Classical Planning with LLM-Generated Heuristics: Challenging the State of the Art with Python Code

Augusto B. Corrêa
University of Oxford
United Kingdom

André G. Pereira
Federal University of Rio Grande do Sul
Brazil

Jendrik Seipp
Linköping University
Sweden

Abstract

In recent years, large language models (LLMs) have shown remarkable performance in many problems. However, they fail to plan reliably. Specialized attempts to improve their planning capabilities still produce incorrect plans and fail to generalize to larger tasks. Furthermore, LLMs designed for explicit "reasoning" fail to compete with automated planners while increasing computational costs, which reduces one of the advantages of using LLMs. In this paper, we show how to use LLMs to always generate correct plans, even for out-of-distribution tasks of increasing size. For a given planning domain, we ask an LLM to generate several domain-dependent heuristic functions in the form of Python code, evaluate them on a set of training tasks with a greedy best-first search, and choose the best one. The resulting LLM-generated heuristic functions solve substantially more unseen out-of-distribution test tasks than end-to-end LLM planning, particularly for non-reasoning LLMs. Moreover, they also solve many more tasks than state-of-the-art domain-independent heuristics for classical planning, and are competitive with the strongest learning algorithm for domain-dependent planning. These results are impressive given that our implementation is based on a Python planner and the baselines all build upon highly optimized C++ code. In some domains, the LLM-generated heuristics expand fewer states than the baselines, showing that they are not only efficiently computable but also more informative than the state-of-the-art heuristics. Overall, our results show that sampling a set of planning heuristic functions can significantly improve the planning capabilities of LLMs.

1 Introduction

Classical planning is a fundamental problem in Artificial Intelligence (AI), with applications ranging from robotics to computational chemistry [25]. Given the *initial state* of the world, a description of the *goal*, and a set of deterministic *actions* that can be executed in a fully-observable environment, the task is to find a sequence of actions transforming the initial state into a state that satisfies the goal. Nowadays, most classical *planners* rely on *heuristic search* algorithms to find plans [6, 39, 35, 55, 45, 75, 65]. The efficiency of these planners depends on the quality of the *heuristic functions* that estimate the cost of reaching the goal from a given state. Traditionally, these heuristics have been either *domain-independent*, offering generality at the expense of accuracy; *manually crafted* for specific domains, requiring significant human effort and expertise; or *learned on a per-domain basis*, incurring costs for training a new heuristic whenever we want to use a new domain.

In this work, we propose a new way of producing heuristics: we use LLMs to automatically generate domain-dependent heuristic functions for classical planning. Our hypothesis is that LLMs, given sufficient context and examples, can generate heuristic functions that outperform generic domain-independent heuristics.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

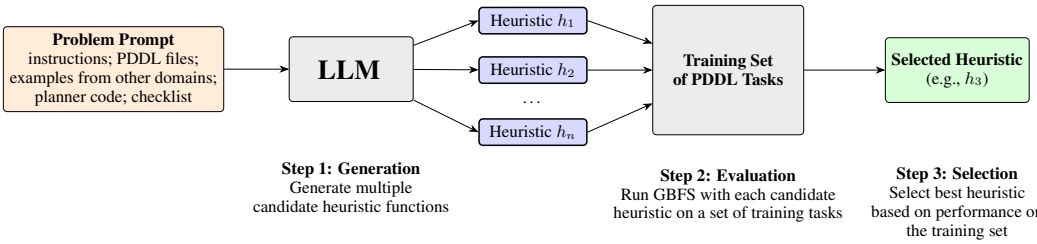


Figure 1: Our pipeline for generating domain-dependent heuristics with LLMs: we prompt the LLM n times to generate n candidate heuristics, evaluate each of these heuristics on a set of training tasks and choose the strongest one.

Our overall pipeline is much simpler than previous work: we simply pass to an LLM the domain description, example planning tasks, example domain-dependent heuristics for other domains, and the relevant planner code. Then we request that the LLM generates a heuristic for the given domain. We specifically request that the LLM *generates the code*, in Python, to compute the heuristic. We execute the same prompt n times to obtain a pool of n candidate heuristics, evaluate each of them on a training set, and select the best one. Figure 1 shows the overall pipeline. This discards the necessity of a back-and-forth communication between the planner and the LLM, making the overall procedure straightforward. Our approach generates a constant number of heuristics *per domain*, and then uses the selected heuristic for any new task of this domain. This drastically reduces the costs for LLM inference compared to invoking an LLM for each task in a domain.

We implement our pipeline on top of Pyperplan [1] as a proof of concept, and then evaluate the generated heuristics on the domains of the Learning Track of the International Planning Competition (IPC) 2023 [73]. The LLM-generated heuristic functions solve significantly more tasks than LLMs prompted to generate plans end-to-end, and this difference is especially pronounced for non-reasoning LLMs. The heuristics also outperform state-of-the-art heuristics, such as h^{FF} [39] in terms of solved tasks and are competitive in the number of required state expansions. Although Pyperplan is much slower than state-of-the-art planners [35, 45, 63] due to its Python implementation, our method still outperforms h^{FF}, even when using the standard C++ implementation available in Fast Downward [35], as well as the strongest learning-based domain-dependent planner [11], which is also built on Fast Downward. This is an impressive result, as this implementation is a cornerstone of most of the state-of-the-art planners in the literature.

2 Background

We consider *classical planning*, where a single agent applies deterministic actions in a fully-observable, discrete environment. Classical planning tasks are usually described using the Planning Domain Definition Language (PDDL) [47, 33]. To understand our work, an informal description of a fragment of PDDL is sufficient. We introduce it alongside examples from a simple Logistics domain.

A PDDL task consists of a set of objects (e.g., representing vehicles, packages, locations); a set of predicates representing relations between these objects (e.g., using the `at` predicate, the *ground atom* `(at car1 city2)` represents that object `car1` is `at` `city2`); a set of actions that change the relations (e.g., driving a car changes its location); an initial state which is a set of ground atoms (e.g., where all packages and vehicles are initially and which locations are connected) and a goal description that lists the ground atoms that must hold at the end of a plan (e.g., the desired locations of all packages). Applying an action changes the current state of the environment by removing or adding ground atoms. PDDL tasks are commonly separated into a *domain* and an *task* part, where the domain part holds the common actions and predicates, while the task part describes the specific objects, initial state and the goal. The two parts are typically represented as two separate files.

The objective of a *planner* is to find a sequence of actions, called a *plan*, that leads from the initial state to a state where all goal atoms hold. Most planners nowadays use *state-space search* to find a plan. The search is usually guided by a *heuristic* function h which maps each state s to a value $h(s) \in \mathbb{R}_0^+ \cup \infty$ that estimates the cost of reaching a goal state from s [51]. In heuristic search algorithms—such as A*[32], weighted-A*[52], and greedy best-first search [19]—this heuristic

guides the search towards promising states and thereby reduces the search effort. The performance of a planner is heavily influenced by the accuracy and computational efficiency of the heuristic function.

In our work, we assume that all actions have cost one, and we consider *satisficing planning*, where any plan is acceptable, irrespective of its length or cost. We focus on planners that use greedy best-first search (GBFS). While there are lots of search improvements that one could evaluate on top of GBFS [e.g., 39, 45, 54, 57], we limit ourselves to "pure" GBFS planners as this is the most commonly used version in the classical planning literature [e.g., 18, 22, 38].

3 Using LLMs to Generate Heuristics for Classical Planning

In our pipeline, we give as input to an LLM the PDDL description of our target domain together with some additional information (described below). Then we ask the LLM for a domain-dependent heuristic function for the given domain, implemented in Python, which we then inject into the Pyperplan planner [1]. We choose Python because LLMs generate correct code for Python more often than for other languages [44], and because code injection is simple in Python.

We send n identical requests to the LLM with the same prompt, collect all returned heuristic functions $h_1, \ldots, h_n$, and then evaluate them on the training tasks. Then, we automatically select the best heuristic $h_{\text{best}} \in \{h_1, \ldots, h_n\}$ and use it in the test set evaluation (see below for details on this selection). Figure 1 shows the graphical representation of our method.

When used for planning, LLMs often produce incorrect plans [79, 80]. In contrast, our pipeline ensures that all found plans are correct: the greedy best-first search algorithm only produces correct plans, and the heuristic created by the LLM only influences how efficiently such a solution is found.

Prompt Our prompt instructs the LLM to generate a domain-dependent heuristic for a given domain D and provides some advice, such as that the heuristic should minimize the number of expanded states and balance accuracy with computational efficiency. It then includes the following file contents:

1. the PDDL domain file of domain D
2. the smallest and the largest PDDL tasks of domain D in the training set
3. for each of the two example domains Gripper and Logistics [47]: the PDDL domain file, a task file and a domain-dependent heuristic implemented in Pyperplan
4. an example of how a state of domain D is represented in Pyperplan
5. an example of how the static information of domain D is represented in Pyperplan
6. the Python code from Pyperplan for representing a planning task and an action
7. a checklist of common pitfalls

Items 1 and 2 provide the context about the domain D that we are interested in. Item 3 illustrates the Python implementation of domain-dependent heuristics using Pyperplan's interface. For Gripper, we provide a Python function computing the perfect heuristic as input, while for Logistics, we encode the simple "single visit and load/unload counting heuristic" by Paul et al. [50]. These functions show the LLM what a heuristic could do and illustrate how to manipulate the available data. Items 4–6 give more context about Pyperplan. Last, the checklist consists of tips based on our own observations of LLM responses. Appendix A shows the complete prompt used for the Blocksworld domain [47].

Heuristic Function Selection We prompt the model n times with the input above to generate n heuristics. For each task in the training set, we then run the n heuristics within a GBFS for at most five minutes. Then, we select the heuristic that solves the largest number of tasks from the training set. If there is a tie, we choose the one minimizing the accumulated *agile score* [1] over the training set. We use the term *training set* to follow IPC terminology. However, our approach does not involve any training. We merely use this set of tasks to evaluate the generated heuristics and select the best one.

[1]The agile score is a common metric from IPCs and awards heuristics that lead to finding plans quickly. If the search needs less than 1 second, then the score is 1. If the search runs out of time (in our case, 300 seconds) the score is 0. Intermediate values are interpolated with the logarithmic function $1 - \frac{\log(t)}{\log(300)}$, where t is the run time (in seconds) of the search. The accumulated score is the sum over all training tasks.

Table 1: Size of training and testing tasks for each domain based on their main parameters. Parameters by domain: n blocks in Blocksworld, c children in Childsnack, t tiles in Floortile, p passengers in Miconic, r rovers in Rovers, b boxes in Sokoban, s spanners in Spanner, and v vehicles in Transport.

Domain	Training	Testing
Blocksworld	$n \in [2, 29]$	$n \in [5, 488]$
Childsnack	$c \in [1, 10]$	$c \in [4, 292]$
Floortile	$t \in [2, 30]$	$t \in [12, 980]$
Miconic	$p \in [1, 10]$	$p \in [1, 485]$
Rovers	$r \in [1, 4]$	$r \in [1, 30]$
Sokoban	$b \in [1, 4]$	$b \in [1, 79]$
Spanner	$s \in [1, 10]$	$n \in [1, 487]$
Transport	$v \in [1, 7]$	$n \in [3, 50]$

4 Experimental Results

For running our experiments, we use Downward Lab [64] on AMD EPYC 7742 processors running at 2.25 GHz. As mentioned, we use Pyperplan [1] for all our configurations. This allows us to evaluate the different heuristics (domain-independent and LLM-generated ones) in a single framework. We use PyPy 7.3.9 to run Pyperplan, as it proved to be slightly faster than CPython. The source code and experimental data are publicly available online [14].

In the training phase, we limit each run to 5 minutes and 8 GiB. In the testing phase, each run is limited to 30 minutes and 8 GiB, following recent International Planning Competitions [73].[2] To diversify the pool of generated heuristics, we increase the temperature parameter of the models to 1.0 [7].

We use the domains and training/test tasks from the IPC 2023 Learning Track to generate and evaluate heuristics. However, since Pyperplan does not support two of these ten domains (Ferry and Satellite), we exclude them from our experiments. The resulting test set has 90 tasks for each of the 8 domains. The distribution of tasks in the training and test sets differs a lot: the test tasks are generally much larger than the training ones. Table 1 shows the distributions of parameters. In addition to size differences, tasks may also vary in structure. For example, Sokoban mazes can be arranged in different layouts. The full details about the task sets are available online [62].

We also include experiments on novel domains that were not seen during the training of the LLMs. This helps us to identify whether our method is robust to new domains, and also addresses the potential concern that the LLMs are retrieving memorized heuristics instead of generating them by reasoning about the domain description.

Generating Heuristics To generate the heuristics, we use two families of LLMs: Gemini [27, 28], with the models Gemini 2.0 Flash (stable release 001) and Gemini 2.0 Flash Thinking (version 01-21); and DeepSeek [16, 17], with the models DeepSeek V3 (version 0324), DeepSeek R1 Distill Qwen 14B, and DeepSeek R1. We include the distilled version to Qwen 14B [3] to evaluate the impact of smaller models in our pipeline. We had free API access to Gemini 2.0 models and ran R1 Distill locally. The total API costs for generating all V3 and R1 heuristics were $0.25 and $6.12 (USD).

We prompt the LLM n times and receive n different heuristic functions. But how large should n be? We ran an experiment with Gemini 2.0 Flash to evaluate this. For all domains, the biggest increase in average *coverage* (i.e., number of solved tasks) results from going from 1 to 5 heuristics, and after that, we see diminishing returns. In six of the eight domains, using $n = 25$ is enough to consistently find heuristics solving the entire training set. In two domains, Childsnack and Floortile, coverage increased for $n > 25$, but only slightly. Due to these results, we set $n = 25$ for all experiments below.

[2]We used a disjoint set of ten domains from the Autoscale benchmark set [76] for exploratory experiments while developing our pipeline. This split allowed us to test different prompts, hyperparameters, and models without the risk of overfitting to the IPC 2023 benchmark set or causing some LLM APIs to cache our prompts.

Table 2: Number of solved tasks when using LLMs for end-to-end planning compared to a greedy best-first search within Pyperplan using the blind heuristic h^0, h^{FF} and our LLM-generated heuristics.

	End-to-End				Pyperplan						
	Gemini 2.0		DeepSeek				**Gemini 2.0**		**DeepSeek**		
Domain	–	Think.	V3	R1	h^0	h^{FF}	**–**	**Think.**	**V3**	**R1 D.**	**R1**
Blocksworld (90)	2	40	2	17	6	24	35	37	45	34	66
Childsnack (90)	3	59	12	40	9	17	32	14	55	16	22
Floortile (90)	0	0	0	0	1	10	4	8	3	3	4
Miconic (90)	6	21	10	24	30	74	90	88	64	30	90
Rovers (90)	0	5	0	10	12	28	32	39	34	32	32
Sokoban (90)	0	14	0	8	24	31	31	32	31	24	30
Spanner (90)	6	39	21	47	30	30	30	30	69	30	70
Transport (90)	2	24	3	28	8	29	42	57	42	45	59
Sum (720)	19	202	48	174	120	243	296	305	343	214	373

Comparisons to Domain-Independent Heuristics in Pyperplan We now compare the LLM-generated heuristics to two baselines: breadth-first search (h^0), which uses no heuristic guidance,[3] and GBFS with the h^{FF} heuristic [39], which is one of the most commonly used heuristics for satisficing planning [e.g., 8, 13, 26]. These two baselines are also implemented in Pyperplan, which allows us to evaluate exactly the impact of the generated heuristics. As the right part of Table 2 shows, all LLM-generated heuristics outperform h^0 and h^{FF} regarding total coverage, except for the distilled version of DeepSeek R1. In all but one domain (Floortile), the best performing heuristic is an LLM-generated one.

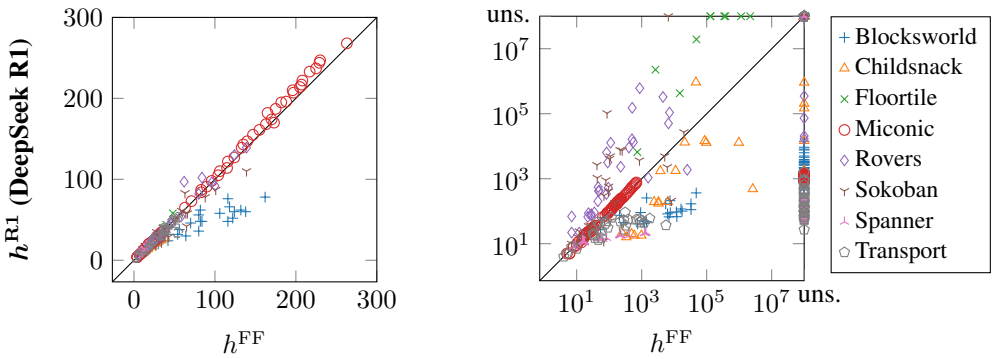
DeepSeek R1 heuristics have the highest coverage with 373 solved tasks, while DeepSeek V3 places second with 343 solved tasks. Gemini 2.0 Flash Thinking solves 68 fewer tasks (total 305). The best baseline h^{FF} solves only 234 tasks. The selected DeepSeek R1 heuristic is particularly impressive in the Blocksworld domain, where it solves almost 40% more tasks as the second best heuristic (DeepSeek V3).

The non-reasoning models—Gemini 2.0 Flash and DeepSeek V3—perform worse than their reasoning counterparts. DeepSeek R1 Distill Qwen 14B heuristics are the only LLM-based models that underperform in comparison to h^{FF}. However, this is mostly due to its low coverage in the Miconic domain. In fact, DeepSeek R1 Distill Qwen 14B outperforms h^{FF} in 3 domains, while being outperformed in 4. This shows that the smaller distilled models might be competitive with existing heuristics in some domains.

Figure 2a compares the plan lengths obtained with h^{FF} and the heuristics generated by DeepSeek R1. In general, the two methods yield plans with similar lengths. The only domains where one approach has a clear edge are Blocksworld, where the DeepSeek R1 plans are consistently shorter, and Miconic, where DeepSeek R1 plans get longer than h^{FF} plans as the tasks get larger.

Last, we compare the informativeness of the traditional and the LLM-generated heuristics by inspecting the number of states expanded during the search. Ideally, a heuristic only expands states traversed by a plan. Figure 2b compares expansions between h^{FF} and DeepSeek R1 heuristics. The results vary by domain: DeepSeek R1 has an edge in Blocksworld, Spanner, Transport and in most of the Childsnack tasks, whereas h^{FF} is more informative in Floortile, Rovers and Sokoban. In all the domains where DeepSeek R1 expands fewer states than h^{FF}, it also solves many more tasks than h^{FF}. On the flip side, the only domain where h^{FF} expands fewer states and solves many more tasks than DeepSeek R1 is Floortile. In the Rovers domain, DeepSeek R1 has higher coverage than h^{FF}, while in Sokoban h^{FF} solves only two more tasks. This indicates that despite being less informed in these two domains, the heuristics generated by DeepSeek R1 perform better because they are more efficient to compute.

[3]This is identical to running GBFS with the blind heuristic h^0, where $h^0(s) = 0$ iff s is a goal state and $h^0(s) = 1$ otherwise.



(a) Plan length.

(b) Expansions.

Figure 2: Comparison of plan length and expansions for h^{FF} and the heuristics generated by DeepSeek R1. We only show plan lengths up to 300 (only plans in Miconic exceed this limit).

Comparisons to End-to-End Plan Generation with LLMs We also analyze the end-to-end plan generation capabilities of LLMs. We prompt the LLMs with general instructions, the PDDL files for the domain and the specific task, and a checklist of common pitfalls. To guide the model, the prompt also includes two examples: a PDDL task and PDDL domain for Gripper and Logistics (the same ones used in our prompt to generate heuristics), along with a complete optimal plan for each of the two tasks. The LLM is then asked to generate a plan, which is subsequently validated using the automated plan validation tool VAL [40]. Appendix C shows an example of our end-to-end prompt for the smallest Blocksworld task in our test set.

The left part of Table 2 shows that DeepSeek R1 solves 174 tasks and Gemini 2.0 Flash Thinking solves 202 tasks, both outperforming GBFS with the blind heuristic (h^0) in Pyperplan. However, both LLMs solve fewer tasks than all LLM-generated heuristics in Pyperplan, including the small DeepSeek R1 Distill Qwen 14B. With the end-to-end plan generation, the cost of using the LLMs also increases significantly. The cost for using V3 increases from \$0.25 to \$2.79, while the cost of R1 increases from \$6.12 to \$13.62. The end-to-end experiment requires 720 LLM calls (one for each of the 90 tasks across 8 domains), compared to 200 calls when generating 25 heuristics per domain. In the case of DeepSeek R1, the end-to-end experiment takes several days to complete, while the heuristics can be generated in a few hours. Furthermore, heuristics are reusable for solving new tasks, whereas end-to-end plans are not, making heuristics a more versatile and cost-effective approach.

We now compare the performance of the same LLM when used to solve two different problems: end-to-end planning versus heuristic generation. For the reasoning models Gemini 2.0 Flash Thinking and DeepSeek R1, using them for heuristic generation increases the number of solved tasks from 202 and 174 (end-to-end) to 305 and 373 (heuristic generation), respectively. The increase in total coverage is even more significant for non-reasoning models: with Gemini 2.0 Flash, the number of solved tasks jumps from 19 (end-to-end) to 296 (heuristic generation), and with DeepSeek V3 it increases from 48 (end-to-end) to 343 (heuristic generation). This shows that our heuristic generation approach can bridge the performance gap between expensive reasoning models and cheaper non-reasoning ones, while also increasing the overall number of solved tasks.

Comparison to State-of-the-Art Heuristics Implemented in C++ Our experimental setup has an obvious caveat: we are using Pyperplan, which is an educational, unoptimized Python planner, while all state-of-the-art planners are implemented in compiled languages such as C++. For example, the winners of all tracks of the last IPC, in 2023, are implemented in C++ [73]. Moreover, all of these planners are implemented on top of the Fast Downward planning system [35]. Even though Python is slower than C++ and uses more memory, we compare our best method, GBFS in Pyperplan using DeepSeek R1 heuristics, to GBFS in Fast Downward using the many satisficing heuristics implemented in the planner: the goal-count heuristic h^{GC} [23]; the landmark count heuristic h^{lmc} [56, 9]; the C++ implementation of the FF heuristic h^{FF} [39]; the context-enhanced additive heuristic h^{cea} [36]; the causal graph heuristic h^{cg} [34]; and the additive heuristic h^{add} [6]. We also compare it to $h^{\mathrm{WLF}}_{\mathrm{GPR}}$ [11], which uses Gaussian Process Regression [53] over features derived with the Weisfeiler-

Table 3: Coverage for different heuristics implemented in Fast Downward, and in Pyperplan. Heuristics h^{V3} and h^{R1} indicate the heuristics generated by DeepSeek V3 and by DeepSeek R1.

	Fast Downward (C++)							Pyperplan		
Domain	h^{GC}	h^{lmc}	h^{FF}	h^{cea}	h^{cg}	h^{add}	$h^{\mathrm{WLF}}_{\mathrm{GPR}}$	h^{FF}	$\boldsymbol{h^{\mathrm{V3}}}$	$\boldsymbol{h^{\mathrm{R1}}}$
Blocksworld (90)	32	39	27	40	34	44	72	24	45	66
Childsnack (90)	23	13	25	29	29	29	31	17	55	22
Floortile (90)	3	3	12	10	7	14	2	10	3	4
Miconic (90)	90	90	90	79	90	90	90	74	64	90
Rovers (90)	38	41	34	36	39	33	37	28	34	32
Sokoban (90)	42	43	36	33	35	33	38	31	31	30
Spanner (90)	30	30	30	30	30	30	73	30	69	70
Transport (90)	36	36	41	49	54	51	28	29	42	59
Sum (720)	294	295	295	306	318	324	371	243	343	373

Leman algorithm [69] to learn domain-dependent heuristics, and is considered the state-of-the-art in classical planning for heuristic learning. $h^{\mathrm{WLF}}_{\mathrm{GPR}}$ is also implemented on top of Fast Downward.

From now on, we denote the heuristics generated by DeepSeek V3 as h^{V3} and the ones by DeepSeek R1 as h^{R1}. Table 3 shows that GBFS in Pyperplan with h^{V3} and with h^{R1} solves more tasks in total than *any of the traditional Fast Downward heuristics*. Moreover, h^{R1} is also *competitive with the state-of-the-art,* $h^{\mathrm{WLF}}_{\mathrm{GPR}}$, and achieves slightly higher total coverage. This is quite an unexpected result, as Pyperplan is not as engineered and receives little attention compared to Fast Downward. It indicates that the heuristics generated by DeepSeek R1 are indeed powerful, being capable of surpassing the performance gap between Python and C++ implementations. Additionally, it shows that even the non-reasoning model DeepSeek V3 can outperform classical planners with our method.

Ablation Study We now analyze the impact of the different components in our prompt in an ablation study. For this, we use Gemini 2.0 Flash Thinking to generate 25 heuristics for several variants of our prompt, each of which alters or removes a single component of the original prompt. To reduce the effects of randomness, we run *all* generated heuristics on the test set, instead of selecting only the single best heuristic. We limit each GBFS run in this experiment to 5 minutes.

Table 4 shows the results. For the first ablation (second column in the table), we replace the long instructions at the beginning of the prompt by a simple request to generate a heuristic function, omitting details such as the request to minimize expansions. For the "Heuristics" ablation, we replace the provided domain-dependent example heuristics with domain-independent heuristics available in Pyperplan, namely the goal-count heuristic and four heuristics based on delete-relaxation [1].[4] For all other ablations, we remove the corresponding component from the prompt.

Comparing the heuristics with the best coverage score among all ablations, we see that the original prompt (solving 423 tasks) outperforms all other ablations. This score is 38.7% higher than the one obtained by using our method of selecting heuristics based on performance on the training set (see Table 2), and it even surpasses h^{R1}. This shows that while our heuristic selection method is good enough to outperform state-of-the-art planners, it is still not the optimal selection strategy. We leave the challenge of devising better selection strategies for future work.

Focusing on the worst possible coverage and the average coverage, our original prompt performs worse than other ablations. This result is to be expected: as our method aims to have a very diverse pool by generating multiple heuristics at a high temperature, we expect a great variance in performance. This is supported by the large standard deviation in our results. To obtain more consistent results, one could simply use a lower temperature, for example. However, this is not desirable: it is much better to have a single well-performing heuristic than several average ones.

[4]We do so because completely removing the example heuristics yields a coverage of 0 for all generated heuristics in all domains.

Table 4: Ablation study on the impact of different prompt components using Gemini 2.0 Flash Thinking. Components are ordered according to the original prompt. The symbol "$-$" indicates that the component was removed, and the symbol "$\leftrightarrows$" indicates that the component was replaced with a weaker version (see text for details). A failed heuristic is a heuristic that solves no tasks.

	Original Prompt	$\leftrightarrows$ Instruction	$-$ PDDL Domain	$-$ PDDL Tasks	$\leftrightarrows$ Heuristics	$-$ State Repr.	$-$ Static Repr.	$-$ Pyperplan Code	$-$ Checklist
Best Coverage	423	359	368	401	404	402	404	401	382
Worst Coverage	114	138	118	90	126	133	92	110	63
Avg. Coverage	267.0	242.5	237.3	261.5	263.2	253.7	243.8	270.0	260.0
Std. Deviation	$\pm$94.3	$\pm$59.6	$\pm$70.4	$\pm$86.8	$\pm$87.3	$\pm$85.0	$\pm$95.0	$\pm$97.4	$\pm$87.8
Failed Heuristics	64	57	52	42	64	68	57	97	57

For average coverage, we exclude heuristics that fail by crashing or not solving any task in our analysis. In this analysis, most of the components of our prompt are beneficial, with the most impactful component being the PDDL domain description. The only component whose removal actually slightly increases the average coverage is the Pyperplan code. However, its removal also increases the number of heuristics (out of 200) that fail by almost 50%. These results show that all prompt components are important for generating high-quality heuristics.

Examples of LLM-Generated Heuristic Functions We illustrate the heuristic functions generated by DeepSeek R1 using Blocksworld and Spanner (see Appendix B). Both domains have polynomial algorithms [30]. In Blocksworld, n blocks are rearranged by moving blocks from stacks. The selected heuristic identifies misplaced goal blocks A, adding 1 to the heuristic value plus 2 for each block B on top of A, using an auxiliary function for stack traversal. In Spanner, an agent traverses a one-way corridor to pick spanners (each used once) and tighten n nuts; moving without a required spanner can lead to an unsolvable state. The Spanner heuristic greedily assigns the closest available spanner to each loose nut. The cost calculation depends on whether the spanner is already picked up (agent-to-nut distance + 1) or not (agent-to-spanner + spanner-to-nut distances + 2), with a large penalty for unassigned nuts. This heuristic precomputes all-pairs shortest paths using breadth-first search during initialization. We describe and analyze the generated heuristics for the remaining domains in Appendix B.

Memorization vs. Reasoning To verify whether our method is robust to new domains, we run three additional experiments testing if the LLMs retrieve memorized heuristics or reason over the provided PDDL domain. Each experiment introduces a new PDDL domain.

We begin with a *qualitative* experiment: a variant of the Spanner domain. In the original Spanner domain, an agent traverses a one-way corridor toward a gate while picking up spanners. It must tighten n nuts to open the gate, and spanners break after a single use. Consequently, the agent must collect at least n spanners. In this variant, spanners no longer break after use. If the LLM merely recalled a standard Spanner heuristic, it would likely ignore this change and produce longer plans. Instead, the generated heuristic adapts to the new semantics. The best LLM-generated heuristic is perfect for this modified domain, yielding optimal plan lengths for all states. This suggests the LLM reasoned about the given domain rather than retrieving a memorized solution.

Next, we *obfuscate* Blocksworld following Valmeekam et al. [78]: all symbols (action names, predicate symbols, objects) are renamed to random strings. We refer to this as Obfuscated Blocksworld. In this experiment, h^{FF} (in Fast Downward) solves a similar number of tasks in both versions, 27 for Blocksworld and 28 for Obfuscated Blocksworld. This is expected, as h^{FF} does not rely on the semantics of the PDDL names. In contrast, h^{R1} (in Pyperplan) solves 66 tasks in Blocksworld but only 40 in Obfuscated Blocksworld. This indicates that while semantic cues from the PDDL names are useful, the LLM can still reason about the domain's logical structure without token semantics. We note that this is a highly adversarial setting for LLMs, which are trained to exploit token semantics rather than operate on random identifiers.

Finally, we use a completely new domain, *Rod-Rings*, which has not been released online nor exposed to an LLM. The domain features sticks with stacks of rings and a single "held" ring. Two moves are allowed: (a) place the held ring at the bottom of a stick (swapping with the current top ring), or (b) place the held ring at the top of a stick (swapping with the bottom ring). The goal is to arrange specified rings on specified sticks in a defined order. To avoid leakage, we use an OpenAI model (o3) via their API for this experiment. For Rod-Rings, h^{o3} guiding a GBFS in Pyperplan solves 58 tasks, nearly matching Fast Downward with h^{FF} (59 tasks).[5] This suggests the LLM reasons about this unseen domain.

Together, these three experiments indicate that the success of our method is due to the LLMs' ability to generate domain-dependent heuristic functions by reasoning about the logical structure of the domain.

5 Related Work

The combination of planning and learning to create heuristic functions has a long tradition [61, 12, 60, 2]. There are two main paradigms for learning heuristic functions in classical planning: task-dependent [20, 21, 49, 4] and domain-dependent [68, 71, 10, 31]. In this paper, we consider the second paradigm. Currently, the strongest approach in domain-dependent heuristic learning is $h^{\text{WLF}}_{\text{GPR}}$ [11], which we compare to above.

Recently, LLMs entered the picture. Yet, Valmeekam et al. [78] show that LLMs cannot reliably solve even small classical planning tasks when used for end-to-end plan generation. Moreover, techniques such as supervised fine-tuning and chain-of-thought [43] fail to generalize to out-of-distribution tasks [5, 72]. This holds even when using more advanced prompting techniques such as Tree of Thoughts [81] or Algorithm of Thoughts (AoT) [66]. While the strongest variant of this line of work, AoT+ [67], finds valid plans for up to 82% of the tasks in their Blocksworld benchmark set, the largest task in their set only has 5 blocks [78]. In contrast, our h^{R1} heuristic solves tasks with up to 216 blocks and never yields incorrect plans. As seen in our experiments, even LLMs explicitly designed for reasoning tasks are not competitive with state-of-the-art planners.

Nonetheless, Rossetti et al. [59] and Huang et al. [41] show that LLMs trained to plan can achieve competitive performance when training and test sets share the same distribution. Furthermore, LLMs can also help to solve classical planning tasks when combined with other techniques. For example, there is an extensive body of work exploring the potential of LLMs to convert problems described in natural language into PDDL tasks [e.g., 29, 24, 48, 46].

The most closely related approaches to ours are those that use LLMs to generate code for solving planning tasks. Katz et al. [42] highlight the high computational cost of using LLMs for end-to-end plan generation, particularly when multiple inferences are required. They propose to use LLMs to generate Python code for successor generation and goal testing, which are then used with standard search algorithms. While more efficient than end-to-end LLM planning, their method requires human feedback for incorrect code and is limited to small tasks due to its reliance on uninformed search. Our approach could address this scalability issue by providing a heuristic for an informed search algorithm.

Silver et al. [70] also use LLMs to generate Python code for solving classical planning tasks. However, they focus on *generalized planning*, where the aim is to find a strategy that efficiently solves any task of a given domain. In their approach, an LLM generates a "simple" Python program that does not rely on search. The key difference to our approach is that they address a different problem: there are many planning domains, such as the Sokoban domain we use above, for which no simple strategy exists to efficiently produce plans. For such domains, heuristic functions can be useful.

The work by Tuisov et al. [77] is the most similar to ours, and we draw inspiration from their work. They also use LLMs to generate heuristic function code for automated planning, though, their focus is on numeric planning. Their approach differs from ours in three main ways. First, they require a manual translation of each PDDL domain into Rust, including the implementation of a successor generator and a goal test. The cost of translating domains to Rust prevents easy comparisons on all IPC domains. In contrast, our approach generates heuristics directly from the PDDL description with the resulting code integrated into an off-the-shelf planner. Second, their heuristics are task-dependent,

[5] On the eight IPC 2023 domains, h^{o3} heuristics solve 354 tasks, 19 fewer than h^{R1} (373 tasks).

requiring new LLM inferences for each task, while ours are domain-dependent and reusable, reducing costs. Finally, while their approach outperforms all domain-independent heuristics they compare against, their LLM-generated heuristics result in fewer expansions for only one task. This suggests that while their heuristics may be computationally faster, they are not necessarily more informative. In our experiments, the LLM-generated heuristics are often more informative than the traditional domain-independent ones.

Beyond PDDL planning, Romera-Paredes et al. [58] use a search in function space to solve combinatorial problems. Their algorithm, FunSearch, samples different initial programs, similar to our set of candidate heuristics. However, FunSearch feeds the best initial candidates back into the LLM to improve them. In contrast, our pipeline never feeds the functions back into the LLM. Although our results are already positive, this feedback loop could further strengthen our results.

6 Limitations

Our approach, while effective, has limitations. One such limitation is its dependency on a formal PDDL description. This dependence makes our method less general than end-to-end LLM planning approaches that can use natural language. However, recent approaches translate natural language into PDDL representations [e.g., 24, 74], and can bridge this gap.

A second limitation is that our method, which selects the heuristic, is itself heuristic. This selection does not guarantee that the chosen heuristic generalizes to the out-of-distribution test set. However, our empirical results indicate that the selected heuristic consistently outperforms other approaches, indicating its robustness.

The effectiveness of our methods also depends on a multi-component prompt. Our ablation study confirmed that while the method is robust, the best performance indeed requires all components of the prompt.

Finally, our current implementation is a proof-of-concept that uses Pyperplan, an educational planner which is significantly slower and less memory-efficient than state-of-the-art C++ planners like Fast Downward. This implementation difference means that a C++ implementation would be necessary to conduct a direct performance comparison and verify the full potential of the LLM-generated heuristics. For example, Appendix E shows that Fast Downward expands up to 669 times more states per second than Pyperplan in certain domains.

7 Conclusions

In this paper, we show how to use LLMs to generate domain-dependent heuristic functions for classical planning domains. Our approach uses LLMs to produce a pool of candidate heuristics, which we then evaluate on a training set in order to choose the best heuristic from the pool. The selected heuristic is then used to solve unseen out-of-distribution test tasks.

We provide a proof-of-concept implementation of this pipeline in Pyperplan, an educational classical planner written in Python. We show that our LLM-generated heuristics outperform directly using LLMs end-to-end prompted to produce plans. This difference is particularly noticeable when using non-reasoning LLMs. Comparing the Python-based heuristics, we see that our LLM-generated heuristics outperform state-of-the-art domain-independent heuristics in most of the domains of our benchmark set. In particular, heuristics generated by reasoning LLMs such as DeepSeek R1 show a significant improvement compared to the domain-independent heuristic.

We show that Pyperplan equipped with the heuristics from DeepSeek V3 (h^{V3}) and DeepSeek R1 (h^{R1}) outperform heuristics implemented in Fast Downward [35], a state-of-the-art planner written in C++. Moreover, h^{R1} is also competitive with $h^{\mathrm{WLF}}_{\mathrm{GPR}}$ [11], the state-of-the-art in heuristic learning for classical planning implemented on top of Fast Downward. These results are surprising, as Pyperplan is much less optimized than Fast Downward, and DeepSeek R1 is not trained for a specific domain, while $h^{\mathrm{WLF}}_{\mathrm{GPR}}$ is. Our results show the potential of LLM-generated heuristics in classical planning as an efficient and effective approach to improve the planning capabilities of LLMs.

Acknowledgments

We thank Elliot Gestrin for designing the Rod-Rings domain, and Malte Helmert for giving us access to the cluster of the AI group at the University of Basel.

Jendrik Seipp was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. André G. Pereira acknowledges support from FAPERGS with project 21/2551-0000741-9. This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil* (CAPES) – Finance Code 001.

References

[1] Yusra Alkhazraji, Matthias Frorath, Markus Grützner, Malte Helmert, Thomas Liebetraut, Robert Mattmüller, Manuela Ortlieb, Jendrik Seipp, Tobias Springenberg, Philip Stahl, and Jan Wülfing. Pyperplan. `https://doi.org/10.5281/zenodo.3700819`, 2020.

[2] Shahab J. Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175:2075–2098, 2011.

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. arXiv:2309.16609 [cs.CL], 2023.

[4] Rafael V Bettker, Pedro P Minini, André G Pereira, and Marcus Ritt. Understanding sample generation strategies for learning heuristic functions in classical planning. *Journal of Artificial Intelligence Research*, 80:243–271, 2024.

[5] Bernd Bohnet, Azade Nova, Aaron T Parisi, Kevin Swersky, Katayoon Goshvadi, Hanjun Dai, Dale Schuurmans, Noah Fiedel, and Hanie Sedghi. Exploring and benchmarking the planning capabilities of large language models. arXiv:2406.13094 [cs.CL], 2024.

[6] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1): 5–33, 2001.

[7] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. arXiv:2407.21787 [cs.LG], 2024.

[8] Clemens Büchner, Remo Christen, Augusto B. Corrêa, Salomé Eriksson, Patrick Ferber, Jendrik Seipp, and Silvan Sievers. Fast Downward Stone Soup 2023. In *IPC-10 Planner Abstracts*, 2023.

[9] Clemens Büchner, Salomé Eriksson, Thomas Keller, and Malte Helmert. Landmark progression in heuristic search. In *Proc. ICAPS 2023*, pages 70–79, 2023.

[10] Dillon Z. Chen, Sylvie Thiébaux, and Felipe Trevizan. Learning domain-independent heuristics for grounded and lifted planning. In *Proc. AAAI 2024*, pages 20078–20086, 2024.

[11] Dillon Z. Chen, Felipe Trevizan, and Sylvie Thiébaux. Return to tradition: Learning reliable heuristics with classical machine learning. In *Proc. ICAPS 2024*, pages 68–76, 2024.

[12] Jens Christensen and Richard E Korf. A unified theory of heuristic evaluation functions and its application to learning. In *Proc. AAAI 1986*, pages 148–152, 1986.

[13] Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo, and Jendrik Seipp. Scorpion Maidu: Width search in the Scorpion planning system. In *IPC-10 Planner Abstracts*, 2023.

[14] Augusto B. Corrêa, André G. Pereira, and Jendrik Seipp. Code and experiment data from the NeurIPS 2025 paper "Classical planning with LLM-generated heuristics: Challenging the state of the art with Python code". https://doi.org/10.5281/zenodo.17400964, 2025.

[15] Joseph C. Culberson. Sokoban is PSPACE-complete. Technical Report TR 97-02, Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada, 1997.

[16] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J.L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, and Jingchang Chen et al. Deepseek-V3 technical report. arXiv:2412.19437 [cs.CL], 2024.

[17] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z.F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, and Guanting Chen et al. Deepseek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. arXiv:2501.12948 [cs.CL], 2025.

[18] Carmel Domshlak, Jörg Hoffmann, and Michael Katz. Red-black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence*, 221:73–114, 2015.

[19] James E. Doran and Donald Michie. Experiments with the graph traverser program. *Proceedings of the Royal Society A*, 294:235–259, 1966.

[20] Patrick Ferber, Malte Helmert, and Jörg Hoffmann. Neural network heuristics for classical planning: A study of hyperparameter space. In *Proc. ECAI 2020*, pages 2346–2353, 2020.

[21] Patrick Ferber, Florian Geißer, Felipe Trevizan, Malte Helmert, and Jörg Hoffmann. Neural network heuristic functions for classical planning: Bootstrapping and comparison to other methods. In *Proc. ICAPS 2022*, pages 583–587, 2022.

[22] Maximilian Fickert and Jörg Hoffmann. Online relaxation refinement for satisficing planning: On partial delete relaxation, complete hill-climbing, and novelty pruning. *Journal of Artificial Intelligence Research*, 73:67–115, 2022.

[23] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[24] Elliot Gestrin, Marco Kuhlmann, and Jendrik Seipp. NL2Plan: Robust LLM-driven planning from minimal text descriptions. In *ICAPS Workshop on Human-Aware and Explainable Planning*, 2024.

[25] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[26] Daniel Gnad, Álvaro Torralba, and Alexander Shleyfman. DecStar-2023. In *IPC-10 Planner Abstracts*, 2023.

[27] Gemini Team Google, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, and Timothy Lillicrap et al. Gemini: A family of highly capable multimodal models. arXiv:2312.11805 [cs.CL], 2023.

[28] Gemini Team Google, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, Soroosh Mariooryad, Yifan Ding, Xinyang Geng, Fred Alcober, Roy Frostig, Mark Omernick, Lexi Walker, Cosmin Paduraru, Christina Sorokin, Andrea Tacchetti, Colin Gaffney, Samira Daruki, Olcan Sercinoglu, Zach Gleicher, and Juliette Love et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530 [cs.CL], 2024.

[29] Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. In *Proc. NeurIPS 2023*, pages 79081–79094, 2023.

[30] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2–3):223–254, 1992.

[31] Mingyu Hao, Felipe Trevizan, Sylvie Thiébaux, Patrick Ferber, and Jörg Hoffmann. Guiding GBFS through learned pairwise rankings. In *Proc. IJCAI 2024*, pages 6724–6732, 2024.

[32] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107, 1968.

[33] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*, volume 13 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool, 2019.

[34] Malte Helmert. A planning heuristic based on causal graph analysis. In *Proc. ICAPS 2004*, pages 161–170, 2004.

[35] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[36] Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In *Proc. ICAPS 2008*, pages 140–147, 2008.

[37] Malte Helmert, Robert Mattmüller, and Gabi Röger. Approximation properties of planning benchmarks. In *Proc. ECAI 2006*, pages 585–589, 2006.

[38] Manuel Heusner, Thomas Keller, and Malte Helmert. Understanding the search behaviour of greedy best-first search. In *Proc. SoCS 2017*, pages 47–55, 2017.

[39] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[40] Richard Howey and Derek Long. VAL's progress: The automatic validation tool for PDDL2.1 used in the International Planning Competition. In *ICAPS 2003 Workshop on the Competition*, 2003.

[41] Sukai Huang, Trevor Cohn, and Nir Lipovetzky. Chasing progress, not perfection: Revisiting strategies for end-to-end LLM plan generation. arXiv:2412.10675 [cs.CL], 2024.

[42] Michael Katz, Harsha Kokel, Kavitha Srinivas, and Shirin Sohrabi. Thought of search: Planning with language models through the lens of efficiency. In *Proc. NeurIPS 2024*, pages 138491–138568, 2024.

[43] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Proc. NeurIPS*, volume 35, pages 22199–22213, 2022.

[44] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. arXiv:2203.07814 [cs.PL], 2022.

[45] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *Proc. AAAI 2017*, pages 3590–3596, 2017.

[46] 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:2304.11477 [cs.CL], 2023.

[47] Drew McDermott. The 1998 AI Planning Systems competition. *AI Magazine*, 21(2):35–55, 2000.

[48] James T. Oswald, Kavitha Srinivas, Harsha Kokel, Junkyu Lee, Michael Katz, and Shirin Sohrabi. Large language models as planning domain generators. In *Proc. ICAPS 2024*, pages 423–431, 2024.

[49] Stefan O'Toole, Miquel Ramirez, Nir Lipovetzky, and Adrian R. Pearce. Sampling from pre-images to learn heuristic functions for classical planning (extended abstract). In *Proc. SoCS 2022*, pages 308–310, 2022.

[50] Gerald Paul, Gabriele Röger, Thomas Keller, and Malte Helmert. Optimal solutions to large logistics planning domain problems. In *Proc. SoCS 2017*, pages 73–81, 2017.

[51] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[52] Ira Pohl. First results on the effect of error in heuristic search. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 219–236. Edinburgh University Press, 1969.

[53] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

[54] Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS 2009*, pages 273–280, 2009.

[55] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[56] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *Proc. AAAI 2008*, pages 975–982, 2008.

[57] Gabriele Röger and Malte Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proc. ICAPS 2010*, pages 246–249, 2010.

[58] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[59] Nicholas Rossetti, Massimiliano Tummolo, Alfonso Emilio Gerevini, Luca Putelli, Ivan Serina, Mattia Chiari, and Matteo Olivato. Learning general policies for planning through GPT models. In *Proc. ICAPS 2024*, pages 500–508, 2024.

[60] Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristics. In *Proc. AAAI 2008*, pages 357–362, 2008.

[61] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 1959.

[62] Javier Segovia and Jendrik Seipp. Benchmark repository of IPC 2023 - learning track. `https://github.com/ipc2023-learning/benchmarks`, 2023.

[63] Jendrik Seipp. Dissecting Scorpion: Ablation study of an optimal classical planner. In *Proc. ECAI 2024*, pages 39–42, 2024.

[64] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. `https://doi.org/10.5281/zenodo.790461`, 2017.

[65] Jendrik Seipp, Thomas Keller, and Malte Helmert. Saturated cost partitioning for optimal classical planning. *Journal of Artificial Intelligence Research*, 67:129–167, 2020.

[66] Bilgehan Sel, Ahmad Al-Tawaha, Vanshaj Khattar, Ruoxi Jia, and Ming Jin. Algorithm of thoughts: enhancing exploration of ideas in large language models. In *Proceedings of the 41st International Conference on Machine Learning*. JMLR.org, 2024.

[67] Bilgehan Sel, Ruoxi Jia, and Ming Jin. LLMs can plan only if we tell them. *arXiv preprint arXiv:2501.13545*, 2025.

[68] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning domain-independent planning heuristics with hypergraph networks. In *Proc. ICAPS 2020*, pages 574–584, 2020.

[69] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

[70] Tom Silver, Soham Dan, Kavitha Srinivas, Josh Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. Generalized planning in PDDL domains with pretrained large language models. In *Proc. AAAI 2024*, pages 20256–20264, 2024.

[71] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *Proc. ICAPS 2022*, pages 629–637, 2022.

[72] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. Chain of thoughtlessness? an analysis of CoT in planning. In *Proc. NeurIPS 2024*, pages 29106–29141, 2024.

[73] Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. The 2023 International Planning Competition. *AI Magazine*, 45(2):280–296, 2024. doi: 10.1002/aaai.12169.

[74] Marcus Tantakoun, Christian Muise, and Xiaodan Zhu. LLMs as planning formalizers: A survey for leveraging large language models to construct automated planning models. In *Findings of the Association for Computational Linguistics: ACL 2025*.

[75] Álvaro Torralba, Carlos Linares López, and Daniel Borrajo. Symbolic perimeter abstraction heuristics for cost-optimal planning. *Artificial Intelligence*, 259:1–31, 2018.

[76] Álvaro Torralba, Jendrik Seipp, and Silvan Sievers. Automatic instance generation for classical planning. In *Proc. ICAPS 2021*, pages 376–384, 2021.

[77] Alexander Tuisov, Yonatan Vernik, and Alexander Shleyfman. LLM-generated heuristics for AI planning: Do we even need domain-independence anymore? arXiv:2501.18784 [cs.AI], 2025.

[78] Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of large language models – a critical investigation. In *Proc. NeurIPS 2023*, pages 75993–76005, 2023.

[79] Karthik Valmeekam, Sarath Sreedharan, Matthew Marquez, Alberto Olmo Hernandez, and Subbarao Kambhampati. On the planning abilities of large language models (A critical investigation with a proposed benchmark). arXiv:2305.15771 [cs.AI], 2023.

[80] Karthik Valmeekam, Kaya Stechly, Atharva Gundawar, and Subbarao Kambhampati. Planning in strawberry fields: Evaluating and improving the planning and scheduling capabilities of LRM o1. arXiv:2410.02162 [cs.CL], 2024.

[81] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

A Heuristic Generation Prompt: Blocksworld

Below we show our prompt used for heuristic generation in the Blocksworld domain. The only parts of the prompt that change between domains are the names of the domain and heuristic, and the `domain-file`, `instance-file-example-1` and `instance-file-example-2` sections. To reduce the number of pages, we do not display the entire domain and instance files, but just the beginning of each. The complete domain and instance files can be found online [14].

```
<problem-description>
You are a highly-skilled professor in AI planning and a proficient Python programmer creating a domain-dependent heuristic function for the PDDL domain <domain>blocksworld</domain>. The heuristic function you create will be used to guide a greedy best-first search to solve instances from this domain. Therefore, the heuristic does not need to be admissible. For a given state, the heuristic function should estimate the required number of actions to reach a goal state as accurately as possible, while remaining efficiently computable. The name of the heuristic should be blocksworld1Heuristic. The heuristic should be efficiently computable, and it should minimize the number of expanded nodes during the search. Next, you will receive a sequence of file contents to help you with your task and to show you the definition of the blocks world domain.
</problem-description>

This is the PDDL domain file of the blocksworld domain, for which you need to create a domain-dependent heuristic:
<domain-file>
(define (domain blocksworld)
[...]
</domain-file>

This is an example of a PDDL instance file of the blocksworld domain:
<instance-file-example-1>
(define (problem blocksworld-01)
[...]
</instance-file-example-1>

This is a second example of a PDDL instance file of the blocksworld domain:
<instance-file-example-2>

(define (problem blocksworld-99)
 (:domain blocksworld)
 [...]
</instance-file-example-2>

This is the PDDL domain file of another domain, called Gripper, which serves as an example:
<gripper-domain-file>
(define (domain gripper-strips)
[...]
</gripper-domain-file>

This is an example of an instance file from the Gripper domain:
<gripper-instance-file-example>
(define (problem strips-gripper-x-20)
[...]
</gripper-instance-file-example>

This is an example of a domain-dependent heuristic for Gripper:
<code-file-heuristic-1>
from fnmatch import fnmatch
from heuristics.heuristic_base import Heuristic

class GripperHeuristic(Heuristic):
    """
```

```
    A domain-dependent heuristic for the Gripper domain.

    # Summary
    This heuristic estimates the number of actions needed to transport all balls
    from `rooma` to `roomb`.

    # Assumptions:
    - The robot has two grippers, allowing it to carry up to two balls per trip.
    - The robot must return to rooma after each trip, except for the final trip.
    - If the robot starts in roomb, it must move to rooma first.

    # Heuristic Initialization
    - Implicitly assume that all balls must be in `roomb` at the end.

    # Step-By-Step Thinking for Computing Heuristic
    1. Identify the number of balls still in `rooma` that need to be transported.
    2. Determine if the robot is currently carrying balls (it may start with 1 or
    ↪  2 already).
    3. Check whether the robot is in `rooma` or `roomb`:
       - If in room B, it may need to drop the carried balls first before moving
       ↪  to A.
       - If in room A, it can immediately begin planning the transport.
    4. Handle the case where the robot starts with balls in the grippers:
       - If carrying 2 balls, it should move to B, drop them, and return to A.
       - If carrying 1 ball and an odd number remains in `rooma`, it may pick up
       ↪  another ball before moving.
       - If carrying 1 ball and an even number remains, it transports the single
       ↪  ball first.
    5. Compute the number of full two-ball trips needed:
       - This is `balls_in_rooma // 2` (since up to 2 balls are moved per trip).
       - Each full two-ball trip costs 6 actions (except for the last trip).
    6. Handle the last remaining ball (if the total number of balls is odd):
       - If one ball is left, the robot moves to A, picks it up, moves to B, and
       ↪  drops it.
    """

    def __init__(self, task):
        """Initialize the heuristic by extracting goal conditions and static
        ↪  facts."""
        # The set of facts that must hold in goal states. We assume that all balls
        ↪  must be in `roomb` at the end.
        self.goals = task.goals
        # Static facts are not needed for this heuristic.
        static_facts = task.static

    def __call__(self, node):
        """Estimate the minimum cost to transport all remaining balls from room A
        ↪  to room B."""
        state = node.state

        def match(fact, *args):
            """
            Utility function to check if a PDDL fact matches a given pattern.
            - `fact`: The fact as a string (e.g., "(at ball1 rooma)").
            - `args`: The pattern to match (e.g., "at", "*", "rooma").
            - Returns `True` if the fact matches the pattern, `False` otherwise.
            """
            parts = fact[1:-1].split()  # Remove parentheses and split into
            ↪  individual elements.
            return all(fnmatch(part, arg) for part, arg in zip(parts, args))

        # Count how many balls are currently in room A.
        balls_in_room_a = sum(1 for fact in state if match(fact, "at", "*",
        ↪  "rooma"))

```

```
        # Count the number of balls currently held by the robot.
        balls_in_grippers = sum(1 for fact in state if match(fact, "carry", "*",
        ↪  "*"))

        # Check if the robot is in room A.
        robot_in_room_a = "(at-robby rooma)" in state

        # Define the cost of each individual action for readability.
        move_cost = 1  # Moving between rooms.
        pick_cost = 1  # Picking up a ball.
        drop_cost = 1  # Dropping a ball.

        total_cost = 0  # Initialize the heuristic cost.

        # Handle cases where the robot is already carrying balls.
        if robot_in_room_a:
            if balls_in_grippers == 2:
                # Both grippers are full, so move to room B and drop both balls.
                total_cost += move_cost + 2 * drop_cost
            elif balls_in_grippers == 1 and balls_in_room_a % 2 == 1:
                # Pick one more ball to fill both grippers, then move and drop
                ↪  both.
                total_cost += pick_cost + move_cost + 2 * drop_cost
                balls_in_room_a -= 1  # Since we moved one extra ball.
            elif balls_in_grippers == 1 and balls_in_room_a % 2 == 0:
                # Move with one ball and drop it, leaving an even number of balls.
                total_cost += move_cost + drop_cost
        else:
            # If the robot is in room B, it must drop any carried balls.
            total_cost += balls_in_grippers * drop_cost

        if balls_in_room_a > 0:
            # Move back to room A to continue transporting balls.
            total_cost += move_cost

            # Compute the number of trips with two balls.
            num_two_ball_trips = balls_in_room_a // 2

            # Each trip includes: 2 picks, 1 move to B, 2 drops and 1 move back to
            ↪  A (except for the last trip).
            total_cost += num_two_ball_trips * (2 * pick_cost + move_cost + 2 *
            ↪  drop_cost + move_cost) - 1

            # If there's a single ball left after the two-ball trips, go back to A
            ↪  and move the ball by itself.
            if balls_in_room_a % 2 == 1:
                total_cost += move_cost + pick_cost + move_cost + drop_cost

        # Return the estimated cost to goal state.
        return total_cost
</code-file-heuristic-1>

This is the PDDL domain file of another domain, called Logistics, to serve as a
↪  second example:
<logistics-domain-file>
(define (domain logistics-strips)
  (:requirements :strips)
  (:predicates         (OBJ ?obj)
                       (TRUCK ?truck)
                       (LOCATION ?loc)
               (AIRPLANE ?airplane)
               (CITY ?city)
               (AIRPORT ?airport)
               (at ?obj ?loc)
               (in ?obj1 ?obj2)
```

```
                (in-city ?obj ?city))

(:action LOAD-TRUCK
  :parameters
   (?obj
    ?truck
    ?loc)
  :precondition
   (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
   (at ?truck ?loc) (at ?obj ?loc))
  :effect
   (and (not (at ?obj ?loc)) (in ?obj ?truck)))
[...]
</logistics-domain-file>

This is an example of an instance file from the Logistics domain:
<logistics-instance-file-example>
(define (problem strips-log-y-5)
   (:domain logistics-strips)
   (:objects package5 package4 package3 package2 package1 city8
             city7 city6 city5 city4 city3 city2 city1 truck15 truck14
             truck13 truck12 truck11 truck10 truck9 truck8 truck7 truck6
             truck5 truck4 truck3 truck2 truck1 plane1 city8-2 city8-1
             city7-2 city7-1 city6-2 city6-1 city5-2 city5-1 city4-2
             city4-1 city3-2 city3-1 city2-2 city2-1 city1-2 city1-1
             city8-3 city7-3 city6-3 city5-3 city4-3 city3-3 city2-3
             city1-3)
   (:init (obj package5)
          (obj package4)
[...]
</logistics-instance-file-example>

This is an example of a domain-dependent heuristic for Logistics:
<code-file-heuristic-2>
from fnmatch import fnmatch
from heuristics.heuristic_base import Heuristic


def get_parts(fact):
    """Extract the components of a PDDL fact by removing parentheses and splitting
    ↪  the string."""
    return fact[1:-1].split()


def match(fact, *args):
    """
    Check if a PDDL fact matches a given pattern.

    - `fact`: The complete fact as a string, e.g., "(in-city airport1 city1)".
    - `args`: The expected pattern (wildcards `*` allowed).
    - Returns `True` if the fact matches the pattern, else `False`.
    """
    parts = get_parts(fact)
    return all(fnmatch(part, arg) for part, arg in zip(parts, args))


class LogisticsHeuristic(Heuristic):
    """
    A domain-dependent heuristic for the Logistics domain.

    # Summary
    The heuristic estimates the number of necessary actions (load, unload, and
    ↪  transport) in order to transport each package to its goal based on its
    ↪  current state.

```

```
    # Assumptions
    - Packages can be on the ground, in a truck, or in a plane.
    - Trucks move within a city, while planes move between cities.
    - If a package is already at the goal, no extra actions are needed.

    # Heuristic Initialization
    - Extract the goal locations for each package and the static facts (e.g., `in-city` relationships and airport locations) from the task.

    # Step-by-Step Thinking for Computing the Heuristic Value
    Below is the thought process for computing the heuristic for a given state:

    1. Extract Relevant Information:
    - Identify the current location of every package.
    - Identify whether a package is inside a vehicle (truck or plane), and if so, find the physical location of that vehicle.

    2. Distinguish Between Intra-city and Inter-city Transport:
    - Determine the current city and goal city for each package by checking its location-to-city mapping.
    - If the current city is the same as the goal city, follow the intra-city package movement rules.
    - If the current city is different from the goal city, follow the inter-city package movement rules.

    3. Handle Intra-city Transport:
    - If the package is already at its goal location, no action is required.
    - If the package is in a plane, it must be unloaded.
    - If the package is not in a truck and not already at its goal, it must be loaded into a truck.
    - If the package is in a truck or not yet at its final location, it must be unloaded from the truck at the goal.

    4. Handle Inter-city Transport:
    - Step 1: Move the package to the airport in the current city.
        - If the package is not inside a truck and not at an airport, it must be loaded into a truck.
        - If the package is not at an airport or inside a truck, it must be unloaded from the truck at the airport.
    - Step 2: Fly the package to the destination city.
        - If the package is not in a plane, it must be loaded into a plane.
        - The package must always be unloaded from the plane at the airport of the destination city.
    - Step 3: Move the package from the airport to its final location.
        - If the goal location is not an airport, the package must be loaded into a truck at the airport.
        - Finally, the package must be unloaded from the truck at the goal location.

    5. Summing the Actions:
    - The total heuristic value is the sum of all necessary actions.
    - Loading and unloading costs are counted exactly based on the required actions.
    - Transport movements (trucks or planes) are counted only when necessary.
    """

    def __init__(self, task):
        """
        Initialize the heuristic by extracting:
        - Goal locations for each package.
        - Static facts (`in-city` relationships and airport locations).
        """
        self.goals = task.goals  # Goal conditions.
        static_facts = task.static  # Facts that are not affected by actions.

```

```
        # Map locations to their respective cities using "in-city" relationships.
        self.location_to_city = {
            get_parts(fact)[1]: get_parts(fact)[2]
            for fact in static_facts
            if match(fact, "in-city", "*", "*")
        }

        # Identify all airport locations.
        self.airports = {
            get_parts(fact)[1]
            for fact in static_facts
            if match(fact, "airport", "*")
        }

        # Store goal locations for each package.
        self.goal_locations = {}
        for goal in self.goals:
            predicate, *args = get_parts(goal)
            if predicate == "at":
                package, location = args
                self.goal_locations[package] = location

    def __call__(self, node):
        """Compute an estimate of the minimal number of required actions."""
        state = node.state  # Current world state.

        # Track where packages and vehicles are currently located.
        current_locations = {}
        for fact in state:
            predicate, *args = get_parts(fact)
            if predicate in ["at", "in"]:  # Track both direct location and inside
            ↪  vehicle.
                obj, location = args
                current_locations[obj] = location

        total_cost = 0  # Initialize action cost counter.

        for package, goal_location in self.goal_locations.items():
            # Get the current location of the package (could be a city location,
            ↪  truck or plane).
            current_location = current_locations[package]

            # Check if the package is inside a vehicle.
            in_vehicle = current_location not in self.location_to_city

            if in_vehicle:
                # Identify type of vehicle (truck or plane).
                in_plane = current_location.startswith("plane")
                in_truck = current_location.startswith("truck")
                assert in_plane ^ in_truck, f"Invalid state: {current_location}"

                # Retrieve the physical location of the vehicle.
                current_location = current_locations[current_location]
            else:
                in_plane = False
                in_truck = False

            # Get the city of the package's current location and goal.
            current_city = self.location_to_city[current_location]
            goal_city = self.location_to_city[goal_location]

            # Intra-city Transport (Same City)
            if current_city == goal_city:
                if in_plane:
                    total_cost += 1  # Unload from the plane.
```

```
                if current_location != goal_location and not in_truck:
                    total_cost += 1  # Load into a truck.

                if current_location != goal_location or in_truck:
                    total_cost += 1  # Unload from the truck.

            # Inter-city Transport (Different Cities)
            else:
                # Step 1: Move to the airport in the current city.
                if current_location not in self.airports and not in_truck:
                    total_cost += 1  # Load into a truck.

                if current_location not in self.airports or in_truck:
                    total_cost += 1  # Unload from the truck at the airport.

                # Step 2: Fly to the destination city.
                if not in_plane:
                    total_cost += 1  # Load into a plane.

                total_cost += 1  # Unload from the plane.

                # Step 3: Transport from airport to the goal (if required).
                if goal_location not in self.airports:
                    total_cost += 1  # Load into a truck at destination airport.
                    total_cost += 1  # Unload at the destination.

        return total_cost
</code-file-heuristic-2>

This is how an example state from the blocksworld domain is represented internally by the planner. Note that PDDL facts are represented as strings, for example '(predicate_name object1 object2)'.
<state>
frozenset({'(on-table b9)', '(on b8 b9)', '(on-table b2)', '(on b10 b2)', '(on b11 b10)', '(on b4 b8)', '(on b6 b11)', '(on b5 b6)', '(on b14 b5)', '(on b13 b4)', '(on b12 b13)', '(on b1 b12)', '(on b7 b1)', '(clear b3)', '(on b3 b7)', '(holding b15)', '(clear b14)'})
</state>

This is an example for how the static information is represented internally by the planner:
<static>
frozenset()
</static>

This is the source code for representing operators and tasks in the planner:
<code-file-task>
"""
Classes for representing a STRIPS planning task
"""


class Operator:
    """
    The preconditions represent the facts that have to be true
    before the operator can be applied.
    add_effects are the facts that the operator makes true.
    delete_effects are the facts that the operator makes false.
    """

    def __init__(self, name, preconditions, add_effects, del_effects):
        self.name = name
        self.preconditions = frozenset(preconditions)
        self.add_effects = frozenset(add_effects)
```

```
        self.del_effects = frozenset(del_effects)

    def applicable(self, state):
        """
        Operators are applicable when their set of preconditions is a subset
        of the facts that are true in "state".

        @return True if the operator's preconditions is a subset of the state,
                False otherwise
        """
        return self.preconditions <= state

    def apply(self, state):
        """
        Applying an operator means removing the facts that are made false
        by the operator from the set of true facts in state and adding
        the facts made true.

        Note that therefore it is possible to have operands that make a
        fact both false and true. This results in the fact being true
        at the end.

        @param state The state that the operator should be applied to
        @return A new state (set of facts) after the application of the
                operator
        """
        assert self.applicable(state)
        assert type(state) in (frozenset, set)
        return (state - self.del_effects) | self.add_effects

    def __eq__(self, other):
        return (
            self.name == other.name
            and self.preconditions == other.preconditions
            and self.add_effects == other.add_effects
            and self.del_effects == other.del_effects
        )

    def __hash__(self):
        return hash((self.name, self.preconditions, self.add_effects,
        ↪  self.del_effects))

    def __str__(self):
        s = "%s\n" % self.name
        for group, facts in [
            ("PRE", self.preconditions),
            ("ADD", self.add_effects),
            ("DEL", self.del_effects),
        ]:
            for fact in facts:
                s += "  {}: {}\n".format(group, fact)
        return s

    def __repr__(self):
        return "<Op %s>" % self.name


class Task:
    """
    A STRIPS planning task
    """

    def __init__(self, name, facts, initial_state, goals, operators, static):
        """
        @param name The task's name
```

```
        @param facts A set of all the fact names that are valid in the domain
        @param initial_state A set of fact names that are true at the beginning
        @param goals A set of fact names that must be true to solve the problem
        @param operators A set of operator instances for the domain
        @param static_info A set of facts that are true in every state
        """
        self.name = name
        self.facts = facts
        self.initial_state = initial_state
        self.goals = goals
        self.operators = operators
        self.static = static

    def goal_reached(self, state):
        """
        The goal has been reached if all facts that are true in "goals"
        are true in "state".

        @return True if all the goals are reached, False otherwise
        """
        return self.goals <= state

    def get_successor_states(self, state):
        """
        @return A list with (op, new_state) pairs where "op" is the applicable
        operator and "new_state" the state that results when "op" is applied
        in state "state".
        """
        return [(op, op.apply(state)) for op in self.operators if
        ↪  op.applicable(state)]

    def __str__(self):
        s = "Task {0}\n  Vars:  {1}\n  Init:  {2}\n  Goals: {3}\n  Ops:   {4}"
        return s.format(
            self.name,
            ", ".join(self.facts),
            self.initial_state,
            self.goals,
            "\n".join(map(repr, self.operators)),
        )

    def __repr__(self):
        string = "<Task {0}, vars: {1}, operators: {2}>"
        return string.format(self.name, len(self.facts), len(self.operators))
</code-file-task>

Provide only the Python code of the domain-dependent heuristic for the blocksworld
↪  domain. Here is a checklist to help you with your code:
1) The code for extracting objects from facts remembers to ignore the surrounding
↪  brackets.
2) The heuristic is 0 only for goal states.
3) The heuristic value is finite for solvable states.
4) All used modules are imported.
5) The information from static facts is extracted into suitable data structures in
↪  the constructor.
6) Provide a detailed docstring explaining the heuristic calculation. For this,
↪  divide the docstring into sections "Summary", "Assumptions", "Heuristic
↪  Initialization" and "Step-By-Step Thinking for Computing Heuristic".
```

B Generated Heuristics (Selection)

We present the selected DeepSeek R1 heuristics from our experiments. We provide descriptions and source code for the Blocksworld and Spanner domains, and only the descriptions for the rest.

B.1 DeepSeek R1 Heuristic for Blocksworld

In the *Blocksworld* domain, stacks of n blocks must be rearranged from an initial state to a goal condition. The available actions move blocks that are on top of a stack onto a different stack or the table. Blocksworld is *2-approximable* [30]: simply "destroy" all stacks (by placing their blocks onto the table) and then build the goal stacks.

The best heuristic generated by the LLM does not use this method though. It first computes for each block A mentioned in the goal whether A is misplaced and, if so, adds 2 to the heuristic value for each block B on top of A, plus 1. For this, the heuristic uses an auxiliary function that traverses the stack on top of A. It is easy to see that this heuristic can overestimate the optimal plan length.

```python
from fnmatch import fnmatch
from collections import defaultdict
from heuristics.heuristic_base import Heuristic

class blocksworld9Heuristic(Heuristic):
    """
    A domain-dependent heuristic for the Blocksworld domain.

    # Summary
    This heuristic estimates the number of actions required to achieve the goal by considering the number of blocks that need to be moved and the blocks above them in the current state. For each block not in its goal position, the cost is 2 times the number of blocks above it plus 2. Held blocks not in the goal position add 1 to the cost.

    # Assumptions
    - The goal specifies the required 'on' and 'on-table' predicates for certain blocks.
    - Blocks not mentioned in the goal do not affect the heuristic.
    - Moving a block requires unstacking all blocks above it first.
    - The arm can only carry one block at a time.

    # Heuristic Initialization
    - Extract the goal conditions to determine the target positions for each block.
    - Store the goal parent (block or 'table') for each block mentioned in the goal.

    # Step-By-Step Thinking for Computing Heuristic
    1. Parse the current state to determine the current parent (block or 'table') and children (blocks on top) for each block.
    2. Check if any block is currently being held.
    3. For each block mentioned in the goal:
        a. If the block is held and not in the goal position, add 1 action (putdown or stack).
        b. If the block is not held and not in the goal position, calculate the number of blocks above it and add 2 actions per block (unstack and putdown) plus 2 actions for the block itself.
    4. Sum all calculated costs to get the heuristic value.
    """

    def __init__(self, task):
        """Initialize the heuristic by extracting goal conditions."""
        self.goal_parent = {}
        for goal in task.goals:
            parts = goal[1:-1].split()
            if parts[0] == 'on':
                block, parent = parts[1], parts[2]
                self.goal_parent[block] = parent
            elif parts[0] == 'on-table':
                block = parts[1]
                self.goal_parent[block] = 'table'
```

```python

    def __call__(self, node):
        """Compute the heuristic value for the given state."""
        state = node.state
        current_parent = {}
        current_children = defaultdict(list)
        held_block = None

        # Parse current state
        for fact in state:
            parts = fact[1:-1].split()
            if parts[0] == 'on':
                child, parent = parts[1], parts[2]
                current_parent[child] = parent
                current_children[parent].append(child)
            elif parts[0] == 'on-table':
                block = parts[1]
                current_parent[block] = 'table'
            elif parts[0] == 'holding':
                held_block = parts[1]

        # Calculate cost
        cost = 0

        # Check held block
        if held_block is not None and held_block in self.goal_parent:
            current_pos = 'held'
            goal_pos = self.goal_parent[held_block]
            # If held block is not in goal position, add 1 action
            if (goal_pos == 'table' and current_pos != 'table') or \
               (goal_pos != 'table' and current_pos != goal_pos):
                cost += 1

        # Process each block in the goal
        for block in self.goal_parent:
            if block == held_block:
                continue  # Already handled

            current_parent_block = current_parent.get(block, 'table')
            goal_parent_block = self.goal_parent[block]

            if current_parent_block != goal_parent_block:
                # Calculate number of blocks above the current block
                def count_above(x):
                    cnt = 0
                    stack = [x]
                    while stack:
                        current = stack.pop()
                        for child in current_children.get(current, []):
                            cnt += 1
                            stack.append(child)
                    return cnt

                above = count_above(block)
                cost += 2 * (above + 1)

        return cost
```

B.2 DeepSeek R1 Heuristic for Spanner

In the *Spanner* domain, an agent must move through a corridor, pick up spanners, and tighten n nuts at the gate, using each picked-up spanner at most once. The agent can move in the direction of the gate but not backwards. Thus, moving without first picking up a required spanner, results in an

unsolvable state. Optimal plans can be computed in polynomial time by simply pick up exactly the first n spanners and then tighten the n nuts at the gate.

However, this is not the strategy the LLM uses to implement the heuristic function. The heuristic greedily assigns to each loose nut (in a fixed order) the closest spanner still available. If the spanner has already been picked up, the cost of tightening the nut is the distance from the agent to the nut location plus 1. If not, then the cost is the distance from the agent to the location of the spanner plus the distance from the location of the spanner to the location of the nut plus 2. Each spanner can be used at most once, so if a nut has no assigned spanner, the heuristic adds a large number to the cost of the state. The heuristic performs a breadth-first search during the initialization phase to compute the shortest path between all locations. This can also overestimate the optimal plan length. Arguably, the LLM could have created a simpler heuristic if the implicit assumptions of the domain were explicit: the PDDL domain allows for arbitrary connections between locations, but all instances assume a one-way corridor.

```python
from fnmatch import fnmatch
from collections import defaultdict, deque
from heuristics.heuristic_base import Heuristic

class spanner10Heuristic(Heuristic):
    """
    A domain-dependent heuristic for the Spanner domain.

    # Summary
    Estimates the number of actions required to tighten all loose nuts by considering the minimal path to collect usable spanners and reach each nut's location.

    # Assumptions
    - The man (bob) can carry multiple spanners, but each spanner can be used only once.
    - The shortest path between locations is precomputed using static link information.
    - The man's name is 'bob' as per the problem examples.

    # Heuristic Initialization
    - Extracts static link facts to build a directed graph of locations.
    - Precomputes shortest paths between all pairs of locations using BFS.

    # Step-By-Step Thinking for Computing Heuristic
    1. Identify the man's current location.
    2. Collect all loose nuts and their locations.
    3. Identify all usable spanners (carried or on the ground).
    4. For each loose nut, compute the minimal cost to tighten it using the best available spanner.
    5. Assign spanners to nuts greedily, marking used spanners to avoid reuse.
    6. Sum the minimal costs for all nuts to get the heuristic value.
    """

    def __init__(self, task):
        self.goals = task.goals
        self.static_links = defaultdict(list)
        for fact in task.static:
            parts = fact[1:-1].split()
            if parts[0] == 'link':
                start, end = parts[1], parts[2]
                self.static_links[start].append(end)
        # Precompute shortest paths between all locations
        self.shortest_paths = {}
        all_locations = set(self.static_links.keys())
        for ends in self.static_links.values():
            all_locations.update(ends)
        all_locations = list(all_locations)
        for loc in all_locations:
```

```
            distances = {loc: 0}
            queue = deque([loc])
            while queue:
                current = queue.popleft()
                for neighbor in self.static_links.get(current, []):
                    if neighbor not in distances:
                        distances[neighbor] = distances[current] + 1
                        queue.append(neighbor)
            self.shortest_paths[loc] = distances

    def __call__(self, node):
        state = node.state
        # Find man's location (assumed to be 'bob')
        man_location = None
        for fact in state:
            if fact.startswith('(at bob '):
                parts = fact[1:-1].split()
                man_location = parts[2]
                break
        if not man_location:
            return float('inf')  # Invalid state

        # Collect loose nuts and their locations
        loose_nuts = []
        nut_locations = {}
        for fact in state:
            if fact.startswith('(loose '):
                parts = fact[1:-1].split()
                loose_nuts.append(parts[1])
            elif fact.startswith('(at ') and 'nut' in fact:
                parts = fact[1:-1].split()
                obj, loc = parts[1], parts[2]
                nut_locations[obj] = loc

        # Collect usable spanners and their locations
        usable_spanners = []
        carried_spanners = []
        spanner_locations = {}
        for fact in state:
            if fact.startswith('(usable '):
                parts = fact[1:-1].split()
                usable_spanners.append(parts[1])
            if fact.startswith('(carrying bob '):
                parts = fact[1:-1].split()
                carried_spanners.append(parts[2])
            if fact.startswith('(at ') and 'spanner' in fact:
                parts = fact[1:-1].split()
                spanner_locations[parts[1]] = parts[2]

        # Prepare available spanners (carried or on ground)
        available_spanners = []
        for spanner in usable_spanners:
            if spanner in carried_spanners:
                available_spanners.append((spanner, man_location, True))
            elif spanner in spanner_locations:
                available_spanners.append((spanner, spanner_locations[spanner],
                ↪  False))

        # Assign spanners to nuts greedily
        total_cost = 0
        used_spanners = set()
        for nut in loose_nuts:
            nut_loc = nut_locations.get(nut)
            if not nut_loc:
                continue  # Skip if nut location not found
```

```
            min_cost = float('inf')
            best_spanner = None
            for spanner_info in available_spanners:
                spanner, s_loc, is_carried = spanner_info
                if spanner in used_spanners:
                    continue

                if is_carried:
                    distance = self.shortest_paths[man_location].get(nut_loc,
                    ↪  float('inf'))
                    cost = distance + 1
                else:
                    d1 = self.shortest_paths[man_location].get(s_loc,
                    ↪  float('inf'))
                    d2 = self.shortest_paths[s_loc].get(nut_loc, float('inf'))
                    cost = d1 + 1 + d2 + 1 if d1 != float('inf') and d2 !=
                    ↪  float('inf') else float('inf')

                if cost < min_cost:
                    min_cost = cost
                    best_spanner = spanner

            if best_spanner is not None:
                total_cost += min_cost
                used_spanners.add(best_spanner)
            else:
                total_cost += 1000000  # Penalize for missing spanner

        return total_cost
```

B.3 Childsnack

In *Childsnack*, one must prepare and deliver sandwiches to children, some of whom are allergic to gluten. Ingredients are stored at the kitchen and consumed when a sandwich is made. If both ingredients are gluten-free, the resulting sandwich is gluten-free. Sandwiches must be placed on trays at the kitchen, which can be moved between locations, while the children wait at specific locations. The objective is to serve every child by producing sandwiches that respect their allergies.

In the initialization step, the generated heuristic counts the total numbers of allergic and non-allergic children. Then, it counts the unserved allergic and non-allergic children and the number of available gluten-free and regular sandwiches. Next, it estimates the cost to produce and place the missing sandwiches and the cost to move trays to each waiting location. The heuristic value is the sum of all these costs. Because it ignores tray reuse and that gluten-free sandwiches can be served to non-allergic children, it can overestimate the optimal plan length.

B.4 Floortile

In the *Floortile* domain, robots must paint a grid of tiles with specific colors. Robots may move only onto clear tiles, and moving onto a tile or painting it makes that tile “not clear”. Each robot holds a single color and can change to any available color. Painting is performed from an adjacent tile above or below a tile while holding the target color. The objective is to paint all required tiles without blocking access to unpainted tiles that need painting.

The DeepSeek R1 heuristic identifies all required unpainted tiles and their target colors. For each such tile, it calculates the Manhattan distance to every robot and selects the closest ones as candidates. The cost for that tile is estimated as this minimum distance, plus one if none of the candidate robots hold the required color, and another one for the paint action. The total heuristic value is the sum of these costs over all these tiles. This heuristic can overestimate the optimal plan length because it considers tiles independently. Furthermore, by ignoring the blocking constraints that arise from painted tiles, it fails to identify dead-end states.

B.5 Miconic

The *Miconic* domain models an elevator transporting passengers between floors. This domain is *2-approximable* [37].

The selected heuristic first builds an undirected graph of the floors and precomputes all-pairs shortest paths using breadth-first search. Then, for each unserved passenger, it estimates the remaining cost. If a passenger is already on board, the cost is the distance from the floor of the elevator to its destination plus one (for the depart action). If a passenger is waiting, the cost is the distance from the floor of the elevator to its origin, plus the distance from its origin to its destination, plus two (for the board and depart actions). The total heuristic value is the sum of these costs. Because it ignores that the elevator could serve multiple passengers at once, this heuristic can overestimate the optimal plan length.

B.6 Rovers

In the *Rovers* domain, a team of rovers explores the surface of a planet to collect data and communicate it back to a lander. Data can be soil or rock samples, or images. Rovers have specific equipment for each task. To collect a sample, a rover must travel to a waypoint, have the right equipment, and an empty storage unit. To take an image, a rover with a camera must first calibrate it at a specific location and then move to a waypoint from which the objective is visible.

During initialization, the selected heuristic processes static information, such as waypoint visibility, builds a traversal graph for each rover, and records their equipment. When evaluating a state, the heuristic iterates through each unachieved goal and estimates the cost to achieve it. If the data has already been collected, the cost is the shortest path for that rover to a waypoint visible from the lander, plus one for the communication action. If the data has not been collected, the cost is estimated by finding an equipped rover that can achieve the goal with minimum cost. This cost includes moving to the goal location, performing the collection or imaging action, moving to a communication waypoint, and communicating. For images, the cost of calibration is also included. Distances are computed on-the-fly using a breadth-first search. The total heuristic value is the sum of these costs over all unachieved goals. This heuristic can also overestimate the optimal plan length.

B.7 Sokoban

Sokoban is a classic **PSPACE**-complete problem [15] where an agent must push boxes to specific goal locations within a grid. The agent can move between adjacent empty locations. To push a box, the agent moves to an adjacent location occupied by a box, and the box is pushed to the next location in the same direction, which must be clear. Since the agent can only push boxes, never pull them, Sokoban has dead-end states.

The heuristic first precomputes all-pairs shortest paths between locations using a breadth-first search. When evaluating a state, it sums the shortest-path distances from each box to its goal location. To this sum, it adds the shortest-path distance from the agent to the closest box not yet at its goal. This heuristic cannot overestimate the optimal plan length, but it ignores push constraints and box interactions and thus fails to identify dead-end states.

B.8 Transport

In the *Transport* domain, vehicles must deliver packages between locations connected by a road network. Vehicles can move between connected locations, pick up packages, and drop them. The key constraint is that each vehicle has a limited capacity. Picking up a package consumes and dropping it frees one unit of capacity. The objective is to transport all packages to their specified goal locations.

The selected heuristic first precomputes all-pairs shortest paths between locations using a breadth-first search on the road network. When evaluating a state, it iterates through each package not yet at its goal. If a package is already in a vehicle, the cost is the shortest-path distance from the current location of the vehicle to the goal of the package, plus one for the drop action. If the package is at a location, the heuristic greedily assigns it to a vehicle that minimizes the cost, which is calculated as the sum of the travel distance of the vehicle to the package, the travel distance of the package from its current location to its goal, and two actions for pick-up and drop. The total heuristic value is the sum

of these costs for all packages. This heuristic can overestimate the optimal plan length because it does not account for one vehicle delivering multiple packages.

C End-to-End Plan Generation Prompt: Blocksworld

Below we show our prompt used for end-to-end plan generation for the smallest task in the test set of the Blocksworld domain. The parts of the prompt that change for different tasks and domains are the name of the domain, the `domain-file`, and `instance-file-1`. As before, to reduce the number of pages, we do not display the entire domain and instance files, but just the beginning of each. We also show only the first few actions of each example plan.

```
<problem-description>
You are a highly-skilled professor in AI planning searching a plan for a PDDL task from the domain <domain>blocksworld</domain>. You will be given the PDDL domain and the PDDL instance, and you need to return the plan in the format shown below. Next, you will receive a sequence of examples for your task and finally the definition of the blocksworld domain.
</problem-description>

This is the PDDL domain file of the blocksworld domain:
<domain-file>
(define (domain blocksworld)
[...]
</domain-file>

This is the PDDL instance file, for which you need to find a plan:
<instance-file-1>
(define (problem blocksworld-01)
[...]
</instance-file-1>

This is the PDDL domain file of another domain, called Gripper, which serves as an example:
<gripper-domain-file>
(define (domain gripper-strips)
[...]
</gripper-domain-file>

This is an example of an instance file from the Gripper domain:
<gripper-instance-file-example>
(define (problem strips-gripper-x-20)
[...]
</gripper-instance-file-example>

This is a plan for the Gripper instance above:
<plan-gripper>
(pick ball9 rooma right)
(move rooma roomb)
(drop ball9 roomb right)
(move roomb rooma)
[...]
</plan-gripper>

This is the PDDL domain file of another domain, called Logistics, to serve as a second example:
<logistics-domain-file>
(define (domain logistics-strips)
[...]
</logistics-domain-file>

This is an example of an instance file from the Logistics domain:
<logistics-instance-file-example>
(define (problem strips-log-y-5)
```

```
[...]
</logistics-instance-file-example>

This is a plan for the Logistics instance above:
<plan-logistics>
(load-truck package2 truck12 city5-2)
(drive-truck truck12 city5-2 city5-1 city5)
[...]
</plan-logistics>

Provide only the plan for the given instance. Here is a checklist to help you with your task:
1) The plan must be in the same format as the examples above.
2) Use the tags "<plan>...</plan>" around the plan.
3) The actions in the plan must be from the set of actions in the domain blocksworld, that is, they must use the same name and same number of parameters as one of the action schemas.
4) The plan must be valid, that is, each action must be applicable in the state it is applied in and the plan must end in a goal state.
```

D Ablation Study: Simplified Prompt Instructions for Blocksworld

Below we show the simplified instruction component used for the Blocksworld domain within our ablation study. This simplified version replaces the standard detailed instructions, while other parts of the prompt are retained. Only the modified instructions are shown below.

```
<problem-description>
Create a Python domain-dependent heuristic function for the PDDL domain <domain>blocksworld</domain>. The heuristic function you create will be used to guide a greedy best-first search to solve instances from this domain. The name of the heuristic should be blocksworldHeuristic. Next, you will receive a sequence of file contents to help you with your task and to show you the definition of the blocksworld domain.
</problem-description>
```

E Runtime Comparison of h^{FF} in Pyperplan and Fast Downward

Table 5 reports state expansions per second for the h^{FF} heuristic with GBFS in the Python planner (Pyperplan) and the C++ planner (Fast Downward) on the subset of tasks solved by both. The "Pyperplan" and "Fast Downward" columns report the total number of state expansions divided by the total search time (in seconds) taken to solve the tasks in each domain. This does not include preprocessing time (e.g., for grounding). The "Performance Increase" column is "Fast Downward" divided by "Pyperplan".

Table 5: Expansions per second for GBFS with h^{FF} in Pyperplan and Fast Downward.

Domain (# Tasks)	Pyperplan	Fast Downward	Performance Increase
Blocksworld (24)	111.96	8 559.94	76.46
Childsnack (17)	893.26	15 890.78	17.79
Floortile (10)	2 758.31	87 681.02	31.79
Miconic (74)	1.24	829.57	669.00
Rovers (27)	109.04	28 710.18	263.30
Sokoban (31)	149.45	20 059.83	134.22
Spanner (30)	1 375.35	3 255.00	2.37
Transport (29)	3.11	305.49	98.23

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims are supported by the extensive discussion of our experimental results. We define the scope to be classical planning already in the abstract.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper discusses the limitations of the proposed approach in Section 4. Specifically, we discuss the impact of different LLMs used in our experiments, the generated heuristics' limitations, and the limitations of the specific prompt approach used with the ablation study.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: We do not present theoretical results in the paper.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The paper fully discloses all information necessary to reproduce the main experimental results. This includes: the complete prompt with instructions; the domain-dependent heuristics developed; specifications of the planning software and large language models used; and a detailed description of the experimental setup. To further ensure reproducibility, the complete source code, execution logs, and all generated heuristics are also provided.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: All code, the prompt, generated heuristics, execution logs, and scripts required to reproduce our main experimental results are provided as supplemental material. A README file contains detailed instructions. Upon acceptance, these materials will be made available in a public repository.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We specify the data used (from IPC 2023), the hyperparameters used by the LLMs, the hardware used, and the experimental setup in Section 4.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

Justification: The paper does not report error bars or formal statistical significance tests for the primary experimental results (e.g., coverage on test sets). Our evaluation focuses on metrics such as tasks solved (coverage), plan length, and state expansions. Once a heuristic is generated by the LLM and selected through our described process, its performance on a given planning task is deterministic. While the heuristic generation involves LLM sampling, the main reported results are for a single, deterministically chosen heuristic per approach. This evaluation methodology, emphasizing direct comparison of performance metrics like coverage on benchmark tasks, is standard practice in the planning community. Moreover, Table 4 reports the averages and standard deviations for all generated heuristics with the our best method.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 4 reports this information.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Our work used publicly available datasets. No harms were caused by this research, and there are no foreseen potential harmful impacts from our results.

   Guidelines:

- The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: There is no direct societal impact from our results. Our work is situated on a more foundational level (i.e., how to plan using LLMs) and does not address specific uses of it.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The LLMs and the datasets we use are already publicly available and widespread, and they already have their safeguards in place.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All assets which were not created by us are available online and are credited in the paper.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The created assets (code) are documented and explained.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The paper describes the usage of LLMs in the core methodology, specifically for generating heuristics for classical planning. The paper also discusses the limitations of the generated heuristics and the impact of different LLMs on performance.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.