On Using Large Language Models to Generate Plans

Anonymous authors
Paper under double-blind review

Abstract

Automated planning is concerned with developing efficient algorithms to generate plans or sequences of actions to achieve a specific goal in a given environment. Emerging Large Language Models (LLMs) can answer questions, write high-quality programming code, and predict protein folding, showcasing their versatility in solving various tasks beyond language-based problems. This paper explores if and how LLMs can also be used for automated planning given the diverse ways LLMs are modeled and trained. To do so, we seek to answer four key questions. Firstly, we want to understand the effectiveness of different LLM architectures for plan generation. Secondly, we aim to identify which pre-training data (general purpose vs code specific) effectively facilitates plan generation. Thirdly, we investigate whether fine-tuning or prompting is a more effective approach for plan generation. Finally, we explore whether LLMs are capable of plan generalization. By answering these questions, the study seeks to shed light on the capabilities of LLMs in solving complex planning problems and provide insights into the most effective approaches for using LLMs in this context.

1 Introduction

Automated Planning (Ghallab et al., 2004) focuses on generating sequences of actions, called plans, for an agent to navigate from an initial state to a desired goal state. Automated planning is crucial for many real-world applications, such as robotics, dialog systems, game playing, and more.

In the literature, usually planning problems are described using the Planning Domain Definition Language (PDDL) (Aeronautiques et al., 1998), a Lisp-inspired declarative language with the specifications for the (partial) transition model, initial & goal states and constraints (such as preconditions and effects). Automated planners must follow these specifications to generate valid, optimal, or near-optimal plans without violating constraints. Recently, LLMs such as GPT-4 (OpenAI, 2023) have demonstrated the ability to generate executable pieces of code in some programming languages (Poldrack et al., 2023). PDDL and programming languages use a similar formal syntax and share concepts such as variables, functions, and control structures. This resemblance motivates us to investigate the capabilities of LLMs for plan generation.

LLMs are built using neural networks with millions/billions of learnable parameters, pre-trained with a large corpus of natural language data. LLMs such as GPT-4 have been shown to generate human-like, coherent, and diverse texts. There has been recent interest in exploring the capabilities of LLMs beyond the applications in many natural language processing tasks (Li, 2022; Zhao et al., 2023) with more details on LLM architectures provided in the Appendix A.2. For example, in code generation (Wang et al., 2021; Feng et al., 2020; Chen et al., 2021) and protein folding (Unsal et al., 2022; Ferruz & Höcker, 2022). Recent literature has also explored the application of LLMs for reasoning tasks within simulated textual environments, such as Alexa Arena (Gao et al., 2024), AlfWorld (Shridhar et al., 2020), and Minecraft (Wang et al., 2023a). However, these models still struggle to solve straightforward classical planning problems (Valmeekam et al., 2022; Kambhampati, 2024).

In this paper, we comprehensively analyze LLMs capabilities for automated planning. Toward this goal, we address the following four research questions: **(1)** To what extent can different LLM architectures solve planning problems? **(2)** What pre-training data is effective for plan generation? **(3)** Does fine-tuning and prompting improve LLMs plan generation? **(4)** Are LLMs capable of plan generalization?

Table 1: Definitions for the terms used in the paper.

Term	Definition
PDDL	A formal language used to describe classical planning problems. It requires a domain and problem file.
Domain File	Defines the set of actions, their preconditions and effects, objects and their relations, and predicates that can describe a planning problem within a specific domain.
Problem File	Define the initial state of a planning problem, along with the goal state(s) that needs to be achieved.
Planner	An algorithmic tool that generates a plan of actions to achieve a desired goal state, given the domain and problem PDDL files. An example is the tool FastDownward (Helmert, 2006).
Plan	A sequence of actions that transforms the initial state into one that respects the goal conditions.
Satisficing plan	A plan that achieves the goal state.
Optimal plan	A plan that achieves the goal state with the minimum possible cost (such as time or resources).
Plan Length	A numerical value representing the number of actions or steps required to achieve a given goal.
Degree of Correctness	It is the ratio of solved goals and the total number of goals.
Plan Verification Tool	Determines whether a plan achieves the specified goals while satisfying any constraints and/or requirements.

To answer these questions, we compare different LLMs on six classical planning domains using fine-tuning and prompting approaches. We propose a metric to measure plan generalization and introduce three new tasks to evaluate LLMs on plan generalization. Despite the inapt claims on LLMs for automated planning (Kambhampati, 2024), we show favorable outcomes with appropriate selection of LLM, data preparation, and fine-tuning. We claim that LLMs pre-trained on code generation can benefit from further fine-tuning with problems from several automated planning domains, although their generalization capabilities seem limited. We recommend further research in LLM for better plan generalization.

Our key contributions in this paper are: **(a)** a diverse set of benchmark problems to evaluate LLMs for automated planning (along with a publicly available codebase with fine-tuned model weights to drive future research). **(b)** a metric to evaluate plan generalization and introduce three new tasks to measure it. **(c)** a thorough empirical analysis of LLMs on planning-related metrics and insights on an appropriate use of LLMs for automated planning.

2 Background and Related Work

2.1 Automated Planning

Automated Planning, or simply planning, is a branch of AI that focuses on creating strategies or action sequences, typically for execution by intelligent agents, autonomous robots, and unmanned vehicles. A basic category in planning is a Classical Planning Problem (CPP) (Russell & Norvig, 2003), which is a tuple $\mathcal{M} = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ with domain $\mathcal{D} = \langle F, A \rangle$ - where F is a set of fluents that define a state $s \subseteq F$, and A is a set of actions - and initial and goal states $\mathcal{I}, \mathcal{G} \subseteq F$. Action $a \in A$ is a tuple $(c_a, \mathit{pre}(a), \mathit{eff}^{\pm}(a))$ where c_a is the cost, and $\mathit{pre}(a), \mathit{eff}^{\pm}(a) \subseteq F$ are the preconditions and add/delete effects, i.e., $\delta_{\mathcal{M}}(s, a) \models \perp s$ *if* $s \not\models \mathit{pre}(a)$; *else* $\delta_{\mathcal{M}}(s, a) \models s \cup \mathrm{eff}^{+}(a) \setminus \mathrm{eff}^{-}(a)$ where $\delta_{\mathcal{M}}(\cdot)$ is the transition function. The cumulative transition function is $\delta_{\mathcal{M}}(s, (a_1, a_2, \ldots, a_n)) = \delta_{\mathcal{M}}(\delta_{\mathcal{M}}(s, a_1), (a_2, \ldots, a_n))$. A plan for a CPP is a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ that transforms the initial state $\mathcal{I}$ into the goal state $\mathcal{G}$ using the transition function $\delta_{\mathcal{M}}$. Traditionally, a CPP is encoded using a symbolic representation, where states, actions, and transitions are explicitly enumerated. This symbolic approach, often implemented using Planning Domain Definition

Language or PDDL (McDermott et al., 1998), ensures precise and unambiguous descriptions of planning problems. This formalism allows for applying search algorithms and heuristic methods by automated planners such as FastDownward (Helmert, 2006), LAMA (Richter & Westphal, 2010), and Graphplan (Blum & Furst, 1997) to find a sequence of actions that lead to the goal state. While recognizing the existence of various planning types, such as epistemic (Bolander & Andersen, 2011) and hierarchical (Georgievski & Aiello, 2015), our research primarily utilizes CPP as the foundational basis for investigating the capabilities of LLMs in planning.

For the reader's convenience and to facilitate a clearer understanding throughout this paper, we have summarized the planning-related terminology in Table 1 and provided an example from `Blocksworld` in Appendix A.3.

2.2 LLMs for Plan Generation

LLMs for planning have constituted a focal point of extensive research in recent years (Pallagani et al., 2024). Pallagani et al. (2024) outline eight distinct categories based on the application of LLMs in addressing various aspects of planning problems. In this paper, we concentrate solely on the category of plan generation. In Figure 1, we conduct a survey of contemporary literature where LLMs have been leveraged for plan generation, stratifying them into three primary categories: **(a)** LLMs as Planners **(b)** LLMs with Planners, and **(c)** Cognitive Architectures

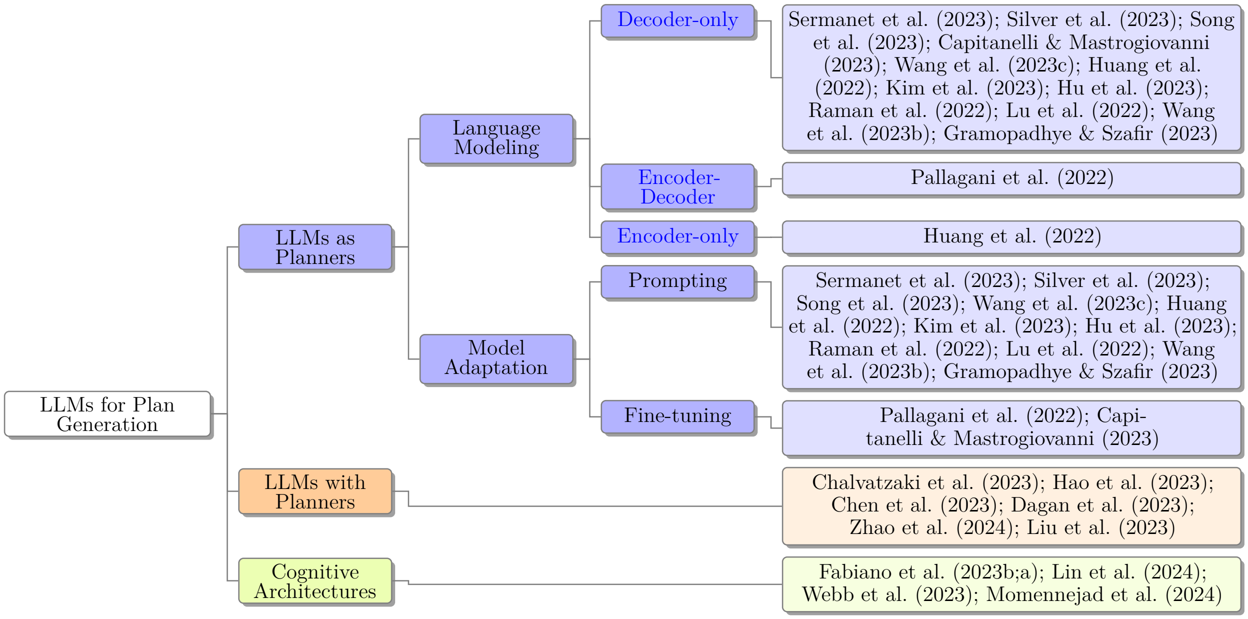


Figure 1: Taxonomy of recent research in LLMs for plan generation.

LLMs as Planners.
In this category, LLMs are tasked with generating plans for planning problems predominantly represented in natural language. Silver et al. (2023) also employ the PDDL representation. This area of study is subdivided based on the language modeling architecture employed and the model adaptation methods to generate plans. Sermanet et al. (2023); Kim et al. (2023) highlight the capabilities of decoder-only models in addressing long-horizon planning tasks in robotics, leveraging multimodal data. Silver et al. (2023) introduced an innovative prompting mechanism that enable decoder-only models to generate programming code capable of generating plans for planning problems expressed in PDDL. This approach is similar to the common-sense prompting strategies described in Capitanelli & Mastrogiovanni (2023); Hu et al. (2023); Lu et al. (2022) and Wang et al. (2023b). Further, Wang et al. (2023c); Raman et al. (2022); Wang et al. (2023a) describe how iterative re-prompting with feedback can assist decoder-only models in solving open-ended planning problems, for example, in the domain of Minecraft. Huang et al. (2022) demonstrate how prompting different

LLMs in a pre-defined sequential order can facilitate generating grounded plans, making use of a decoder-only model for generating abstract plans and an encoder-only model for grounding the abstract plan. Pallagani et al. (2022) explore fine-tuning of a encoder-decoder model for plan generation, which has been pre-trained on programming code. While prompting is the predominant technique employed across most studies, a few have adopted fine-tuning, reporting better outcomes in plan generation. It is critical to acknowledge that irrespective of the language modeling architecture employed, LLMs have been observed to exhibit hallucinations during plan generation (Momennejad et al., 2024) and often produce invalid plans (Valmeekam et al., 2023a). This phenomenon has sparked significant debate regarding the actual capacity of LLMs to engage in reasoning and planning.

LLMs with Planners.
To address the limitations inherent in employing LLMs solely as planners, recent research has explored the integration of LLMs as natural language parsers, which are then combined with symbolic planners. Unlike LLMs as planners, which work auto-regressively, these symbolic planners utilize search algorithms and heuristics to generate sound and complete plans. Chalvatzaki et al. (2023) leverage LLMs at a higher level of abstraction, effectively suggesting sub-goals formulated as PDDL problems to be solved by FastDownward, a symbolic planner. Chen et al. (2023) adopt a similar strategy for addressing task and motion planning problems. Hao et al. (2023) capitalize on the world knowledge acquired by LLMs to predict a set of actions suitable for a given problem, subsequently employed the Monte Carlo Tree Search (MCTS) algorithm to solve for the plan, a methodology similar to (Dagan et al., 2023; Zhao et al., 2024). Furthermore, Liu et al. (2023) utilize an LLM to generate PDDL domain and problem files from natural language descriptions, using a planner to solve for plans optimally.

Cognitive Architectures.
To enhance reasoning capabilities in LLMs, researchers are exploring architectures inspired by human cognition. Webb et al. (2023) introduces an architecture inspired by the prefrontal cortex, which demonstrates improved plan generation compared to current prompting strategies, notably eliminating hallucinations. Momennejad et al. (2024) employ cognitive maps to assess the planning capabilities in LLMs. Additionally, Fabiano et al. (2023a;b); Lin et al. (2024) present cognitive architectures inspired by the dual-process theory of thinking, often characterized as the fast and slow thinking mechanism by Daniel Kahneman, further advancing the field.

Despite the growing body of research investigating the potential of LLMs in automated planning, there remains a notable lack of comprehensive insights into the plan generation capabilities of LLMs. To date, evaluations of these capabilities have predominantly focused on decoder-only models and utilized prompting techniques. *In response to this gap, the current study aims to empirically and comprehensively evaluate LLMs plan generation capabilities. This analysis will span three distinct language modeling architectures, three types of input representations, and four model adaptation methods, focusing on assessing their generalization abilities.*

3 Research Questions

In this study, we aim at exploring the capabilities of LLMs in solving planning problems. To do that, we address the following four research questions (RQ):

- **RQ1 - What are the effective types of language modeling architectures (decoder-only, encoder-decoder, decoder-only) for plan generation using LLMs?** Automated planning necessitates reasoning abilities to formulate plans that satisfy predefined constraints. We aim to assess the effectiveness of various architectures — decoder-only, encoder-decoder, decoder-only for plan generation using LLMs. For this purpose, we conduct evaluations on two LLMs per architecture, employing fine-tuning and prompting approaches where feasible to provide a qualitative analysis of their performance. Additionally, we utilize a plan verification tool (VAL (Howey & Long, 2003)) to validate LLM-generated plans, assessing them based on standard planning-related metrics such as satisficing, optimality, identification of invalid plans, and overall correctness.
- **RQ2 - What pre-training data is effective for plan generation?** LLMs, regardless of architectural nuances, undergo training using either general-purpose corpora such as textual documents

or domain-specific data like programming code. Given the relevance of the task at hand—plan generation using PDDL—to code generation, particularly due to the similarities between PDDL and Lisp, we focus on comparing the performance of LLMs pre-trained exclusively on general-purpose corpora against those trained on code-specific datasets. We aim to provide insights to guide future researchers in selecting the most suitable LLMs for plan generation tasks.

- **RQ3 - Which approach between fine-tuning and prompting improves plan generation?** Our objective is to compare the effectiveness of fine-tuning and prompting approaches for plan generation. Fine-tuning LLM updates the model's parameters using a labeled dataset from the target task. Prompting controls the input to an LLM using a template or a cue to elicit the desired output.
- **RQ4 - Are LLMs capable of plan generalization?** To the best of our knowledge, the current literature provides a limited understanding of the plan generalization capabilities of LLMs (Valmeekam et al., 2023b; Webb et al., 2023). Only a handful of studies have studied and quantitatively measured them. To better evaluate the generalization capabilities of LLMs within the scope of automated planning, we think that the current definition of plan generalization needs to be clarified due to its limited scope, as it fails to account for all possible scenarios that may arise when using LLMs to plan. Therefore, we propose three new tasks to quantify the plan generalization capabilities of LLMs accurately.

4 Experimental Setup

This section describes our planning dataset and discusses the difficulty of the planning domains. We also provide an overview of the evaluated LLMs and the experimental setup for plan generation. In what follows, let the training dataset be $D_{train} = \{(x_i, y_i) \mid x_i \text{ is a planning problem and } y_i \text{ is the optimal plan for } x_i, \text{ for } i = 1, \ldots, n\}$. Let the testing dataset be $D_{test} = \{(x_i, y_i) \mid x_i \text{ is a previously unseen planning problem and } y_i \text{ is the optimal plan for } x_i, \text{ for } i = n+1, \ldots, m\}$. Note that D_{train} and D_{test} consist of pairs of planning problems and their corresponding optimal plans, generated using FastDownward (Helmert, 2006), a classical planning system based on heuristic search.

4.1 Planning Datasets

The International Planning Competition (IPC) (ICAPS, 2022) is a biennial event that benchmarks the state-of-the-art automated planning and scheduling systems. In this study, we consider six classical planning domains represented in PDDL, released as part of the IPC, to assess the planning capabilities of LLMs. We utilize problem generators provided with these domains (Seipp et al., 2022) to generate a planning dataset with 18,000 problems for each domain, encoded in PDDL. We use a random seed to generate the problems and further enforce that there are no duplicate problems in the dataset. Generating ground truth optimal plans for these problems is accomplished using the FastDownward planner, leveraging the A^* search algorithm with LM-Cut heuristics (Helmert & Domshlak, 2011).

The planning domains considered in this study are categorized into three difficulty levels — Easy, Medium, and Hard. This classification allowed us to better abstract 'how difficult' is to solve domains based on the generated and evaluated states for our baseline, *i.e.,* FastDownward when using A^* + LM-Cut heuristic, as shown in Table 2. Generated states refer to all the possible states that can be reached from the initial state of a planning problem by applying valid actions. These generated states may include redundant or irrelevant states that do not contribute to finding a solution to the problem. Evaluated states refer to the subset of generated states the search algorithm has examined to determine their suitability for inclusion in the final solution plan. Let g be the number of generated states and e be the number of evaluated states. The total number of states T is $T = g + e$. The difficulty classification C based on T is given by:

$$C = \begin{cases} \text{Easy} & \text{if } T < 150, \\ \text{Medium} & \text{if } 150 \leq T \leq 300, \\ \text{Difficult} & \text{if } T > 300. \end{cases}$$

In addition to preparing the planning dataset in PDDL, we have translated the problems into both natural language (NL) and a Compact representation. This tripartite approach facilitates a comprehensive evaluation of the plan generation capabilities of LLMs. Including an NL representation is particularly significant, as it offers a more accessible and human-readable format, aligning well with the general-purpose orientation of LLMs. The conversion of PDDL problems into an NL representation is obtained using AutoPlanBench (Stein & Koller, 2023).

Table 2: Our classification of planning domains by the difficulty of solving for FastDownward using A* + LM-Cut. Reported Generated states (Gen.) and Evaluated States (Eval.) are average values for all problems belonging to that domain.

Planning Domain	Difficulty	State Space	Branching Factor	Gen.	Eval.
`Ferry`	Easy	$O(2^n * 2 * m * n!)$, n is no. of cars, m is no. of locations	$O(n+1)$	47	21
`Blocksworld`	Easy	$O(3^n)$, n is no. of blocks	$O(4n/2+1)$	51	35
`Tower of Hanoi`	Medium	$O(3^n)$, n is no. of disks	$O((k-1)k/2)$, k is no. of pegs	141	55
`Miconic`	Medium	$O(n^{(m+1)} * 2^m * m!)$, n is no. of floors, m is no. of passengers	$O(m+1)$	197	72
`Grippers`	Hard	$O(2^n * 3^{nr})$, n is no. of balls, r is no. of robots	$O(3nr+r)$	707	334
`Driverlog`	Hard	$O(l^{(d+t+p)} * k^p * d * t * 2^t)$, l is no. of locations, d is no. of drivers, t is no. of trucks, p is no. of packages	$O(l*(d+t+p+dt+td))$	33520	1347

Conversely, both the PDDL and Compact representations retain the structural syntax akin to programming code, with the latter serving as a streamlined version of the PDDL schema. The Compact representation is a syntactic reduction of PDDL to minimize the number of tokens while maintaining the same semantic values. Appendix A.6 shows the token length variation between PDDL and Compact representation. Notably, the representation of plans derived from FastDownward remains unchanged across the three formats—PDDL, NL, and Compact.

For preliminary testing, the test dataset was configured to mirror the plan length distribution observed in the training dataset, as illustrated in Figure 12b. Subsequent experiments also explore the performance of LLMs with plan lengths that fall outside the training distribution. This methodological diversity is designed to assess not only LLMs proficiency in handling traditional planning languages like PDDL but also their adaptability to interpret and solve planning problems presented in formats that mimic natural human discourse. The evaluation of the three representations employs four distinct methods: zero-shot prompting, one-shot prompting, chain-of-thought (CoT) prompting, and fine-tuning. The prompts for the planning problems are generated using AutoPlanBench Stein & Koller (2023). Examples of PDDL and compact representation are provided in Appendix A.4, A.5 respectively, and the prompting template with NL representation is detailed in Appendix A.7 for reference. For the fine-tuning process, we allocate 80% of the total dataset for each representation, corresponding to 86,400 unique problems across six domains.

4.2 Large Language Models

Table 3 gives an overview of the LLMs employed in this study, including critical design parameters such as the number of layers, heads, embedding size (head dimension), maximum context length, and the nature of the training data source, distinguishing between general-purpose and code-specific models. Our investigation encompasses six LLMs, divided into two models per category of language modeling—specifically, one model tailored for general-purpose applications and another optimized for code-related tasks.

In the category of decoder-only models, we explore GPT-4 (model name on OpenAI documentation is gpt-4-0613) and code-davinci-03. Given the non-modifiable architecture of these models, our exploration is confined to various prompting methods, excluding the possibility of fine-tuning. The exact specifications of these models are unknown, and we write it as "not available" (NA) in our table. For encoder-decoder models,

Table 3: LLM architectures used in our study.

Model-Parameters	Architecture	Pre-training	Layers	Heads	Embed. Size	Context Length
GPT-4 (no-vision)	decoder-only	General-purpose	NA	NA	NA	8192
code-davinci-03	decoder-only	Code-specific	NA	NA	NA	8000
T5-base-220M	encoder-decoder	General-purpose	12	12	768	512
CodeT5-base-220M	encoder-decoder	Code-specific	12	12	768	512
BERT-base-110M	encoder-only	General-purpose	12	12	768	512
StarEncoder-125M	encoder-only	Code-specific	12	12	768	1024

we employ T5 and CodeT5 models. These models are subjected to a broader experimental framework that includes zero-shot, one-shot prompting, and fine-tuning, but excludes CoT prompting, as these models lack compatibility with this specific method. Within encoder-only, we examine BERT and StarEncoder, tested with all methods except for CoT due to lack of compatibility. This diverse selection of models allows for a thorough exploration of the capabilities and limitations of current LLMs in plan generation across both general-purpose and code-specific contexts. Despite the rapid emergence of new LLMs, our study chooses representative models from each category to investigate our research questions.

4.3 Plan Generalization

In this paper, we are interested in the ability of LLMs to generate plans that are close to optimal for unseen, out-of-distribution problems from different planning domains. We need a metric to measure the distance between a pair of plans. Measuring the distance between plans is based on differences in actions, states, or causal structures (Srivastava et al., 2007). This paper uses an action-based criterion for LLM-generated plan vs optimal plan, i.e., the Hamming distance between the order of actions in two plans, motivated by the recent literature (Katz & Sohrabi, 2020). This metric measures how much LLMs deviate from the sequences of actions in the optimal plan. More formally, the plan generalization error E_{pg} is defined as:

$$E_{pg} = \frac{1}{|m-n|} \sum_{i=n+1}^{m} \frac{H(y_i, \hat{y}_i)}{|A|} \tag{1}$$

where $\hat{y}_i$ is the optimal plan for $x_i \in D_{test}$ generated by FastDownward, y_i is the LLM generated plan, $H(\cdot, \cdot)$ is the Hamming distance between two plans, which are made comparable by padding shorter plans to match the length of the longer plan. $|A|$ denotes the total number of actions in the planning domain, providing a normalization factor that adjusts for domain complexity. A lower E_{pg} means that LLM can produce plans more similar to the optimal plans generated by a traditional planning system. To evaluate the plan generalization capabilities of LLMs and provide insights to address RQ4, we propose the following three tasks:

- **Task 1 - Plan Length Generalization:** We evaluate the ability of LLMs to generalize to plan lengths that are out of the distribution of the training set. Given a set of planning domains in the training set $\mathcal{D} = D_1, D_2, ..., D_n$, we select a plan length l_i for each domain D_i that is outside the range of plan lengths considered in the training set. We then pose the planning problem for each domain with the selected plan length to the LLM.
- **Task 2 - Object Name Randomization:** This task examines the LLMs capability to generate plans using randomized object names not encountered in the training set. For each domain in the dataset, we generate a set of randomized object names using the IPC problem generator. These names replace the original object names in the test set's planning problems across each domain, testing the model's ability to understand and adapt to novel identifiers.
- **Task 3 - Unseen Domain Generalization:** We evaluate the LLMs ability to generalize to planning problems from new domains not included in the training set. We select a set of planning domains $\mathcal{D}' = D'_1, D'_2, ..., D'_m$ that differs from the training set's domains. This task challenges the models to apply their learned planning capabilities to unfamiliar contexts.

5 Experimental Results

This section presents the quantitative and qualitative results obtained using LLMs to generate plans for classical planning domains of varying difficulty and generalization abilities across three tasks. All the reported results in this paper are averaged over five independent runs. We evaluate the performance of the considered LLMs on a test set with 3600 planning problems per domain classified into easy (E), medium (M), and hard (H) categories. We assess the generated plans using various planning-related metrics, including satisficing plans (*Sat. Plans*), optimal plans (*Opt. Plans*), degree of correctness (*Deg. Corr.*), and inference time (*Inf. Time*). We can determine the number of invalid plans by subtracting the percentage of satisficing plans from 100 percent.

Table 4: Evaluation of plan generation capabilities of LLMs. For each model, we report the percentage of satisficing plans (Sat. Plans), the percentage of optimal plans (Opt. Plans), and the degree of correctness (Deg. Corr.). Table with all rows is present in Appendix A.8

Model	Input Representation	Method	Sat. Plans (%)			Opt. Plans (%)			Deg. Corr.		
			E	M	H	E	M	H	E	M	H
GPT-4 (no-vision)	Compact	Zero-shot	1.6	0	0	0	0	0	0.04	0	0
		One-shot	7.21	2.66	1.78	2.52	0.07	0	0.05	0.02	0.01
		CoT	21.28	16.74	11.33	10.74	8.21	4.22	0.27	0.25	0.21
	NL	Zero-shot	15.72	10.41	7.78	7.29	3.84	1.25	0.21	0.19	0.07
		One-shot	24.78	22.91	22.78	7.31	3.05	1.25	0.28	0.17	0.07
		CoT	35.02	34.76	32.93	12.88	10.14	9.21	0.41	0.41	0.37
	PDDL	Zero-shot	8.78	6.43	0	3.92	0	0	0.27	0.21	0
		One-shot	10.82	6.90	3.31	7.23	2.84	1.21	0.32	0.17	0.04
		CoT	22.18	18.55	18.97	9.87	5.21	3.22	0.27	0.18	0.19
code-davinci-03	Compact	Zero-shot	4.78	0	0.97	0	0	0	0.21	0	0.04
		One-shot	11.54	11.18	10.14	3.27	1.94	0.07	0.21	0.17	0.11
		CoT	10.77	4.18	1.92	3.74	1.86	0	0.35	0.31	0.27
	NL	One-shot	8.23	7.66	5.21	0	0	0	0.04	0.04	0.04
		CoT	11.29	9.07	5.22	2.84	1.45	1.01	0.02	0.01	0.01
	PDDL	Zero-shot	17.42	11.77	4.38	8.23	6.11	1.84	0.38	0.27	0.21
		One-shot	43.52	37.48	31.89	17.57	15.85	14.69	0.57	0.38	0.31
		CoT	27.71	19.55	19.97	20.11	14.27	11.25	0.31	0.20	0.23
T5	Compact	FT	0.11	0	1.16	0.11	0	0.36	0.02	0	0.03
	NL	Zero-shot	0.16	0	0	0	0	0	0.01	0	0
		One-shot	1.28	1.01	0	0.17	0	0	0.01	0.01	0
		FT	21.56	18.56	11.23	9.02	2.89	1.01	0.28	0.17	0.06
	PDDL	FT	3.78	0.72	0	0	0	0	0.21	0.06	0
CodeT5	Compact	Zero-shot	2.7	0.6	0	1.73	0.6	0	0.07	0	0
		One-shot	1.02	0	0	0.03	0	0	0.01	0	0
		FT	97.57	92.46	89.54	86.21	90.36	66.71	0.99	0.95	0.95
	NL	FT	3.82	3.01	3.03	0	0	0	0.03	0.03	0.03
	PDDL	FT	43.89	37.41	30.42	40.84	31.55	21.02	0.61	0.52	0.47
BERT	Compact	FT	10.89	8.42	6.77	3.57	0	0	0.28	0.07	0.01
	NL	FT	1.78	0.91	0	0.07	0	0	0.01	0.01	0
	PDDL	FT	1.21	0	0	0	0	0	0.01	0	0
StarEncoder	Compact	Zero-shot	4.21	1.79	0.91	1.44	0.07	0	0.12	0.02	0.01
		One-shot	11.02	5.42	1.71	2.08	0	0	0.18	0.07	0.02
		FT	19.44	10.25	5.69	8.92	2.55	2.07	0.25	0.17	0.07
	NL	One-shot	4.05	0	0	1.11	0	0	0.05	0	0
		FT	4.11	1.97	0	0.55	0	0	0.06	0.01	0
	PDDL	Zero-shot	7.91	6.22	5.97	5.01	2.11	1.94	0.18	0.08	0.07
		One-shot	6.91	5.28	3.97	3.01	1.04	0.24	0.17	0.08	0.07
		FT	21.01	15.66	9.89	17.61	9.78	3.88	0.37	0.21	0.18

5.1 Plan Generation

Table 4 offers a detailed evaluation of each LLM across different architectures, utilizing four distinct methods: zero-shot, one-shot, CoT, and fine-tuning (abbreviated as FT in Table 4. Among the vanilla models tested using the zero-shot method across all input representations, code-davinci-03 demonstrates better performance. **We observe limited planning capabilities overall in pre-trained models under zero-shot conditions**, aligning with the findings of (Valmeekam et al., 2022).

In our analysis of decoder-only, encoder-decoder, and encoder-only language models, distinct performance patterns emerge across these architectures. Specifically, code-davinci-03 stands out among causal models by achieving 43.52% satisficing plans in the easy domain, with PDDL input representation and one-shot method. This level of performance, while notable, is modest compared to the significantly higher success rates observed with seq2seq models like CodeT5 in subsequent analysis. The relative performance of code-davinci-03 can be understood through architectural and training differences. With their autoregressive prediction capabilities, decoder-only models are naturally inclined towards tasks with a sequential output structure, such as plan generation. However, their effectiveness is contingent on receiving well-defined contextual cues, which may partly explain their limited success rate in our study compared to more specialized or fine-tuned approaches.

The fine-tuned CodeT5 model, utilizing a compact representation, achieves 97.57% satisficing plans, with 86.21% being optimal. This performance highlights the encoder-decoder model's capacity for adaptability and efficiency in tasks requiring encoding and decoding, further enhanced by fine-tuning on domain-specific datasets. The advantage of the compact representation over PDDL, in terms of performance, can be attributed to its reduced token usage, which alleviates the context length limitations. This reduction in token usage enables the model to maintain more relevant information within the context window, thus supporting a more efficient planning process. Moreover, the performance of the fine-tuned CodeT5 model with PDDL input closely matches that of code-davinci-03 under one-shot prompting. This result indicates that fine-tuning may help to minimize performance discrepancies across different model architectures and input formats.

In contrast, encoder-only models demonstrate modest performance across all input formats and methodologies, highlighting their challenges in producing sequential outputs such as plans. This outcome may be linked to the architectural focus of encoder-only models on bidirectional context processing, which, despite its utility in tasks involving text comprehension, may not be ideally suited for tasks requiring forward-sequential content generation.

Research Question 1

What are the effective types of language modeling (Causal, Seq2Seq, Masked) for plan generation using LLMs?

Answer

In our experiments, decoder-only and encoder-decoder models show better plan generation. Specifically, the fine-tuned CodeT5 model using compact representation outperforms other models, achieving up to 97.57% satisficing plans, of which 86.21% are optimal.

Not only does the architecture influence performance, but the nature of the pre-training data and the choice of input representation are also crucial for the plan generation capabilities of LLMs. Analysis of Table 4 reveals that LLMs pre-trained on general-purpose corpora, predominantly consisting of textual data, exhibit enhanced performance with NL representation of the planning problems. However, the best performance achieved by general-purpose models is observed with GPT-4, which achieves 35.02% satisficing plans using CoT prompting in easy domains. Code-specific models show improved effectiveness when using compact or PDDL representations. This improvement is anticipated due to the structured programming code and the semantics on which these models have been trained. Consistently, code-specific models outperform general-purpose LLMs in plan generation tasks.

Research Question 2

What pre-training data is effective for plan generation?

Answer

In our experiments, LLMs pre-trained on programming code-related datasets demonstrate superior performance in plan generation tasks compared to LLMs trained on natural language.

In this study, fine-tuning the CodeT5 model demonstrated superior performance compared to other methods. When comparing the fine-tuned CodeT5 using compact representation with the highest performing prompting method observed in code-davinci-03 employing one-shot prompting and PDDL input, it is evident that the fine-tuned CodeT5 excels across various domains in terms of satisficing plans, optimal plans, and correctness. This suggests that fine-tuning notably enhances optimal plan generation capabilities, particularly for CodeT5. However, this comparison is not exhaustive due to the inability to fine-tune GPT-4 and code-davinci-03, which limits a comprehensive evaluation of how causal models might perform if similarly optimized.

Research Question 3

Which approach between fine-tuning and prompting improves plan generation?

Answer

In our experiments, fine-tuning improves plan generation capabilities more than prompting (Zero-shot, One-shot, Chain-of-Thought).

To address RQ4, we report the results for the tasks described in Section 4.3.

5.2 Plan Generalization

Task 1 - Plan Length Generalization. In the context of generalization experiments, we focus on the most proficient models derived from fine-tuning and prompting methodologies. Specifically, we utilize the fine-tuned CodeT5 model with a compact representation alongside the one-shot prompting (represented as FS) of code-davinci-03 (or simply code-davinci) with a PDDL representation approach. We subject these models to an empirical evaluation by testing them on ten problems per difficulty class. Notably, the ten problems selected are characterized by a plan length outside the distribution of the training set. Figure 2 depicts the plan length generalization assessment outcomes across the three difficulty classes. Our findings demonstrate that the **fine-tuned CodeT5 model can generalize to plan lengths to some extent, while the one-shot prompting of code-davinci generates only a single valid plan** for a hard domain having a short plan. While neither model produces optimal plans, we observe that the average correctness score of the plans generated by the fine-tuned CodeT5 model is 0.46, while that of code-davinci is 0.04. With regard to E_{pg}, the fine-tuned CodeT5 model has an average score of 0.69, whereas code-davinci has an average score of 1.

Task 2 - Object Name Randomization. For object name randomization, we created three versions of randomized variables for the problems in D_{test} to evaluate the plan generalization capabilities of LLMs. In version 1, we used only single-digit numeric values as object names. In version 2, we used alphanumeric strings of length 2 (similar to the convention followed by IPC generators), where the combinations of alphabets and numerals used were unseen during training. Version 3 consisted of object names named after three alphabets. For the one-shot prompting of code-davinci, we used an example from the training set and posed a problem with randomized object names for which the plan needed to be generated. We also implemented a symbol table that maps the randomized object names to the same vocabulary as the training set to comprehensively evaluate the dependence of LLMs on the training data or the prompting example for plan generation. Table 5 captures the performance of the models considered for generalization, and it is observed that code-davinci's performance is unaffected by object name randomization and retains the same performance as seen in Table 4. However, code-davinci has a high E_{pg}, showing poor plan generalization capabilities. Fine-tuned CodeT5 has the best performance for version 2 and better plan generalization capabilities than any other model (without a symbol table). We observed a decrease in performance when the length of object names was increased, as in version 3, because the model confuses action names and object names. We further noticed an

Table 5: Evaluating the capabilities of LLMs in handling randomized object names.

Object Names	Model	ST	Sat. Plans (%) / Opt. Plan (%)			Deg. Corr.			E_{pg}		
			E	M	H	E	M	H	E	M	H
Version 1	CodeT5(FT)	✗	47.12% / 33.78%	42.74% / 38.68%	39.58% / 21.22%	0.57	0.51	0.51	0.82	0.84	0.84
		✓	97.57% / 86.21%	92.46% / 90.36%	89.54% / 66.71%	0.99	0.95	0.95	0.15	0.18	0.18
	code-davinci(FS)	✗	43.52% / 17.57%	37.48% / 15.85%	31.89% / 14.69%	0.57	0.38	0.31	0.77	0.83	0.96
		✓	43.52% / 17.57%	37.48% / 15.85%	31.89% / 14.69%	0.57	0.38	0.31	0.77	0.83	0.96
Version 2	CodeT5(FT)	✗	66.01% / 64.72%	61.98% / 52.80%	55.17% / 37.5%	0.79	0.72	0.58	0.47	0.49	0.67
		✓	97.57% / 86.21%	92.46% / 90.36%	89.54% / 66.71%	0.99	0.95	0.95	0.15	0.18	0.18
	code-davinci(FS)	✗	43.52% / 17.57%	37.48% / 15.85%	31.89% / 14.69%	0.57	0.38	0.31	0.77	0.83	0.96
		✓	43.52% / 17.57%	37.48% / 15.85%	31.89% / 14.69%	0.57	0.38	0.31	0.77	0.83	0.96
Version 3	CodeT5(FT)	✗	11.82% / 2.10%	4.92% / 1.47%	0.17% / 0%	0.24	0.04	0.01	0.87	0.95	1
		✓	97.57% / 86.21%	92.46% / 90.36%	89.54% / 66.71%	0.99	0.95	0.95	0.15	0.18	0.18
	code-davinci(FS)	✗	43.52% / 17.57%	37.48% / 15.85%	31.89% / 14.69%	0.57	0.38	0.31	0.77	0.83	0.96
		✓	43.52% / 17.57%	37.48% / 15.85%	31.89% / 14.69%	0.57	0.38	0.31	0.77	0.83	0.96

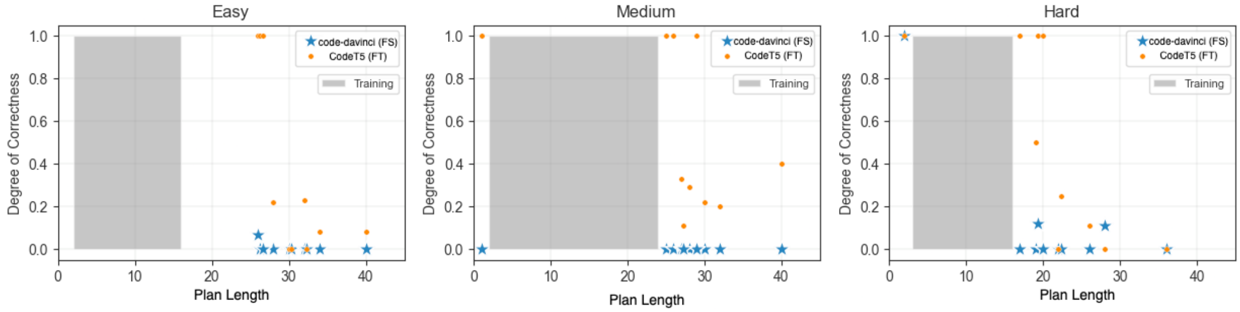


Figure 2: Fine-tuned CodeT5 and code-davinci with one-shot prompting show poor plan length generalization capabilities: E_{pg} for both models is high but plans from fine-tuned CodeT5 overall have a higher degree of correctness. The x-axis represents the plan length, and the y-axis represents the degree of correctness. The training plan lengths are highlighted in grey.

improvement in plan generalization capabilities when a symbol table (ST) was used to map the problem name with randomized object names to the same vocabulary as the training set.

Ground Truth Plan - π^* (*Domain:* `logistics`)

`load-airplane p2 a0 l1-0, fly-airplane a0 l1-0 l0-0, unload-airplane p2 a0 l0-0, load-airplane p0 a0 l0-0, fly-airplane a0 l0-0 l1-0, unload-airplane p0 a0 l1-0`

Fine-tuned CodeT5 (*Inference Time:* 1.12s)

`load-airplane driver0 l0, load-truck package3 truck0 s0, fly-airplane truck0 s0 s1 driver0, load-truck package2 truck0 s1, unload-airplane p0 a0 l1-0, drive-truck truck0 s1 s0 driver0, unload-truck package2 truck0 s0` ✗

One-shot prompting code-davinci (*Inference Time:* 3.51s)

`load-truck p0 t0 l0-0, load-truck p1 t1 l1-0, drive-truck t0 l0-0 l1-0 c1, unload-truck p0 t0 l1-0, drive-truck t1 l1-0 l0-0 c0, unload-truck p1 t1 l0-0, load-airplane p2 a0 l1-0` ✗

Figure 3: Example of an incorrect generation from LLMs for a problem from an unseen domain - `Logistics`.

Task 3 - Unseen Domain Generalization We selected three new classical planning domains, namely `childsnack`, `depots`, and `satellite`, and created ten problems per domain. We randomly chose an example prompt and its corresponding plan from the six training domains for each problem. We report the results averaged over five random seeds. Our findings reveal that both models failed to generate valid plans in this task, resulting in an E_{pg} of 1. We observed that fine-tuned CodeT5 often confused the action and object names present in the test with those seen during training, showing no capabilities to generalize to unseen domains. On the other hand, code-davinci generated relevant actions but incorrect plans for all test cases. To further illustrate our observations, we present in Figure 3 a comparison between the ground truth plan generated by a planner for the `logistics` domain and the output produced by the considered LLMs. This comparison highlights the incorrect combination of action and object names. Fine-tuned models struggle to generate plans for unseen domains. While LLMs, such as code-davinci, show promising developments, they fail to generate valid plans.

Research Question 4

Are LLMs capable of plan generalization?

Answer

Our experiments found limited generalization even for the best-performing LLMs with prompting (code-davinci-03) and fine-tuning (CodeT5).

6 Conclusion and Future Work

The intersection of LLMs and Automated Planning is rapidly gaining interest in AI research, aiming to understand the reasoning capabilities of LLMs. While some researchers have pointed out that LLMs may not inherently possess planning abilities, suggesting their performance might merely reflect sophisticated information retrieval (Kambhampati, 2024), the diverse architectures of LLMs and their training methodologies have not been systematically studied in the context of enhancing their planning capabilities. This paper addresses this gap by comprehensively analyzing LLMs for classical planning problems across architectures, model adaptation methods, and input representations. We outline the relative merits of these approaches, providing a foundation for future research on effectively utilizing LLMs for planning tasks, either independently or in conjunction with other solvers (Fabiano et al., 2023b).

Our study in this paper explores the usability of LLMs in solving automated planning problems. To do this, we defined and addressed four research questions through a comprehensive experimental analysis of several LLMs and a diverse set of planning domains. The study finds that: **(1)** Off-the-shelf, pre-trained LLMs cannot effectively solve planning problems. **(2)** LLMs pre-trained on natural language and programming code are more capable of plan generation than natural language-only models. **(3)** Fine-tuning contributes to improved plan generation. **(4)** LLMs have limited plan generalization abilities. Moreover, our research highlights that fine-tuning aids in partial generalization to plan lengths not encountered during the training phase while maintaining a higher level of correctness than prompting. Notably, when object names are randomized, fine-tuned models exhibit satisfactory performance only when the randomized vocabulary aligns with the training set. Both prompting and fine-tuning approaches prove ineffective when solving problems from unfamiliar domains. In future work, we plan to investigate recent techniques, such as scratchpad fine-tuning and prompting methodologies, that have been shown to enhance the length generalization capabilities of LLMs. These methods could improve the planning capabilities of LLMs and open up new avenues for their use in solving complex planning problems.

References

Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. PDDL – the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

Thomas Bolander and Mikkel Birkegaard Andersen. Epistemic planning for single-and multi-agent systems. *Journal of Applied Non-Classical Logics*, 21(1):9–34, 2011.

Alessio Capitanelli and Fulvio Mastrogiovanni. A framework to generate neurosymbolic pddl-compliant planners. *arXiv preprint arXiv:2303.00438*, 2023.

Georgia Chalvatzaki, Ali Younes, Daljeet Nandha, An Thai Le, Leonardo FR Ribeiro, and Iryna Gurevych. Learning to reason over scene graphs: a case study of finetuning gpt-2 into a robot language model for grounded task planning. *Frontiers in Robotics and AI*, 10, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yongchao Chen, Jacob Arkin, Yang Zhang, Nicholas Roy, and Chuchu Fan. Autotamp: Autoregressive task and motion planning with llms as translators and checkers. *arXiv preprint arXiv:2306.06531*, 2023.

Gautier Dagan, Frank Keller, and Alex Lascarides. Dynamic planning with a llm. *arXiv preprint arXiv:2308.06391*, 2023.

Francesco Fabiano, Vishal Pallagani, Marianna Bergamaschi Ganapini, Lior Horesh, Andrea Loreggia, Keerthiram Murugesan, Francesca Rossi, and Biplav Srivastava. Fast and slow planning. *arXiv preprint arXiv:2303.04283*, 2023a.

Francesco Fabiano, Vishal Pallagani, Marianna Bergamaschi Ganapini, Lior Horesh, Andrea Loreggia, Keerthiram Murugesan, Francesca Rossi, and Biplav Srivastava. Plan-sofai: A neuro-symbolic planning architecture. In *Neuro-Symbolic Learning and Reasoning in the era of Large Language Models*, 2023b.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Noelia Ferruz and Birte Höcker. Controllable protein design with language models. *Nature Machine Intelligence*, pp. 1–12, 2022.

Qiaozi Gao, Govind Thattai, Suhaila Shakiah, Xiaofeng Gao, Shreyas Pansare, Vasu Sharma, Gaurav Sukhatme, Hangjie Shi, Bofei Yang, Desheng Zhang, et al. Alexa arena: A user-centric interactive platform for embodied ai. *Advances in Neural Information Processing Systems*, 36, 2024.

Ilche Georgievski and Marco Aiello. Htn planning: Overview, comparison, and beyond. *Artif. Intell.*, 222: 124–156, 2015. URL `https://api.semanticscholar.org/CorpusID:42064508`.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice.* The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, Amsterdam, 2004. ISBN 978-1-55860-856-6. URL `http://www.sciencedirect.com/science/book/9781558608566`.

Maitrey Gramopadhye and Daniel Szafir. Generating executable action plans with environmentally-aware language models. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3568–3575. IEEE, 2023.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.

Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Malte Helmert and Carmel Domshlak. Lm-cut: Optimal planning with the landmark-cut heuristic. *Seventh international planning competition (IPC 2011), deterministic part*, pp. 103–105, 2011.

R. Howey and D. Long. Val's progress: The automatic validation tool for pddl2.1 used in the international planning competition. In *ICAPS 2003 workshop on "The Competition: Impact, Organization, Evaluation, Benchmarks", Trento, Italy*, 2003.

Hanxu Hu, Hongyuan Lu, Huajian Zhang, Wai Lam, and Yue Zhang. Chain-of-symbol prompting elicits planning in large langauge models. *arXiv preprint arXiv:2305.10276*, 2023.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pp. 9118–9147. PMLR, 2022.

ICAPS. International planning competitions at international conference on automated planning and scheduling (icaps). In *https://www.icaps-conference.org/competitions/*, 2022.

Subbarao Kambhampati. Can large language models reason and plan? *Annals of the New York Academy of Sciences*, 2024.

Michael Katz and Shirin Sohrabi. Reshaping diverse planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06):9892–9899, Apr. 2020. doi: 10.1609/aaai.v34i06.6543. URL `https://ojs.aaai.org/index.php/AAAI/article/view/6543`.

Gyeongmin Kim, Taehyeon Kim, Shyam Sundar Kannan, Vishnunandan LN Venkatesh, Donghan Kim, and Byung-Cheol Min. Dynacon: Dynamic robot planner with contextual awareness via llms. *arXiv preprint arXiv:2309.16031*, 2023.

Hang Li. Language models: Past, present, and future. *Commun. ACM*, 65(7):56–63, jun 2022. ISSN 0001-0782. doi: 10.1145/3490443. URL `https://doi.org/10.1145/3490443`.

Bill Yuchen Lin, Yicheng Fu, Karina Yang, Faeze Brahman, Shiyu Huang, Chandra Bhagavatula, Prithviraj Ammanabrolu, Yejin Choi, and Xiang Ren. Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks. *Advances in Neural Information Processing Systems*, 36, 2024.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

Yujie Lu, Weixi Feng, Wanrong Zhu, Wenda Xu, Xin Eric Wang, Miguel Eckstein, and William Yang Wang. Neuro-symbolic procedural planning with commonsense prompting. *arXiv preprint arXiv:2206.02928*, 2022.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

Ida Momennejad, Hosein Hasanbeig, Felipe Vieira Frujeri, Hiteshi Sharma, Nebojsa Jojic, Hamid Palangi, Robert Ness, and Jonathan Larson. Evaluating cognitive maps and planning in large language models with cogeval. *Advances in Neural Information Processing Systems*, 36, 2024.

OpenAI. Gpt-4 technical report, 2023.

Vishal Pallagani, Bharath Muppasani, Keerthiram Murugesan, Francesca Rossi, Lior Horesh, Biplav Srivastava, Francesco Fabiano, and Andrea Loreggia. Plansformer: Generating symbolic plans using transformers. *arXiv preprint arXiv:2212.08681*, 2022.

Vishal Pallagani, Kaushik Roy, Bharath Muppasani, Francesco Fabiano, Andrea Loreggia, Keerthiram Murugesan, Biplav Srivastava, Francesca Rossi, Lior Horesh, and Amit Sheth. On the prospects of incorporating large language models (llms) in automated planning and scheduling (aps). *arXiv preprint arXiv:2401.02500*, 2024.

Russell A Poldrack, Thomas Lu, and Gašper Beguš. Ai-assisted coding: Experiments with gpt-4, 2023.

Shreyas Sundara Raman, Vanya Cohen, Eric Rosen, Ifrah Idrees, David Paulius, and Stefanie Tellex. Planning with large language models via corrective re-prompting. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010. doi: 10.1613/jair.2972.

S. Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach. Second Edition.* 01 2003.

Jendrik Seipp, Álvaro Torralba, and Jörg Hoffmann. PDDL generators. `https://doi.org/10.5281/zenodo.6382173`, 2022.

Pierre Sermanet, Tianli Ding, Jeffrey Zhao, Fei Xia, Debidatta Dwibedi, Keerthana Gopalakrishnan, Christine Chan, Gabriel Dulac-Arnold, Sharath Maddineni, Nikhil J Joshi, et al. Robovqa: Multimodal long-horizon reasoning for robotics. *arXiv preprint arXiv:2311.00899*, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. Generalized planning in pddl domains with pretrained large language models. *arXiv preprint arXiv:2305.11014*, 2023.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 2998–3009, 2023.

Biplav Srivastava, Tuan Anh Nguyen, Alfonso Gerevini, Subbarao Kambhampati, Minh Binh Do, and Ivan Serina. Domain independent approaches for finding diverse plans. In Manuela M. Veloso (ed.), *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pp. 2016–2022, 2007. URL `http://ijcai.org/Proceedings/07/Papers/325.pdf`.

Katharina Stein and Alexander Koller. Autoplanbench:: Automatically generating benchmarks for llm planners from pddl. *arXiv preprint arXiv:2311.09830*, 2023.

Serbulent Unsal, Heval Atas, Muammer Albayrak, Kemal Turhan, Aybar C Acar, and Tunca Doğan. Learning functional properties of proteins with language models. *Nature Machine Intelligence*, 4(3):227–245, 2022.

Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). *arXiv preprint arXiv:2206.10498*, 2022.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023a. URL `https://openreview.net/forum?id=YXogl4uQUO`.

Karthik Valmeekam, Sarath Sreedharan, Matthew Marquez, Alberto Olmo, and Subbarao Kambhampati. On the planning abilities of large language models (a critical investigation with a proposed benchmark). *arXiv preprint arXiv:2302.06706*, 2023b.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023a.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023b.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021.

Zihao Wang, Shaofei Cai, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023c.

Taylor Webb, Shanka Subhra Mondal, Chi Wang, Brian Krabach, and Ida Momennejad. A prefrontal cortex-inspired architecture for planning in large language models. *arXiv preprint arXiv:2310.00194*, 2023.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2023.

Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. *Advances in Neural Information Processing Systems*, 36, 2024.

A Appendix

A.1 Frequently Asked Questions

A.1.1 Will the codebase be made publicly available?

Yes, the codebase along with all the fine-tuning, prompting, and inference scripts along with the fine-tuned model weights for all applicable models will be released for public access after the review period in accordance with the Findability, Accessibility, Interoperability, and Reuse (FAIR) of digital assets principles. The codebase will be made available via Github and the datasets as well as model weights will be released on Zenodo.

A.1.2 What is the reason for taking 18,000 problems per domain for fine-tuning?

In our experiment, we examined different numbers of planning problems, starting from 3000 per planning domain and increasing them by multiples of 3. We employed an 80%-20% train-test split to evaluate the models. Upon reaching 18,000 problems, we observed the best performance across all the models. Subsequently, we conducted fixed point iteration to assess model performance up to 24,000 problems. However, no significant changes in model performance were observed beyond 18,000 problems. Therefore, we chose 18,000 problems per domain as the de facto standard. In Table 6, we show the variation in satisficing plans generated by fine-tuned CodeT5 with respect to the dataset size. We only show the results of fine-tuned CodeT5 for brevity, but similar results are observed across all models considered for fine-tuning.

Table 6: Variations in the performance of satisficing plans generated by fine-tuned CodeT5 using different dataset sizes per domain. After reaching 18,000 datapoints per domain, i.e., a total of 108,000 total problems belonging to six domains, we see a saturation in the performance of plan generation with increase in data points. The values in the brackets show the increase/decrease in performance in comparison with the values in bold to showcase fixed point iteration.

Model	Dataset size (per domain)	Satisficing Plans (%)		
		Easy	Medium	Hard
Fine-tuned CodeT5	3000	0.20%	0%	0%
	6000	19.72%	12.11%	6.57%
	9000	47.78%	32.65%	27.92%
	12000	69.82%	41.17%	32.67%
	15000	86.91%	73.88%	68.71%
	18000	**97.57%**	**92.46%**	**89.54%**
	21000	97.57% (↑ 0%)	92.46% (↑ 0%)	89.51% (↓ 0.03%)
	24000	97.77% (↑ 0.02%)	92.46% (↑ 0%)	89.32% (↓ 0.19%)

A.1.3 What is the significance of using FastDownward planner to generate the dataset?

FastDownward is a traditional planning system that searches the space of world states associated with a planning task in the forward direction using heuristics. In the 4th International Planning Competition at ICAPS 2004, FastDownward secured first place in the "traditional (i.e. propositional, non-optimizing) track". FastDownward comes equipped with a variety of search algorithms by default. We utilize the A* + LM-Cut heuristic since it can produce the optimal plans. Thus, we use FastDownward to generate a planning dataset consisting of optimal plans.

A.2 Language Modeling Architectures

Large language models are neural network models with upwards of a million parameters trained on extremely large corpora of natural language data. These models are proficient in interpreting, generating, and contextualizing human language, leading to applications ranging from text generation to language-driven reasoning tasks. The evolution of LLMs in NLP began with rule-based models, progressed through statistical models, and achieved a significant breakthrough with the introduction of neural network-based models. The shift to sequence-based neural networks, with Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, marked a notable advancement due to their capability to process information and context over long sequences.

However, shortcomings in RNNs and LSTMs due to vanishing gradients and, consequently, loss of *very long* sequence contexts led to the development of the transformer model, which leverages a novel architecture without the need for recurrence. The transformer consists of an encoder and decoder, each comprising a stack of layers that use self-attention and position-wise feed-forward networks. This design allows each position in the encoder to attend overall positions in the previous layer of the encoder, enabling parallelization and handling of longer contexts. Similarly, in the decoder, the self-attention (SA) layers provide attention to positions up to and including the current position. This architecture has substantially improved speed and performance on long-sequence tasks.

The SA mechanism at the heart of the transformer enables it to focus on different parts of a long input sequence in parallel, which enhances the understanding of contextual nuances in language patterns over extremely long sequences. This mechanism is also complemented with positional encodings in transformers to enable the model to maintain an awareness of word/token order, which is required to understand accurate grammar and syntax. The self-attention mechanism, central to transformers, uses a query, key, and value system to contextualize dependencies in the input sequence. Informally, the SA concept is inspired by classical information retrieval systems where the query is the input sequence context, the key refers to a "database" contained within the parametric memory, and the value is the actual value present at that reference. The SA operation is mathematically expressed in Equation 2.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2}$$

In this equation, Q, K, and V denote the query, key, and value matrices. The scaling factor $\sqrt{d_k}$, where d_k is the dimension of the keys, is employed to standardize the vectors to unit variance for ensuring stable softmax gradients during training. Since the introduction of LLMs with self-attention, several architectural variants have depended on the downstream tasks.

Causal Language Modeling (CLMs): CLMs, such as GPT-4, are decoder-only models designed for tasks where text generation is sequential and dependent on the preceding context. They predict each subsequent word based on the preceding words, modeling the probability of a word sequence in a forward direction. This process is mathematically formulated as shown in Equation 3.

$$P(T) = \prod_{i=1}^{n} P(t_i | t_{<i}) \tag{3}$$

In this formulation, $P(t_i|t_{<i})$ represents the probability of the i-th token given all preceding tokens, $t_{<i}$. This characteristic makes CLMs particularly suitable for applications like content generation, where the flow and coherence of the text in the forward direction are crucial.

Masked Language Modeling (MLMs): Unlike CLMs, MLMs like BERT are trained to understand the bidirectional context by predicting words randomly masked in a sentence. This approach allows the model to learn both forward and backward dependencies in language structure. MLMs are encoder-only models. The MLM prediction process can be represented as Equation 4.

$$P(T_{\text{masked}} | T_{\text{context}}) = \prod_{i \in M} P(t_i | T_{\text{context}}) \tag{4}$$

Here, T_{masked} is the set of masked tokens in the sentence, T_{context} represents the unmasked part of the sentence, and M is the set of masked positions. MLMs have proven effective in NLP tasks such as sentiment analysis or question answering.

Sequence-to-Sequence (Seq2Seq) Modeling: Seq2Seq models, like T5, are encoder-decoder models designed to transform an input sequence into a related output sequence. They are often employed in tasks that require a mapping between different types of sequences, such as language translation or summarization. The Seq2Seq process is formulated as Equation 5.

$$P(T_{\text{output}}|T_{\text{input}}) = \prod_{i=1}^{m} P(t_{\text{output}_i}|T_{\text{input}}, t_{\text{output}_{<i}}) \tag{5}$$

In Equation 5, T_{input} is the input sequence, T_{output} is the output sequence, and $P(t_{\text{output}_i}|T_{\text{input}}, t_{\text{output}_{<i}})$ calculates the probability of generating each token in the output sequence, considering both the input sequence and the preceding tokens in the output sequence.

In addition to their architectural variants, the utility of LLMs is further enhanced by specific model utilization strategies, enabling their effective adaptation to various domains at scale. One key strategy is fine-tuning, which applies to pre-trained LLMs. Pre-trained LLMs are models already trained on large datasets to understand and generate language, acquiring a broad linguistic knowledge base. Fine-tuning involves further training pre-trained LLMs on a smaller, task-specific dataset, thereby adjusting the neural network weights for particular applications. This process is mathematically represented in Equation 6.

$$\theta_{\text{fine-tuned}} = \theta_{\text{pre-trained}} - \eta \cdot \nabla_{\theta} L(\theta, D_{\text{task}}) \tag{6}$$

Here, $\theta_{\text{fine-tuned}}$ are the model parameters after fine-tuning, $\theta_{\text{pre-trained}}$ are the parameters obtained from pre-training, η is the learning rate, and $\nabla_{\theta} L(\theta, D_{\text{task}})$ denotes the gradient of the loss function L with respect to the parameters θ on the task-specific dataset D_{task}.

$$P(T|C) = \prod_{i=1}^{n} P(t_i|t_{<i}, C) \tag{7}$$

Complementing the fine-tuning approach is in-context learning, an alternative strategy particularly characteristic of models like the GPT series. This method diverges from fine-tuning by enabling the model to adapt its responses based on immediate context or prompts without necessitating further training. The efficacy of in-context learning is a direct consequence of the comprehensive pre-training phase, where models are exposed to diverse textual datasets, thereby acquiring a nuanced understanding of language and context. Given a context C, the model generates text T that is contextually relevant, as shown in Equation 7. Here, $P(T|C)$ is the probability of generating text T given the context C, and $P(t_i|t_{<i}, C)$ is the probability of generating the i-th token t_i given the preceding tokens $t_{<i}$ and the context C.

A.3 Example of a Planning Problem

To illustrate the application of CPP in a more tangible context, let's consider the well-known `Blocksworld` domain. This example is a practical demonstration of how the formal elements of CPP are instantiated in a specific scenario. In the `Blocksworld` domain, the set of fluents F is defined by the predicates describing the locations of blocks, their clear status, and the state of the agent's hand. Specifically,

$$F = \{\texttt{on}(x, y), \texttt{onTable}(x), \texttt{clear}(x), \texttt{handempty}\},$$

where x and y are variables representing blocks. The actions set A includes ***pick-up***, ***put-down***, ***stack***, and ***unstack***, with specific preconditions and effects as follows:

- ***pick-up***(b) where b is a block:

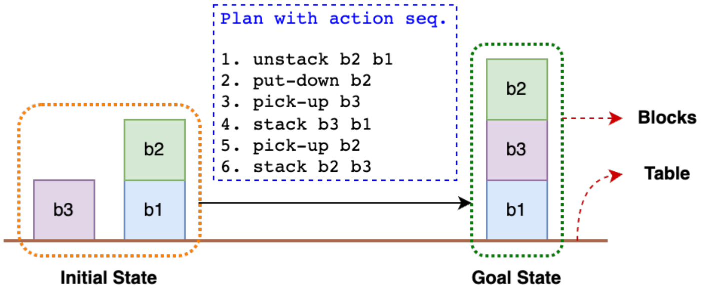


Figure 4: An illustrative example of the `Blocksworld` planning problem. The left side of the figure shows the initial state with blocks b1, b2, and b3. The right side of the figure shows the goal state after applying a sequence of actions.

  - Preconditions: $\texttt{clear}(b), \texttt{onTable}(b), \texttt{handempty}$
  - Effects: $\neg\texttt{onTable}(b), \neg\texttt{handempty}, \texttt{holding}(b)$

- $\boldsymbol{put\text{-}down}(b)$:
  - Preconditions: $\texttt{holding}(b)$
  - Effects: $\texttt{onTable}(b), \texttt{handempty}, \neg\texttt{holding}(b)$
- $\boldsymbol{stack}(b, t)$ where t is the target block:
  - Preconditions: $\texttt{holding}(b), \texttt{clear}(t)$
  - Effects: $\texttt{on}(b, t), \texttt{handempty}, \neg\texttt{clear}(t), \neg\texttt{holding}(b)$
- $\boldsymbol{unstack}(b, t)$:
  - Preconditions: $\texttt{on}(b, t), \texttt{clear}(b), \texttt{handempty}$
  - Effects: $\texttt{holding}(b), \texttt{clear}(t), \neg\texttt{on}(b, t), \neg\texttt{handempty}$

The initial state $\mathcal{I}$ is defined by the fluents $\{\texttt{onTable}(b1), \texttt{on}(b2, b1), \texttt{onTable}(b3), \texttt{clear}(b2), \texttt{clear}(b3)\}$. The goal state $\mathcal{G}$ is defined by the fluents $\{\texttt{onTable}(b1), \texttt{on}(b3, b1), \texttt{on}(b2, b3), \texttt{clear}(b2)\}$, which describes the desired end configuration where b1 is on the table, b3 is on b1, and b2 is on b3, with b2 being clear.

Figure 4 illustrates the transition from the initial state $\mathcal{I}$ to the goal state $\mathcal{G}$ through a sequence of actions that modify the state according to the planning domain's transition function $\delta_{\mathcal{M}}$, which is explained in detail below -

1. $\boldsymbol{unstack}(b2,\ b1)$: The state changes according to $\delta_{\mathcal{M}}(s, \boldsymbol{unstack}(b2, b1))$, resulting in $s = s \cup \{\texttt{clear}(b1), \texttt{holding}(b2)\} \setminus \{\texttt{on}(b2, b1), \texttt{clear}(b2)\}$.
2. $\boldsymbol{put\text{-}down}(b2)$: Applying $\delta_{\mathcal{M}}(s, \boldsymbol{put\text{-}down}(b2))$ modifies the state to $s = s \cup \{\texttt{onTable}(b2), \texttt{clear}(b2)\} \setminus \{\texttt{holding}(b2)\}$.
3. $\boldsymbol{pick\text{-}up}(b3)$: The state transition function $\delta_{\mathcal{M}}(s, \boldsymbol{pick\text{-}up}(b3))$ yields $s = s \cup \{\texttt{holding}(b3)\} \setminus \{\texttt{onTable}(b3), \texttt{clear}(b3)\}$.
4. $\boldsymbol{stack}(b3,\ b1)$: By $\delta_{\mathcal{M}}(s, \boldsymbol{stack}(b3, b1))$, the state is updated to $s = s \cup \{\texttt{on}(b3, b1)\} \setminus \{\texttt{holding}(b3), \texttt{clear}(b1)\}$.
5. $\boldsymbol{pick\text{-}up}(b2)$: The action $\delta_{\mathcal{M}}(s, \boldsymbol{pick\text{-}up}(b2))$ alters the state to $s = s \cup \{\texttt{holding}(b2)\} \setminus \{\texttt{onTable}(b2), \texttt{clear}(b2)\}$.
6. $\boldsymbol{stack}(b2,\ b3)$: The final action $\delta_{\mathcal{M}}(s, \boldsymbol{stack}(b2, b3))$ leads to the goal state with $s = s \cup \{\texttt{on}(b2, b3)\} \setminus \{\texttt{holding}(b2), \texttt{clear}(b3)\}$.

A.4 Planning Dataset

This section provides a brief overview of the domains used in our study, accompanied by visualizations of their corresponding PDDL domain and problem files.

A.4.1 Description of the domains

A.4.2 `Ferry`

`Ferry` is a classical planning domain involving a ferry crossing a river to transport passengers and their vehicles from one side of the river. The ferry can carry a limited number of vehicles at a time and must return to the starting point to pick up more cars if passengers are still waiting. The domain includes constraints such as vehicle capacity, safety conditions, and scheduling. The goal of the domain is to transport all passengers and their vehicles to the other side of the river safely and efficiently. The `Ferry` domain is a benchmark problem for classical planners and has been used in various planning competitions. Based on the domain definition of `Ferry`, the state space and branching factor can be calculated as follows:

- The state space is $O(2^n * 2 * m * n!)$, where n is the number of cars and m is the number of locations. This is because each car can be either on the ferry or at a location (2^n possibilities), the ferry can be at any location (m possibilities), the order of the cars on the ferry matters ($n!$ possibilities), and the ferry can be either empty or full (2 possibilities).
- The branching factor is $O(n+1)$, where n is the number of cars. This is because there are only two operators (board and debark) with one parameter each (car) and one operator (sail) with no parameters. For each car, there are two possible operators that can be applied to it (board or debark), but only one of them is applicable at any given state. Additionally, there is one operator that can be applied to any state (sail), which moves the ferry from one location to another.

Figure 5 shows the domain and a sample problem file in PDDL for `Ferry`.

A.4.3 `Blocksworld`

`Blocksworld` is a classical planning domain that involves a world of blocks arranged on a table. The domain includes various actions such as picking up, putting down, and stacking or unstacking blocks. The goal of the domain is to reach a specific arrangement of blocks on the table from an initial configuration, which may involve stacking blocks on top of each other or moving blocks to different positions. It is commonly used as a benchmark problem for classical planners and has been used in various planning competitions. The domain is simple and abstract yet complex enough to demonstrate different planning challenges, such as state explosion and search complexity. The state space and branching factor can be calculated as follows:

- The state space of the four action `Blocksworld` is $O(3^n)$ because each block can be in one of three possible locations: on the table, on another block, or in hand. Therefore, the number of possible states is 3^n, where n is the number of blocks.
- The branching factor of the four action `Blocksworld` is O(4n/2 + 1) because, at each state, four types of actions can be applied to any block: pick up, put down, stack, and unstack. However, not all moves apply to all blocks at all times. For example, a block cannot be picked up if it is unclear, and a block cannot be stacked if the hand is empty. Therefore, the number of applicable actions is at most $4n/2+1$, where n is the number of blocks.

Figure 6 shows the `Blocksworld` domain and a sample problem file in PDDL.

A.4.4 `Miconic`

`Miconic` planning domain is a model of an elevator system that transports passengers between building floors. The elevator can move up or down one floor at a time and can board or depart passengers on each floor.

Listing 1: Domain description of `Ferry`

```
(define (domain ferry)
   (:predicates (not-eq ?x ?y)
                (car ?c)
                (location ?l)
                (at-ferry ?l)
                (at ?c ?l)
                (empty-ferry)
                (on ?c))

   (:action sail
       :parameters  (?from ?to)
       :precondition (and (not-eq ?from ?to)
                          (location ?from) (location ?to) (at-ferry ?from))
       :effect (and  (at-ferry ?to)
                     (not (at-ferry ?from))))


   (:action board
       :parameters (?car ?loc)
       :precondition  (and  (car ?car) (location ?loc)
                            (at ?car ?loc) (at-ferry ?loc) (empty-ferry))
       :effect (and (on ?car)
                    (not (at ?car ?loc))
                    (not (empty-ferry))))

   (:action debark
       :parameters  (?car  ?loc)
       :precondition  (and  (car ?car) (location ?loc)
                            (on ?car) (at-ferry ?loc))
       :effect (and (at ?car ?loc)
                    (empty-ferry)
                    (not (on ?car)))))
```

Listing 2: Problem description of `Ferry`

```
(define (problem ferry-1)
    (:domain ferry)
    (:objects
        location-1 location-2 location-3 car-1 car-2)
    (:init
        (location location-1)
        (location location-2)
        (location location-3)
        (car car-1)
        (car car-2)
        (at car-1 location-1)
        (at car-2 location-1)
        (at-ferry location-2)
        (empty-ferry)
        (not-eq location-1 location-2)
        (not-eq location-1 location-3)
        (not-eq location-2 location-3))
    (:goal
        (and (at car-1 location-2)
             (at car-2 location-2))))
```

Figure 5: Domain and problem descriptions of `Ferry`

Listing 3: Domain description of `Blocksworld`

```
(define (domain blocksworld)
(:requirements :strips)
(:predicates
(on ?x ?y)
(ontable ?x)
(clear ?x)
(handempty)
(holding ?x))

(:action pick-up
:parameters (?x)
:precondition (and (clear ?x) (ontable ?x) (handempty))
:effect (and (not (ontable ?x))
(not (clear ?x))
(not (handempty))
(holding ?x)))

(:action put-down
:parameters (?x)
:precondition (holding ?x)
:effect (and (not (holding ?x))
(clear ?x)
(handempty)
(ontable ?x)))

(:action stack
:parameters (?x ?y)
:precondition (and (holding ?x) (clear ?y))
:effect (and (not (holding ?x))
(not (clear ?y))
(clear ?x)
(handempty)
(on ?x ?y)))

(:action unstack
:parameters (?x ?y)
:precondition (and (on ?x ?y) (clear ?x) (handempty))
:effect (and (holding ?x)
(clear ?y)
(not (clear ?x))
(not (handempty))
(not (on ?x ?y)))))
```

Listing 4: Problem description of `Blocksworld`

```
(define (problem blocksworld-1)
(:domain blocksworld)
(:objects a b c d e f)
(:init
(clear a) (clear b) (clear c)
(ontable d) (ontable e) (ontable f)
(handempty)
(on a b) (on b c)
(on d e) (on e f)
)
(:goal (and (on a c) (on d b) (on e f)))))
```

Figure 6: Domain and problem descriptions of `Blocksworld`

The goal is to deliver all the passengers to their desired floors. The state space and branching factor can be calculated as follows:

- The state space of the `Miconic` domain is $O(n^{(m+1)} * 2^m * m!)$ because each state is determined by the following factors:
  - The floor of the elevator (n possibilities, where n is the number of floors)
  - The destination floor of each passenger (n possibilities for each of the m passengers, where m is the number of passengers)
  - The location of each passenger (2 possibilities for each passenger: inside or outside the elevator)
  - The permutation of passengers inside the elevator ($m!$ possibilities, assuming the order matters)
- The branching factor of the `Miconic` domain is $O(m+1)$ because, at each state, there are $m+1$ possible actions: move up, move down, or board/depart a passenger. However, not all actions are applicable at all times. For example, the elevator cannot move up if it is on the top floor and cannot load a passenger if it is full. Therefore, the number of applicable actions is at most $m+1$.

Figure 7 shows the `Miconic` domain and a sample problem file in PDDL.

A.4.5 `Tower of Hanoi`

The `Tower of Hanoi` planning domain is a puzzle model involving moving disks of different sizes between pegs. The puzzle starts with all the disks stacked on one peg in decreasing order of size, and the goal is to move all the disks to another peg, following two rules: only one disk can be moved at a time, and a larger disk cannot be placed on top of a smaller disk. The state space and branching factor of this domain can be calculated as follows:

- The state space of the `Tower of Hanoi` is $O(3^n)$ because each disk can be on one of three possible pegs. Therefore, the number of possible states is 3^n, where n is the number of disks.
- The branching factor of the `Tower of Hanoi` is $O((k-1)k/2)$ because, at each state, there are k possible pegs to move a disk from and $k-1$ possible pegs to move a disk to. Therefore, the number of valid moves is at most $(k-1)k/2$, where k is the number of pegs. However, not all moves are valid, as some may violate the puzzle's rules.

Figure 8 shows the `Tower of Hanoi` domain and a sample problem file in PDDL.

A.4.6 `Grippers`

The `Grippers` planning domain is a robot model that can move between rooms and pick up or drop balls using its grippers. The robot has two left and right grippers and can hold at most one ball in each gripper. The goal is to move all the balls from one room to another. The state space and branching factor of this domain can be calculated as follows:

- The state space of the `Grippers` domain is $O(2^n * 3^{(nr)})$ because the following factors determine each state:
  - The presence of a robot in the room (2 possibilities).
  - The room of each ball (2 possibilities for each of the n balls, where n is the number of balls)
  - The gripper of each ball (3 possibilities for each of the n balls: left, right, or none)
  - The ball in each gripper (n possibilities for each of the r grippers, where r is the number of robots)
- The branching factor of the `Grippers` domain is $O(3nr+r)$ because, at each state, three types of actions can be applied to any ball: pick up, drop, or do nothing. Additionally, there are r possible actions for moving the robot to another room. Therefore, the number of possible actions is at most $3nr+r$.

Listing 5: Domain description of `Miconic`

```
(define (domain miconic)
  (:requirements :strips)
  (:types passenger - object
          floor - object
         )

(:predicates
(origin ?person - passenger ?floor - floor)
(destin ?person - passenger ?floor - floor)
(above ?floor1 - floor  ?floor2 - floor)
(boarded ?person - passenger)
(not-boarded ?person - passenger)
(served ?person - passenger)
(not-served ?person - passenger)
(lift-at ?floor - floor)
)

(:action board
  :parameters (?f - floor ?p - passenger)
  :precondition (and (lift-at ?f) (origin ?p ?f))
  :effect (boarded ?p))

(:action depart
  :parameters (?f - floor ?p - passenger)
  :precondition (and (lift-at ?f) (destin ?p ?f)
                     (boarded ?p))
  :effect (and (not (boarded ?p))
               (served ?p)))

(:action up
  :parameters (?f1 - floor ?f2 - floor)
  :precondition (and (lift-at ?f1) (above ?f1 ?f2))
  :effect (and (lift-at ?f2) (not (lift-at ?f1))))

(:action down
  :parameters (?f1 - floor ?f2 - floor)
  :precondition (and (lift-at ?f1) (above ?f2 ?f1))
  :effect (and (lift-at ?f2) (not (lift-at ?f1))))
)
```

Listing 6: Problem description of `Miconic`

```
(define (problem miconic-1)
(:domain miconic)
(:objects
p1 - passenger p2 - passenger f1 - floor f2 - floor f3 - floor
)
(:init
(lift-at f1)(above f1 f2)(above f2 f3)
(origin p1 f1)(origin p2 f2)(destin p1 f3)
(destin p2 f3)(not-boarded p1)(not-boarded p2)
(not-served p1)(not-served p2)
)
(:goal
(and (served p1) (served p2))
)
)
```

Figure 7: Domain and problem descriptions of `Miconic`

Listing 7: Domain description of `Tower of Hanoi`

```
(define (domain hanoi)
(:requirements :strips)
(:predicates (clear ?x)
             (on ?x ?y)
             (smaller ?x ?y))

(:action move
:parameters (?disc ?from ?to)
:precondition (and (smaller ?to ?disc)
                   (on ?disc ?from)
                   (clear ?disc)
                   (clear ?to))
:effect  (and (clear ?from)
              (on ?disc ?to)
              (not (on ?disc ?from))
              (not (clear ?to)))
 ))
```

Listing 8: Problem description of `Tower of Hanoi`

```
(define (problem hanoi-1)
(:domain hanoi)
(:objects
d1 d2 d3 - disc
p1 p2 p3 - peg
)
(:init
(smaller d1 d2)(smaller d1 d3)(smaller d2 d3)
(on d1 p1)(on d2 p1)(on d3 p1)
(clear p2)(clear p3)(clear d1)
)
(:goal
(and (on d1 p3) (on d2 p3) (on d3 p3))
)
)
```

Figure 8: Domain and problem descriptions of `Tower of Hanoi`

Figure 9 shows the `Grippers` domain and a sample problem file in PDDL.

Listing 9: Domain description of `Grippers`

```
(define (domain grippers)
 (:requirements :strips :typing)
 (:types room object robot gripper)
 (:predicates (at-robby ?r - robot ?x - room)
              (at ?o - object ?x - room)
              (free ?r - robot ?g - gripper)
              (carry ?r - robot ?o - object ?g - gripper))

   (:action move
       :parameters  (?r - robot ?from ?to - room)
       :precondition (and  (at-robby ?r ?from))
       :effect (and  (at-robby ?r ?to)
                     (not (at-robby ?r ?from))))

   (:action pick
       :parameters (?r - robot ?obj - object ?room - room ?g - gripper)
       :precondition  (and  (at ?obj ?room) (at-robby ?r ?room) (free ?r ?g))
       :effect (and (carry ?r ?obj ?g)
                    (not (at ?obj ?room))
                    (not (free ?r ?g))))

   (:action drop
       :parameters (?r - robot ?obj - object ?room - room ?g - gripper)
       :precondition  (and  (carry ?r ?obj ?g) (at-robby ?r ?room))
       :effect (and (at ?obj ?room)
                    (free ?r ?g)
                    (not (carry ?r ?obj ?g)))))
```

Listing 10: Problem description of `Grippers`

```
(define (problem grippers-1)
(:domain grippers)
(:objects
robby - robot
ball cube - object
room-a room-b - room
grip-1 grip-2 - gripper
)
(:init
(at-robby robby room-a)(at ball room-a)(at cube room-b)
(free robby grip-1)(free robby grip-2)
)
(:goal
(and (at ball room-b) (at cube room-a))
)
)
```

Figure 9: Domain and problem descriptions of `Grippers`

A.4.7 `Driverlog`

The `Driverlog` planning domain is a transportation system model involving drivers, trucks, and packages. The drivers can drive trucks between locations, load and unload packages from trucks, or walk between

Listing 11: Domain description of `Driverlog`

```
(define (domain driverlog)
  (:requirements :typing)
  (:types location locatable - object
          driver truck obj - locatable )
  (:predicates
                (at ?obj - locatable ?loc - location) (in ?obj1 - obj ?obj - truck)
                (driving ?d - driver ?v - truck) (link ?x ?y - location)
            (path ?x ?y - location) (empty ?v - truck)
)
(:action load-truck
  :parameters
   (?obj - obj ?truck - truck ?loc - location)
  :precondition
   (and (at ?truck ?loc) (at ?obj ?loc))
  :effect
   (and (not (at ?obj ?loc)) (in ?obj ?truck)))

(:action unload-truck
  :parameters
   (?obj - obj ?truck - truck ?loc - location)
  :precondition
   (and (at ?truck ?loc) (in ?obj ?truck))
  :effect
   (and (not (in ?obj ?truck)) (at ?obj ?loc)))

(:action board-truck
  :parameters
   (?driver - driver ?truck - truck ?loc - location)
  :precondition
   (and (at ?truck ?loc) (at ?driver ?loc) (empty ?truck))
  :effect
   (and (not (at ?driver ?loc)) (driving ?driver ?truck) (not (empty ?truck))))

(:action disembark-truck
  :parameters
   (?driver - driver ?truck - truck ?loc - location)
  :precondition
   (and (at ?truck ?loc) (driving ?driver ?truck))
  :effect
   (and (not (driving ?driver ?truck)) (at ?driver ?loc) (empty ?truck)))

(:action drive-truck
  :parameters
   (?truck - truck ?loc-from - location ?loc-to - location ?driver - driver)
  :precondition
   (and (at ?truck ?loc-from)
   (driving ?driver ?truck) (link ?loc-from ?loc-to))
  :effect
   (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))

(:action walk
  :parameters
   (?driver - driver ?loc-from - location ?loc-to - location)
  :precondition
   (and (at ?driver ?loc-from) (path ?loc-from ?loc-to))
  :effect
   (and (not (at ?driver ?loc-from)) (at ?driver ?loc-to)))
)
```

Listing 12: Problem description of `Driverlog`

```
(define (problem driverlog-1)
  (:domain driverlog)
  (:objects
    driver-1 - driver truck-1 - truck obj-1 - obj loc-1 loc-2 loc-3 - location
  )
  (:init
    (at truck-1 loc-1)(at obj-1 loc-2)(at driver-1 loc-3)(link loc-1 loc-2)(link loc-2 loc-3)(path loc-1
        ↪ loc-2)(path loc-2 loc-1)(path loc-2 loc-3)(path loc-3 loc-2)(empty truck-1)
  )
  (:goal
    (and
      (at obj-1 loc-1)(at driver-1 loc-1)(empty truck-1)
    )))
```

Figure 10: Domain and problem descriptions of `Driverlog`

adjacent locations. The trucks can move between locations if they have a driver. The packages can be loaded or unloaded from trucks at any location. The goal is to deliver all the packages to their destinations. The state space and branching factor of this domain can be calculated as follows:

- The state space of the `Driverlog` domain is $O(L^{(D+T+P)} * K^P * D * T * 2^T)$ because the following factors determine each state:
  - The location of each driver, truck, and package (L possibilities for each of the $D+T+P$ entities, where L is the number of locations)
  - The destination of each package (K possibilities for each of the P packages, where K is the number of possible destinations)
  - The driver of each truck (D possibilities for each of the T trucks, where D is the number of drivers)
  - The truck of each driver (T possibilities for each of the D drivers, where T is the number of trucks)
  - The subset of trucks that are loaded with packages (2^T possibilities, where T is the number of trucks)
- The branching factor of the `Driverlog` domain is $O(L * (D + T + P + DT + TD))$ because, at each state, five types of actions can be applied to any entity: drive, walk, load, unload, or do nothing. However, not all actions are applicable at all times. For example, a driver can only drive a truck at the exact location, and a package can only be loaded if it is clear. Therefore, the number of applicable actions is at most $L * (D + T + P + DT + TD)$, where L is the number of locations.

Figure 10 shows the `Driverlog` domain and a sample problem file in PDDL.

A.5 Visualization of compact form

For fine-tuning LLMs, we make use of the compact form. Figure 11 shows the compact form representation of the PDDL problems for `blocksworld` domain from the training dataset as an example. The implemented python code to perform the conversion of PDDL domain and problem files to compact form is given in Listing 13.

Compact form for `Blocksworld`

```
<GOAL>on b1 b3, on b2 b4, ontable b3, on b4 b1, on b5 b2, clear b5<INIT>handempty, on b1 b3, on b2
b1, on b3 b5, on b4 b2, clear b4, ontable b5<ACTION> pick-up <PRE> clear x, ontable x, handempty
<EFFECT> not ontable x, not clear x, not handempty, holding x <ACTION> put-down <PRE> holding
x <EFFECT> not holding x, clear x, handempty, ontable x <ACTION> stack <PRE> holding x, clear y
<EFFECT> not holding x, not clear y, clear x, handempty, on x y <ACTION> unstack <PRE> on x y,
clear x, handempty <EFFECT> holding x, clear y, not clear x, not handempty, not on x y
```

Figure 11: Example of compact form representation obtained from PDDL domain and problem files for the `blocksworld` domain considered in the planning dataset. This example is from the training set.

Listing 13: Python code for converting PDDL domain and problem files to compact form

```python
import re
import sys

def find_parens(s):
  """Finds all parentheses in the string ‘s‘ and returns a dictionary mapping
  the start index of each parenthesis to its end index.

  Args:
    s: A string.

```

```
  Returns:
    A dictionary mapping the start index of each parenthesis to its end index.
  """

  toret = {}
  pstack = []
  flag = 0
  for i, c in enumerate(s):

    if flag == 1 and len(pstack) == 0:
      return toret

    if c == '(':
      pstack.append(i)
      flag = 1
    elif c == ')':
      toret[pstack.pop()] = i

  return toret

def prompt_action(data):
  """Generates a string representation of the action in the given data.

  Args:
    data: A string containing the definition of an action.

  Returns:
    A string representation of the action.
  """

  # Get the name of the action.

  action_name = data.split('\n')[0].split('␣')[1].lower()

  # Get the precondition of the action.

  precondition = data[data.find(':precondition') + 14:data.find(':effect')]
  precondition_parens = find_parens(precondition)
  precondition_strings = []
  for start, end in precondition_parens.items():
    precondition_strings.append(precondition[start:end + 1].strip('()?'))

  # Get the effect of the action.

  effect = data[data.find(':effect') + 10:]
  effect_parens = find_parens(effect)
  effect_strings = []
  for start, end in effect_parens.items():
    effect_strings.append(effect[start:end + 1].strip('()?'))

  # Return a string representation of the action.

  return f'<ACTION>␣{action_name}␣{",␣".join(precondition_strings)}␣{",␣".join(
      ↪ effect_strings)}␣</ACTION>'


def prompt_problem(data):
  """Generates a string representation of the problem in the given data.
```

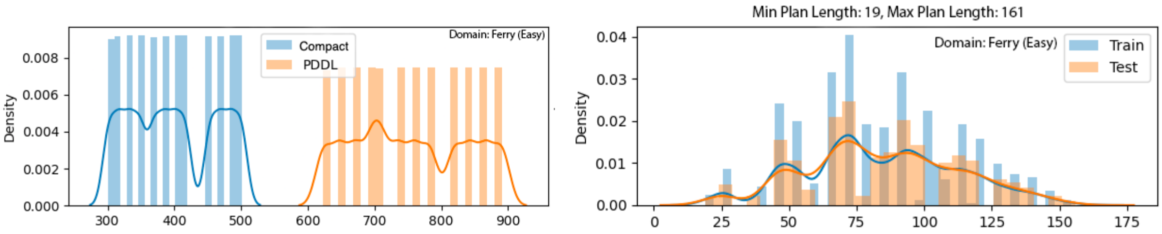
```
  Args:
    data: A string containing the definition of a problem.

  Returns:
    A string representation of the problem.
  """

  # Get the initial state of the problem.

  init = data[data.find(’(:init’) + 8:data.find(’(:goal’)]
  init_parens = find_parens(init)
  init_strings = []
  for start, end in init_parens.items():
    init_strings.append(init[start:end + 1].strip(’()?’))

  # Get the goal state of the problem.

  goal = data[data.find(’(:goal’) + 7:]
  goal_parens = find_parens(goal)
  goal_strings = []
  for start, end in goal_parens.items():
    goal_strings.append(goal[start:end + 1].strip(’()?’))

  # Return a string representation of the problem.

  return f’<INIT>␣{",␣".join(init_strings)}␣</INIT>␣<GOAL>␣{",␣".join(goal_strings
     ↪ )}␣</GOAL>’


def get_prompt(domain_file, problem_file):
  """Generates a string representation of the domain and problem files.

  Args:
    domain_file: The name of the domain file.
    problem_file: The name of the problem file.

  Returns:
    A string representation of the domain and problem files.
  """

  # Read the domain file.

  with open(domain_file, ’r’) as f:
    domain_data = f.read()

  # Get the name of the domain.

  domain_name = re.findall(r’(?<=domain␣)\w+’, domain_data)[0]
```

A.6 Token lengths for PDDL and Compact representation

Figure 12a exemplifies the variance in token lengths when utilizing PDDL versus the Compact representation for the `Ferry` domain

A.7 Visualization of prompting techniques

We prompt the LLMs using zero-shot, one-shot, and Chain of Thought (CoT) prompting strategies. The prompts for the planning problems in our dataset, which spans six domains, are obtained using AutoPlanBench



(a) Token lengths for PDDL vs Compact Form

(b) Plan length distribution of train and test sets

Figure 12: (a) Token length for the `Ferry` domain. Using the compact form achieves an average 40% reduction in token length compared to PDDL. (b) Plan length distribution of both train and test sets are carefully chosen to be similar and within the contextual limitation of the LLMs considered in this paper. Token lengths are in the x-axis and number of problems/plans per token length are in y-axis.

(Stein & Koller, 2023). An example of the natural language representation for a planning problem from the `blocksworld` domain, utilizing zero-shot, one-shot, and CoT prompting, is depicted in Figure 14, 13, 15, 16. AutoPlanBench also facilitates the parsing of output generated by the LLMs, particularly for CoT prompting, into a list of action sequences. These sequences are comparable to those generated by a symbolic planner (e.g., FastDownward) and can be evaluated using the VAL tool.

A.8 Experimental Results

Table 4 in the main paper is presented with non-contributory rows removed. In contrast, Table 7 includes all rows.

Table 7: Evaluation of plan generation capabilities of Causal, Seq2Seq, and Masked LLMs. For each model, we report the percentage of satisficing plans (Sat. Plans), the percentage of optimal plans (Opt. Plans), and the degree of correctness (Deg. Corr.)

Model	Input Representation	Method	Sat. Plans (%)			Opt. Plans (%)			Deg. Corr.		
			E	M	H	E	M	H	E	M	H
GPT-4 (no-vision) Causal	Compact	Zero-shot	1.6	0	0	0	0	0	0.04	0	0
		One-shot	7.21	2.66	1.78	2.52	0.07	0	0.05	0.02	0.01
		CoT	21.28	16.74	11.33	10.74	8.21	4.22	0.27	0.25	0.21
		FT	-	-	-	-	-	-	-	-	-
	NL	Zero-shot	15.72	10.41	7.78	7.29	3.84	1.25	0.21	0.19	0.07
		One-shot	24.78	22.91	22.78	7.31	3.05	1.25	0.28	0.17	0.07
		CoT	35.02	34.76	32.93	12.88	10.14	9.21	0.41	0.41	0.37
		FT	-	-	-	-	-	-	-	-	-
	PDDL	Zero-shot	8.78	6.43	0	3.92	0	0	0.27	0.21	0
		One-shot	10.82	6.90	3.31	7.23	2.84	1.21	0.32	0.17	0.04
		CoT	22.18	18.55	18.97	9.87	5.21	3.22	0.27	0.18	0.19
		FT	-	-	-	-	-	-	-	-	-
code-davinci-03 Causal	Compact	Zero-shot	4.78	0	0.97	0	0	0	0.21	0	0.04
		One-shot	11.54	11.18	10.14	3.27	1.94	0.07	0.21	0.17	0.11
		CoT	10.77	4.18	1.92	3.74	1.86	0	0.35	0.31	0.27
		FT	-	-	-	-	-	-	-	-	-
	NL	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	8.23	7.66	5.21	0	0	0	0.04	0.04	0.04
		CoT	11.29	9.07	5.22	2.84	1.45	1.01	0.02	0.01	0.01
		FT	-	-	-	-	-	-	-	-	-
	PDDL	Zero-shot	17.42	11.77	4.38	8.23	6.11	1.84	0.38	0.27	0.21
		One-shot	43.52	37.48	31.89	17.57	15.85	14.69	0.57	0.38	0.31
		CoT	27.71	19.55	19.97	20.11	14.27	11.25	0.31	0.20	0.23
		FT	-	-	-	-	-	-	-	-	-

Continued on next page

Table 7: Evaluation of plan generation capabilities of LLMs. (continued)

Model	Input Representation	Method	Sat. Plans (%)			Opt. Plans (%)			Deg. Corr.		
			E	M	H	E	M	H	E	M	H
T5 Seq2Seq	Compact	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	0	0	0	0	0	0	0	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	0.11	0	1.16	0.11	0	0.36	0.02	0	0.03
	NL	Zero-shot	0.16	0	0	0	0	0	0.01	0	0
		One-shot	1.28	1.01	0	0.17	0	0	0.01	0.01	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	21.56	18.56	11.23	9.02	2.89	1.01	0.28	0.17	0.06
	PDDL	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	0	0	0	0	0	0	0	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	3.78	0.72	0	0	0	0	0.21	0.06	0
CodeT5 Seq2Seq	Compact	Zero-shot	2.7	0.6	0	1.73	0.6	0	0.07	0	0
		One-shot	1.02	0	0	0.03	0	0	0.01	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	97.57	92.46	89.54	86.21	90.36	66.71	0.99	0.95	0.95
	NL	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	0	0	0	0	0	0	0	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	3.82	3.01	3.03	0	0	0	0.03	0.03	0.03
	PDDL	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	0	0	0	0	0	0	0	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	43.89	37.41	30.42	40.84	31.55	21.02	0.61	0.52	0.47
BERT Masked	Compact	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	0	0	0	0	0	0	0	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	10.89	8.42	6.77	3.57	0	0	0.28	0.07	0.01
	NL	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	0	0	0	0	0	0	0	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	1.78	0.91	0	0.07	0	0	0.01	0.01	0
	PDDL	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	0	0	0	0	0	0	0	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	1.21	0	0	0	0	0	0.01	0	0
StarEncoder Masked	Compact	Zero-shot	4.21	1.79	0.91	1.44	0.07	0	0.12	0.02	0.01
		One-shot	11.02	5.42	1.71	2.08	0	0	0.18	0.07	0.02
		CoT	-	-	-	-	-	-	-	-	-
		FT	19.44	10.25	5.69	8.92	2.55	2.07	0.25	0.17	0.07
	NL	Zero-shot	0	0	0	0	0	0	0	0	0
		One-shot	4.05	0	0	1.11	0	0	0.05	0	0
		CoT	-	-	-	-	-	-	-	-	-
		FT	4.11	1.97	0	0.55	0	0	0.06	0.01	0
	PDDL	Zero-shot	7.91	6.22	5.97	5.01	2.11	1.94	0.18	0.08	0.07
		One-shot	6.91	5.28	3.97	3.01	1.04	0.24	0.17	0.08	0.07
		CoT	-	-	-	-	-	-	-	-	-
		FT	21.01	15.66	9.89	17.61	9.78	3.88	0.37	0.21	0.18

Zero-shot Prompt

You are an assistant for giving instructions to successfully complete small tasks.
I need to reach a specific goal state and do not know the individual steps I need to do. Please instruct me how to complete my task.
I can only use objects that are observable in the situation.

My current situation is as follows: There are 4 objects: object_0, object_1, object_2, object_3
Currently, object_0 is on the table, object_1 is clear, object_1 is on object_2, object_2 is on object_3, object_3 is on object_0, the hand is empty

My task is to execute actions until reaching my goal. My goal is that in the end object_1 is on object_2, object_3 is on object_0, object_2 is on table, object_0 is on table.

I can carry out the following actions:
pick up an object A from the table
put down an object A
stack an object A on top of an object B
unstack an object A from an object B

I have the following restrictions on my actions:
I can only pick up an object A from the table if it is the case that A is clear and A is on the table and the hand is empty
I can only put down an object A if it is the case that A is being held
I can only stack an object A on top of an object B if it is the case that B is clear and A is being held
I can only unstack an object A from an object B if it is the case that A is on B and A is clear and the hand is empty

The actions have the following effects on the state:
Once I pick up an object A from the table, it is the case that A is being held
Once I pick up an object A from the table, it is not the case anymore that A is clear and it is not the case anymore that A is on the table and it is not the case anymore that the hand is empty
Once I put down an object A, it is the case that A is clear and the hand is empty and A is on the table
Once I put down an object A, it is not the case anymore that A is being held
Once I stack an object A on top of an object B, it is the case that the hand is empty and A is clear and A is on B
Once I stack an object A on top of an object B, it is not the case anymore that B is clear and it is not the case anymore that A is being held
Once I unstack an object A from an object B, it is the case that A is being held and B is clear
Once I unstack an object A from an object B, it is not the case anymore that A is on B and it is not the case anymore that A is clear and it is not the case anymore that the hand is empty

Remember: My goal is that in the end object_1 is on object_2, object_3 is on object_0.
As output, only provide the plan which is a list of actions and their associated objects having the following structure ["action_name object_name", "action_name object_name"]. DO NOT OUT ANY OTHER INFORMATION.

GPT-4

["unstack object_1 from object_2", "put down object_1", "unstack object_2 from object_3", "put down object_2", "unstack object_3 from object_0", "put down object_3", "stack object_3 on object_0", "pick up object_1 from the table", "stack object_1 on object_2"]

Figure 13: An example of zero-shot prompting approach for `Blocksworld` domain and the output obtained from GPT-4

One-shot Prompt

You are an assistant for giving instructions to successfully complete small tasks.
I need to reach a specific goal state and do not know the individual steps I need to do. Please instruct me how to complete my task.
I can only use objects that are observable in the situation.

Exemplar:
My current situation is as follows: There are 2 objects: object_0, object_1
Currently, object_0 is clear, object_1 is clear, the hand is empty.
My goal is that in the end object_0 is on object_1
Plan: ["stack object_0 on object_1"]

My current situation is as follows: There are 4 objects: object_0, object_1, object_2, object_3 Currently, object_0 is on the table, object_1 is clear, object_1 is on object_2, object_2 is on object_3, object_3 is on object_0, the hand is empty

My task is to execute actions until reaching my goal. My goal is that in the end object_1 is on object_2, object_3 is on object_0, object_2 is on table, object_0 is on table.

I can carry out the following actions:
pick up an object A from the table
put down an object A
stack an object A on top of an object B
unstack an object A from an object B

I have the following restrictions on my actions:
I can only pick up an object A from the table if it is the case that A is clear and A is on the table and the hand is empty
I can only put down an object A if it is the case that A is being held
I can only stack an object A on top of an object B if it is the case that B is clear and A is being held
I can only unstack an object A from an object B if it is the case that A is on B and A is clear and the hand is empty

The actions have the following effects on the state:
Once I pick up an object A from the table, it is the case that A is being held
Once I pick up an object A from the table, it is not the case anymore that A is clear and it is not the case anymore that A is on the table and it is not the case anymore that the hand is empty
Once I put down an object A, it is the case that A is clear and the hand is empty and A is on the table
Once I put down an object A, it is not the case anymore that A is being held
Once I stack an object A on top of an object B, it is the case that the hand is empty and A is clear and A is on B
Once I stack an object A on top of an object B, it is not the case anymore that B is clear and it is not the case anymore that A is being held
Once I unstack an object A from an object B, it is the case that A is being held and B is clear
Once I unstack an object A from an object B, it is not the case anymore that A is on B and it is not the case anymore that A is clear and it is not the case anymore that the hand is empty

Remember: My goal is that in the end object_1 is on object_2, object_3 is on object_0.
As output, only provide the plan which is a list of actions and their associated objects having the following structure ["action_name object_name", "action_name object_name"]. DO NOT OUT ANY OTHER INFORMATION.

GPT-4

["unstack object_3 from object_0", "put down object_3", "unstack object_2 from object_1", "put down object_2", "unstack object_1 from object_2", "put down object_1", "pick up object_1 from the table", "stack object_1 on object_2", "pick up object_3 from the table", "stack object_3 on object_0"]

Figure 14: An example of one-shot prompting approach for `Blocksworld` domain and the output obtained from GPT-4

Chain-of-Thought Prompt

You are an assistant for giving instructions to successfully complete small tasks.
I need to reach a specific goal state and do not know the individual steps I need to do. Please instruct me how to complete my task.
I can only use objects that are observable in the situation.

My current situation is as follows: There are 4 objects: object_0, object_1, object_2, object_3 Currently, object_0 is on the table, object_1 is clear, object_1 is on object_2, object_2 is on object_3, object_3 is on object_0, the hand is empty

My task is to execute actions until reaching my goal. My goal is that in the end object_1 is on object_2, object_3 is on object_0, object_2 is on table, object_0 is on table.

I can carry out the following actions:
pick up an object A from the table
put down an object A
stack an object A on top of an object B
unstack an object A from an object B

I have the following restrictions on my actions:
I can only pick up an object A from the table if it is the case that A is clear and A is on the table and the hand is empty
I can only put down an object A if it is the case that A is being held
I can only stack an object A on top of an object B if it is the case that B is clear and A is being held
I can only unstack an object A from an object B if it is the case that A is on B and A is clear and the hand is empty

The actions have the following effects on the state:
Once I pick up an object A from the table, it is the case that A is being held
Once I pick up an object A from the table, it is not the case anymore that A is clear and it is not the case anymore that A is on the table and it is not the case anymore that the hand is empty
Once I put down an object A, it is the case that A is clear and the hand is empty and A is on the table
Once I put down an object A, it is not the case anymore that A is being held
Once I stack an object A on top of an object B, it is the case that the hand is empty and A is clear and A is on B
Once I stack an object A on top of an object B, it is not the case anymore that B is clear and it is not the case anymore that A is being held
Once I unstack an object A from an object B, it is the case that A is being held and B is clear
Once I unstack an object A from an object B, it is not the case anymore that A is on B and it is not the case anymore that A is clear and it is not the case anymore that the hand is empty

Here are some examples
My goal is that in the end object_2 is on object_1
I: My current initial situation is as follows:
There are 4 objects: object_0, object_1, object_2, object_3
Currently, object_0 is clear, object_0 is on the table, object_1 is being held, object_2 is clear, object_2 is on the table, object_3 is clear, object_3 is on the table
You:
Think: [TODO: ADD REASONING THOUGHT]
Instruction: put down object object_1
I: I put down object object_1.
You:
Think: [TODO: ADD REASONING THOUGHT]
Instruction: pick up object object_2 from the table
I: I pick up object object_2 from the table.
You:
Think: [TODO: ADD REASONING THOUGHT]
Instruction: stack object object_2 on top of object object_1
I: I stack object object_2 on top of object object_1.
You:
Think: object_2 is on object_1
Instruction: You are finished.
I: Great!

Please provide me a step-by-step instruction for how to complete my task. Remember: My goal is that in the end object_1 is on object_2, object_3 is on object_0.
Please provide each step in a new line.

Figure 15: An example of chain-of-thought prompting approach for `Blocksworld` domain, Part 1

GPT-4

Think: First, I need to clear object_0 so that it can be moved. To do this, I need to unstack object_3 from object_0.
Instruction: unstack object_3 from object_0

Think: Now that object_0 is clear, I can pick it up.
Instruction: pick up object_0 from the table

Think: Now that I'm holding object_0, I need to place it on object_2. But first, I need to clear object_2. To do this, I need to unstack object_1 from object_2.
Instruction: unstack object_1 from object_2

Think: Now that object_2 is clear, I can stack object_0 on it.
Instruction: stack object_0 on top of object_2

Think: Now that object_0 is on object_2, I need to place object_3 on object_0. But first, I need to pick up object_3.
Instruction: pick up object_3 from the table

Think: Now that I'm holding object_3, I can stack it on object_0.
Instruction: stack object_3 on top of object_0

Think: Now object_0 is on object_2 and object_3 is on object_0, which is the final goal.
Instruction: You are finished.

Figure 16: An example of chain-of-thought prompting approach for `Blocksworld` domain, Part 2