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

Yuanfei Lin\(^1\), Student Member, IEEE, Chenran Li\(^2\), Student Member, IEEE, Mingyu Ding\(^2\), Student Member, IEEE, Masayoshi Tomizuka\(^2\), Life Fellow, IEEE, Wei Zhan\(^2\), Member, IEEE, and Matthias Althoff\(^1\), Member, IEEE

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, our framework returns repaired planners with detailed diagnostic descriptions. Furthermore, our framework advances iteratively with continuous feedback from the evaluation of the repaired outcomes. Our approach is validated using both search- and sampling-based motion planners for automated vehicles; experimental results highlight the need for demonstrations in the prompt and show the ability of our framework to effectively identify and rectify elusive issues.

Index Terms—Integrated planning and learning, motion and path planning, intelligent transportation systems, large language models, automated software repair.

I. INTRODUCTION

Motion planners for automated vehicles are responsible for computing safe, physically feasible, and comfortable motions [1]. A major challenge is the excessive manual effort required to tune motion planners, which entails diagnosing the planner based on a variety of critical test scenarios and evaluation metrics. To address this, we establish a framework that leverages the remarkable emergent abilities of large language models (LLMs) [2]–[4] to automatically provide and apply diagnostic solutions for a motion planner of automated vehicles, as illustrated in Fig. 1

Manuscript received: March 12, 2024; Revised June 08, 2024; Accepted July 24, 2024. This paper was recommended for publication by Editor H. Kurniawati upon evaluation of the Associate Editor and Reviewers’ comments. The authors gratefully acknowledge partial financial support by the German Federal Ministry for Digital and Transport (BMDV) for the project KoSi, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) for the project SFB 1608, under Grant 501798263, and by the Berkeley DeepDrive. The work was developed during Y. Lin’s visit to the University of California, Berkeley. (Corresponding author: Wei Zhan.)

1 Y. Lin and M. Althoff are with the School of Computer, Information and Technology, Technical University of Munich, 85748 Garching, Germany. (e-mail: yuanfei.lin@tum.de; althoff@tum.de).
2 C. Li, M. Ding, M. Tomizuka, and W. Zhan are with the Department of Mechanical Engineering, University of California, Berkeley, CA 94720, USA. (e-mail: chenran_li@berkeley.edu; myding@berkeley.edu; tomizuka@berkeley.edu; wzh@berkeley.edu).

Digital Object Identifier (DOI): see top of this page.

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
the focus on performance improvement with [36], our work uniquely addresses the challenges posed by the larger and more intricate codebases of motion planners. Another branch of work focuses on repairing the outcome of given software [37]–[39] or addressing specified diagnostic criteria [40].

2) Language Models for Motion Planning: With their indispensable role of common sense reasoning and generalization [41]–[43], LLMs have been applied in motion planning for autonomous driving to make high-level decisions [44]–[48].

III. DrPlanner

This section presents our prompt engineering with a nuanced diagnostic description. We begin by introducing the overall algorithm, followed by a more detailed presentation.

A. Overall Algorithm

A general overview of using DrPlanner is presented in Fig. 3 and Alg. 1. Before initiating the process, the user fills in the placeholders enclosed in square brackets. 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 using 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 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 [59]. 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 the current best performance $\min$ 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
However, to date, there exists no open-source dataset containing input-output examples of motion planners. Additionally, finetuning usually only provides modest improvements in solving challenging and complex tasks compared to in-context learning [34, 35, 57]. Regardless of the approach, when deploying the repaired planners on roads, a safety layer is always required [12].

### B. Diagnostic Description

As discussed in Sec. II-B, prompt design is challenging, particularly when considering the limited information about the diagnostic object in the pretrained LLM. To enhance conclusions, 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 states is omitted in the prompt. Alternatively, these tasks can be addressed by additional modules, such as those employing LLM-embedded agents (cf. Sec. II-B).

#### 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

---

**Algorithm 1** DiagnoseAndRepairPlanner

**Input:** planning problem \( P \), motion planner \( M \), target value \( J^* \), system prompt \( \ell_{\text{system}} \), LLM

**Output:** diagnoses and prescriptions \( \ell_{dp} \), repaired planner \( M_{\text{rep}} \)

1. \( \chi \leftarrow M.PLAN(P) \)
2. \( J \leftarrow \text{EVALUATE}(\chi) \)
3. \( \ell_{\text{user}} \leftarrow \text{DESCRIBE}(M, J, J^*) \)  
\( \triangleright \text{Sec. III-B} \)
4. \( J_{\text{min}} \leftarrow J, \ell_{dp} \leftarrow \emptyset, M_{\text{rep}} \leftarrow \emptyset \)
5. \( \text{while not REACTOKENLIMIT}(\text{LLM}) \) and \( J_{\text{min}} - J^* > \epsilon \) \( \text{do} \)
6. \( (\ell_{dp}, p_{dp}) \leftarrow \text{LLM.QUERY}(\ell_{\text{system}}, \ell_{\text{user}}) \)  
\( \triangleright \text{Sec. III-C} \)
7. \( M_{\text{rep}} \leftarrow \text{REPAIR}(M, p_{dp}) \)
8. \( \chi \leftarrow M_{\text{rep}}.PLAN(P) \)
9. \( J_{\text{rep}} \leftarrow \text{EVALUATE}(\chi) \)
10. \( \ell_{\text{user}} \leftarrow \text{ADDFEEDBACK}(\ell_{\text{user}}, J, J_{\text{rep}}, \ell_{dp}) \)  
\( \triangleright \text{Sec. III-C} \)
11. \( \text{if } J_{\text{rep}} < J_{\text{min}} \) \( \text{then} \)
12. \( J_{\text{min}} \leftarrow J_{\text{rep}}, \ell_{dp} \leftarrow \ell_{dp}, M_{\text{rep}} \leftarrow M_{\text{rep}} \)
13. \( \text{end if} \)
14. \( \text{end while} \)
15. \( \text{return} \ (\ell_{dp}, M_{\text{rep}}) \)
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 code of the key components as an additional input modality. As mentioned in Sec. II-A1 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 planned trajectory and track its improvement. These measures include the cost function [54], criticality measures [60], courtesy to other traffic participants [61], and degree of traffic rule compliance [11], [38]. 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. 4

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.

Motivated by how LLMs are utilized in improving technical systems [34], [45], [62], [63], 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. 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 motion planners from the CommonRoad platform [54], which are 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\(^2\) as our LLM and use its function calling feature to generate structured outputs. It should be noted that our framework is not limited to GPT-4-Turbo and can be easily adapted for use with other LLMs by modifying the interface. The patched programs are then stringified in a JSON object and directly parsed 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. [26] Fig. 5). Code and exemplary prompts are available at https://github.com/CommonRoad/drplanner

A. Setup

1) Search-Based Motion Planner\(^3\) We adapt the anytime A* search algorithm using lattice-based graphs [64]. This implementation features a time-limited search cut-off and 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. The number of explored nodes in the graph is denoted as \( N_n \). Motion primitives are typically referenced by IDs encoded with configurable parameters:\(^4\)

\[
\text{MP} = "V_{\text{min}} - V_{\text{max}} \_Vstep_ \_\Delta v_ \_\delta_{\text{min}} - \delta_{\text{max}} - S\_A\_\delta_{\text{Model}} - T_\tau - m " ,
\]

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. Therefore, the heuristic function and motion primitives constitute the key components. We provide the entire code block of the heuristic function along with descriptions of the involved subfunctions in natural language. In the description of motion primitives, the explanation includes the naming convention, followed by their ID.

2) Sampling-Based Motion Planner\(^5\) Similarly, we evaluate our approach on the sampling-based motion planner of [65], which computes jerk-optimal trajectories using polynomials to connect sampled end states with the initial state. From the set of feasible trajectory samples, the optimal trajectory is selected based on a cost function. Consequently, the cost function and sampling configurations, such as the sampling time horizon \( t_s \), are the key components.

---

\(^2\)ID gpt-4-turbo-preview in the API of OpenAI.

\(^3\)https://commonroad.in.tum.de/tools/commonroad-search

\(^4\)All parameters are given in SI units.

\(^5\)https://commonroad.in.tum.de/tools/commonroad-reactive-planner
There are some pre-defined helper functions that can be directly called in the heuristic function:
def calc_acceleration_cost(self, path: List[KSState]) -> float:
    """returns the acceleration costs."""

Examples:
(input)
def heuristic_function(self, node_current: PriorityNode) -> float:
    ... cost = angle_to_goal
    return cost
(output)

Diagnosis: the acceleration is not considered
Prescription: add the acceleration cost to the heuristic function
def heuristic_function(self, node_current: PriorityNode) -> float:
    acceleration_cost =
        self.calc_acceleration_cost(node_current.list_paths[-1])
    ... cost = angle_to_goal + acceleration_cost
    return cost

Feasible motion primitives with the same name format that you can directly use:
"V_0.0_20.0_Vstep_1.0_SA_-1.066_1.066_SAstep_2.13_T_0.5_Model_BMW_320i",
...
allows the vehicle to travel further forward with fewer explored nodes in the search graph due to the coarser motion primitives applied (see Tab. I and Fig. 5a). In contrast, the diagnostic result from the second iteration leads to a **KeyError** (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 “**KeyError in heuristic function**”) but also retains the previous diagnostic results that lead to a positive impact on the planner. As a result, the planner significantly improves its performance, with a substantial reduction in $J_{SMI}$ from 752.56 to 4.65, achieved by further balancing the objective components (cf. Tab. I). Moreover, it can be observed from Fig. 7d 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. 7c). The resulting patched programs align precisely with these diagnoses and prescriptions.

The initial configuration snippet of the sampling-based planner is shown in Fig. 6a. A similar repair pattern to the search-based planner can be observed in Tab. I and Fig. 5b. For brevity, we only show the diagnostic details for the third iteration in Fig. 9 which achieves the best performance among all iterations. The cost function improves through weight tuning and adding more items, and a larger $t_s$ is selected, leading to a noticeable reduction in $J_{SMI}$.

### C. Performance Evaluation

We further evaluate the performance of DrPlanner by analyzing 50 randomly selected critical CommonRoad scenarios, along with 50 A*-search-based motion planners in various setups from the CommonRoad challenges. The former is benchmarked against the search-based planner configured as shown in Fig. 6a. The latter evaluation utilizes the scenario illustrated in Fig. 5 and employs the pass@k metric. We use its unbiased version as proposed in [26 Sec. 2.1], 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 $J_{SMI}$ for the returned planner as the criterion for passing. As a baseline, the performance of DrPlanner is compared with a genetic approach [14], where the program of the heuristic function is repaired to minimize $J_{SMI}$. The solution space consists of 10 chromosomes, and the process runs for 100 generations. Additionally, we conduct ablation studies to examine the impact of omitting two specific components within the framework across different planners: few-shots and feedback. For each study, we execute the framework 10 times to collect solution samples.

Fig. 8 and Tab. I present the results of the performance evaluation. Overall, DrPlanner effectively diagnoses and repairs motion planners under various setups, outperforming the baseline approach in all metrics, with a pass rate of 98.0% at $k = 10$ and an average reduction of 54.5% in $J_{SMI}$. The benchmark results in Fig. 8 further indicate robust performance across diverse scenarios, with an average $J_{SMI}$ decrease of

```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]):
        heur_time = 1e6 # A large but not infinite cost
    else:
        heur_time = self.time_desired.start - path_last[-1].time_step

    cost = 20 * orientationToGoalDiff + 0.5 * cost_time + heur_time
    steering_velocity_cost = self.calc_steering_velocity_cost(path_last)
    velocity = node_current.list_paths[-1][-1].velocity
    if self.position_desired is None:
        return 0
    cost = 10 * orientationToGoalDiff + 0.5 * cost_time + steering_angle_cost + steering_velocity_cost + acceleration_cost + path_efficiency + steering_angle_cost + steering_velocity_cost

    return cost
```

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

The identical program patches in the first and third iteration are

**Diagnosis**

Orientation weight too high
- Missing acceleration cost
- Missing path efficiency
- Missing steering angle cost
- Missing steering velocity cost
- Infinite heuristic for zero velocity
- Sparse motion primitives

**Prescription**

Adjust the weight for orientation to goal difference
- Include acceleration cost in heuristic
- Include path efficiency in heuristic
- Include steering angle cost in heuristic
- Include steering velocity cost in heuristic
- Handle zero velocity case appropriately
- Recommend motion primitives with higher branching factor

![Fig. 8: Benchmarked reduction of $J_{SMI}$ across scenarios using DrPlanner.](image)

For better visibility, outliers in the box plot are not shown.
Fig. 9: Diagnostic and repair result of the third iteration for the sampling-based motion planner in Fig. [20].

Table II: Performance evaluation and ablation studies across planners on the design of DrPlanner. Values in bold denote the best performance.

<table>
<thead>
<tr>
<th>Method</th>
<th>pass@k</th>
<th>k = 1↑</th>
<th>k = 5↑</th>
<th>k = 10↑</th>
<th>Decrement of JSM1 Avg. ↑ Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genetic [14]</td>
<td>0.8%</td>
<td>0.4%</td>
<td>7.7%</td>
<td>0.1%</td>
<td>1.6%</td>
</tr>
<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>49.6%</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>

REFERENCES


ACKNOWLEDGMENT

The authors kindly thank Sebastian Iliing for implementing the experiments for the sampling-based planner.