diagnosis and Repair of Motion Planners Using Large Language Models

Yuanfei Lin\textsuperscript{1,2}, Chenran Li\textsuperscript{2}, Mingyu Ding\textsuperscript{2}, Masayoshi Tomizuka\textsuperscript{2}, Wei Zhan\textsuperscript{2}, and Matthias Althoff\textsuperscript{1}

Abstract—Motion planners are essential for the safe operation of automated vehicles across various scenarios. However, no motion planning algorithm has achieved perfection in the literature, and improving its performance is often time-consuming and labor-intensive. To tackle the aforementioned issues, we present DrPlanner, the first framework designed to automatically diagnose and repair motion planners using large language models. Initially, we generate a structured description of the planner and its planned trajectories from both natural and programming languages. Leveraging the profound capabilities of large language models in addressing reasoning challenges, our framework returns repaired planners with detailed diagnostic descriptions. Furthermore, the framework advances iteratively with continuous feedback from the evaluation of the repaired outcomes. Our approach is validated using search-based motion planners; experimental results highlight the need of demonstrations in the prompt and the ability of our framework in identifying and rectifying elusive issues effectively.

I. INTRODUCTION

Motion planners for automated vehicles are responsible for computing safe, physically feasible, and comfortable motions [1]. However, to the best of our knowledge, no universal algorithm currently exists that can safely and reliably solve the motion planning problem in all scenarios (see Sec. [A.1]). Therefore, it is crucial to continuously evaluate and enhance the performance of a motion planner during its development. A major challenge is the excessive manual effort required, which entails diagnosing the planner based on a variety of critical test scenarios and evaluation metrics. This process requires not only deep expertise in motion planner functionalities but also a comprehensive understanding of how various aspects of the algorithm correlate with performance. Furthermore, the discrepancy between the design of the algorithm and its practical implementation is another significant factor to consider. To address these challenges, we establish a nuanced framework that leverages the remarkable emergent abilities of large language models (LLMs) [2]–[4]

Fig. 1: An example usage of DrPlanner: In a critical scenario, our imperfect motion planner plans a trajectory. The description of the trajectory and the planner is then fed into DrPlanner. By harnessing the strengths of LLMs in understanding common sense and programming languages, we adeptly diagnose and repair the deficiencies within the planner.

to automatically provide and apply diagnostic solutions for a motion planner, which is shown in Fig. 1

A. Related Work

Subsequently, we review the literature on the imperfections of motion planners, automated software repair, and the application of LLMs in motion planning.

1) Imperfections of Motion Planners: Although many motion planning algorithms can tackle a diverse range of tasks, they often face issues related to probabilistic completeness, computational complexity, or real-time constraints in finding the optimal solution [1], [5], [6]. For example, the authors of [7] examine the solvability of the planning problem using multiple trials and a specified time budget. They demonstrate that both their planner and state-of-the-art alternatives [8], [9] might fail to find a feasible solution, even when one exists. Moreover, most previous studies do not benchmark motion planners across a variety of scenarios, even though they are developed with a few use cases in mind. This lack of comprehensive evaluation makes it challenging to assess and compare the algorithms, let alone improve them. Besides, advanced deep learning algorithms [10]–[12] have been applied over the past decades to handle complex scenarios and learn from experience. However, guaranteeing safety, rule compliance, and social compatibility of motion planners remains a challenge [13]–[15].

2) Automated Software Repair: With the increasing complexity and size of software, automatic debugging and repair techniques have been developed to reduce the extensive manual effort required to fix faults and to improve quality [16].
For instance, human-designed templates are used to repair certain types of bugs in code [17]–[21], but their effectiveness is limited to the hard-coded patterns. To overcome these limitations, deep-learning-based approaches utilize neural machine translation [22] to learn from existing patches, treating the repaired code as a translation of the buggy one [23]–[26]. However, the performance of these approaches is limited by the quality and quantity of the training data as well as its representation format [27]. As LLMs have shown emergent abilities in solving programming tasks [28]–[31], they are applied for generating program patches [32]–[34], self-debugging [35], [36], and cleaning code [37]. Unlike simply maintaining functional equivalence, we aim to both rectify imperfections and boost the performance of the planning algorithms. The aspect of performance improvement aligns with [38], but our work distinguishes itself by focusing on motion planners with a larger codebase. Another branch of work focuses on repairing the outcome of given software [39]–[41] or addressing specified diagnostic criteria [42].

**B. Contributions**

In this work, we introduce DrPlanner, the first framework to autonomously diagnose and repair motion planners, harnessing the power of LLMs that improve as they scale with additional data and model complexity. In particular, our contributions are:

1) establishing a structured and modular description for motion planners across both natural and programming language modalities to exploit the capabilities of LLMs for diagnosis and repair;
2) leveraging the in-context learning capabilities of LLMs by providing demonstrations to the model at the point where it infers diagnostic results;
3) and enhancing the understanding of underlying improvement mechanisms by generating continuous feedback in a closed-loop manner.

The remainder of this work is structured as follows: Sec. II lists necessary preliminaries. The proposed framework for diagnosing and repairing motion planners is described in Sec. III. We demonstrate the benefits of our approach in Sec. IV and conclude the paper in Sec. V.

## II. PRELIMINARIES

### A. Motion Planning

We refer to the vehicle for which trajectories are planned as the *ego vehicle*. As illustrated in Fig. 2, motion planning algorithms are tasked with ensuring that the ego vehicle travels from an initial state to a goal region within a specified time [56]. Additionally, the solution, denoted by $\chi$, must satisfy common and safety-relevant requirements, such as being drivable, collision-free, and rule-compliant [40], [57]. Meanwhile, the motion planner typically minimizes a given objective function $J(\chi)$, e.g., by penalizing the travel time or passenger discomfort [1, Sec. IV]. Finally, we denote a motion planner by $p$ and a motion planning problem by $P$.

### B. Prompt Engineering for LLMs

The technique of using a textual string $\ell$ to condition LLMs for probabilistic predictions is referred to as *prompting* [58]. This approach enables LLMs to be pretrained on a massive amount of data and subsequently adapt to new use cases with few or no labeled data. To enhance the in-context learning capabilities, the prompt may include a few human-annotated examples of the task, known as *few-shot prompting* [2], or utilize chain-of-thought reasoning [43], [59]. We divide the input prompt $\ell$ into two components: the system prompt $\ell_{\text{system}}$, which outlines the task for the LLMs, and the user prompt $\ell_{\text{user}}$, providing context for the diagnostic task. The labels, manual inputs, and automatically generated content within the prompt are marked with angle brackets, square brackets, and curly brackets, respectively. The output consists of both a list of diagnosis-prescription pairs and patched programs, collectively denoted by $\ell_{dp}$ and $p_p$. It is important to note that LLMs typically have a limit on the number of tokens [60] they can process, which essentially means there is a maximum length for the prompt.
The described algorithm is illustrated in the framework in Fig. 3. Afterwards, the obtained patched programs are applied to the motion planner by integrating the modifications into the existing codebase (see line 7). However, it is important to note that the output generated may include errors such as hallucinations and inaccurate analyses [61]. To mitigate these issues, we employ an iterative prompting strategy repeatedly refining the process. The iteration is terminated when a notable improvement in the planner is observed, e.g., when the difference between $J_{\min}$ and a target value $J^*$ is smaller than a threshold $\epsilon \in \mathbb{R}_+$, or when the token limit of the LLM is reached (see lines 5-14). Finally, the repaired planner demonstrating the best improvement, if any, along with the corresponding diagnoses and prescriptions, is returned (see line 15). Another regime is to finetune the LLM to the diagnostic and repair task. However, to date, there exists no dataset containing input-output examples as the cumbersome improvement process of motion planners is typically neither open-sourced nor well documented. Additionally, finetuning...

For a given scenario, the motion planner $M$ is first deployed to address the associated planning problem $P$ (see line 1). Subsequently, the planned trajectory $\chi$ is evaluated against the objective function $J$ (see line 2). Following this, a diagnostic description $\ell_{\text{user}}$ encompassing the diagnostic instructions, the description of the planner, the evaluation of the trajectory, and the few-shot examples are formulated (see line 3). This description, along with the system prompt $\ell_{\text{system}}$, is then fed into the LLM (see line 6). The structure of the input prompt is illustrated in the center of the framework in Fig. 3. Afterwards, the obtained patched programs are applied to the motion planner by integrating the modifications into the existing codebase (see line 7). However, it is important to note that the output generated may include errors such as hallucinations and inaccurate analyses [61]. To mitigate these issues, we employ an iterative prompting strategy repeatedly refining the process. The iteration is terminated when a notable improvement in the planner is observed, e.g., when the difference between $J_{\min}$ and a target value $J^*$ is smaller than a threshold $\epsilon \in \mathbb{R}_+$, or when the token limit of the LLM is reached (see lines 5-14). Finally, the repaired planner demonstrating the best improvement, if any, along with the corresponding diagnoses and prescriptions, is returned (see line 15). Another regime is to finetune the LLM to the diagnostic and repair task. However, to date, there exists no dataset containing input-output examples as the cumbersome improvement process of motion planners is typically neither open-sourced nor well documented. Additionally, finetuning...
usually only provides modest improvement in solving challenging and complex tasks compared to in-context learning [36], [37], [59].

B. Diagnostic Description

As discussed in Sec. II-B, prompt design is challenging, particularly when considering the limited information about the diagnostic objective in the pretrained LLM. To enhance reasoning outcomes, we design a structured and comprehensive description of the motion planner, emulating the process of a real doctor. Its overall skeleton is depicted in the lower part of Fig. 3. As we assume that the motion planner internally handles goal-reaching and drivability-checking of the trajectory in the scenario (cf. Sec. II-A), a detailed description of the scenario, motion planning problem, and trajectory is omitted in the prompt. Alternatively, these tasks can be addressed by additional modules, such as those employing LLM-embedded agents (cf. Sec. I.A.3).

1) Instructions: The instruction provides general guidance for the LLM, detailing the expected output and reasoning constraints. In addition, we can include the commonly used rule-of-thumb from expert knowledge. For instance, “merely adjusting the weighting or coefficients is often cumbersome and not very effective”.

2) Motion Planner: The description of the motion planner begins with the selection and a brief introduction to the planning algorithm. This is followed by a general description of the key components that primarily affect the performance of the planner. To gain a better understanding of how the algorithm is practically implemented, we also include the program of the key components as an additional input modality. As mentioned in Sec. II-A, the LLM is then able to generate repaired programs given corresponding instructions. Motivated by the chain of thought (cf. Sec. II-B), we incorporate existing explanations found within the docstrings of subfunctions to provide natural language summaries for the code blocks. The description adheres to the format of {subfunction name} followed by its {docstring}. For instance, an automatically generated {detailed description} is: “self.calc_angle_to_goal returns the orientation of the goal with respect to current position; ...” (cf. Fig. 6).

3) Planned Trajectory: There are various measures to quantitatively evaluate the quality of the planned trajectory and track its improvement. These measures include the cost function [56], criticality measures [62], courtesy to other traffic participants [63], and degree of traffic rule compliance [40]. To align the LLM with the desired behavior, we present not only the evaluation results for the selected measures but also incorporate the target value \( J^* \), which can be, e.g., sourced from the motion planning benchmark leaderboard. In addition, the numerical data of the values and weights of the objective components is translated into a narrative description by mapping them to their corresponding placeholders.

4) Few-Shots: As it is not necessary for LLMs to have prior knowledge of the other part of the large-scale motion planner, we provide existing helper functions and their exemplary usage in the prompt. Furthermore, several human-annotated examples for improving the performance of the specific type of motion planner can be added here, with examples available in Fig. 5.

C. LLM Querying and Iterative Prompting

When querying the LLM, it is essential to specify the desired output format. To achieve this, one can guide the LLM by emphasizing the diagnoses, prescriptions, and key components of the planner (cf. Sec. II-A) in the prompt as desired responses or employ other third-party tools such as LangChain. Consequently, the structured patched results can directly replace the original elements to repair the planner.

Human diagnosis experts often gauge the effectiveness of prescriptions by analyzing the outcomes of individuals and conducting follow-up consultations after the initial diagnosis. Thus, motivated by [36], [47], [64], [65], we examine the repaired planner by executing it and then pass the evaluation result back to the LLM. In case of compilation or execution errors, the previous diagnostic result is combined with the information indicating where the error occurred and what it entails, as feedback. Otherwise, the combination is made with a comparison of the performance between the updated planned trajectory and the original one.

IV. EVALUATION

We evaluate our approach using the open-source graph-search-based motion planner from the CommonRoad platform [56], which is written in Python. As CommonRoad provides customizable challenges and annual competitions where users can compete against each other on predefined benchmarks, we can continuously integrate enhancements into DrPlanner based on insights from a broad user base. Furthermore, we choose GPT-4-Turbo as our LLM and use its function calling feature to generate structured outputs. The patched programs are then stringified in a JSON object and directly passed to the motion planner, followed by execution through the exec function in Python. The token limit is set to 8,000, the threshold \( \epsilon \) is equal to 10, and we choose the sampling temperature of the LLM at 0.6 (cf. [29, Fig. 5]). Code and exemplary prompts are available at https://github.com/CommonRoad/drplanner

A. A* Search using Motion Primitives

We adapt the standard A* search algorithm using lattice-based graphs [66] (see Fig. 1), which employs a cost function and an estimated cost to the goal, namely, a heuristic function, to guide the search process. The graph is constructed with motion primitives—short trajectories generated offline through a forward simulation of a given vehicle model.

1) Description of the Motion Planner: The heuristic function and motion primitives constitute the key components of the planner. We provide the entire code block of the heuristic function along with descriptions of the involved subfunctions
in natural language. Furthermore, motion primitives are referenced by IDs encoded with configurable parameters:

\[ MP = \text{V}_{\text{min}} \text{V}_{\text{max}} \text{V}_{\text{step}} \Delta \text{V}_{\text{SA}} \delta_{\text{min}} \delta_{\text{max}} \text{SA}_{\text{step}} \Delta \delta \tau \text{Model}_m \]

where \( v_{\text{min}} \) and \( v_{\text{max}} \) are the sampling velocity limits, \( \delta_{\text{min}} \) and \( \delta_{\text{max}} \) are the sampling steering angle bounds, \( \Delta v \) and \( \Delta \delta \) specify their respective step sizes, \( \tau \) is the time duration of each motion primitive, and \( m \) is the model identifier of the ego vehicle. All parameters are given in SI units. In the description, the explanation of the above naming convention is included, followed by the ID of motion primitives used in the original planner.

2) Measures of the Planned Trajectory: To evaluate the quality of the planned trajectory, we utilize the standardized objective function \( J_{\text{SMI}} \) from CommonRoad [56, Sec. VI]:

\[ J_{\text{SMI}}(\chi) = \omega_A J_A + \omega_{SA} J_{SA} + \omega_{SR} J_{SR} \]
\[ + \omega_{LC} J_{LC} + \omega_{O} J_{O} + \omega_{V} J_{V}, \]

where \( \omega_A, \omega_{SA}, \omega_{SR}, \omega_{LC}, \omega_{O}, \) and \( \omega_{V} \) are the weights assigned to the respective objective components, and their values are listed in Tab. I. These components include the cost for acceleration \( J_A \), steering angle \( J_{SA} \), steering rate \( J_{SR} \), distance and orientation offset to the centerline of the road \( J_{LC} \) and \( J_{O} \), and velocity offset \( J_{V} \). For the evaluation scenario in Fig. 4, the target value of \( J_{\text{SMI}} \) is extracted from the CommonRoad benchmark leaderboard\(^4\) and is \( J_{\text{SMI}}^* = 0.16 \).

3) Few-Shots: To gain a deeper insight into the planner, we include method definitions and docstrings for existing helper functions within the planner class. As shown in Fig. 4, we also supply a list of IDs corresponding to offline-generated motion primitives, from which the LLM can select.

B. Case Study

We choose an intersection scenario from the CommonRoad platform (cf. Fig. 4), which is generated by the scenario factory for safety-critical traffic scenarios [62], [67]. In the urban environment, the search-based motion planner is responsible for navigating the ego vehicle from the initial state for 3.3s without colliding with any obstacles. The time increment of the scenario is 0.1s. Moreover, we use the planner with the setup illustrated in Fig. 6. The planned trajectory by the initial planner is shown in Fig. 7 where the ego vehicle brakes and steers slightly to the right, leading to a high value of \( J_{\text{SMI}} \) (cf. Tab. I).

The diagnostic results using our approach are illustrated in Fig. 7b. In the first iteration, the provided helper functions are automatically included in the heuristic function by the LLM (cf. Fig. 7a). Meanwhile, some hyperparameters are adjusted, such as the orientation weight and the heuristic for zero velocity, and coarser motion primitives are applied. Considering all the above factors, the repaired planner results in a decrease in \( J_{\text{SMI}} \) of the planned trajectory, particularly in \( J_A, J_{SA}, J_{SR}, \) and \( J_{O} \) (cf. Tab. I), and leads to the vehicle traveling further forward. In contrast, the diagnostic result from the second iteration leads to a KeyErro (cf. Fig. 7b), indicating that the repaired heuristic function is not provided by the LLM. With the iterative prompting, the error message is incorporated as feedback into the prompt for the third iteration. As shown in Fig. 7c our approach not only helps the LLM avoid the errors from previous iterations (cf. the diagnosis “KeyErro in heuristic function”) but also retains the previous diagnostic results that lead to a positive impact.

### Table I: Comparison of the planned trajectories before and after repair.

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Initial Planner</th>
<th>1. Iteration</th>
<th>2. Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>( J_A )</td>
<td>50</td>
<td>91.7333</td>
<td>14.9333</td>
<td>0.0000</td>
</tr>
<tr>
<td>( J_{SA} )</td>
<td>50</td>
<td>0.0850</td>
<td>0.0102</td>
<td>0.0147</td>
</tr>
<tr>
<td>( J_{SR} )</td>
<td>50</td>
<td>0.2525</td>
<td>0.0968</td>
<td>0.0673</td>
</tr>
<tr>
<td>( J_{LC} )</td>
<td>1</td>
<td>0.3175</td>
<td>0.3504</td>
<td>0.3393</td>
</tr>
<tr>
<td>( J_{O} )</td>
<td>50</td>
<td>0.0614</td>
<td>0.0038</td>
<td>0.0041</td>
</tr>
<tr>
<td>( J_{V} )</td>
<td>20</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

\( J_{\text{SMI}} \) | - | 4606.93 | 752.56 | 4.65 |
on the planner. As a result, the planner significantly improves its performance, with a substantial reduction in $J_{SM1}$ from 752.56 to 4.65, achieved by further balancing the objective components (cf. Tab. II). Moreover, it can be observed from Fig. [7] that DrPlanner can provide fine-grained diagnoses and prescriptions based on both the prompt design and fundamental aspects of programming, such as aliasing (cf. lines 10, 13, 15 in Fig. [7]). The resulting patched programs align precisely with these diagnoses and prescriptions.

C. Performance Evaluation

We further evaluate the performance of DrPlanner using the scenario illustrated in Fig. [4] and 50 A*-search-based motion planners with different setups obtained from the CommonRoad challenges. The evaluation employs the pass@k metric and uses its unbiased version proposed in [29, Sec. 2.1], which is defined as the probability that at least one of the top $k \in \mathbb{N}$, generated code samples for a problem passes the given tests. Here, we use a decrease of $J_{SM1}$ for the returned planner as the criterion for passing. Additionally, we conduct ablation studies to examine the impact of omitting two specific components within the framework: few-shots and feedback. For each study, we execute the framework 10 times to collect solution samples.

Tab. [II] presents the evaluation results for DrPlanner and its variants. Overall, DrPlanner exhibits robust capabilities for diagnosing and repairing motion planners, achieving a pass rate of 98.6% at $k = 10$ and an average reduction of 54.5% in $J_{SM1}$. Note that, similar to the case study in Sec. [IV], the value of $J_{SM1}$ does not converge with the iterations due to diagnostic inaccuracies. However, the average number of iterations required to observe its first decrease is 1.4. Furthermore, the results demonstrate that both the few-shot learning (cf. Sec. III-B.3) and the iterative prompting (cf. Sec. III-C) play crucial roles in enhancing the effectiveness of the planner.

![Fig. 6: Heuristic function and ID of motion primitives used in the initial planner that are to be diagnosed and repaired.](image-url)

```python
def heuristic_function(self, node_current: PriorityNode) -> float:
    path_last = node_current.list_paths[-1]
    angle_to_goal = self.calc_angle_to_goal(path_last[-1])
    orientation_to_goal_diff = self.calc_orientation_diff(angle_to_goal, path_last[-1].orientation)
    cost_time = self.calc_time_cost(path_last)
    if self.reached_goal(node_current.list_paths[-1][-1].time, node_current.list_paths[-1][-1].time_step):
        return 0
    time = self.time
    if self.position_desired is None:
        time = 0.0
    else:
        velocity = node_current.list_paths[-1][-1].velocity
        cost = 20 * orientation_to_goal_diff + 0.5 * cost_time + heur_time
        if np.isclose(velocity, 0):
            cost = 0
        else:
            cost = self.calc_euclidean_distance(node_current.list_paths[-1][-1]) / velocity
            if cost < 0:
                cost = 0
            elif cost > 1e6:
                cost = 1e6
    return cost
```

![Fig. 7: Diagnostic and repair result for the motion planner in Fig. 6. The](image-url)

```python
def heuristic_function(self, node_current: PriorityNode) -> float:
    acceleration_cost = self.calc_acceleration_cost(path_last)
    path_efficiency = self.calc_path_efficiency(path_last)
    steering_angle_cost = self.calc_steering_angle_cost(path_last)
    steering_velocity_cost = self.calc_steering_velocity_cost(path_last)
    if self.reached_goal(node_current.list_paths[-1][-1].time, node_current.list_paths[-1][-1].time_step):
        time = 0.0
    elif self.position_desired is None:
        time = self.time
    else:
        velocity = node_current.list_paths[-1][-1].velocity
        cost = 20 * orientation_to_goal_diff + 0.5 * cost_time + heur_time
        if np.isclose(velocity, 0):
            cost = 0
        else:
            cost = self.calc_euclidean_distance(node_current.list_paths[-1][-1]) / velocity
            if cost < 0:
                cost = 0
            elif cost > 1e6:
                cost = 1e6
    return cost
```

![Fig. 8: Diagnostic and repair result for the motion planner in Fig. 6. The](image-url)

<table>
<thead>
<tr>
<th>Diagnosis</th>
<th>Prescription</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orientation weight too high</td>
<td>Adjust the weight for orientation to goal difference</td>
</tr>
<tr>
<td>Missing acceleration cost</td>
<td>Include acceleration cost in heuristic</td>
</tr>
<tr>
<td>Missing path efficiency</td>
<td>Include path efficiency in heuristic</td>
</tr>
<tr>
<td>Missing steering angle cost</td>
<td>Include steering angle cost in heuristic</td>
</tr>
<tr>
<td>Missing steering velocity cost</td>
<td>Include steering velocity cost in heuristic</td>
</tr>
<tr>
<td>Infinite heuristic for zero velocity</td>
<td>Handle zero velocity case appropriately</td>
</tr>
<tr>
<td>Sparse motion primitives</td>
<td>Recommend motion primitives with higher branching factor</td>
</tr>
</tbody>
</table>

![Fig. 9: Diagnostic and repair result for the motion planner in Fig. 6. The](image-url)
of DrPlanner. In particular, the few-shots prompting is more effective since the LLM is intrinsically unaware of the other supportive components of the planner, e.g., the available motion primitives. Additionally, since the initial planners are not buggy but underperforming, the results without using few-shots show that they cannot be easily improved with only the descriptions of the planner and the planned trajectory.

V. CONCLUSION

We present the first framework for diagnosing and repairing motion planners that leverages both common sense and domain-specific knowledge about causal mechanisms in LLMs. Through a modular and iterative prompt design, our approach automates the generation of descriptions for the planner and continuously enhances diagnostic performance. The major limitation of our approach is that the improvement of the planner cannot be guaranteed. However, as the capabilities of LLMs advance, we anticipate the paradigm to enhance significantly over time. Future work will involve developing datasets by monitoring user submissions over time, specifically focusing on sequences of edits that lead to performance improvements. This effort will examine the impact of different objective functions and their target values. We encourage researchers using DrPlanner to refine their motion planners and contribute towards establishing a large-scale framework that encompasses a variety of planner types for diagnostic and repair tasks.

REFERENCES


Table II: Ablation studies on the design of DrPlanner. Values in bold denote the best performance.

<table>
<thead>
<tr>
<th>Method</th>
<th>pass@k k = 1↑</th>
<th>k = 5↑</th>
<th>k = 10↑</th>
<th>Decrement of JSMI Avg. ↑</th>
<th>Std. Dev. ↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o Few-Shots</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>w/o Feedback</td>
<td>45.4%</td>
<td>86.2%</td>
<td>92.0%</td>
<td>49.0%</td>
<td>36.3%</td>
</tr>
<tr>
<td>DrPlanner</td>
<td>68.0%</td>
<td>95.1%</td>
<td>98.0%</td>
<td>54.5%</td>
<td>34.9%</td>
</tr>
</tbody>
</table>


