Teaching LLMs to Plan with Executable Value Iteration Code

Milan Gautam
gautammi@oregonstate.edu

Eric Wang
wanger@oregonstate.edu

Charishma Khandapu
khandapc@oregonstate.edu

Oregon State University
Corvallis, OR

Aayam Shrestha
shrestaa@oregonstate.edu

Abstract

In this work we explore integration of Large Language Models with exactly planning frameworks like value iteration in Markov Decision Processes(MDPs). We propose finetuning LLMs to frame planning problems as MDPs and and write executable value iteration code for deriving the optimal solution to the MDP. This planning by executable planning code is simple yet powerful and may enable LLMs to plan effectively in different environments.We fine-tune Code-Llama-Instruct 7B model to output an executable Python script that frames maze problems as MDPs and includes solver code. We demonstrate that the finetuned model significantly outperforms the baseline in finding correct solutions and exhibits generalization capabilities to out-of-distribution maze sizes and structures. Overall, this work highlights the potential of integrating LLMs with exact planning frameworks to enhance their problem-solving abilities and generalization to unseen scenarios.

I. Introduction

Large Language Models (LLMs) have demonstrated impressive linguistic and knowledge retrieval capabilities, leading many to investigate their potential for planning and reasoning tasks. However, LLMs by themselves lack the systematic reasoning and correctness guarantees needed for robust planning [1]. Rather than ascribing unwarranted planning abilities to LLMs, the proposed LLM-Modulo Framework suggests leveraging LLMs as approximate knowledge sources and candidate generators that work in conjunction with sound external planning systems.

Building on this perspective, we explore integrating LLMs with exact planning frameworks like value iteration in Markov Decision Processes (MDPs). Specifically, we propose fine-tuning LLMs to frame planning problems as MDPs and generate executable value iteration code to derive provably optimal solutions. This planning-by-code approach is conceptually simple yet powerful, enhancing the ability of LLMs to effectively plan in various environments while inheriting the theoretical guarantees of the underlying planning algorithms.

To investigate this idea, we fine-tune the Code-Llama-Instruct 7B model to output executable Python scripts that formulate maze problems as MDPs and include solver code based on value iteration. Our experiments demonstrate that the fine-tuned model significantly outperforms the baseline in finding correct solutions. Moreover, it exhibits promising generalization to out-of-distribution maze sizes and structures not seen during training.

The key contributions of this work are: (1) A novel approach for integrating LLMs with exact planning algorithms by fine-tuning them to generate executable planning code and (2) Experimental results showing this method enables LLMs to effectively solve maze planning problems and generalize to unseen scenarios.

Overall, this work highlights the potential synergies between the broad knowledge of LLMs and the rigorous reasoning of symbolic planning frameworks. By having LLMs generate problem formulations for algorithms like value iteration, we can enhance their planning capabilities while sidestepping their limitations in autonomous reasoning. This points to a promising neuro-symbolic direction where LLMs are thoughtfully combined with traditional

AI planning techniques to tackle complex reasoning challenges.

II. Preliminaries

A. Large Language Models for Planning

Recent years have seen growing interest in leveraging the knowledge and generative capabilities of Large Language Models (LLMs) for planning tasks. Several works have explored using LLMs to directly generate plans from natural language problem descriptions. For example, [2] proposed an "Inner Monologue" approach where LLMs engage in self-talk to reason about plans. [3] uses an external environment snd self-reflection to improve reasoning and decision-making. However, the ability of LLMs to reliably generate correct plans remains questionable. [4] conducted a systematic evaluation of various LLMs on benchmark planning problems, finding that generated plans are often flawed due to the lack of sound reasoning capabilities. Moreover, the plan generation is largely dependent on the specific prompts and in-context examples provided. These limitations have spurred efforts to integrate LLMs with dedicated planning systems that can provide guarantees on plan validity.

B. Markov Decision Processes and Value Iteration

Markov Decision Processes (MDPs) provide a principled mathematical framework for sequential decision making under uncertainty [5]. An MDP is defined by a tuple $< S, A, T, T, \lambda >$ where S is the state space, A is the action space, T specifies the transition dynamics, R defines the rewards, and λ is a discount factor. The goal is to find an optimal policy $\pi*$ that maximizes the expected cumulative discounted reward. Value Iteration (Bellman, 1957) is a fundamental dynamic programming algorithm for solving MDPs. It computes the optimal value function V* via an iterative update:

$$V_{t+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) V_t(s') \right]$$

The optimal policy can then be derived as:

$$\pi^(s) = \underset{a}{argmax} \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) V^(s') \right]$$

Value Iteration provides a theoretically sound approach for finding optimal policies in finite MDPs. However, its effectiveness is limited by the need to explicitly specify the MDP components, which can be challenging for complex real-world problems.

C. Fine-Tuning and Low Rank Adaptation

Fine-tuning is a process where a pre-trained model is further trained on a given dataset from which it adapts a knowledge to perform particular tasks or the assigned tasks. Fine-tuning involves adjusting the weights and parameters on the pre-trained model with our own data; this enables the model to make accurate predictions customized for the new task. We fine-tuned LLaMA with our own dataset. We used Supervised Fine-tuning (where the model is trained on a task-specific dataset) with LoRA (Low-Rank Adaptation) [6] to make our fine-tuning process more efficient.

LoRA (Low-Rank Adaptation) reduces the number of trainable parameters which helps in a significant reduction in the GPU memory requirements. LoRA has been found to be efficient even with fewer trainable parameters compared to other fine-tuning methods. [7]

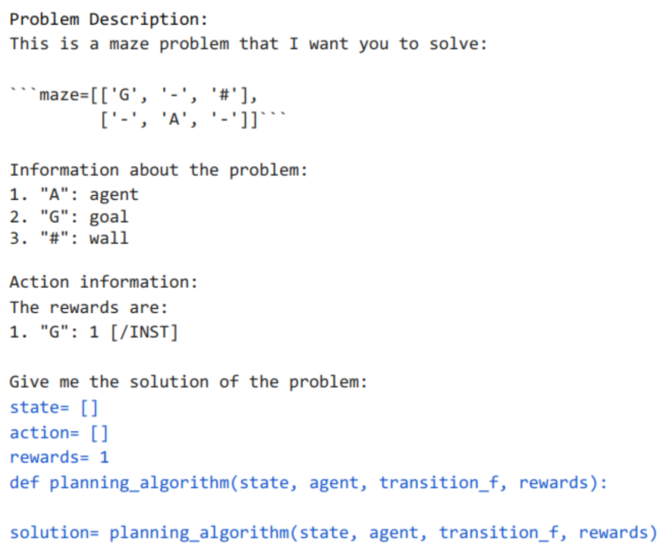
D. LLM-Modulo Planning

[1] proposed the LLM-Modulo framework as a principled way to integrate LLMs with symbolic planning systems. The key idea is to use LLMs as generators of planning problem formulations that are then solved by sound external solvers. This modular architecture allows leveraging the broad knowledge of LLMs while ensuring plan correctness through the use of dedicated planning algorithms.

Several recent works align with the LLM-Modulo paradigm. [8] introduced LLM+P which uses LLMs to map natural language problems to PDDL specifications that are then solved by classical planners. [9] takes a similar approach, using LLM outputs to construct planning problems for the Fast Downward planner. [10] showed how LLMs can help acquire approximate symbolic world models that enable the use of model-based planners. In the context of MDPs, [11] used LLMs to generate reward functions from natural language that are then combined with a learned world model to solve the resulting MDP. Our work builds on these efforts by investigating the use of LLMs to generate complete MDP specifications along with executable value iteration code. This extends the LLM-Modulo framework to the stochastic sequential decision making setting while preserving optimality guarantees.

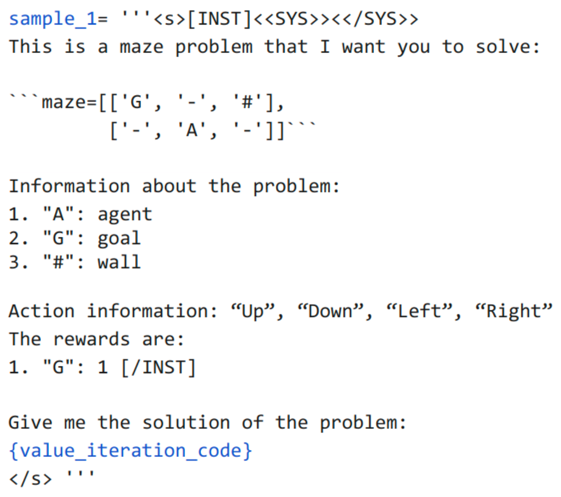
III. Problem Statement and Methodology

To train our LLM to recognize MDP problems and implement value iteration, we constructed our own dataset of training examples. For each training example we included a textual user prompt that describes the maze problem and a Python response that implements the solution. We generated a total of 1000 training examples programmatically, randomizing different configurations for the problem statement. Pseudocode for a training example is shown in 1



~~~
Problem Description:
This is a maze problem that I want you to solve:

```maze=[['G', '-', '#'],
        ['-', 'A', '-']]```

Information about the problem:
1. "A": agent
2. "G": goal
3. "#": wall

Action information:
The rewards are:
1. "G": 1 [/INST]

Give me the solution of the problem:
state= []
action= []
rewards= 1
def planning_algorithm(state, agent, transition_f, rewards):

solution= planning_algorithm(state, agent, transition_f, rewards)
~~~

Figure 1. Example maze problem and the solution



~~~
sample_1= '''<s>[INST]<<SYS>><</SYS>>
This is a maze problem that I want you to solve:

```maze=[['G', '-', '#'],
         ['-', 'A', '-']]```

Information about the problem:
1. "A": agent
2. "G": goal
3. "#": wall

Action information: “Up”, “Down”, “Left”, “Right”
The rewards are:
1. "G": 1 [/INST]

Give me the solution of the problem:
{value_iteration_code}
</s> '''
~~~

Figure 2. Example maze problem and the solution

IV. Methodology

We developed a large language model (LLM) specifically focused on code generation and tailored it to solve navigation problems involving agents in an environment. The model’s objective is to process descriptive inputs about scenarios with an agent whose task is navigating to a designated goal. From these descriptions, the LLM constructs a solution comprising two main components:

1. Markov Decision Process (MDP) Representation: The model first formulates the problem as an MDP. This involves defining the states, actions, transition probabilities, and rewards based on the environmental dynamics and the agent’s interactions described in the input.

2. Value Iteration Planner: Utilizing the MDP framework, the model employs a value iteration algorithm to compute an optimal policy. This iterative method evaluates the expected utility of actions at each state, generating a transition matrix. The matrix encapsulates the strategy that guides the agent’s decisions to reach its goal efficiently.

A. Problem Statement

In this work, we selected the maze problem as a planning problem that we want to solve. The LLM will receive a) information about the maze’s environment, i.e., the position of the Agent, the Goal, and obstacles, and b) the set of actions that an agent can perform in a state. Figure 2 shows an example of data used for supervised fine-tuning of the LLM. Here we can see the maze of size 2*3, which has ”A”, ”G”, and ”#” as agents, goals and obstacles, respectively. Similarly, ”Up”, ”Down”, ”Left”, and ”Right” are the actions that an agent can perform in a state.

To train our LLM to recognize maze problems and implement value iteration, we constructed our own dataset of training examples. For each training example we included a textual user prompt that describes the maze problem and a Python response that implements the solution. We generated a total of 1000 training examples programmatically, randomizing different configurations for the problem statement.

B. Training Data

*1) **Problem Configurations**:* In the training examples, we varied the maze dimensions as well as obstacle placement. We also varied the character symbols used to describe the maze by creating 10 different possible configurations for maze descriptions. Each configuration comes with its own reward value and action set. The action set always includes the basic directions: up, down, left and right. Alternative actions sets include diagonal directions or 2-jumps. Here are 4 example configurations:

```
{
    'space_rover': {
    'agent': 'R',
    'goal': 'P',
    'wall': 'O',
    'empty': ".",
    'actions': [(0,1), (1,0), (0,-1), (-1,0)],
    'reward': 3
  },
  'worm_in_garden': {
    'agent': 'R',
    'goal': 'P',
    'wall': 'O',
    'empty': ".",
    'actions': [(0,1), (1,0), (0,-1), (-1,0)],
    'reward': 3
  },
  'desert_trek': {
    'agent': 'T',
    'goal': 'O',
```

```
    'wall': '@',
    'empty': ".",
    'actions': [(0,1), (1,0), (0,-1), (-1,0)],
    'reward': 2
  },
}
```

When generating each training example, a random number was chosen between 9 and 14 for the width and another for the height. One of the 10 symbol configurations was then chosen at random to determine which character symbols to designate the maze. The four boundaries are set as walls. Then $width \times height \times 0.2$ walls are placed at random positions in the maze to act Finally, the agent is positioned at (1, 1) and the goal at $(width - 2, height - 2)$.

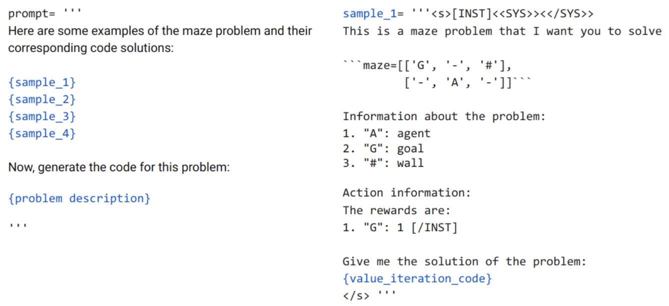
*2) **Solutions**:* The agent response examples are complete python scripts that implement solutions to the problems described in the prompt. The solution includes two bindings in the local scope: `maze` and `value_iteration`. The first is an exact copy of the python maze from the problem and the second is a function implementation of value iteration on that maze. Applying the value iteration function on `maze` returns a value iteration matrix assigning an expected value to each state in the maze.

C. Model Architecture

The LLama2 model, initially pre-trained on billions of tokens sourced from the internet, is a versatile but not task-specific model. Given its generalist nature, adapting LLama2 for code generation tasks would necessitate substantial computational resources and extensive coding datasets. This challenge led us to select the recently unveiled CodeLLama Instruct. This variant begins with the foundational LLama2 architecture and undergoes further specialization, being trained on 500 billion tokens of coding data. Additionally, it receives training on an extra 5 billion tokens comprising instructions and code to enhance its ability to generate code from natural language instructions. We implemented two popular approaches to explore its capabilities: i. prompt tuning and ii—supervised fine-tuning. We selected the 7 Billion variant of CodeLLama Instruct for all experiments, aiming to optimize the balance between performance and resource efficiency.

D. Experiment approach

*1) **Prompt tuning**:* Given that the CodeLLama Instruct model was specifically fine-tuned for code generation tasks, we hypothesized that it would be inherently well-suited to address our maze-solving planning problem. This assumption led us to believe that with strategic prompt tuning, we could further optimize the model's performance to produce accurate and efficient solutions. To this end, we crafted prompts for CodeLLama Instruct, detailing a description of the maze problem. To enhance the quality of the code generation, we also included examples of similar problems and their corresponding solutions based on value iteration code.



```
prompt= '''
Here are some examples of the maze problem and their
corresponding code solutions:

{sample_1}
{sample_2}
{sample_3}
{sample_4}

Now, generate the code for this problem:

{problem description}

'''
```

~~~
sample_1= '''<s>[INST]<<SYS>><</SYS>>
This is a maze problem that I want you to solve

```maze=[['G', '-', '#'],
        ['-', 'A', '-']]```

Information about the problem:
1. "A": agent
2. "G": goal
3. "#": wall

Action information:
The rewards are:
1. "G": 1 [/INST]

Give me the solution of the problem:
{value_iteration_code}
</s> '''
~~~

Figure 3. Prompt Example

In Figure 3, we can see an example of a prompt that we used. For each sample, we have a problem description and the solution. Similarly, we have utilized the prompt format of CodeLLama and included the problem description within the [INST] token.

*2) **Fine tuning**:* Numerous studies have demonstrated that supervised fine-tuning of the Large Language Model (LLM) on domain-specific tasks significantly improves the model's performance. However, due to the extensive number of parameters in LLMs, such as LLama3-base, which has 8 billion parameters, fine-tuning would require a substantial amount of computational resources. To address this issue, we are employing low-rank adaptations (LoRA), a method that modifies only a small fraction of the model's parameters while keeping the rest frozen. Specifically, we're using LoRA with a rank of 65 and an alpha of 50, which has effectively reduced the number of parameters from 8 billion to 33 million. This approach significantly enhances the fine-tuning process's computational efficiency by leveraging the power of LoRA to maintain performance while minimizing the required computational resources.

E. Experimental Setup

1) Model Configuration: Initially, we planned to evaluate the performance of the baseline CodeLLaMa and our new fine-tuned model using a test set different from the training set. We also aimed to assess the robustness of the fine-tuned model against variations in the symbols and action sets. This approach helps determine whether the model is overfitting to the 10 specific configurations used during training. To conduct this evaluation, we created three additional test sets that include new symbols and actions not present in the training data.

2) Evaluation Setup: For the purposes of evaluation, we define two metrics for an LLM's ability to solve mazes with Python code. One is code execution success and the

	Same Actions as Training Set	Changed Actions
Same Symbols as Training Set	Test set #1	Test set #3
Changed Symbols	Test set #2	Test set #4

Table I. The 4 Test Datasets

second is solution accuracy. Code execution success is counted if the LLM generated code output is able to run without any errors from the Python interpreter. Solution accuracy is counted if the path for the agent based on the value iteration matrix is identical to the path based on human generated code

F. Test Data Creation

Initially, we considered simply comparing the baseline CodeLLaMa and our new fine-tuned model based on a newly generated test set that is different from the training set. We also wanted to test to see if our fine-tuned model would be robust against changes in the symbols used and action sets, in case the model was overfitting to only work with the 10 example configurations used for generating the training data. To test this, we generated 3 new test sets to use new symbols and actions not found in the training data.

*1) **Test Set #1:*** Our first test set is 100 examples generated using the same configuration that produced the training data.

ii. Test Set #2: Test set #2 uses a different set of maze configurations to generate the data. Here, new symbols are introduced to represent the agent, goal, walls and empty positions that are not found in the training data at all. The action sets are the same as the training data.

iii. Test Set #3: Test set #3 uses a different set of maze configurations to generate the data. Here, new action sets are introduced that allow the agent to move in ways that are not possible in the original training data. In this test set 3-jumps and L-shaped movements like the way knights move in chess are introduced. The symbols used to represent the maze are the same as the training data.

iv. Test Set #4: Test set #4 uses the new character symbols from test set #2 and the new actions sets from test set #3.

V. Results

The results on all 4 test datasets are summarized in Table II, providing a comprehensive overview of the model's performance under varying conditions.

A. Test Set 1 Analysis

Comparing the original CodeLLaMa performance with our fine-tuned model on data set #1, we can see that our fine-tuning raised the code execution success rate from 81% to 100% and the solution accuracy rate from 62% to 100%. This was surprising to us because we didn't expect LLMs to have such high accuracy for generating complete executable code in the first place and didn't have any familiarity with using CodeLLaMa either. When examining the output for the original CodeLLaMa, one thing we noticed is that about half the time the LLM chose to solve the problem using breadth first search and half the time using value iteration. Sometimes breadth first search actually produces the correct solution, for the passing case, some of those are actually implementations of breadth first search. Failing cases included failing to correctly determine the agent or goal location based on the input maze. Another common failing case is for CodeLLaMa to assume that the agent should start at position (0, 0) when in all of our test cases the agent actually starts at (1, 1).

B. Test Set 2 Analysis

When evaluating our fine-tuned model on test set #2, we saw a code execution success rate of 97% and a solution accuracy of 84%. We see that the code execution success rate remains very high but there is a significant dip in solution accuracy. Examining the 16% of failure cases, we noticed a commonality around misidentifying the agent and goal symbols. The symbols might have been interpreted in reverse where the agent position is interpreted as the goal. This led to a faulty value iteration matrix calculation that led to a failing solution.

C. Test Set 3 Analysis

When evaluating our fine-tuned model on test set #3, we saw a code execution success rate of 100% and a solution accuracy of 99%. While introducing new action sets, we see that there was not a significant impact on accuracy. We theorize that this is because the new action sets did not introduce any foreign symbols the LLM did not recognize. The basic task required is to copy a python snipped from the input text, a skill that transferred to this new test set.

D. Test Set 4 Analysis

Test set #4, combining new symbols and action sets, presented the most complex challenge. The model achieved a 99% code execution success rate and a 92% solution accuracy, underscoring its strong adaptability. However, the slight drop in solution accuracy highlights areas for improvement, particularly in the model's ability to simultaneously process novel symbols and actions within the context of maze navigation.

E. Comparative Analysis and General Observations

In our training data, we represented the agents and states within the Markov Decision Process (MDP) using character symbols. Remarkably, even after altering the state symbols in Test Sets 2 and 4, the model continued

Model	Test Dataset	Size	Code Execution Success	Solution Accuracy
CodeLLaMa baseline	test set 1 (nothing change)	100	81%	62%
Fine-tuned	test set 1 (nothing change)	100	100%	100%
Fine-tuned	test set 2 (symbols changed)	100	97%	84%
Fine-tuned	test set 3 (actions changed)	100	100%	99%
Fine-tuned	test set 4 (both changed)	100	99%	92%

Table II. Test Evaluation of Fine-Tuned CodeLLaMa

to perform well, demonstrating its ability to generalize to novel states and actions. This adaptability was further tested by changing state representations in the test set to words rather than symbols. The generated code outputs were logically and syntactically correct in most cases. However, there were instances where the model inaccurately placed a symbol as a state, leading to erroneous representations of state. These observations lead us to believe that with sufficient exposure to varied state representations, the fine-tuned Large Language Model (LLM) is capable of solving maze problems with any representation of state, showcasing its potential for broad applicability in problem-solving scenarios.

In the training data for our value iteration solution, we set the goal value to 2 whenever an agent reached a goal, maintaining this value consistently across the entire training set. To our surprise, we observed a range of goal values from 2 to 7 in the code solutions across multiple test sets. This observation suggests that the Large Language Model (LLM) has determined the goal value to be independent of the actual solution, allowing for any value greater than zero. We theorized that Code Llama might have encountered value iteration code with varying goal values during its creation. However, this theory is contradicted by the results we observed in prompt tuning, where, even after including samples of value iteration solutions, the model predominantly generated solutions based on the breadth-first search algorithm. Therefore, we currently cannot provide an explanation for this phenomenon.

VI. Conclusion

In conclusion, this whitepaper has showcased the innovative integration of Large Language Models with the Markov Decision Process framework to address the inherent limitations of LLMs in planning and decision-making tasks. By fine-tuning the Code-Llama-Instruct 7B model on a specialized maze problem, we have successfully demonstrated the model's enhanced ability to not only understand and generate MDP frameworks but also to solve complex problems through value iteration code. This approach significantly improved the model's generalization capabilities, enabling it to effectively navigate through problems involving novel states and actions that were not present during its training phase.

Our findings reveal that the fine-tuned model achieved remarkable performance across various test sets, consistently showing high rates of code execution success and solution accuracy. This underscores the model's robustness and its ability to adapt to changes in problem configurations, such as new symbols and action sets. Particularly noteworthy is the model's proficiency in handling different symbols for MDP components, indicating a deep understanding of the problem structure beyond mere symbol recognition.

This work represents a significant step forward in the field of artificial intelligence, bridging the gap between traditional LLMs and structured problem-solving paradigms. By leveraging the strengths of both approaches, we have created a tool that enhances the problem-solving capabilities of LLMs, paving the way for more autonomous and reliable AI systems in the future. The success of this project opens up new avenues for research and application, suggesting that similar integrations could be applied to a wide range of real-world problems, further expanding the utility and impact of LLMs in various domains.

References

[1] S. Kambhampati, K. Valmeekam, L. Guan, K. Stechly, M. Verma, S. Bhambri, L. Saldyt, and A. Murthy, "Llms can't plan, but can help planning in llm-modulo frameworks," *ArXiv*, vol. abs/2402.01817, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267413178

[2] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. R. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman, and B. Ichter, "Inner monologue: Embodied reasoning through planning with language models," in *Conference on Robot Learning*, 2022.

[3] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang, "Language agent tree search unifies reasoning acting and planning in language models," *ArXiv*, vol. abs/2310.04406, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID: 263829963

[4] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati, "On the planning abilities of large language models - a critical investigation," *ArXiv*, vol. abs/2305.15771, 2023.

[5] M. L. Puterman, "Markov decision processes: Discrete stochastic dynamic programming," 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:122678161

[6] H. Hu, A. Singh, Y. Yang, and J. Gao, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[7] A. Aghajanyan, L. Zettlemoyer, S. Gupta, and A. Shrivastava, "Intrinsic dimensionality explains the effectiveness of language model fine-tuning," *arXiv preprint arXiv:2010.03648*, 2020.

[8] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone, "Llm+p: Empowering large language models with optimal planning proficiency," *ArXiv*, vol. abs/2304.11477, 2023.

[9] T. Silver, S. Dan, K. Srinivas, J. B. Tenenbaum, L. P. Kaelbling, and M. Katz, "Generalized planning in pddl domains with pretrained large language models," in *AAAI Conference on Artificial Intelligence*, 2023.

[10] L. Guan, K. Valmeekam, S. Sreedharan, and S. Kambhampati, "Leveraging pre-trained large language models to construct and utilize world models for model-based task planning," *ArXiv*, vol. abs/2305.14909, 2023.

[11] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar, "Eureka: Human-level reward design via coding large language models," *ArXiv*, vol. abs/2310.12931, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:264306288