Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)

# Successor-Generator Planning with LLM-generated Heuristics

**Alexander Tuisov**[1], **Yonatan Vernik**[2], **Alexander Shleyfman**[2]

[1]Independent Researcher
[2]Computer Science Department, Bar-Ilan University
queldelan@gmail.com, yonatanw55@gmail.com, alexander.shleyfman@biu.ac.il

## Abstract

Heuristics are a central component of deterministic planning, particularly in domain-independent settings where general applicability is prioritized over task-specific tuning. This work revisits that paradigm in light of recent advances in large language models (LLMs), which enable the automatic synthesis of heuristics directly from problem definitions – bypassing the need for handcrafted domain knowledge. We present a method that employs LLMs to generate problem-specific heuristic functions from planning tasks specified through successor generators, goal tests, and initial states written in a general-purpose programming language. These heuristics are compiled and integrated into standard heuristic search algorithms, such as greedy best-first search. Our approach achieves competitive, and in many cases state-of-the-art, performance across a broad range of established planning benchmarks. Moreover, it enables the solution of problems that are difficult to express in traditional formalisms, including those with complex numeric constraints or custom transition dynamics. We provide an extensive empirical evaluation that characterizes the strengths and limitations of the approach across diverse planning settings, demonstrating its effectiveness.

## Introduction

Deterministic AI planning involves finding a sequence of actions that moves an agent from an initial state to a goal state, given a formal description of the environment's dynamics (Ghallab, Nau, and Traverso 2004). Because state spaces are usually exponential or even infinite, effective search strategies require heuristics – functions that estimate the distance to the goal and help prioritize promising states (Pearl 1984; Bonet and Geffner 2001). Heuristic search has therefore become a central paradigm in classical planning, underpinning many of the most successful planners developed over the past decades (Hoffmann 2003; Helmert 2006; Scala, Haslum, and Thiébaux 2016; Aldinger and Nebel 2017).

Classical planning has traditionally focused on domain-independent heuristics, which offer broad applicability and avoid the need for task-specific design (Bonet and Geffner 2001). However, the performance of such heuristics can vary with the representation language and the structural characteristics of the domain. In more expressive settings, general heuristics may encounter challenges and need adaptation to remain effective across different versions of PDDL (McDermott et al. 1998; Fox and Long 2003; Edelkamp and Hoffmann 2004) (e.g., cf. classical vs. numeric settings (Hoffmann 2001, 2003; Helmert and Domshlak 2009; Kuroiwa et al. 2022)). For further discussion of current limitations within PDDL see Edelkamp (2003); Rintanen (2015).

In this work, we propose a method for *automatically* generating problem-specific heuristic functions from formal problem definitions using LLMs. The generated heuristics are then used by a sound search algorithm, such as greedy best-first search (GBFS). This setup preserves the systematic and provably correct behavior of classical search methods, while making heuristic construction fully automatic and eliminating the need for expert-designed heuristics.

We represent each planning task using three components, implemented in a general-purpose programming language: a successor generator, a goal test, and an initial state (Russell and Norvig 1995; Guan et al. 2023; Oswald et al. 2024). We refer to this representation as an *Explicit Successor Generator* (ESG). These ESG components, together with a prompt, are provided as input to the LLM, which returns a heuristic function (expressed in the same programming language) tailored to the structure of the given problem. The heuristic is then compiled and used to guide search without any further interaction with the LLM.

This approach offers several benefits. It removes the need for repeated LLM calls during planning—a source of significant inefficiency in prior work (Katz et al. 2024; Valmeekam et al. 2022; Kambhampati et al. 2024). It also provides transparent search components: both the LLM-generated heuristic and the task specification are expressed in a general-purpose programming language (Rust in this work), allowing inspection, debugging, and testing. In addition, it accommodates planning tasks that involve constructs difficult to model in traditional frameworks, including recursive goals, intricate numeric conditions, the creation of new variables, or custom transition logic.

We show that LLM-generated heuristic can strengthen general-purpose ESG planning. Our evaluation includes a set of standard numerical planning benchmarks represented in the ESG form (using Rust), as well as new benchmarks that are challenging to encode in PDDL. Across many domains, the resulting heuristics achieve strong performance,

matching the level of state-of-the-art numeric planners in numerous cases. Furthermore, the method is able to solve planning tasks that currently lack formal encodings in existing planning languages.

## Related Work

This paper revisits AI planning by heuristic search, by replacing domain-independent heuristics with LLM-generated domain-specific ones. Below, we review related work on LLMs and planning. For a more comprehensive overview of LLMs as planning modelers, see the surveys by (Tantakoun, Zhu, and Muise 2025) and by (Aghzal et al. 2025).

**Planning with LLMs** (Valmeekam et al. 2022) presents an early evaluation of LLM-based planning, showing that even on simple benchmarks, LLMs are not consistently reliable planners. While they cannot yet generate plans efficiently, our work shows they are effective within a broader planning framework, for example by generating heuristics and search components rather than planning directly.

A closely related approach is presented by (Katz et al. 2024), who argue that existing LLM-based planning methods are neither sound nor complete and incur high computational costs due to repeated LLM calls. They propose an alternative, *Thought of Search (ToS)*, in which LLMs are used to generate symbolic search components—successor functions and goal tests. These can then be executed efficiently without further model queries. Although their work focuses on small, finite search problems such as the *24 game* and *Mini Crosswords*, they provide compelling evidence that such an approach can drastically reduce computational cost and improve correctness (solved by BFS). The work shows that with respect to planning their approach overperforms the planning directly with LLM approaches such as: Chain-of-Thought (Wei et al. 2022), ReAct (Yao et al. 2023b), ReWOO (Xu et al. 2023), Reasoning via Planning (Hao et al. 2023), Tree of Thoughts (Yao et al. 2023a), Graph of Thoughts (Besta et al. 2024), Reflexion (Shinn et al. 2023), Algorithm of Thought (Sel et al. 2024). The ToS approach runs a breadth-first search (BFS) using components—successor functions and goal tests generated by LLM. Our approach extends this intuition by employing heuristic search methods to address problems of significantly greater complexity than those solvable through naive BFS—specifically in AI planning, where transition systems often become infinite due to the presence of unbounded numeric fluents. Thus, generating effective heuristics is crucial, underscoring that the symbolic utilization of LLMs can be successfully applied beyond toy domains.

A recent study in a similar vein to our work is presented in (Corrêa, Pereira, and Seipp 2025). While their approach targets planning tasks specified in PDDL, ours departs from this framework by operating directly on successor generators and goal-check functions. This shift affords greater representational flexibility, enabling the modeling of tasks that PDDL cannot express effectively. However, abandoning PDDL's structured representation also limits our ability to exploit domain structure for heuristic derivation.

**Heuristic Generation Outside of Planning** LLMs have also been explored for heuristic generation beyond automated planning, notably in domains such as online bin packing and TSP (Liu et al. 2024), as well as other combinatorial optimization problems (Ye et al. 2024). These approaches typically combine LLMs with evolutionary algorithms to evolve heuristics that guide the optimization process. However, due to fundamental differences in problem structure, these methods are not directly applicable to planning tasks without substantial adaptation.

**Problem Generation** Recent work has explored using LLMs to generate domains and problem descriptions using PDDL2.1 (Fox and Long 2003), among such works are (Guan et al. 2023; Liu et al. 2023; Oswald et al. 2024). For instance, (Silver et al. 2024) demonstrated that GPT-4 can output plans from parsed PDDL2.1 inputs using simple, non-search-based strategies. (Guan et al. 2023) proposed an approach in which PDDL2.1 actions are generated one at a time via LLM queries, refined using human feedback. A related concept appears in (Zhou et al. 2024), who use LLMs to generate, execute, and refine Python code for solving mathematical reasoning problems. Their findings support the idea that LLMs can produce verifiable code with minimal manual intervention. To reduce human involvement even further, we propose generating Rust code instead of Python. Since Rust is a compiled language with strong static typing, much of the burden of code verification can be offloaded to the compiler.
**Code Generation** Our approach relies on LLMs not to solve problems directly, but to generate search guidance in the form of state evaluation heuristic code. (Zhou et al. 2024) pursued a similar strategy in mathematical domains, showing that LLMs can generate and iteratively improve executable code. Our work builds on this insight, but targets search guidance functions within a planning framework. The viability of this approach is supported by recent advances in program synthesis and LLM-based code generation (Madaan et al. 2023; Zhang et al. 2023; Chen et al. 2024; Muennighoff et al. 2024; Zhong, Wang, and Shang 2024), which show that language models are increasingly capable of producing correct, type-safe, and semantically meaningful programs.

## Deterministic AI Planning

AI planning refers to the problem of computing a sequence of actions that transforms a known initial state into one that satisfies a specified goal condition. Unlike reinforcement learning, which learns behavior through trial-and-error interaction with an environment, AI planning assumes full knowledge of the state transition dynamics and the goal. The planner searches the combinatorial space of states reachable via actions to construct a valid plan.

Formally, a deterministic planning problem can be defined as a tuple $\Pi = \langle V, A, T, s_0, G \rangle$, where: $V$ is a set of variables with either finite or numeric domains, $A$ is a set of symbolic actions, $T$ is a deterministic transition function, $s_0 \in S$ is the initial state, and $G$ is the goal description. A state $s \in S$ is a full assignment over the variables in $V$. Since the state space $S$ is typically exponential in $|V|$ (or even in-

finite in the numeric case), it is not explicitly represented in the input. The goal description $G$ is usually represented as a set of conditions. A state $s_* \in S$ is considered a goal state if it satisfies $G$, i.e., for each $\varepsilon \in G$ the state $s_*$ satisfies $\varepsilon$. The transition function $T$ maps a state-action pair to the resulting state, i.e., $T(s, a) = s'$. A solution (or *plan*) for $\Pi$ is a sequence of actions $\langle a_1, \ldots, a_n \rangle$ such that applying them in order leads from $s_0$ to a goal state $s_n$.

Planning problems can be classified as either *satisficing*, where the goal is to find any valid plan, or *optimal*, where the goal is to find a cost-minimizing plan. In this work, we focus on satisficing planning, which prioritizes informative heuristics over theoretically sound ones (e.g., admissibility).

Solutions to planning problems are typically found via heuristic search algorithms such as GBFS or A* (Hart, Nilsson, and Raphael 1968; Pearl 1984). These algorithms are guided by heuristic functions that estimate the cost or distance from a given state to the goal (Pearl 1984). A heuristic function $h : S \to \mathbb{R}$ provides such estimates, often derived using domain-independent approximations, such as relaxations of the planning problem. Traditional heuristics are based on symbolic representations and formal action models (e.g., in PDDL2.1) (Fox and Long 2003), but recent work also explores learned heuristics (Oswald et al. 2024; Chen, Thiébaux, and Trevizan 2024). Informative heuristics greatly reduce the number of explored states, making them essential for practical planning. In this work, we use an LLM to generate a heuristic function based on the formal problem description written in a general-purpose programming language (Rust). This heuristic is integrated into a GBFS planner to obtain satisficing solutions.

## Methodology

We start with the **overview** of our approach. The process begins by accepting two types of input: structured representations provided in PDDL2.1 (Fox and Long 2003) format and general problem descriptions given in natural language or in other non-standardized formats. The problems specified in PDDL2.1 are manually translated into Rust functions that implement a *successor generator* and a *goal-testing* function. For general problem descriptions, we encode them into Rust according to their specifications. Moreover, the initial and the goal states are specified separately in a JSON file. The manual translation is necessary because current LLMs are not sufficiently reliable for this task, and existing tools lack the maturity required to perform it effectively. The automatic translations to ESG formalism is left for future work.

With the foundational components in place, we employ an LLM to generate the heuristic function. The LLM is provided with the Rust code for the successor generator and goal-testing function, along with a prompt designed to produce a heuristic function in Rust suited to the problem. The prompt[1] has three components:

1. Conditioning the model to be a senior engineer in the GPL used (Anam 2025) and providing the format the resulting heuristic must follow. This component is spread across both the system prompt and the user message[2]
2. Requesting that the LLM generate a heuristic.
3. Providing the model with the GPL domain implementation (the ESG).

As an additional variation of our method, the user message may also contain a fourth component:

4. The JSON representation of the instance to be solved.

We analyze the impact of instance-specificity in the heuristic generation pipeline in the Ablation Study section.

The heuristic, successor generator, and goal-testing functions are then integrated into a standard GBFS framework, where the search is guided by the generated heuristic. To ensure correctness, all solutions are verified using appropriate mechanisms based on the input type. For problems specified in PDDL2.1, we rely on a standard PDDL verifier—VAL (Howey, Long, and Fox 2004), which cross-checks the solution against the formal specification of the problem. For general problem descriptions, we verify by applying the actions in the plan sequentially, verify the preconditions are met, and that the last state fulfills goal-test. For the visualization of the workflow see Fig. 1. This workflow integrates LLM capabilities with automated code generation and classical search techniques, enabling efficient and flexible problem-solving for a wide range of planning domains. Note that the only element of the workflow that still requires human intervention is the specification of the problem.

## Empirical Evaluation

The experiments were conducted on a 13th Gen Intel® Core™ i9-13900 processor running at 2.00 GHz, supported by a 64-bit operating system and 32 GB of RAM. To ensure consistency and reliability, each experiment was constrained by a time limit of 10 minutes and a memory usage limit of 8 GB. We test our approach in the context of satisficing numeric planning. We have evaluated our approach on the domains from the Numeric International Planning Competition (IPC) 2023 (Taitler et al. 2024).[3] Note that even problems with three unbounded numeric variables have infinite size state spaces and in general are undecidable (Gnad et al. 2023). As a proof-of-concept, we also include two domains that cannot be easily expressed using PDDL:

**Twin Prime:** The initial state comprises a set of integers and buffers, each holding a single integer. The actions include addition, subtraction, multiplication, and integer division between two registers, with results stored in one register. The goal is to produce a twin prime number exceeding a specified threshold in one of the buffers—a goal that is not easily expressible in PDDL.

**Deterministic Pacman:** The game is defined on a grid containing walls, pellets, ghosts, power-ups, and a single Pacman agent. Pacman moves in one of four cardinal directions per turn, consuming pellets and power-ups upon entry into

[1] The exact prompt for both variations is given in the Appendix.

[2] We include guidance to avoid compilation errors, however the guidance could be *far* better optimized. This optimization, however, is out of scope for this work.

[3] https://github.com/ipc2023-numeric

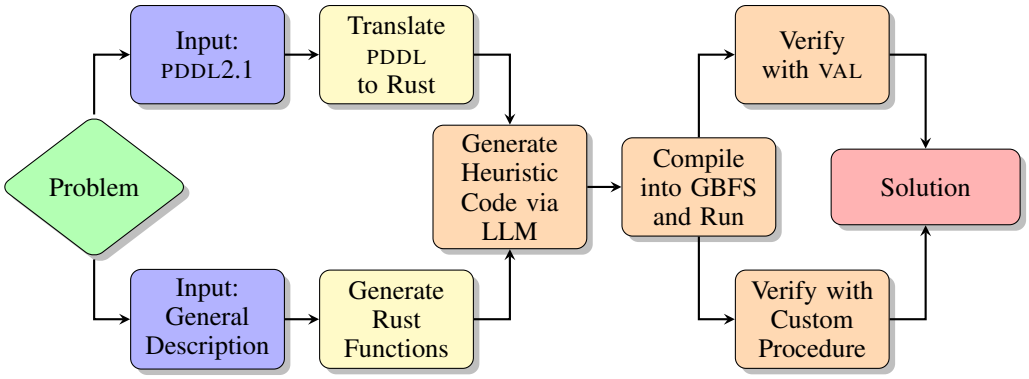



Figure 1: Procedure flowchart. A general problem description is manually written using Rust components—successor generator, goal test, and initial state—which are then fed into the system. For problems already described in PDDL2.1, the Rust translation is derived directly from the encoding.

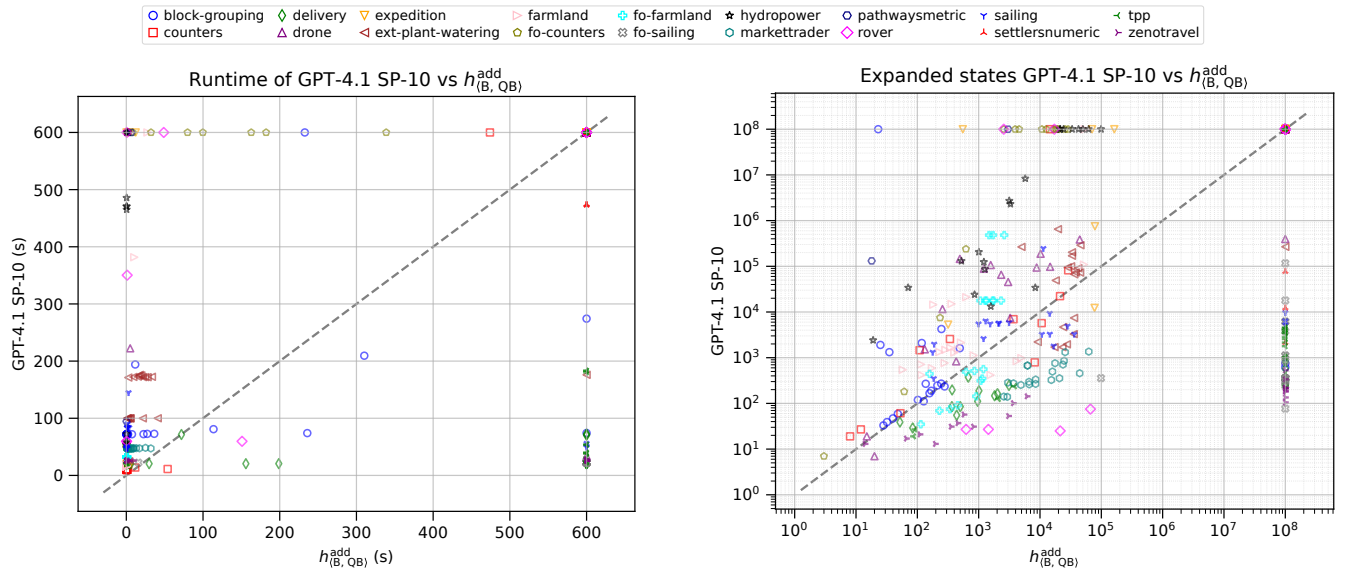
| | | Baseline | | Domain-Independent Planners | | | | | FIRSTCOMPILATION (FC) | | | UNTILSUCCESS (US) | | | SELFPORTFOLIO-10 (SP-10) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Rust | | ENHSP-20 | | | NFD | MFF | GPT | | Claude | GPT | | Claude | GPT | | Claude |
| Domain | | BFS | $h^{\text{md}}$ | $h^{\text{md}}$ | $h^{\text{add}}_{\langle \text{B, QB} \rangle}$ | $P(3h\|\|3n)^{\ddagger}$ | $h_{\text{LMC}}$ | $h_{\text{FF}}$ | 4.1 | 5.1$^{\dagger}$ | Sonnet 4.5$^{\dagger}$ | 4.1 | 5.1$^{\dagger}$ | Sonnet 4.5$^{\dagger}$ | 4.1 | 5.1$^{\dagger}$ | Sonnet 4.5$^{\dagger}$ |
| Block Grouping | (20) | 0 | 9 | 15 | 18 | **20**$^{\ddagger}$ | 0 | 2 | 16 | 11 | 15 | 19 | 17 | 18 | 19 | 18 | 18 |
| Counters | (20) | 3 | 10 | 13 | 11 | **20**$^{\ddagger}$ | 12 | 15 | 10 | 10 | 5 | 10 | 10 | 5 | 10 | 10 | 5 |
| Delivery | (20) | 1 | 9 | 17 | 12 | 18$^{\ddagger}$ | 9 | **20** | 16 | 16 | 16 | **20** | 16 | **20** | **20** | 17 | **20** |
| Drone | (20) | 3 | 16 | **20** | 15 | **20**$^{\ddagger}$ | 16 | 16 | 15 | 16 | 16 | 16 | 16 | 17 | 16 | 16 | **20** |
| Expedition | (20) | 3 | 5 | 3 | **6** | **6**$^{\ddagger}$ | 5 | 5 | 3 | 3 | 2 | 3 | 3 | 4 | 3 | 3 | 4 |
| Farming | (20) | 3 | **20** | **20** | **20** | **20**$^{\ddagger}$ | 15 | 9 | 18 | **20** | **20** | 19 | **20** | **20** | 19 | **20** | **20** |
| FO-Counters | (20) | 3 | 7 | 9 | 15 | 15$^{\ddagger}$ | 6 | **20** | 4 | 5 | 6 | 4 | 6 | 6 | 4 | 6 | 6 |
| FO-Farming | (20) | 3 | **20** | **20** | **20** | **20**$^{\ddagger}$ | 11 | 16 | **20** | 17 | **20** | **20** | **20** | **20** | **20** | **20** | **20** |
| FO-Sailing | (20) | 0 | 0 | 0 | 1 | 3$^{\ddagger}$ | 16 | 11 | **20** | 0 | 11 | **20** | **20** | 18 | **20** | **20** | 18 |
| Hydropower | (20) | 8 | 4 | 4 | **20** | **20**$^{\ddagger}$ | 6 | 1 | 8 | 6 | 8 | 12 | 12 | 8 | 12 | **20** | 8 |
| Market Trader | (20) | 0 | 7 | 17 | **20** | **20**$^{\ddagger}$ | 0 | 0 | **20** | 0 | 0 | **20** | 1 | **20** | **20** | 1 | **20** |
| Pathways | (20) | 0 | 1 | 0 | 2 | **13**$^{\ddagger}$ | 2 | **13** | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| Plant Watering | (20) | 0 | 1 | **20** | 19 | **20**$^{\ddagger}$ | **20** | 13 | 9 | 10 | 6 | **20** | **20** | **20** | **20** | 19 | **20** |
| Rover | (20) | 2 | 4 | 10 | 6 | **14**$^{\ddagger}$ | 4 | 10 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Sailing | (20) | 0 | 0 | 0 | 17 | **20**$^{\ddagger}$ | 10 | 2 | 11 | 19 | 19 | **20** | **20** | 19 | **20** | **20** | 19 |
| Settlers | (20) | 0 | 1 | 0 | 0 | **8**$^{\ddagger}$ | 0 | 6 | 1 | 1 | 3 | 1 | 1 | 3 | 3 | 1 | 3 |
| TPP | (20) | 1 | 16 | **20** | 4 | **20**$^{\ddagger}$ | 2 | 4 | 13 | 0 | 0 | 16 | 4 | 2 | 16 | 4 | 2 |
| Zenotravel | (20) | 4 | 11 | 14 | 11 | 18$^{\ddagger}$ | 10 | 0 | **20** | **20** | 14 | **20** | **20** | 17 | **20** | **20** | 17 |
| $\sum$ | (360) | 34 | 141 | 202 | 217 | **295**$^{\ddagger}$ | 144 | 163 | 207 | 158 | 165 | 245 | 211 | 222 | 247 | 220 | 225 |
| Twin Prime | (20) | 4 | **20** | - | - | - | - | - | 18 | 17 | 17 | 18 | 17 | 17 | 18 | **20** | 17 |
| Pacman | (20) | 1 | 14 | - | - | - | - | - | 9 | 13 | 10 | 9 | 14 | **16** | **16** | **16** | **16** |
| $\sum$ | (400) | 39 | 175 | - | - | - | - | - | 234 | 188 | 192 | 272 | 242 | 255 | **281** | 256 | 258 |

Table 1: Coverage results of the baseline, domain-independent planners, the best portfolio, and the LLM-generated heuristics. We report our results from GPT-4.1, GPT-5.1, and Claude Sonnet 4.5 with configurations FIRSTCOMPILATION (FC), UNTILSUCCESS (US), and SELFPORTFOLIO-10 (SP-10). All configuration except BFS and MFF (which uses EHCS) use GBFS. Models marked with $^{\dagger}$ are reasoning models. For results on all models and configurations see Appendix. $^{\ddagger}$ The $P(3h||3n)$ portfolio uniformly divides time among three "best" ENHSP heuristics and three "best" novelty heuristics (Chen and Thiébaux 2024), where the notion of "best" is derived from performance on the same problem set used in our evaluation, rather than being fixed independently of it.

their cell. The goal is to clear all pellets while avoiding collisions with ghosts that behave in a deterministic manner. Consuming a power-up gives Pacman a temporary ability to banish ghosts and avoid harm from them. This domain cannot be easily expressed in PDDL or its extensions due to the complex interactions between Pacman, ghosts, and power-ups, which require temporal dynamics and conditional effects beyond PDDL's representational capabilities.

We evaluated our approach against the following state-of-the-art methods on the IPC domains. All of the planners

[4]A portfolio between the two achieves a coverage of 304, however this is a post-hoc optimization and not a fair comparison. We do however consider Domain-Independent+LLM a fruitful direction fo future work.




(a) Total Time (including failed generation + run pairs)

(b) Expanded States

Figure 2: Per-instance comparisons of the Total Time (left) and expanded states (right) between GPT-4.1 with SelfPortfolio-10 and $h^{\text{add}}_{\langle \text{B, QB} \rangle}$. Points below the diagonal favor our approach. On problems both solve they seem to provide similar levels of heuristic guidance.[4]

below support linear expressions in both conditions and assignment effects. As a *baseline*, we implement a "blind" BFS and GBFS with a manually implemented $h^{\text{md}}$ heuristic (Chen and Thiébaux 2024), running on a Rust implementation of the problems based on (Green and Izhaki 2025).

The $h^{\text{md}}$ heuristic estimates the total distance from a state $s$ to satisfying all goal conditions, inspired by the $h^{\text{md}}$ norm:

$$h^{\text{md}}(s) := \sum_{\varepsilon \in G} d(\varepsilon, s),$$

where $G$ is the set of goal conditions, and $d(\varepsilon, s)$ measures the distance from $s$ to satisfying $\varepsilon$.

For propositional goals, $d(\varepsilon, s) = 0$ if $s \models \varepsilon$ and 1 otherwise. For numeric goals of the form $\varepsilon : \psi \bowtie 0$, where $\psi : \mathbb{R}^n \to \mathbb{R}$ and $\bowtie \in \{>, \geq, =, \leq, <\}$, we define:

$$d(\varepsilon, s) := \inf_{y : \psi(y) \bowtie 0} |\psi(y) - \psi(s)|,$$

i.e., the distance from $\psi(s)$ to the nearest satisfying value of $\psi$. Chen and Thiébaux (2024) proposed this approach for linear constraints in PDDL2.1, but this heuristic extends naturally to non-linear formulas, though computing $d$ becomes more expensive.

**ENHSP-20** is a Java-based planner whose satisficing configurations use GBFS by (Scala et al. 2020), using the modified version released by Chen and Thiébaux (2024).[5]. Following (Chen and Thiébaux 2024) we compare against $h^{\text{md}}$, $h^{\text{mrp}}_{\text{+hj}}$, $h^{\text{add}}$, their novelty variants $h^{\text{md}}_{\langle \text{B, QB} \rangle}$, $h^{\text{mrp}}_{\text{+hj}\,\langle \text{B, QB} \rangle}$, $h^{\text{add}}_{\langle \text{B, QB} \rangle}$, as well as the three portfolio settings, $P(3h)$ (non-novelty only), $P(3n)$ (novelty only), and $P(3h \,\|\, 3n)$ (both). We report of them only $h^{\text{md}}$, the best-performing single heuristic $h^{\text{add}}_{\langle \text{B, QB} \rangle}$, and the best portfolio configuration, $P(3h \,\|\, 3n)$, while the full comparison is provided in Appendix.

**Metric-FF (MFF)** (Hoffmann 2003) is used off-the-shelf. Implemented in C, it employs Enforced Hill Climbing Search (EHCS) with the interval-relaxed $h_{\text{FF}}$ heuristic and Helpful Operators.[6] The planner is incomplete and may report problems as unsolvable due to limitations of EHCS.

**Numeric-FD (NFD)** (Aldinger and Nebel 2017) is a C++ planner that employs Lazy GBFS with numeric LM-cut heuristics (Kuroiwa, Shleyfman, and Beck 2022).[7] This planner won the Numeric IPC-2023 (see Taitler et al. 2024).

The **Pacman** and **Twin Prime** domains demand a level of expressiveness that makes encoding them for existing planners highly impractical and cumbersome in practice. As there currently are no available methods to derive a domain-independent heuristic for this type of tasks, we benchmarked them against BFS and $h^{\text{md}}$.

LLM-generated heuristics are not always compilable and vary in quality and efficiency generation-to-generation. To address this, we employ three fallback strategies:

In FIRSTCOMPILATION (FC), we repeatedly query the LLM until a compilable heuristic is obtained and run GBFS with it until a solution is found or resources are exhausted, 600 sec (search + all API calls).

In UNTILSUCCESS (US), we operate the same as FC, except we continue even if a compilable heuristic failed.

[5] github.com/hstairs/jpddlplus/tree/socs24-width-and-mq

[6] fai.cs.uni-saarland.de/hoffmann/metric-ff.html

[7] github.com/ipc2023-numeric/team-1

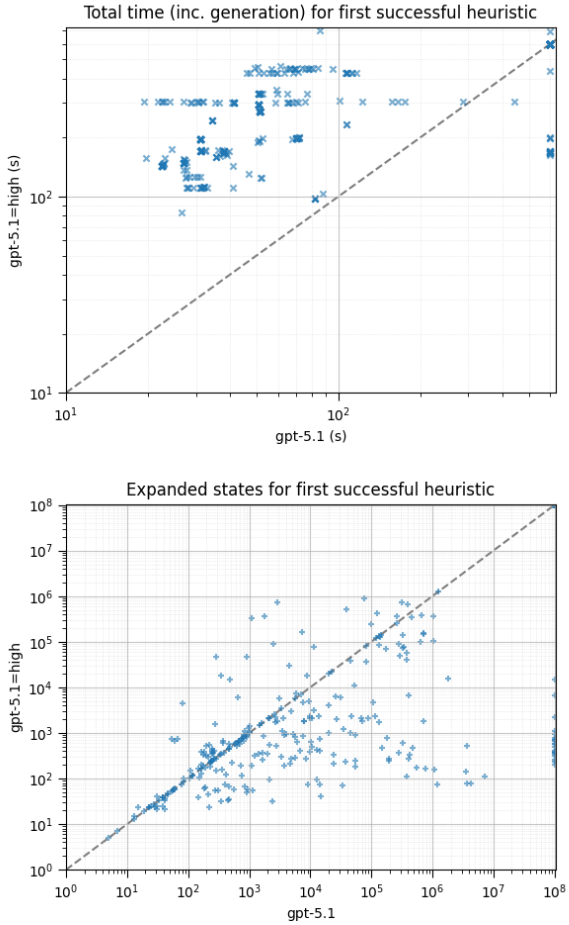



Figure 3: Per-instance comparison of the Total Time (generation + run) (up) and expanded states (down) between GPT-5.1 with and without setting the reasoning_effort to *high*. Points below the diagonal favor high reasoning effort. Allowing increased reasoning effort moderately increases heuristic quality, but comes at a heavy expense of time.

In SELFPORTFOLIO-N (SP-N), we allocate fixed $Time/N$s slices to N heuristics; if a run fails or exceeds memory, we restart with generating a new heuristic. This continues until a solution is found or all slices failed. All API call durations, typically 10–45 seconds depending on the model, are included in the time budget.

We evaluated the following LLMs and settings: OpenAI's GPT-4.1 and GPT-4.1-mini, their reasoning models GPT-5.1 and GPT-5-mini (under both a "low" and "high" reasoning effort setting), and Anthropic's Claude Sonnet 4.5 and Claude Haiku 4.5. Results are presented in Table 1. For practicality, we limit the maximum number of heuristics generated for FC and US to 10, which if passed the instance is declared failed. As is standard in AI planning, we report coverage rather than averages, since state space size typically grows exponentially and averaging would be misleading. As a qualitative measure of the generated heuristics we report the head-to-head time and expanded nodes for solution comparison of SP-10 with GPT-4.1 vs. the best performing $h^{\text{add}}_{\langle \text{B, QB} \rangle}$, which can be seen in Figure 2. Comparisons with additional models can be seen in Appendix.

We conducted ablation studies to evaluate the contribution of three components in the heuristic generation pipeline: (i) the provision of the specific instance (ii) the model and fallback choice (iii) the reasoning effort requested from reasoning models. The study evaluated their effect on heuristic quality, generation time, planning performance, and the variability of our best configuration (Table 4).

**Impact of Instance-Specificity** We evaluate the impact of providing the instance JSON, which contains the initial state and goal, before generation. This incurs a significant cost, as it requires generating heuristics per instance, unlike the *domain-specific-instance-general* approach, which is generated once and applied to all instances. Intuitively, IS may offer the following benefits:

1. The heuristic often contains multiple penalizing elements which are combined with guessed coefficients. Knowing the details of the instance gives the model a better idea on how to tune these coefficients.
2. It tells the model how significant each part of the problem is. Consider a simple instance of Pacman where the ghosts are trapped. The model can avoid spending generation- and run-time on a complex subroutine to quantify the risk from the ghosts.
3. It tells the model the scale of the problem which can provide guidance on efficiency constraints. We hypothesize that there exists a tradeoff between the precision of the heuristic estimates and the computation time, and further that for larger problems it can be worthwhile for the heuristic to sacrifice precision for efficiency. For example, on TPP, Anthropic's models implement path-finding algorithms like Dijkstra, Prim and Floyd-Warshall, about half the time. Although this is a good basis for a "smart" heuristic, it causes a timeout within a few thousand expansions for even medium problems. Meanwhile even Manhattan Distance heuristics achieves 16/20.
4. At times, a solution state is obvious from the initial state (e.g, FO-Counters). In these cases showing the model the instance details allows it to write a heuristic to guide efficiently toward that solution rather than abstractly guiding toward all solutions.

Empirically, IS leads to a decrease in overall coverage (Table 2) with most domains varying slightly but a few increasing or decreasing significantly. IS did not cause heuristic generation time to increase[8]. IS moderately increased compilation errors (see Table 3), and if this effect is normalized against, IS-heuristics have a slightly higher solving rate. We hypothesize that GPT-4.1 is insufficiently strong to make use of this info and that for a stronger model it may be worth revisiting.

[8]Interestingly, despite adding tokens and theoretically adding complexity which can increase output tokens, adding the instance slightly *decreased* generation time. (P=2e-5$<$0.05). We do not have an explanation for this fact.

| Domain | | FC DD | FC IS | US DD | US IS | SP-10 DD | SP-10 IS |
|---|---|---|---|---|---|---|---|
| Block Grouping | (20) | 16 | 9 | 19 | 10 | **19** | 10 |
| Counters | (20) | 10 | 8 | 10 | 11 | 10 | **11** |
| Delivery | (20) | 16 | 10 | 20 | 18 | **20** | 18 |
| Drone | (20) | 15 | 15 | 16 | 16 | **16** | **16** |
| Expedition | (20) | 3 | 1 | 3 | 4 | 3 | **4** |
| Farming | (20) | 18 | 19 | 19 | 20 | 19 | **20** |
| FO-Counters | (20) | 4 | 8 | 4 | 14 | 4 | **14** |
| FO-Farming | (20) | 20 | 18 | 20 | 20 | **20** | **20** |
| FO-Sailing | (20) | 20 | 10 | 20 | 19 | **20** | **20** |
| Hydropower | (20) | 8 | 7 | 12 | 9 | **12** | 9 |
| Market Trader | (20) | 20 | 1 | 20 | 3 | **20** | 3 |
| Pathways | (20) | 0 | 1 | 1 | 1 | **1** | **1** |
| Plant Watering | (20) | 9 | 12 | 20 | 18 | **20** | 18 |
| Rover | (20) | 3 | 2 | 4 | 3 | **4** | 3 |
| Sailing | (20) | 11 | 14 | 20 | 20 | **20** | **20** |
| Settlers | (20) | 1 | 2 | 1 | 3 | **3** | **3** |
| TPP | (20) | 13 | 11 | 16 | 13 | **16** | 15 |
| Zenotravel | (20) | 20 | 14 | 20 | 20 | **20** | **20** |
| $\sum$ | (360) | 207 | 162 | 245 | 222 | **247** | 225 |
| Twin Prime | (20) | 18 | 20 | 18 | 20 | 18 | **20** |
| Pacman | (20) | 9 | 13 | 9 | 13 | **16** | 14 |
| $\sum$ | (400) | 234 | 195 | 272 | 255 | **281** | 259 |

Table 2: Coverage results for GPT-4.1 with domain-dependence only (DD), our standard configuration, against GPT-4.1 with Instance-Specificity (IS), reporting each on FC, US, and their best configuration, SP-10. For most domains GPT-4.1 fails to leverage the additional information, however we highlight FO-Counters and Twin Prime as domains where it lead to the most improvement.

**Impact of model and fallback choice** We tested 6 models: GPT-4.1, GPT-4.1-mini, GPT-5.1, GPT-5-mini, Claude Sonnet 4.5, and Claude Haiku 4.5, each with each of our fallback options including FC, US, SP-5, and SP-10. For the full table see Appendix. Our observations are as follows: (i) for non-reasoning models, under about every configuration, a model's larger variant is preferred. Since they are well-capable of generating and running many heuristics within the allotted time, $SP-10$ is preferred; (ii) reasoning models are more competitive between sizes. They take longer to generate and $US$ performs best, allowing them the time they need; (iii) anecdotally, when reasoning models fail, it is often because they overcomplicate the problem, leading to longer generation times and unpredictable outcomes—sometimes producing better heuristics, other times unusable ones. That said, this risk may be acceptable if we can resample and the samples are sufficiently independent.

**Impact of Reasoning effort** We tested our method on the same model, GPT-5.1 with two configurations: *reasoning_effort='low'* and *reasoning_effort='high'* (for brevity denoted 'GPT-5.1' and 'GPT-5.1=high'). We found that although increasing $reasoning\_effort$ slightly increased heuristic efficiency and quality, and better-than-halved compilation error rate, when we account for generation time the reduced number of heuristics we generate in time does not make this trade worthwhile. With better engineering one could generate many completions simultaneously, which may turn this trade worthwhile. We leave this direction for future work. For further graphs see appendix.

The ablation results suggest that non-reasoning models don't significantly benefit from IS. The best setting of our method is to use a non-reasoning model to generate non-IS heuristics and run them as SelfPortfolio-10 (276 with GPT-4.1). If time is allowed ahead of receiving the instances, the best setting becomes instead to generate ahead of time as many heuristics as possible and run SP-N or US on them (The two fallbacks are approximately equivalent when used on pregenerated heuristics due to near-zero timeout rate).

## Discussion

As shown in Table 1, LLM-based heuristics achieve state-of-the-art performance across the evaluated domains. They are particularly effective in challenging benchmarks such as **(FO-) Sailing** and **Zenotravel**, which are known to be difficult for domain-independent planners. The best-performing LLM-based heuristics were produced by GPT-5.1 and it's mini variant with high reasoning effort, however when accounting for generation time and cost, GPT-4.1 emerges as a clear winner. Nonetheless, certain domains remain challenging for LLMs, likely due to their sensitivity to non-descriptive variable names and opaque goal specifications, as seen in **Pathways** and **Settlers**.

**Expressiveness** In the **Pacman** and **Twin Prime** domains, no available planner can solve problems requiring this level of expressive power, necessitating a comparison against BFS and $h^{\text{md}}$. As expected, LLM-based heuristics outperformed both. Overall, LLM-based heuristics surpass state-of-the-art numerical planners while addressing problems that cannot be adequately expressed in any existing PDDL dialect.

**Representation** One concern is that the performance gains observed with LLM-generated heuristics may result from optimized planner code rather than the heuristics themselves. To test this, we reimplemented $h^{\text{md}}$ within our planner for a direct comparison with ENHSP-20 using the same heuristic. As shown in Table 1, ENHSP-20 with $h^{\text{md}}$ generally achieves better results, particularly in domains such as **Hydropower** and **Plant Watering**, which are highly sensitive to problem representation. This suggests that departing from PDDL-based input to a direct successor-generator representation may, in some cases, degrade performance, allowing us to reject the hypothesis that our representation alone accounts for the observed improvements.

**Costs and Efficiency** Although concerns about LLM efficiency have been raised (Stojkovic et al. 2024), energy use is beyond this work's scope. Our approach supports efficient deployment: heuristics can be generated once per domain and reused across instances, reducing cost even with powerful models. The resulting heuristics are also fast to

[9]For GPT, subset of output tokens. For Claude, they are not provided by the API

| Metric | GPT | | | | | | | Claude | |
|---|---|---|---|---|---|---|---|---|---|
| | **4.1** | **4.1 (IS)** | **4.1 mini** | **5.1**† | **5 mini**† | **5.1 high**† | **5 m-h**† | **Sonnet 4.5**† | **Haiku 4.5**† |
| Input tok. | 3725 | 8102 | 3725 | 3724 | 3724 | 3724 | 3724 | 4761 | 4761 |
| Output tok. | 1510 | 1695 | 1245 | 3253 | 2298 | 18075 | 14485 | 2122 | 1490 |
| Reasoning tok.[9] | - | - | - | 1282 | 589 | 15510 | 11613 | - | - |
| Cost (USD) | 0.02 | 0.03 | 0.003 | 0.037 | 0.006 | 0.185 | 0.03 | 0.046 | 0.012 |
| Generation (sec.) | 35.8 | 27.5 | 19.4 | 44.9 | 44.5 | 235.1 | 202.3 | 33.6 | 12.5 |
| Run (sec.) | 24.7 | 22.1 | 19.2 | 24.2 | 18.0 | 21.4 | 36.7 | 43.4 | 22.2 |
| Compl. error (rate) | 0.300 | 0.430 | 0.215 | 0.150 | 0.405 | 0.070 | 0.205 | 0.195 | 0.165 |
| OOM (rate) | 0.304 | 0.241 | 0.388 | 0.382 | 0.280 | 0.340 | 0.288 | 0.391 | 0.451 |
| Runtime error (rate) | 0.027 | 0.020 | 0.035 | 0.032 | 0.028 | 0.035 | 0.032 | 0.016 | 0.032 |
| Timeout (rate) | 0.015 | 0.014 | 0.004 | 0.007 | 0.009 | 0.007 | 0.031 | 0.032 | 0.002 |
| Success (rate) | 0.355 | 0.295 | 0.358 | 0.430 | 0.278 | 0.548 | 0.444 | 0.365 | 0.351 |

Table 3: Average token usage, cost, latency, and execution outcomes for different GPT and Claude models, independent of downstream applications. Models marked with † are reasoning models. Excepting IS, input tokens across models depend only on the tokenizer. Costs do not account for token caching discounts.

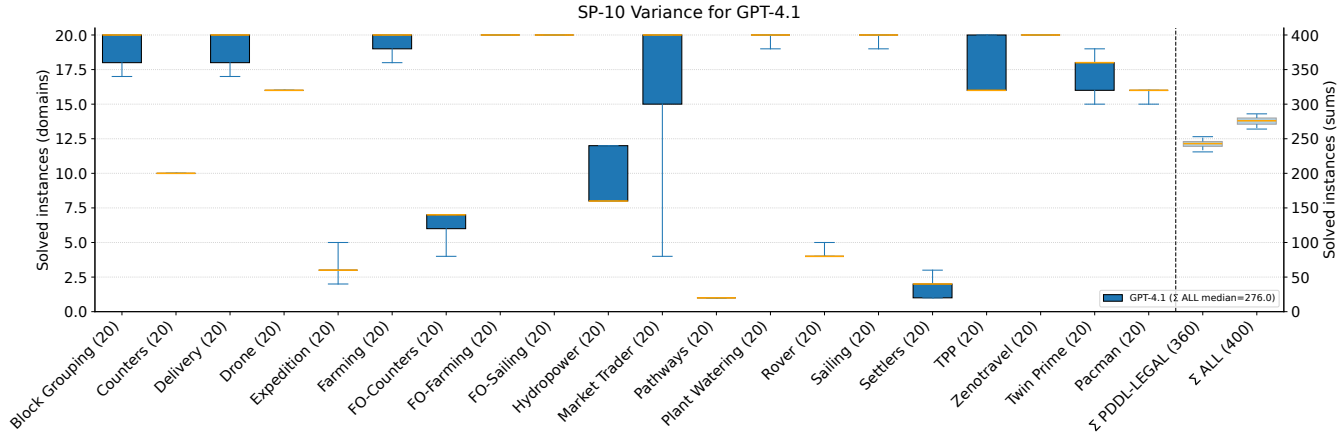



Figure 4: As a variance analysis we generated 40 heuristics with GPT-4.1 in each domain, and performed a Monte-Carlo simulation of our algorithm for 1000 iterations of sampling heuristic order, taking the coverage per-domain and overall each time. The boxes represent the first and third quartiles, the thick orange line the median, and the whiskers the 1st and 99th percentiles. Although limited, the analysis shows the method to be highly consistent for most domains.

run and tend to succeed or fail more quickly than domain-independent ones (see Appendix).

## Conclusion

This paper explored the use of LLMs to generate heuristic functions directly from AI planning task definitions, bypassing traditional domain-independent heuristics written in formal languages like PDDL. Our approach uses general-purpose code (Rust) to represent planning tasks and lets LLMs synthesize tailored heuristics.

Empirically, the results show that LLM-generated heuristics achieve state-of-the-art performance across several established planning benchmarks. The approach excels in domains traditionally difficult for domain-independent planners, such as **Zenotravel**, indicating substantial potential for efficiency gains. Furthermore, it successfully addresses complex planning tasks that are not easily expressible in conventional formalisms, like **Pacman** and **Twin Prime**.

Nonetheless, limitations remain. LLM-generated heuristics are sensitive to the clarity and descriptiveness of task representations; unintuitive state-space encodings degrade their quality. Moreover, while larger models such as GPT-4.1 and Sonnet 4.5 tend to produce high-quality heuristics, they incur greater computational costs. Smaller models offer improved cost-efficiency but often generate less reliable code. This trade-off highlights the importance of selecting models according to task complexity and scalability requirements.

This work opens several promising avenues for future research. These include improving robustness across domains, comparing LLM-generated heuristics to hand-crafted ones, and exploring hybrid strategies combining LLMs with domain knowledge or iterative refinement techniques.

## Acknowledgements

Alexander Shleyfman's and Yonatan Vernik's work was supported by ISF grant 2443/23.

## References

Aghzal, M.; Plaku, E.; Stein, G. J.; and Yao, Z. 2025. A Survey on Large Language Models for Automated Planning. *CoRR*, abs/2502.12435.

Aldinger, J.; and Nebel, B. 2017. Interval Based Relaxation Heuristics for Numeric Planning with Action Costs. In *SOCS*, 155–156.

Anam, R. K. 2025. Prompt Engineering and the Effectiveness of Large Language Models in Enhancing Human Productivity. *ArXiv*, abs/2507.18638.

Besta, M.; Blach, N.; Kubicek, A.; Gerstenberger, R.; Podstawski, M.; Gianinazzi, L.; Gajda, J.; Lehmann, T.; Niewiadomski, H.; Nyczyk, P.; and Hoefler, T. 2024. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. In *AAAI*, 17682–17690. AAAI Press.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33.

Chen, D. Z.; and Thiébaux, S. 2024. Novelty Heuristics, Multi-Queue Search, and Portfolios for Numeric Planning. In *SOCS*, 203–207.

Chen, D. Z.; Thiébaux, S.; and Trevizan, F. W. 2024. Learning Domain-Independent Heuristics for Grounded and Lifted Planning. In *AAAI*, 20078–20086. AAAI Press.

Chen, X.; Lin, M.; Schärli, N.; and Zhou, D. 2024. Teaching Large Language Models to Self-Debug. In *ICLR*. OpenReview.net.

Corrêa, A. B.; Pereira, A. G.; and Seipp, J. 2025. Classical Planning with LLM-Generated Heuristics: Challenging the State of the Art with Python Code. *arXiv preprint arXiv:2503.18809*.

Edelkamp, S. 2003. Limits and Possibilities of PDDL for Model Checking Software. In *Proceedings of the ICAPS 2003 Workshop on the Competition: Impact, Organisation, Evaluation, Benchmarks*.

Edelkamp, S.; and Hoffmann, J. 2004. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 20: 61–124.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Gnad, D.; Helmert, M.; Jonsson, P.; and Shleyfman, A. 2023. Planning over Integers: Compilations and Undecidability. In *ICAPS*, 148–152. AAAI Press.

Green; and Izhaki. 2025. PDDL domains in rust. https://github.com/DavidIzhaki/Domains_Project/tree/11ad675ce4d5689b9a2084af36b68740b664f96d. GitHub repository, commit `11ad675ce4d5689b9a2084af36b68740b664f96d`.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning. In *NeurIPS*.

Hao, S.; Gu, Y.; Ma, H.; Hong, J. J.; Wang, Z.; Wang, D. Z.; and Hu, Z. 2023. Reasoning with Language Model is Planning with World Model. In *EMNLP*, 8154–8173. Association for Computational Linguistics.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *ICAPS*, 162–169.

Hoffmann, J. 2001. FF: The Fast-Forward Planning System. *AI Mag.*, 22(3): 57–62.

Hoffmann, J. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *JAIR*, 20: 291–341.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *ICTAI*, 294–301. IEEE Computer Society.

Kambhampati, S.; Valmeekam, K.; Guan, L.; Verma, M.; Stechly, K.; Bhambri, S.; Saldyt, L.; and Murthy, A. 2024. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *ICML*. OpenReview.net.

Katz, M.; Kokel, H.; Srinivas, K.; and Sohrabi, S. 2024. Thought of Search: Planning with Language Models Through The Lens of Efficiency. In *NeurIPS*.

Kuroiwa, R.; Shleyfman, A.; and Beck, J. C. 2022. LM-Cut Heuristics for Optimal Linear Numeric Planning. In *ICAPS*.

Kuroiwa, R.; Shleyfman, A.; Piacentini, C.; Castro, M. P.; and Beck, J. C. 2022. The LM-Cut Heuristic Family for Optimal Numeric Planning with Simple Conditions. *J. Artif. Intell. Res.*, 75: 1477–1548.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *CoRR*, abs/2304.11477.

Liu, F.; Tong, X.; Yuan, M.; Lin, X.; Luo, F.; Wang, Z.; Lu, Z.; and Zhang, Q. 2024. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*.

Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; Gupta, S.; Majumder, B. P.; Hermann, K.; Welleck, S.; Yazdanbakhsh, A.; and Clark, P. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *NeurIPS*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2.

Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control.

Muennighoff, N.; Liu, Q.; Zebaze, A. R.; Zheng, Q.; Hui, B.; Zhuo, T. Y.; Singh, S.; Tang, X.; von Werra, L.; and Longpre, S. 2024. OctoPack: Instruction Tuning Code Large Language Models. In *ICLR*. OpenReview.net.

Oswald, J. T.; Srinivas, K.; Kokel, H.; Lee, J.; Katz, M.; and Sohrabi, S. 2024. Large Language Models as Planning Domain Generators. In *ICAPS*, 423–431. AAAI Press.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Rintanen, J. 2015. Impact of Modeling Languages on the Theory and Practice in Planning Research. In *AAAI*, 4052–4056. AAAI Press.

Russell, S.; and Norvig, P. 1995. *Artificial Intelligence — A Modern Approach*. Prentice Hall.

Scala, E.; Haslum, P.; and Thiébaux, S. 2016. Heuristics for Numeric Planning via Subgoaling. In *IJCAI*, 3228–3234.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramírez, M. 2020. Subgoaling Techniques for Satisficing and Optimal Numeric Planning. *JAIR*, 68: 691–752.

Sel, B.; Al-Tawaha, A.; Khattar, V.; Jia, R.; and Jin, M. 2024. Algorithm of Thoughts: Enhancing Exploration of Ideas in Large Language Models. In *ICML*. OpenReview.net.

Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; and Yao, S. 2023. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*.

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J. B.; Kaelbling, L. P.; and Katz, M. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In *AAAI*, 20256–20264. AAAI Press.

Stojkovic, J.; Choukse, E.; Zhang, C.; Goiri, I.; and Torrellas, J. 2024. Towards Greener LLMs: Bringing Energy-Efficiency to the Forefront of LLM Inference. *arXiv preprint arXiv:2403.20306*.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fiser, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; Segovia-Aguas, J.; and Seipp, J. 2024. The 2023 International Planning Competition. *AI Mag.*, 45(2): 280–296.

Tantakoun, M.; Zhu, X.; and Muise, C. 2025. LLMs as Planning Modelers: A Survey for Leveraging Large Language Models to Construct Automated Planning Models. *CoRR*, abs/2503.18971.

Valmeekam, K.; Hernandez, A. O.; Sreedharan, S.; and Kambhampati, S. 2022. Large Language Models Still Can't Plan (A Benchmark for LLMs on Planning and Reasoning about Change). *CoRR*, abs/2206.10498.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35: 24824–24837.

Xu, B.; Peng, Z.; Lei, B.; Mukherjee, S.; Liu, Y.; and Xu, D. 2023. ReWOO: Decoupling Reasoning from Observations for Efficient Augmented Language Models. *CoRR*, abs/2305.18323.

Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; and Narasimhan, K. 2023a. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *NeurIPS*.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K. R.; and Cao, Y. 2023b. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*. OpenReview.net.

Ye, H.; Wang, J.; Cao, Z.; Berto, F.; Hua, C.; Kim, H.; Park, J.; and Song, G. 2024. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37: 43571–43608.

Zhang, K.; Li, Z.; Li, J.; Li, G.; and Jin, Z. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *ACL*, 769–787. Association for Computational Linguistics.

Zhong, L.; Wang, Z.; and Shang, J. 2024. Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step. In *ACL (Findings)*, 851–870. Association for Computational Linguistics.

Zhou, A.; Wang, K.; Lu, Z.; Shi, W.; Luo, S.; Qin, Z.; Lu, S.; Jia, A.; Song, L.; Zhan, M.; and Li, H. 2024. Solving Challenging Math Word Problems Using GPT-4 Code Interpreter with Code-based Self-Verification. In *ICLR*. OpenReview.net.