Hierarchical Task Network Planning with LLM-Generated Heuristics

Felipe Meneguzzi
University of Aberdeen, UK
PUCRS, Brazil

Alexandre Buchweitz
Independent Researcher
Toronto, Canada

Augusto B. Corrêa
University of Oxford, UK

Victor Scherer Putrich
Saarland University, Germany

André G. Pereira
Universidade Federal do Rio Grande do Sul, Brazil

Abstract

HTN planning is a variation of classical planning where, instead of searching for a linear sequence of actions, an algorithm decomposes higher-level tasks using a method library until only executable actions remain. On the one hand, this allows one to introduce domain knowledge that can speed up the search for a solution through the method library. On the other hand, it creates challenges that go beyond those of classical state-space search. While recent research produced a number of heuristics and novel algorithms that speed up HTN planning, these heuristics are not yet as informative as those available in classical planning algorithms. We investigate whether large language models (LLMs) can generate effective search heuristics for HTN planning, extending the methodology of Corrêa et al. 2025 from classical to hierarchical planning. Using the PYTRICH planner on seven standard total-order HTN benchmark domains, we evaluate heuristics generated by nine LLMs under domain-specific prompting and compare them against two established domain-independent heuristics in the same planner and against the PANDA planner. A configuration fixed in advance solves more instances than either domain-independent heuristic, the best generated heuristics expand fewer nodes than both on most instances, and the virtual best over all models is on par with PANDA in coverage.

1 Introduction

Hierarchical Task Network (HTN) planning [6, 7, 4] is a widely used formalism for structured, goal-directed tasks in robotics, game AI, and intelligent assistants. Unlike classical planning, which searches over state transitions, HTN planning decomposes abstract tasks into sequences of primitive actions through ***methods***, requiring search over a joint space of world states and task networks. The decomposition hierarchy encodes domain knowledge directly, playing the role that explicit heuristics serve in classical planning: it constrains which actions the algorithm tries and in what order, without requiring a domain-independent estimate of goal distance. This richer structure makes HTN planning substantially harder to scale without effective search guidance.

HTN planning is useful because many domains describe work as procedures rather than as a flat set of goal facts. A robot mission, for example, may choose a survey, delivery, or repair procedure before expanding it into navigation, sensing, and manipulation steps; game agents and assistants use similar routines. A classical encoding can represent the same primitive actions, but it often pushes procedural knowledge into extra predicates, control rules, or a large compiled model. HTN models keep this knowledge explicit in tasks and methods, giving the LLM structured domain information that it can turn into search guidance.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Domain-independent heuristics offer guidance without requiring hand-crafted, domain-specific knowledge; the established methods include the admissible Task Decomposition Graph (TDG) heuristic [3], the relaxed composition (RC) heuristics, which compile the HTN search state into a classical planning problem so that classical heuristics such as FF and LMCUT apply [12, 15], delete- and ordering-relaxation heuristics [14], and landmark-based counting (LMCO) [11, 22]. Despite these advances, heuristic search has only recently taken the lead in HTN planning. HYPERTENSION [18] won the first HTN track of the International Planning Competition (IPC) in 2020 with efficient recursive search and bookkeeping and no explicit heuristic function [1]. Heuristic progression planners were largely absent from that edition, so its result does not measure what a heuristic adds over good bookkeeping. By the 2023 edition, planners built on the PANDA heuristics won every total-order track [23], which makes PANDA the standard against which a new HTN heuristic must be measured. Whether HTN heuristics can become as informative as their classical counterparts remains open.

Recent work shows that large language models (LLMs) can generate effective heuristic functions for classical planning as code [24, 5]. The methodology prompts an LLM with domain context, evaluates candidate heuristics on training problems, selects the best performer, and deploys it on unseen test problems. The resulting heuristics outperform established domain-independent baselines and rival domain-specific methods, a striking result given the absence of any explicit heuristic engineering. Extending this methodology to HTN planning poses additional challenges. A heuristic for HTN search must reason not only over the world state but also over the current ***task network***: the set of pending tasks, their ordering constraints, and the applicable decomposition methods. An LLM must therefore grasp hierarchical decomposition semantics to produce useful guidance, a harder task than reasoning over state-only features. Thus, this paper adapts the LLM heuristic generation methodology of Corrêa et al. [5] to total-order HTN planning. We keep the generate-evaluate-select pipeline deliberately close to theirs, so that differences in outcome are attributable to the hierarchical setting rather than to changes in the pipeline, and we make the following contributions.

1. We implement the pipeline within the PYTRICH planner [22] with a heuristic interface that exposes the task network alongside the world state, and a prompt that teaches the LLM to use it (Section 3).
2. We evaluate heuristics generated by nine LLMs on seven benchmark domains against domain-independent heuristics within the same planner and against PANDA (Section 4). A configuration fixed in advance solves more problems than the domain-independent heuristics in PYTRICH, and the virtual best is on par with PANDA (Section 5).
3. We show that the generated heuristics expand fewer nodes than those heuristics on most shared instances, that their margin shrinks as the decomposition structure of a domain becomes more regular, that no single model dominates, and that greedy search exploits them best (Sections 5 and 6).

2 HTN Planning

An HTN planning problem is a tuple $\mathcal{P}_{\text{HTN}} = \langle s_0, \omega_I, F, A, C, \mathcal{M} \rangle$, where F is a set of ground fluents, A is a set of primitive actions with preconditions and effects over F, C is a set of compound tasks, $\mathcal{M}$ is a set of methods, ω_I is the initial task network, and $s_0 \subseteq F$ is the initial state [6, 7, 22]. A ***task network*** $\omega = \langle T, \prec, \alpha \rangle$ consists of a finite set of task identifiers T, a strict partial order $\prec$ over T encoding ordering constraints, and a labelling function $\alpha : T \to A \cup C$ that assigns each identifier to a primitive action or compound task. A ***method*** $m = \langle c, \omega_m \rangle \in \mathcal{M}$ decomposes a compound task $c \in C$ into a subnetwork ω_m; applying m replaces the occurrence of c in the current network with ω_m. A ***solution*** is a sequence of primitive actions obtained by repeatedly selecting and decomposing compound tasks until the network contains only primitive tasks, such that the resulting sequence is executable from s_0.

This paper focuses on ***total-order*** HTN planning, in which every task network imposes a strict linear order on its tasks. Total-order HTN is widely studied [4] and is the main setting used in the main IPC HTN planning track [1, 23], which supplies the domains and problems for our evaluation. Our evaluation uses problems encoded in HDDL [13], the standard input language for HTN planners. We use the PYTRICH planner [22] as our experimental platform because it exposes a clean Python

interface for plugging in custom heuristic functions, making it straightforward to deploy LLM-generated code.

The TDG heuristic [3] estimates the cost to solve the current task network by computing a relaxed reachability bound over the Task Decomposition Graph, a precomputed structure that encodes which primitive actions can arise from each compound task. It ignores delete effects and ordering constraints, yielding an admissible but often weak estimate.

Landmark-based counting (LMCO) [11] extracts ***landmarks***, facts or tasks that must occur on every solution path, from the task network structure. The heuristic value is the count of landmarks not yet achieved. Putrich et al. [22] extend this approach with bidirectional landmark generation, which finds additional landmarks at polynomial cost.

Both heuristics are domain-independent and exploit only the syntactic structure of the HDDL encoding, requiring no user-supplied domain knowledge. They serve as the primary algorithmic baselines in our evaluation. To follow the protocol of Corrêa et al. of measuring their heuristics against the strongest available planners, we compare against the progression-based PANDA planner [15], the state of the art in HTN planning. PANDA's relaxed composition (RC) heuristics compile the current search state into a classical planning problem and estimate its cost with a classical heuristic.

3 LLM-Generated Heuristics for HTN Planning

For each domain, an LLM writes a Python class with two methods: a preprocessing method that runs once before search, and an evaluation method that maps a search node, a state together with its remaining task network, to a non-negative estimate of the cost to complete it. This class is the heuristic, and the planner uses it as it would any hand-written one. We obtain these classes by adapting to HTN planning the pipeline of Corrêa et al. [5], who show that LLMs can serve as heuristic engineers for classical planning. Their methodology asks an LLM to write a Python function that maps a planning state to a numeric estimate of the distance to the goal. They generate a batch of candidate functions by prompting the model with the domain encoding and a description of the heuristic interface, then discard candidates that raise exceptions or time out. Next, they evaluate survivors on a set of training problems, and select the one that minimises node expansions for deployment on unseen test problems. The pipeline is model-agnostic, and the selection step retains the most effective candidate whichever LLM wrote it.

Adapting this methodology to HTN planning requires two changes. First, the heuristic must accept a task network in addition to the world state, since the remaining work comprises the pending tasks and their decomposition structure rather than a goal condition over fluents alone. Second, the LLM must reason about the hierarchical semantics of the domain, namely which compound tasks remain, how they may be decomposed, and whether the current ordering of tasks suggests a long or short path to completion. These demands are qualitatively harder than reasoning over flat state features, making HTN planning a more challenging test for LLM-generated heuristics.

3.1 Heuristic Function Interface

PYTRICH exposes a Python `Heuristic` base class with two required methods. The initialization method is called once before search and is used for preprocessing; the evaluation method is called at every expansion and must return a non-negative estimate of remaining solution cost. The split keeps per-node evaluation cheap, since expensive operations run once in initialization. The cost a heuristic estimates is the number of primitive actions the planner still needs to complete the task network, since PYTRICH adds one unit for each primitive action it applies and nothing for a decomposition. PYTRICH never prunes a node based on its heuristic value, infinity included, so a candidate that overestimates or returns infinity slows the search but cannot make it incomplete.

The interface also exposes both parts of an HTN search state. From `model`, the heuristic receives the grounded structure (facts, primitive operators, abstract tasks, decomposition methods, and goals). From `node`, it receives the current state and the remaining task network. In particular, states and operator conditions are represented as bitsets, which enables constant-time fact checks and efficient relaxed computations.

```
You are an expert in hierarchical planning
 and heuristic design. Your task is to
analyze the provided domain and create an
effective domain-dependent heuristic for
the **Depots** domain.
[...]
## 12. Your Task: Required Design
Procedure
Follow these steps and **write the
justification in top-of-file comments** of
 your generated heuristic:
1. **Name the bottleneck.** Read the
Depots HDDL and identify the single
constraint that most serializes work. One-
sentence comment.
2. **List 2-3 independent admissible lower
 bounds.** For each, write a one-line
admissibility sketch: why is this term a
lower bound on unavoidable cost?
3. **Implement `initialize(self, model)`.
** Parse fact names, build per-predicate
id maps, precompute goal structures [...]
Preprocessing here is free.
4. **Implement `__call__(self, node)` to
return `max(term1, term2, term3)`.** Each
term must use O(1) bit tests. No iteration
 over model-sized collections.
5. **Add tie-breaking penalties** [...]
Use sub-unit magnitudes (`* 0.001`) if
preserving admissibility matters [...]
```

```python
class DepotsAnthropicOpus46Heuristic9(Heuristic):
    [...]
    def _estimate_put_on_cost(self, crate, target, state):
        key = (crate, target)
        if key in self.on_fact_id and \
                (state >> self.on_fact_id[key]) & 1:
            return 0  # Already on target - nop
        cost = 0
        if not self._is_clear(target, state):
            blocking = self._count_stack_above(target, state)
            cost += blocking * 4  # Lift+Load+Unload+Drop
        target_place = self._get_surface_place(target)
        crate_place = self._get_crate_location(crate, state)
        if crate_place is not None:
            if not self._is_clear(crate, state):
                above = self._count_stack_above(crate, state)
                cost += above * 4
            cost += 2 if crate_place == target_place else 5
        elif self._is_crate_in_truck(crate, state):
            cost += 3  # Drive + Unload + Drop
        [...]
        return cost

    def _compute(self, state, task_network):
        [...]
        for task in task_network:
            [...]
            if task_name == "do_put_on":
                task_cost += self._estimate_put_on_cost(
                    args[0], args[1], state)
        [...]
        return max(goal_cost, task_cost,
                   unsatisfied_goals + clearing_cost)
```

Figure 1: Left, the preamble and the design procedure of the prompt, abridged, as sent for Depots. Right, the core of one heuristic Claude Opus 4.6 wrote from it, abridged. The cost of a pending `do_put_on` task depends on where the crate is and on the stack above its target. We denote elisions as `[...]`; Appendix E lists the heuristic at greater length beside a failed one.

The most important distinction from the classical planning interface of Corrêa et al. [5] is explicit access to the HTN-specific task network, that is, the pending abstract tasks and the decomposition methods applicable to each. A heuristic that ignores the task network degenerates to a classical-planning heuristic and loses all HTN-specific guidance; exploiting this structure is the central challenge the LLM must address to produce useful estimates. We include a detailed explanation of the API in Appendix G for reproducibility.

3.2 Prompt Design

We use a domain-specific prompt regime in which the LLM receives the domain name, the full domain HDDL file, the smallest and largest benchmark instances of the domain, and a worked example of a heuristic for a different domain. Following Corrêa et al. [5], the example teaches the interface and not the domain, and Appendix C shows that it improves code validity without changing search performance.

The prompt directs the LLM through four steps: identify the planning bottleneck by determining which compound tasks decompose into many primitives; list two or three independent lower bounds on the remaining cost and combine them by taking their maximum, or their sum when they count disjoint work; implement the result using the two-method interface, keeping the cost per-node low by moving all loops to the preprocessing step; and add tie-breaking penalties for symmetric states, keeping them below one unit when the estimate must remain a lower bound. Figure 1 reproduces this part of the prompt beside the core of a heuristic Claude Opus 4.6 wrote from it for Depots, which estimates the cost of each pending task from the state rather than from its cheapest method.

Several technical details require explicit treatment in the prompt because LLMs otherwise default to incorrect conventions when working with PYTRICH. The format of grounded fact names differs from HDDL predicate syntax, and the prompt illustrates this difference with wrong-versus-right examples. The prompt also specifies how to query goal facts via the bitwise state encoding, provides correct

import paths for the heuristic base class and model types, and gives full method signatures with type annotations. The prompt asks the LLM to return its response in a structured, parseable format so that candidates can be saved without additional text processing.

The prompt can also carry a per-domain hint block with three parts: representation caveats introduced by grounding; the dominant search bottleneck of the domain; and guidance on lower bounds, symmetry-breaking terms and penalty scales. We wrote the blocks from the records of our development sessions, in which the model summarised the failure modes it met (from which we kept the summary as hint text), so they play the part of the handwritten checklist of Corrêa et al. [5]. We treat the hints as an ablation condition rather than as part of the pipeline. The results of Section 5 use the prompt without hints, and Appendix C shows their effect with Gemini 3 Flash. In summary, they leave coverage and code validity unchanged and reduce expanded nodes on about three quarters of the shared instances, mostly in Barman and Rover. Appendix B reproduces the blocks.

We also implement an iterative refinement variant that augments the base prompt with the previous candidate code, its empirical results against the TDG baseline (expanded nodes, wall time, solution length, and exit status), and auto-generated guidance keyed to the observed failure mode. A candidate that times out receives advice to move all computation to the preprocessing step; a candidate that performs worse than TDG receives advice to add state-awareness or to aggregate decomposition costs across the full task network. Appendix A has the full prompt text and the refinement template.

3.3 Heuristic Selection

For each domain and each LLM model, we generate a batch of candidate heuristics by independent one-shot invocations of the domain-specific prompt, with no feedback between them. The batch holds twenty candidates for Claude Opus 4.6, six for Gemini 3 Flash on the six IPC 2020 domains, and between four and forty for the other models. Appendix D gives the counts for every model and domain. Each candidate runs on the smallest benchmark problem for the domain under a 60-second wall-clock timeout. We discard candidates that fail to parse, raise runtime exceptions, exceed memory limits, or time out. Among the survivors, we select the candidate that minimises expanded nodes on the training problem, breaking ties in favour of the shorter solution. Selection on one instance filters out candidates that crash or misuse the interface, but it does not rank the survivors, several of which tie on the selection instance in Robot and Towers.

We then evaluate the selected candidate without modification on all benchmark problems for that domain, including the selection instance itself. The selection criterion is model-agnostic, since it requires only that the heuristic run without error and produce a result on the single training instance, and it does not reward interpretability. This design means that a weaker model contributing just one viable heuristic can still improve the virtual best over all models, and the selection step acts as an automatic quality filter without requiring the experimenter to implement heuristic code.

We leave a systematic evaluation of the iterative refinement variant to future work; all quantitative results in Section 5 use heuristics generated by the base one-shot prompt.

4 Experimental Setup

We briefly describe the main points of our experimental setup next.[1] Appendix D contains more details, such as specific LLM versions, parameters, and resources used.

Domains. We evaluate on seven total-order HTN domains encoded in HDDL [13]. Six of them, Barman, Blocksworld, Depots, Robot, Rover and Towers, come from the IPC 2020 HTN track [1] and remain in the 2023 track [23]. The seventh, Lamps, is one of the two total-order domains that the 2023 track added. We evaluate every benchmark instance of each domain, 180 problems over the seven domains, and Table 1 gives the number per domain. The Gemini API no longer serves Gemini 2.0 Flash and Gemini 3 Pro, so Lamps has candidates from Claude Opus 4.6, Gemini 2.5 Pro and Gemini 3 Flash only.

Models. We evaluate nine LLMs spanning three providers: Claude Opus 4.6, Claude Sonnet 4.5, Gemini 2.0 Flash, Gemini 2.5 Pro, Gemini 3 Flash, Gemini 3 Pro, GPT-4o, GPT-5, and GPT-5.2.

[1] Code, generated heuristics, result files and scripts: `https://doi.org/10.5281/zenodo.23931424`

For each domain-model pair, we generate heuristic candidates using provider-default sampling and evaluate them under the protocol of Section 3.3. A list of models, generation settings, and model-specific failure accounting are reported in Appendix D; summary failure analysis appears in Section 5.3. We compare the LLM heuristics against two domain-independent heuristics in PYTRICH and four configurations of PANDA.

Baseline Planners and Heuristics. The TDG heuristic [3], as implemented in PYTRICH, precomputes for every compound task the cheapest decomposition into primitive actions over the task decomposition graph, weighting each primitive action by its cost and each method application by one unit, and estimates a node by the sum of these costs over the tasks remaining in its task network. LMCO, also in PYTRICH, counts the landmarks not yet achieved and runs with the bidirectional landmark generation of Putrich et al. [22]. PANDA RC^{FF}, RC^{LMC} and RC^{Add} [15] are the relaxed composition (RC) heuristics of PANDA, which compile the current search state into a classical planning problem and estimate it with the FF [10], LMCut [9] and additive heuristics in turn. LMCO^{P} is PANDA's own landmark-count heuristic with bidirectional landmarks. Both planners use one unit of path cost per primitive action and nothing per decomposition. The LLM heuristics are Python classes written against the PYTRICH interface, and PANDA offers no hook for them, so the comparison across planners rests on coverage under the same limits. We apply the same virtual-best treatment to the PANDA configurations as to the LLM heuristics.

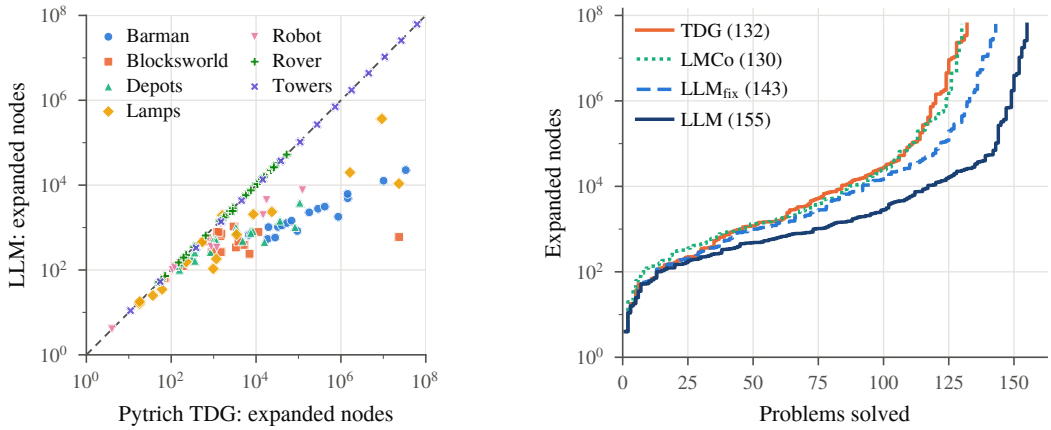
Search Algorithms. We evaluate each LLM-generated heuristic and each PYTRICH baseline under three search algorithms: Standard A* ($f = g + h$), Greedy Best-First Search (GBFS, $f = h$ only, favouring nodes with the lowest heuristic value), and Weighted A* with weight $w = 5$ (WA*, $f = g + 5 \cdot h$), which trades plan optimality for faster search. The weight is the same for every system, LLM heuristics and baselines alike, and we do not tune it. For each system we report the *virtual best* over these three algorithms: a problem counts as solved if any algorithm finds a solution, and we record the node count from the algorithm that solves it with the fewest expansions. This reflects a realistic deployment scenario in which one can choose the search algorithm freely. Results disaggregated by algorithm appear in Section 5.

Evaluation Metrics and Resources. We report three metrics. *Coverage* is the number of benchmark problems we solve within the time and memory limits; it is the primary metric, reported per domain and in aggregate. *Node expansions* is the number of search nodes expanded before finding a solution; lower is better, as it measures heuristic guidance quality independently of hardware speed. We report the median over solved problems and head-to-head win counts on the common set of problems solved by both compared systems. *Plan length* is the number of primitive actions in the returned solution; we report head-to-head comparisons on the common solved set. We run our experiments with 8 GB RAM and a 30-minute runtime limit per problem. All runs use 1 CPU core. We run PANDA experiments under identical resource constraints using a single build of `pandaPIengine`; additional execution details are provided in Appendix D.

5 Results

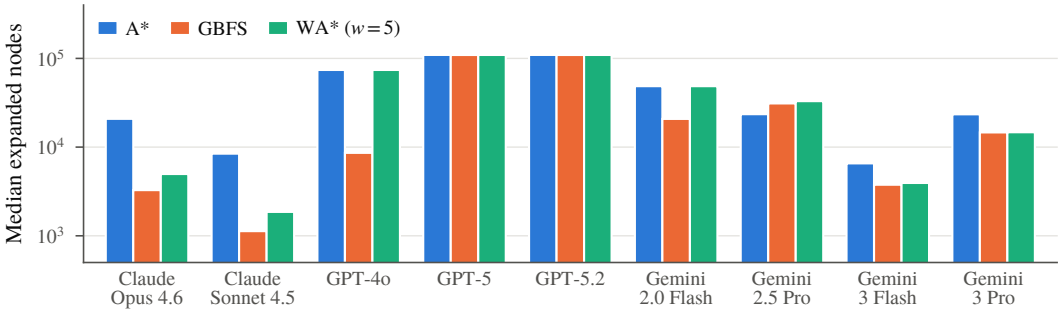
5.1 Coverage

Table 1 reports per-domain and aggregate coverage for all systems over the seven domains. Every column but one is a virtual best over the three search algorithms (GBFS, A*, WA*), so a problem counts as solved if any algorithm solves it. The PANDA block shows the three relaxed-composition engines, RC^{FF}, RC^{LMC} and RC^{Add}, the last with the additive estimate, together with LMCO^{P}, PANDA's own landmark-count heuristic. The PYTRICH block shows TDG, LMCO, which is bidirectional landmark counting [22], the best single model, Claude Opus 4.6 (Opus 4.6), and two columns for the LLM heuristics as a whole. LLM is the virtual best over the five main models and all their candidate heuristics. This is an upper bound that no deployed system can reach, since it chooses the model after seeing the result. LLM_{fix} is its deployable counterpart with one model and one search algorithm per domain, chosen on the smallest instance before running any other instance, taking the best of that model's candidates per instance. We include the smallest instance of each domain (which we use to select candidates and the configuration of LLM_{fix}) in the results of Table 1. Thus, while we do



(a) Expanded nodes for LLM vs. PYTRICH TDG.

(b) Problems solved within a given node-expansion budget in PYTRICH.



(c) Median expanded nodes (log scale) by search algorithm for each LLM model.

Figure 2: Search efficiency and algorithm effects: per-problem expanded nodes of LLM against PYTRICH TDG (a), cumulative coverage of TDG, LMCO, LLM_{fix} and LLM against node-expansion budget (b), and median expanded nodes by search algorithm for each LLM model (c). In (a), points below the diagonal indicate an LLM advantage.

not hold out the selection instance, this favours no column, because every system solves all seven of these instances, and they add one to every cell of their rows alike.

LLM_{fix} solves 143 of the 180 problems, thirteen more than LMCO and eleven more than TDG, and LLM raises the count to 155. Both PYTRICH baselines lose their coverage on the largest instances of each domain. Outside Robot, the LLM heuristics solve all of these instances but one. In aggregate LLM and PANDA RC^{FF} are at parity, one problem apart, and they differ in where they solve. RC^{FF} solves Robot completely, while every PYTRICH configuration, baseline or LLM, stops at the eleven smallest instances within the time limit. On the other six domains LLM matches or exceeds RC^{FF}, and LLM_{fix} matches or exceeds it on Barman, Rover and Towers. The three largest Rover instances account for the aggregate margin, since no PANDA configuration completes its preprocessing on them within the time limit, whereas TDG and the LLM heuristics solve all three. Lamps, which the prompts never saw and for which we wrote no hints, follows the same pattern, with the LLM heuristics at parity with the PANDA engines and ahead of both PYTRICH baselines, and the gain concentrated in a single model, Gemini 3 Flash. The problems that separate LLM_{fix} from LLM are the price of committing to a model on a single small instance, which filters broken candidates but cannot rank the sound ones; we return to this limitation in Section 6.

5.2 Search Efficiency and Plan Quality

We compare search effort within PYTRICH, where the planner, the grounding and the search code are the same and only the heuristic changes, pairing each instance that both sides solve and taking the virtual best over the three algorithms on both sides. Against TDG, LLM expands fewer nodes on

Table 1: Coverage (problems solved) per domain and in aggregate; every column is the virtual best over GBFS, A* and WA* except LLM_{fix}, which fixes one model and one search algorithm per domain in advance, and bold marks the best coverage in each row.

		PANDA				PYTRICH				
Domain	Probs.	RC^{FF}	RC^{LMC}	RC^{Add}	LMCO^{P}	TDG	LMCO	Opus 4.6	LLM_{fix}	LLM
Barman	20	18	15	16	18	18	19	**20**	18	**20**
Blocksworld	30	29	26	**30**	**30**	20	**30**	28	28	29
Depots	30	27	23	22	29	23	27	**30**	26	**30**
Robot	20	**20**	**20**	**20**	**20**	11	11	11	11	11
Rover	30	26	19	27	20	**30**	16	**30**	**30**	**30**
Towers	20	14	13	14	14	**15**	**15**	**15**	**15**	**15**
Lamps	30	19	18	19	15	15	12	14	15	**20**
Total	180	153	134	148	146	132	130	148	143	**155**

107 of the 132 shared instances (81%), with a median node ratio of 0.74, and never expands more except on one Lamps instance. Against LMCO it wins 126 of 129 (98%) at a median ratio of 0.46. Figure 2a shows the comparison against TDG, where nearly every point lies below the diagonal. The margin over TDG depends on the domain. It is largest in Barman, where LLM wins 17 of 18 instances at a median ratio of 0.02, followed by Lamps (13 of 15 at 0.24), Depots (21 of 23 at 0.41) and Blocksworld (20 of 20 at 0.45). In Robot (10 of 11 at 0.77) and Towers (14 of 15 at 0.96) the margin shrinks, and in Rover the two sides tie on 18 of the 30 instances and LLM wins the other 12 by a handful of nodes. The ordering follows the regularity of the decomposition hierarchy. In Towers and Rover the task decomposition graph already yields a tight estimate and leaves little room for semantic guidance, whereas in Barman the ordering constraints of the methods hide most of the remaining work from TDG. Against LMCO the advantage is uniform, with every domain won outright except Blocksworld (27 of 29). Table 14 in Appendix F.5 gives the structure of each domain beside its margin. The node advantage belongs to the portfolio rather than to any configuration fixed in advance. LLM_{fix} is at parity with TDG (54 of 132 wins, median ratio 1.00) and ahead of LMCO (78 of 126 at 0.93). Figure 2b shows where its extra coverage lies. Within ten thousand expanded nodes LLM_{fix} solves 91 problems, against 83 for TDG, 86 for LMCO and 116 for LLM. Its curve stays close to those of the baselines on the smaller instances and separates on the larger ones, which it solves and they do not.

Node counts across planners are not directly comparable, since PANDA and PYTRICH ground differently and count different search nodes, so the node comparison against PANDA RC^{FF} appears in Appendix F.1, for completeness only.

Plan length, counted as primitive actions on both sides, separates the systems less than search effort does. On the 143 instances shared with PANDA RC^{FF}, LLM plans are shorter on 43, equal on 97 and longer on 3, with a median ratio of 1.00, and within PYTRICH the plans of LLM, TDG and LMCO coincide on every shared instance outside Barman and Rover, where LLM is shorter (Appendix F.3).

5.3 LLM Model Comparison

The main comparison covers five models: Claude Opus 4.6, Gemini 2.0 Flash, Gemini 2.5 Pro, Gemini 3 Flash and Gemini 3 Pro. The remaining four models (GPT-4o, Claude Sonnet 4.5, GPT-5 and GPT-5.2) retained an executable heuristic in one domain each, so we exclude them from the per-domain tables. No single model dominates. Claude Opus 4.6 and Gemini 3 Flash each solve 148 of the 180 instances, Gemini 2.0 Flash 112, Gemini 3 Pro 100 and Gemini 2.5 Pro 82, and the virtual best is a portfolio, in which Claude Opus 4.6 carries the Blocksworld and Depots instances, Gemini 3 Pro most of Barman, and Gemini 3 Flash Rover, Towers and Lamps. Over every solved run, the median expanded nodes range from 4,686 for Gemini 3 Flash and 6,599 for Claude Opus 4.6 to 14,908 for Gemini 3 Pro, 24,387 for Gemini 2.5 Pro and 45,294 for Gemini 2.0 Flash, a tenfold spread, wider than the effect of the search algorithm in Section 5.4. The generate-evaluate-select procedure compensates in part for weak models, since Gemini 2.0 Flash and Gemini 2.5 Pro contribute full

coverage on the domains where their selected heuristics run without error. On Robot, no LLM heuristic solves the nine largest instances within that limit, and neither does any PYTRICH baseline.

Across the nine models and seven domains, we generated 928 candidate heuristics, between four and forty per model and domain. Of these, 563 passed the import check that admits a candidate to the evaluation, and 482 solved at least one benchmark instance (Appendix D). Of the 365 candidates the import check rejected, 286 declare a constructor with a required argument, 201 of them a `model` argument that the interface passes to `initialize` instead, 73 fail to import, and 6 give the evaluation method the wrong signature. The 81 candidates that imported but solved no instance fail on every run, most often by treating the bitwise state as a set, although the prompt specifies the representation; Appendix D.4 gives the exceptions and the exit reasons of the baselines' lost instances.

Four of the nine models have results in a single domain. GPT-4o and Claude Sonnet 4.5 were run on Depots only, while GPT-5 and GPT-5.2 were run on all six IPC 2020 domains and passed the import check only on Towers. We did not modify the prompt template to retry these failures, since doing so after observing test results would select against the very benchmark instances used for evaluation. We therefore report a negative finding. For GPT-5 and GPT-5.2, domain-specific prompting at the default sampling temperature produced no executable heuristic on five of the six domains.

5.4 Effect of Search Algorithm

Across all LLM runs, GBFS achieves the lowest median expanded nodes (8,761), followed by WA* (11,934) and A* (21,602), so GBFS is about $2.5\times$ more efficient than A* and $1.4\times$ more efficient than WA* (Figure 2c). Paired per instance, GBFS expands fewer nodes than A* for every model, on 81 of 90 shared instances for Claude Opus 4.6, 110 of 134 for Gemini 3 Flash and 75 of 75 for Gemini 2.0 Flash. The algorithm effect is largest for the strongest heuristics. For Claude Opus 4.6 the median under A* is six times that under GBFS, against twice for Gemini 3 Flash and no gain for Gemini 2.5 Pro.

6 Discussion

The pattern of results differs from the classical-planning case of Corrêa et al. [5], where the LLM virtual best matches or exceeds state-of-the-art baselines across the board. Here, the LLM virtual best reaches parity with PANDA RC^{FF} in coverage, and its advantage shows within PYTRICH, where it expands fewer nodes than TDG on 81% of shared instances and fewer than LMCO on 98%. The two systems differ in the domains that they solve best. RC^{FF} solves every Robot instance, whereas every PYTRICH configuration, baseline or LLM, stops at the eleven smallest within the time limit, so the Robot gap seems to be a limit of the planner rather than of the heuristics. On the other six domains the LLM virtual best matches or exceeds RC^{FF}. Plan length separates the systems less than search effort does. In Towers every run of every system returns a plan of the same length, and in Blocksworld, Depots and Robot the best heuristic of each system returns plans of the same length on almost every shared instance, although weaker candidates return longer ones. Only in Barman and Rover do the best heuristics disagree, and there the LLM heuristics return the shortest plans, ahead of both TDG and RC^{FF}.

The HDDL domain encoding provided in the prompt gives the LLM explicit access to the decomposition structure of the domain: method names, task hierarchies, and operator preconditions and effects. This semantic richness allows the LLM to reason about which compound tasks are costly and which are cheap, producing estimates that are more informative than the purely syntactic relaxation that TDG computes. The gains over TDG are largest where the cost of a task depends on the state, since TDG estimates each pending task by its cheapest method and, in Barman, Lamps, Blocksworld and Depots, that method is the one that applies when the task is already achieved. In Rover the methods fix the cost of each goal and the LLM virtual best only matches TDG, and in Towers at most one method applies in any state, so no heuristic can matter. Appendix F.5 goes through the methods of each domain, with Robot as the exception where neither heuristic guides the search. Appendix E lists a strong and a failed heuristic for Depots.

The tenfold spread in median expanded nodes across the five main models is the most striking model-level finding. Newer models are not uniformly better, since Gemini 2.5 Pro solves fewer instances than Gemini 2.0 Flash and expands four times the nodes of Claude Opus 4.6 and Gemini

3 Flash. The generate-evaluate-select procedure partially compensates for weak models, and even Gemini 2.0 Flash and Gemini 2.5 Pro contribute to full coverage on the domains where their selected heuristics run without error. The efficiency gain from switching to GBFS grows with heuristic quality, as the median for Claude Opus 4.6 under A* is six times that under GBFS, while for Gemini 3 Flash it is twice. This interaction suggests that discarding g-values is most beneficial when the heuristic itself encodes decomposition cost, since in that regime tracking path cost adds noise rather than signal.

7 Related Work

HTN planning encodes domain knowledge in the decomposition hierarchy itself, complementing rather than replacing search heuristics [6, 7]. A line of work develops increasingly informative HTN heuristics, from the TDG heuristic [3] and the relaxed composition (RC) heuristics [15] to landmark counting [11, 22]; the progression-based PANDA planner [15] is the strongest system built on them.

LLMs perform poorly as planners [25], and reliable use requires coupling them with sound search [24, 5]. In the HTN setting, Xu and Muñoz-Avila [26] use LLMs to propose decompositions online, Muñoz-Avila et al. [20] interleave LLM-generated and symbolic decomposition, and Oswald et al. [21] use them to author PDDL domains. We use LLMs only to write the heuristic and delegate correctness to the search algorithm.

Tuisov et al. [24] argue that domain-independence may no longer be the right yardstick once LLMs can synthesise per-domain heuristics on demand. Building on this, Corrêa et al. [5] prompt LLMs to write Python heuristics for classical planning, outperforming domain-independent baselines. We adapt their pipeline to HTN planning, where the heuristic must reason over task networks and decomposition methods rather than flat states.

8 Conclusion

We investigate whether LLM-generated heuristics can serve as effective search guidance for HTN planning, extending the methodology of Corrêa et al. [5] from classical to hierarchical planning. The answer is affirmative. A configuration fixed in advance on one instance per domain solves 143 of 180 benchmark instances, against 132 for TDG and 130 for LMCO in the same planner, and the virtual best over all models and algorithms reaches parity with PANDA RC^{FF}. Within PYTRICH, the LLM heuristics expand fewer nodes than TDG on 81% of the shared instances and LMCO on 98%.

The task hierarchy itself guides search, and HYPERTENSION won the 2020 IPC HTN track with no explicit heuristic function [18]. Prior work has focused on acquiring the methods themselves, from classical operators [17], noisy observations [8], or curriculum learning [16], reducing the manual effort of encoding the hierarchy. We take the method library as given and generate a heuristic that exploits it. Tuisov et al. [24] ask whether domain-independence remains a meaningful goal in the era of LLMs, and our results sharpen this question in the HTN setting, where the task hierarchy already encodes domain knowledge and the LLM heuristic adds a second layer of domain-specific guidance that exploits that encoding directly.

Our results come with four limitations. The evaluation covers seven total-order HTN domains and leaves partial-order planning, numerical fluents and temporal domains open. The virtual best chooses the model after seeing the result and is an upper bound; the configuration fixed in advance is the deployable one, and its single selection instance filters out broken candidates without ranking the sound ones. Generation is stochastic, and every prompt includes the smallest and largest instance of its domain, so Appendix D.5 recomputes every comparison without those instances, which confirms our results. The LLM heuristics are Python programs written against the PYTRICH interface and cannot run inside PANDA, so we compare across planners primarily on coverage.

Extending the approach to partial-order HTN planning is the most important open direction, as the heuristic must then reason about the ordering space of the task network rather than a fixed sequence. A domain-independent prompting condition, without the HDDL file, would determine how much of the observed gain derives from the domain knowledge supplied in the prompt and how much from the LLM's general reasoning about planning structure. A prompt that asks for an estimate of the method applications still required, rather than of the missing primitive actions alone, would let the heuristic reason about the hierarchy itself.

Acknowledgments and Disclosure of Funding

We thank the anonymous reviewers for their comments, which led to the fixed configuration, the prompt-context recomputation and the cost analysis in this version. After acceptance we repeated the full generation protocol for Claude Opus 4.6 with a larger API budget than was available at submission time, and every Claude Opus 4.6 result in this version comes from that batch. We thank Anthropic for the Claude team plan for science, whose monthly API credit paid for that batch. The experiments ran on the Maxwell high-performance computing cluster of the University of Aberdeen. The authors declare no competing interests. Victor Scherer Putrich acknowledges funding from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - GRK 2853/1 "Neuroexplicit Models of Language, Vision, and Action" - project number 471607914. André G. Pereira acknowledges support from FAPERGS with projects 21/2551-0000741-9 and 25/2551-0002590-7. This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil* (CAPES) – Finance Code 001.

References

[1] G. Behnke, D. Höller, and P. Bercher, editors. *Proceedings of the 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, 2021. URL `https://ipc2020.hierarchical-task.net/publications/IPC2020Booklet.pdf`.

[2] P. Bercher, S. Keen, and S. Biundo. Hybrid planning heuristics based on task decomposition graphs. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS)*, pages 35–43. AAAI Press, 2014. doi: 10.1609/socs.v5i1.18323.

[3] P. Bercher, G. Behnke, D. Höller, and S. Biundo. An admissible HTN planning heuristic. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 480–488. IJCAI Organization, 2017. doi: 10.24963/ijcai.2017/68.

[4] P. Bercher, R. Alford, and D. Höller. A survey on hierarchical planning – one abstract idea, many concrete realizations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 6267–6275. IJCAI Organization, 2019. doi: 10.24963/ijcai.2019/875.

[5] A. B. Corrêa, A. G. Pereira, and J. Seipp. Classical planning with LLM-generated heuristics: Challenging the state of the art with Python code. In *Advances in Neural Information Processing Systems 38 (NeurIPS)*. Curran Associates, Inc., 2025. doi: 10.52202/085713-1401.

[6] K. Erol, J. Hendler, and D. S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI)*, pages 1123–1128. AAAI Press / MIT Press, 1994. URL `https://cdn.aaai.org/AAAI/1994/AAAI94-173.pdf`.

[7] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004. doi: 10.1016/B978-1-55860-856-6.X5000-5.

[8] M. Grand, H. Fiorino, and D. Pellier. An accurate HDDL domain learning algorithm from partial and noisy observations. In *Proceedings of the ICAPS 2022 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2022. URL `https://arxiv.org/abs/2206.06882`.

[9] M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 162–169. AAAI Press, 2009. doi: 10.1609/icaps.v19i1.13370.

[10] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001. doi: 10.1613/jair.855.

[11] D. Höller and P. Bercher. Landmark generation in HTN planning. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI)*, pages 11826–11834. AAAI Press, 2021. doi: 10.1609/aaai.v35i13.17405.

[12] D. Höller, P. Bercher, G. Behnke, and S. Biundo. On guiding search in HTN planning with classical planning heuristics. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 6171–6175. IJCAI Organization, 2019. doi: 10.24963/ijcai.2019/857.

[13] D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, and R. Alford. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*, pages 9883–9891. AAAI Press, 2020. doi: 10.1609/aaai.v34i06.6542.

[14] D. Höller, P. Bercher, and G. Behnke. Delete- and ordering-relaxation heuristics for HTN planning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4076–4083. IJCAI Organization, 2020. doi: 10.24963/ijcai.2020/564.

[15] D. Höller, P. Bercher, G. Behnke, and S. Biundo. HTN planning as heuristic progression search. *Journal of Artificial Intelligence Research*, 67:835–880, 2020. doi: 10.1613/jair.1.11282.

[16] R. Li, D. Nau, M. Roberts, and M. Fine-Morris. Automatically learning HTN methods from landmarks. In *Proceedings of the Thirty-Seventh International Florida Artificial Intelligence Research Society Conference (FLAIRS)*. University of Florida George A. Smathers Libraries, 2024. doi: 10.32473/flairs.37.1.135625.

[17] M. C. Magnaguagno and F. Meneguzzi. Method composition through operator pattern identification. In *Proceedings of the ICAPS 2017 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2017. URL `https://www.meneguzzi.eu/felipe/pubs/keps-methods-composition-2017.pdf`.

[18] M. C. Magnaguagno, F. Meneguzzi, and L. de Silva. HyperTensioN and total-order forward decomposition optimizations. *Autonomous Agents and Multi-Agent Systems*, 39(1):Article 24, 2025. doi: 10.1007/s10458-025-09705-9.

[19] F. Meneguzzi, A. Buchweitz, A. B. Corrêa, V. S. Putrich, and A. G. Pereira. Pytrich-LLM: Code, generated heuristics, benchmarks and results of "Hierarchical Task Network Planning with LLM-Generated Heuristics", 2026. URL `https://doi.org/10.5281/zenodo.23960400`.

[20] H. Muñoz-Avila, D. W. Aha, and P. Rizzo. ChatHTN: Interleaving approximate (LLM) and symbolic HTN planning. In *Proceedings of the International Conference on Neuro-symbolic Systems (NeuS)*, volume 288 of *Proceedings of Machine Learning Research*, pages 446–458. PMLR, 2025. URL `https://proceedings.mlr.press/v288/munoz-avila25a.html`.

[21] J. Oswald, K. Srinivas, H. Kokel, J. Lee, M. Katz, and S. Sohrabi. Large language models as planning domain generators (student abstract). In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI)*, pages 23604–23605. AAAI Press, 2024. doi: 10.1609/aaai.v38i21.30491.

[22] V. S. Putrich, F. Meneguzzi, and A. G. Pereira. Landmark generation in HTN planning revisited. In *Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 228–235. AAAI Press, 2025. doi: 10.1609/icaps.v35i1.36123.

[23] A. Taitler, R. Alford, J. Espasa, G. Behnke, D. Fišer, M. Gimelfarb, F. Pommerening, S. Sanner, E. Scala, D. Schreiber, J. Segovia-Aguas, and J. Seipp. The 2023 International Planning Competition. *AI Magazine*, 45(2):280–296, 2024. doi: 10.1002/aaai.12169.

[24] A. Tuisov, Y. Vernik, and A. Shleyfman. LLM-generated heuristics for AI planning: Do we even need domain-independence anymore? arXiv:2501.18784v1 [cs.AI], 2025. URL `https://arxiv.org/abs/2501.18784v1`.

[25] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati. On the planning abilities of large language models – a critical investigation. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, pages 75993–76005. Curran Associates, Inc., 2023. doi: 10.52202/075280-3320.

[26] Y. Xu and H. Muñoz-Avila. Online learning of HTN methods for integrated LLM-HTN planning. In *Proceedings of the Twelfth Annual Conference on Advances in Cognitive Systems (ACS)*, 2025. URL https://arxiv.org/abs/2511.12901.

[27] M. Yousefi, M. Schmautz, P. Haslum, and P. Bercher. How good is perfect? On the incompleteness of A* for total-order HTN planning. In *Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 112–120. AAAI Press, 2025. doi: 10.1609/icaps.v35i1.36107.

A Prompt Templates

A.1 Base Prompt Structure

The base prompt is a structured document of twelve numbered sections, preceded by a task preamble, delivered to the LLM for each domain. Sections 1, 2 and 2.5 supply domain-specific information; sections 3–12 are fixed across all domains.

1. **Preamble.** States the role (*expert in hierarchical planning and heuristic design*), the target domain, and the required class name and parameter name for the generated Python class.
2. **Section 1 — Target domain definition.** The full HDDL domain file, presented verbatim in a fenced code block.
3. **Section 2 — Training set instances.** Two benchmark problems: the smallest (used for heuristic selection) and the largest available, both presented verbatim in HDDL format.
4. **Section 2.5 — Domain-specific hints.** An ablation condition rather than a pipeline input: the reported results use the prompt with this section empty, and Appendix C measures the effect of filling it with the block of the domain (Appendix B).

Sections 3–12 are identical across all domains and prompts.

Section 3 — Grounded fact format. Explains that after grounding, facts are represented as `+predicate[arg1,arg2]` rather than in HDDL syntax, and provides a reference implementation of a parser:

Listing 1: Reference fact-name parser included verbatim in the prompt.

```
def _parse_fact_name(self, name: str):
    name = name.strip()
    if name.startswith("+") or name.startswith("-"):
        name = name[1:]
        bracket_idx = name.find("[")
        if bracket_idx > 0 and name.endswith("]"):
            predicate = name[:bracket_idx]
            args_str = name[bracket_idx+1:-1]
            args = args_str.split(",") if args_str else []
            return predicate, args
    return None, []
```

Section 4 — Goal state access. Explains that `model.goals` is a bitwise integer, not a list of predicates, and shows the correct API call to retrieve goal fact names:

```
goal_fact_names = self.model.state_explicit_repr(self.model.goals)
# Returns: ['+communicated_soil_data[waypoint1]', ...]
```

Section 5 — Bitwise state checks. Contrasts the correct O(1) method against the incorrect string-based alternative:

```
# CORRECT: O(1) bitwise check
is_true = (state_bitwise >> fact.global_id) & 1

# WRONG: O(n) string scan that causes timeouts on large instances
fact.name in self.model.state_explicit_repr(state)
```

Section 6 — Required imports and method signatures. States the mandatory import paths and provides correct and incorrect variants of the two required method signatures:

```
from Pytrich.Heuristics.heuristic import Heuristic
from Pytrich.Search.htn_node import HTNNode
from Pytrich.model import Model

def initialize(self, model: Model, initial_node: HTNNode):
    h_value = self._compute(initial_node.state, initial_node.task_network)
    return super().initialize(model, h_value)
```

```
def __call__(self, parent_node: HTNNode, node: HTNNode) -> int:
    h_value = self._compute(node.state, node.task_network)
    super().update_info(h_value)
    return h_value
```

Section 7 — Working example heuristic. A complete, working heuristic for a different domain, provided verbatim: the Towers heuristic for the other domains and a Snake heuristic, a domain outside the benchmark, for Towers. It teaches the interface and not the domain, following Corrêa et al. [5], and Appendix C shows that it improves code validity without changing search performance. It shows how to parse grounded fact names and how to separate preprocessing in `initialize`, which builds lookup tables from the facts of the model, from per-node evaluation in `__call__`, which tests state bits and never iterates over the facts or operators of the model.

Sections 8 and 9 — Class definitions. Full Python class definitions for `Fact`, `Operator`, `AbstractTask`, `Decomposition`, and `Model` are given verbatim so the LLM knows every attribute and method available at runtime.

Section 10 — Admissibility vs. guidance. Tells the model to add its tie-breaking penalty to the maximum of the lower-bound terms, and that a penalty below one unit (e.g. $\times 0.001$) keeps the result a lower bound whenever the terms are, while still breaking symmetries.

Section 11 — Winning and losing patterns. Six winning patterns (P1–P6) and four anti-patterns (A1–A4) are listed, each with a short code example. The patterns record the failure modes we met while developing the prompt, and the runtime failures in Appendix D.4 are instances of anti-patterns A1 and A3.

Section 12 — Required design procedure. The LLM is instructed to follow five steps and write the justification as top-of-file comments in the generated code: (1) name the domain bottleneck; (2) list two to three independent admissible lower bounds; (3) implement `initialize`; (4) implement `__call__` returning `max(term1, term2, ...)`; (5) add tie-breaking penalties if the domain has interchangeable parameters. The LLM is asked to return a structured response (filename and code) to enable automated parsing.

A.2 Iterative Refinement Prompt

The refinement prompt appends a feedback section to the base prompt. The section includes: the previous candidate code verbatim; a results table comparing the candidate against the TDG baseline on the selection instance (expanded nodes, wall time, solution size, exit status); and auto-generated corrective guidance keyed to the observed failure mode.

Guidance is generated by inspecting the previous code for structural signatures:

- **TIMEOUT**: If the code iterates `self.model.facts` or `self.model.operators` inside `__call__`, the feedback explicitly names this as anti-pattern A3 and instructs the LLM to move that work to `initialize`.
- **Worse than TDG**: If the code never reads state bits (no `>>` or `& 1` patterns), the feedback flags anti-pattern A1 and recommends adding state-aware per-task cost estimation. If no `max(` composition is present, the feedback recommends adding a second admissible lower bound.
- **Better than TDG**: The feedback acknowledges the improvement and suggests tightening an existing term or adding a third `max` component.
- **Runtime error**: The error message is shown verbatim and the LLM is instructed to fix it.

The instruction *"Do NOT start from scratch — modify and improve the previous version"* is highlighted to prevent the LLM from discarding working structure when only a single component needs improvement.

B Details on Hints

The hint block is an ablation condition, not an input of the pipeline. Every result in the main text comes from the prompt without it, and Appendix C measures its effect with Gemini 3 Flash, where it leaves coverage and code validity unchanged and reduces expanded nodes on about three quarters of the shared instances, mostly in Barman and Rover. We wrote one block for each of the six IPC 2020 domains from the records of our development sessions, in which the model summarised the failure modes it had met and we kept the summary as hint text, so the blocks play the part of the hand-written checklist of Corrêa et al. [5]. Lamps has no block. Each block has three kinds of entries: representation caveats introduced by grounding, such as static facts that the grounder strips and method preconditions that it compiles into operators; the dominant search bottleneck of the domain; and guidance on lower bounds, symmetry-breaking terms and penalty scales. The prompt introduces the block with the sentence "These insights were discovered through extensive experimentation on this domain. Use them." The blocks follow verbatim.

Barman.

```
- BOTTLENECK: Parameter-binding symmetry. AchieveContainsShakerIngredient chooses which shot to use - all
N shots are interchangeable, creating N-way branching per ingredient transfer (~N^(2K) for K cocktails)
- GROUNDING WARNING: In grounded operator names, the SHOT argument is at args[0], NOT args[1] as HDDL
parameter order suggests. Always verify by examining grounded fact/operator names
- Goal-shot penalty: Penalize using goal shots (shots that appear in goal facts) as intermediaries.
Penalty must be VERY large (~500/task) - small penalties (15-30) have zero effect because they are washed
out by total h
- Working-shot ordering: Assign increasing penalties (0, 100, 200...) to break symmetry among identical
working shots. STEP=100 >> STEP=50
- Hand preference: Penalize non-preferred hand (50/task) to halve branching
- Combined symmetry-breaking reduced pfile10 from 2.3M nodes (timeout) to 25K nodes (0.2s)
```

Blocksworld.

```
- BOTTLENECK: Stack clearing - to move a buried block, all blocks above must be unstacked first
- Key cost formulas: do_put_on(x,y) non-trivial = clear_cost(x) + clear_cost(y) + 3; clear_cost with stack
 height h = 1 + 2*h
- CRITICAL: Goal-stability / false-trivial detection. on(x,y) currently true but y is "unstable" (will be
moved for another goal) means this goal is NOT free - it costs 5+. This single insight reduced p10 from
2.37M nodes to 5,813
- Nop actions and real (primitive) actions are completely disjoint in this domain, so you can SAFELY SUM
admissible bounds on nop count and real action count (they never overlap)
- Multi-component max of 3 terms (nop+displacement, per-task recursive cost using stack heights, first-
task exact cost) works best
```

Depots.

```
- GROUNDING WARNING: PANDA's grounder strips static facts like (place pallet0 depot0). To recover location
 info, parse grounded operator names: Drop[hoist0,crate1,pallet0,depot0] reveals pallet0 is at depot0
- GROUNDING WARNING: Method preconditions are compiled into __method_precondition_* operators, NOT stored
in decomposition.pos_precons. Check for operators whose name starts with "__method_precondition" when
analyzing decompositions
- Nop detection is non-trivial: "effectively nop" decompositions (all operators are effect-free) != single
 nop operator. Must check for effect-free operator chains
- Balanced additive combination max(goal_h, tn_h) + min(goal_h, tn_h)//2 works but can lose coverage on
large instances; max() alone is safer
- Nop bonus: when a task's nop preconditions are NOT met (meaning the expensive path is required), add +2
to its cost
```

Robot.

```
- BOTTLENECK: Single-carry constraint - robot holds exactly one package at a time. Each delivery is a
round-trip (go to package, pickup, go to goal, putdown)
- Inter-delivery travel bound: with P undelivered packages, the robot must make at least P pickup+putdown
pairs (2P actions) PLUS navigate between them. Sum of min outgoing edges in the delivery graph minus max
edge gives a Hamiltonian path lower bound
- Doors are one-way-open (once opened, stays open forever). Pre-compute all-pairs shortest paths ignoring
door state, then add +1 per closed door on the direct connection as a correction
- Star topology: room "c" (corridor) connects to many rooms - pre-compute BFS distances from corridor hub
- For large instances (50+ packages), keep __call__ to O(P) not O(P^2) - Python overhead matters at 40K+
nodes/second
```

Rover.

```
- GROUNDING WARNING: PANDA's grounder STRIPS static facts (can_traverse, visible, at_lander, store_of,
equipped_for_*). You CANNOT read rover capabilities from the state. Instead, extract static info from
grounded operator names: navigate[rover0,wp1,wp2] tells you rover0 can traverse wp1→wp2
- TDG's weakness: it assumes minimum decomposition cost (rover already at target = nop). Your heuristic
should compute ACTUAL navigation distance when rover is NOT at target: cost = 3*d + 1 per hop (visit +
navigate + unvisit + action)
- For 2+ packages: use Hamiltonian path lower bound on inter-delivery travel (sum of min outgoing edges -
max edge)
- Pre-compute rover navigation graph and all-pairs distances in initialize(). Do NOT pre-select "best"
waypoints - that depends on runtime rover position
- Keep __call__ to O(goals) not O(goals * rovers). O(1) single-rover check per goal beats O(goals*rovers)
iteration (14-24x slower, causes timeout)
```

Towers.

```
- Exact optimal cost for 3-peg Hanoi from any state is computable: 2^k - 1 for largest displaced ring k
- Process rings largest-to-smallest to avoid O(n^2) tower-tracing; build a peg-assignment array in O(n)
during initialize()
- Memoize by state (bitwise int) - many decomposition paths reach the same state
- The heuristic example provided for this domain already demonstrates the optimal approach
```

C Prompt Ablation

This appendix reports a prompt ablation. It asks whether the search performance of the generated heuristics comes from the model or from the two domain-specific inputs the prompt supplies: the per-domain hint block (Section 2.5 of the prompt) and the worked example heuristic (Section 7). All three arms use Gemini 3 Flash, twenty candidates per domain, one-shot, with provider thinking set to medium and temperature 1.0, on the six evaluation domains. Every candidate that runs is evaluated under GBFS, A*, and WA* on all 150 instances of the six domains. The arms differ in exactly one prompt input each relative to the first:

- **Arm 1 (no example).** No domain hints: Section 2.5 is empty. No domain-specific example: Section 7 is replaced by a domain-agnostic stub that returns zero for every node, a syntactic template for the required signatures with no reasoning to imitate.
- **Arm 2 (hints).** Arm 1 plus the full hint block of the domain.
- **Arm 3 (fixed cross-domain example).** Arm 1 plus a working heuristic from a different domain as the worked example: the Towers heuristic for the other five domains, and a Snake heuristic for Towers.

The domain HDDL (Section 1 of the prompt) is present in all three arms and is not ablated. Arms 2 and 3 are each compared with arm 1 only, since they differ from each other in two inputs at once. The comparison is paired per instance: on the instances both arms solve, we count the share where the best candidate of the arm expands fewer nodes than the best candidate of arm 1, and the median of the per-instance node ratio. Pairing controls for instance difficulty and is less sensitive to unequal candidate pools than a median over runs. The result file is `Experiments/results/ablation.csv` in the released repository [19], and the script `Experiments/python_scripts/generate_ablation_table.py` produces the tables from it.

Coverage. Table 2 shows coverage per arm, as the virtual best over the three algorithms, beside the LLM column of Table 1. Coverage saturates: most domains sit at the ceiling for every arm, so coverage cannot discriminate between them, and the signal lies in node counts. The candidate pools are unequal because some generated heuristics fail to run; the pool sizes appear in parentheses. The ablation candidates solve one Robot instance more than any candidate of the main study.

Search effort. Table 3 reports the paired win rate and the median node ratio of arms 2 and 3 against arm 1 under each algorithm. The two arms answer different questions. The hints produce a search effect that does not depend on the algorithm, since arm 2 wins about three quarters of the shared instances under GBFS and WA* at a median node ratio near 0.8, and 59% under A*. The effect is concentrated in two domains, Barman-BDI and Rover-GTOHP, where the median ratio is

Table 2: Coverage per ablation arm, virtual best over GBFS, A*, and WA*, on the 150 instances of the six domains. Parentheses give the number of candidates, out of twenty, that run at all. LLM is the paper's virtual best over the five main models, for reference.

Domain	Probs.	LLM	Arm 1	Arm 2	Arm 3
Barman-BDI	20	20	20 (15)	20 (20)	20 (20)
Blocksworld-GTOHP	30	29	29 (16)	30 (15)	30 (17)
Depots	30	30	28 (18)	27 (16)	27 (16)
Robot	20	11	12 (17)	12 (18)	12 (20)
Rover-GTOHP	30	30	30 (16)	30 (11)	30 (14)
Towers	20	15	15 (17)	15 (18)	15 (20)
Total	150	135	134 (99)	134 (98)	134 (107)

about one third under GBFS, while Blocksworld-GTOHP, Depots, and Robot sit at parity. The fixed cross-domain example leaves search performance unchanged, with arm 3 winning about half of the shared instances under GBFS and WA* at a median node ratio of 1.00 under both. What the example buys is code validity, reported next.

Table 3: Paired comparison of arms 2 and 3 against arm 1 on expanded nodes, per algorithm. Win rate is the share of instances solved by both arms where the arm's best candidate expands fewer nodes than arm 1's; the denominator is the number of shared instances. Ratio is the median per-instance node ratio, arm over arm 1, so values below one favour the arm.

	GBFS		A*		WA*	
Domain	Arm 2	Arm 3	Arm 2	Arm 3	Arm 2	Arm 3
Win rate against arm 1						
Barman-BDI	20/20	13/20	10/10	10/10	19/19	19/19
Blocksworld-GTOHP	26/29	7/29	12/15	11/16	22/25	10/26
Depots	12/27	25/27	13/20	14/20	13/27	25/27
Robot	1/12	0/12	5/12	5/12	1/12	2/12
Rover-GTOHP	30/30	22/30	5/9	3/9	30/30	3/29
Towers	13/15	0/15	3/15	0/15	13/15	0/15
Total	102/133 (77%)	67/133 (50%)	48/81 (59%)	43/82 (52%)	98/128 (77%)	59/128 (46%)
Median node ratio against arm 1						
Barman-BDI	0.32	0.99	0.00	0.94	0.04	0.71
Blocksworld-GTOHP	0.97	1.04	0.89	0.90	0.94	1.04
Depots	1.02	0.82	0.87	0.99	1.02	0.84
Robot	1.00	1.00	1.00	1.00	1.00	1.00
Rover-GTOHP	0.30	0.92	0.98	1.22	0.29	1.25
Towers	0.65	1.00	1.00	1.00	0.70	1.00
Total	0.78	1.00	0.96	0.99	0.81	1.00

Code validity. Table 4 separates candidates that fail to compile from candidates that compile and then fail at run time, measured by running each of the 360 candidates on one small instance of its domain. The worked example cuts runtime failures by more than half, from 17.5% and 19.2% to 7.5%, while compile failures rise from one in arm 1 and none in arm 2 to five in arm 3. Of the 53 runtime failures across the three arms, 28 are attribute errors, all in arms 1 and 2, and 16 of these read the model before the base-class initialiser assigns it, which is the mistake the worked example shows how to avoid. The example teaches the PYTRICH API rather than better Python. The hints do not help validity: arm 2 is marginally worse than arm 1 on both counts, so whatever the hints buy is search guidance.

What the ablation cannot show. Three limits apply. First, A* is incomplete for total-order HTN planning, so the A* columns rest on 82 shared instances rather than 133 and are not comparable to the GBFS figures. Second, twenty candidates per domain can only show the absence of a detectable effect, and we do not measure the variance between independent draws of twenty candidates from the same arm. This applies to arm 3's ratio of 1.00 as much as to the domains where arm 2 is at parity with arm 1. Third, usable on one small instance is a weak bar: it establishes that the code executes,

Table 4: Code validity per arm over the 120 candidates each arm generates, six domains by twenty. A candidate is usable if it imports and completes a run on one small instance of its domain.

Arm	Generated	Usable	Fails to compile	Fails at run time	Failure rate
Arm 1 (no example)	120	98	1 (0.8%)	21 (17.5%)	18.3%
Arm 2 (hints)	120	97	0 (0.0%)	23 (19.2%)	19.2%
Arm 3 (fixed cross-dom. ex.)	120	106	5 (4.2%)	9 (7.5%)	11.7%

not that the heuristic is informative, and a candidate that crashes only on large instances counts as usable here.

D Experimental Details

D.1 Models Used

Table 5 lists all evaluated models.

Table 5: LLM models evaluated. †Run on Depots only. ‡Passed the import check on Towers only. §No Depots candidate loaded (Table 6).

Key in results	Model name	Provider
ClaudeOpus46	Claude Opus 4.6	Anthropic
ClaudeSonnet45†	Claude Sonnet 4.5	Anthropic
Gemini20Flash	Gemini 2.0 Flash	Google
Gemini25Pro	Gemini 2.5 Pro	Google
Gemini3Flash	Gemini 3 Flash	Google
Gemini3Pro§	Gemini 3 Pro	Google
ChatGPT4o†	GPT-4o	OpenAI
GPT5‡	GPT-5	OpenAI
GPT52‡	GPT-5.2	OpenAI

D.2 Generation settings

For candidate generation, we use provider-default sampling settings and cap responses at 16,384 tokens. Each model-domain pair generates its candidates by independent one-shot invocations of the prompt, evaluated and selected as described in Section 3.3. For Claude Opus 4.6 no thinking budget is sent and the temperature is the provider default, which the model does not let us set. Table 6 gives, for every model and domain, the number of candidates generated, the number that pass the import check of the task-list generator (a candidate is admitted if its file imports, defines a `Heuristic` subclass with the two required methods, and has a constructor without required arguments), and the number that solve at least one benchmark instance.

D.3 Execution details

All jobs are executed through SLURM job arrays with one benchmark instance per task. Each task is allocated 1 CPU core, 8 GB RAM, and a 30-minute wall-clock limit. Both PYTRICH and PANDA runs use the same resource limits. All PANDA runs use one build of `pandaPIengine`, with seed 42 for its tie-breaker and path cost counted per action; the released repository builds the parser, grounder and engine from pinned commits. Every run executes on one core of an Intel Xeon Gold 6138 processor in a shared SLURM partition.

D.4 Exit reasons

We read the exit of every unsolved run from its log: the memory limit of 8 GB, the time limit of 30 minutes, or a Python exception. Every unsolved run falls into one of the three. Table 7 gives the exits of the two PYTRICH baselines on the instances they fail, all three search algorithms counted.

Table 6: Candidate funnel per model and domain, as the number generated, the number that passed the import check, and the number that solved at least one instance. A dash marks a pair for which no candidates were generated; † marks the one importable Gemini 3 Pro Depots candidate, whose registry entry was disabled, so it never loaded.

Model	Barman	Blocksworld	Depots	Robot	Rover	Towers	Lamps	Total
Claude Opus 4.6	20/20/20	20/17/13	20/19/19	20/19/14	20/20/20	20/20/20	20/20/20	140/135/126
Gemini 2.0 Flash	29/29/27	39/34/12	40/37/35	39/39/37	40/39/15	39/37/26	–	226/215/152
Gemini 2.5 Pro	20/11/11	20/14/13	10/7/6	20/13/13	6/6/5	29/20/19	20/20/20	125/91/87
Gemini 3 Flash	6/6/6	6/6/6	6/6/6	6/6/6	6/6/6	6/6/6	20/19/19	56/55/55
Gemini 3 Pro	25/5/5	22/2/2	20/1/0†	22/2/2	21/1/1	21/18/18	–	131/29/28
GPT-4o	–	–	4/4/4	–	–	–	–	4/4/4
Claude Sonnet 4.5	–	–	6/6/6	–	–	–	–	6/6/6
GPT-5	20/0/0	20/0/0	20/0/0	20/0/0	20/0/0	20/20/20	–	120/20/20
GPT-5.2	20/0/0	20/0/0	20/0/0	20/0/0	20/0/0	20/8/4	–	120/8/4

TDG fails 48 instances, 123 of its 144 runs on them at the memory limit and 21 at the time limit, and LMCo fails 50, 140 of 150 runs at the memory limit and 10 at the time limit; neither raises an exception. The exceptions are TDG on Barman and most of Towers, and LMCo on Rover p17, which time out. So the baselines lose almost every instance they miss to memory, with all three runs at the limit on 39 of the 48 TDG misses and 46 of the 50 LMCo misses. On the same 65 instances, the five main models' candidates exhaust memory on most of their unsolved runs as well, with time the most common exit for Claude Opus 4.6 alone; their coverage gain comes from the instances they solve, chiefly in Rover, Blocksworld and Depots, and no run of any model solves Robot pfile_10_020 and larger or Towers 16 to 20.

Table 7: Exits of the PYTRICH baselines on the instances they fail, over the three search algorithms, as runs that end at the memory limit (M), at the time limit (T), and the number of instances on which all three runs end at the memory limit (all M).

	TDG				LMCo			
Domain	Unsolved	M	T	all M	Unsolved	M	T	all M
Barman	2	0	6	0	1	3	0	1
Blocksworld	10	28	2	9	0	0	0	0
Depots	7	20	1	6	3	9	0	3
Robot	9	27	0	9	9	27	0	9
Rover	0	0	0	0	14	39	3	13
Towers	5	3	12	0	5	8	7	2
Lamps	15	45	0	15	18	54	0	18
Total	48	123	21	39	50	140	10	46

Of the 563 candidates that pass the import check, 81 solve no instance. Five of them never loaded: four GPT-5.2 Towers candidates and one Gemini 3 Pro Depots candidate have their registry entries disabled, so every run fails on the lookup before the file is read. The other 76 raise the same exception on every one of their runs, and none times out. In 43 the candidate treats the state or the goal, a bitwise integer, as a set or a list, most often by iterating over `model.goals`. In 13 it reads an attribute the search node does not have, such as `uncompleted_tasks` or `method`. Six fail on an import path or an undefined name, and 14 on the candidate's own logic, such as splitting a fact name on a space it does not contain. By model, Gemini 2.0 Flash accounts for 63 of the 76, Claude Opus 4.6 for 9 and Gemini 2.5 Pro for 4. No run in the record used a file that fails today's import check, so the two failure stages do not overlap.

D.5 Prompt-context instances

Every prompt contains the smallest and the largest instance of its domain, and both are in the evaluated set. Table 8 recomputes coverage with those twelve instances removed, 168 remaining. No comparison changes: LLM and RC^{FF} stay at parity, LLM_{fix} stays ahead of both PYTRICH baselines, and the ordering of every column is the same as in Table 1.

Table 8: Coverage with the twelve prompt-context instances removed (168 instances); columns as in Table 1.

Domain	Probs.	RC^{FF}	RC^{LMC}	$LMCO^{P}$	TDG	LMCO	LLM_{fix}	LLM
Barman	18	17	14	17	17	18	17	18
Blocksworld	28	28	25	28	19	28	27	27
Depots	28	26	22	28	22	26	25	28
Robot	18	18	18	18	10	10	10	10
Rover	28	25	18	19	28	15	28	28
Towers	19	13	12	13	14	14	14	14
Lamps	29	18	17	14	14	11	14	19
Total	168	145	126	137	124	122	135	144

D.6 Heuristic cost

Table 9 separates the once-per-problem preprocessing cost of each heuristic from its per-node evaluation cost, measured on the GBFS runs of TDG, LMCO and the selected candidate of each model, all on one core of the same processor; Lamps was not timed. The prompt asks that each term of the per-node evaluation consist of constant-time bit tests and that every loop over the facts or operators of the model move to preprocessing, and the generated heuristics do keep preprocessing cheaper than the baselines in every domain, since the baselines build decomposition graphs and landmark structures while the generated code builds small lookup tables. The generated code does not always comply with this request. Per-node evaluation is comparable to the baselines, at tens of microseconds, but Table 10 shows that the cost of the selected Claude Opus 4.6 heuristic on Blocksworld grows with the instance, from 37 to 261 microseconds per node, while LMCO stays near 20.

Table 9: Median preprocessing cost (init, milliseconds) and per-node evaluation cost (microseconds) per domain under GBFS, LLM heuristics pooled over the five main models.

	LLM heuristics		TDG		LMCO	
Domain	init (ms)	per node (μs)	init (ms)	per node (μs)	init (ms)	per node (μs)
Barman	8.2	28	93	17	157	15
Blocksworld	2.7	37	27	19	19	15
Depots	1.6	26	43	17	74	13
Robot	0.1	11	1.1	11	1.0	11
Rover	4.7	49	2,607	53	6,254	19
Towers	3.6	13	253	18	1,071	12

Table 10: Per-node evaluation cost on Blocksworld instances of increasing size under GBFS.

Instance	p01	p05	p10	p14	p17	p19
LMCO per node (μs)	19	12	17	19	20	21
Claude Opus 4.6 per node (μs)	37	43	196	169	254	261

D.7 Model-specific failures

GPT-4o and Claude Sonnet 4.5 were run on Depots only. GPT-5 and GPT-5.2 were run on the six IPC 2020 domains and passed the import check only on Towers; on the other five domains every candidate declares a constructor that takes the model as a required argument, which the interface passes to `initialize` instead, or fails to import. Gemini 3 Pro passed the import check with one to five candidates on four domains and eighteen on Towers; its one importable Depots candidate never loaded (Table 6). These outcomes are included in the aggregate accounting and discussed in Section 5.3.

E Generated Heuristic Examples

This appendix lists two heuristics from the dataset of record for Depots, the domain in which Claude Opus 4.6 carries the virtual best. The first is the candidate excerpted in Figure 1, selected from the twenty Claude Opus 4.6 candidates on the smallest instance; it solves all 30 Depots instances and expands the fewest nodes of any LLM run on 10 of them. The second is a Gemini 2.5 Pro candidate and shows the failure mode that dominates the weaker models on this domain.

E.1 Strong Heuristic: Claude Opus 4.6 on Depots

The heuristic prices every pending task from the current state. A pending `do_put_on` task costs nothing when its crate is already on the target surface, which is the case in which TDG also returns the cost of the empty `nop` method, and otherwise costs the actions needed to clear the target, clear the crate, and move it, with the move priced by whether the crate is at the same place, at another place, in a truck or on a hoist. The final value is the maximum of three terms: the summed cost of the unsatisfied goals, the summed cost of the pending tasks, and the number of unsatisfied goals plus a clearing term for goal surfaces that the task network does not yet account for. All fact-id maps are built once in `initialize`, so each evaluation only tests bits. The listing abridges `initialize` and omits the helper predicates (`_is_clear`, `_count_stack_above`, `_get_crate_location`, `_is_crate_in_truck`, `_is_crate_being_lifted`, `_get_surface_place`, `_is_on_goal`), each of which is one or two bit tests over the maps built in `initialize`.

Listing 2: Selected heuristic for Depots (Claude Opus 4.6, candidate 9), abridged.

```
"""Domain-Dependent Heuristic for Depots HDDL Domain

Bottleneck Analysis:
    The primary bottleneck in Depots is the need to clear crates stacked on top
    of target surfaces before placing goal crates, combined with the need to
    transport crates between different places using trucks (which requires
    lift/load/drive/unload/drop sequences). [...]

Admissible Lower Bound Terms:
    1. Goal-displacement cost: For each unsatisfied goal (on ?c ?s), we need at
       minimum 1 action (Drop) if the crate is already lifted at the right place,
       2 actions (Lift+Drop) if at the same place on wrong surface, or ~5 actions
       (Lift+Load+Drive+Unload+Drop) if at a different place. [...]
    2. Clearing cost: For each goal surface that needs to be cleared, we need
       at least 2 actions per blocking crate (Lift+Load minimum). [...]
    3. Task network lower bound: Count remaining compound tasks weighted by
       minimum decomposition depth. [...]
"""

class DepotsAnthropicOpus46Heuristic9(Heuristic):

    def initialize(self, model: Model, initial_node: HTNNode):
        """Preprocess: build fact maps, goal structures, place assignments."""
        self.model = model
        self.on_fact_id = {}      # (crate, surface) -> global_id
        self.at_fact_id = {}      # (locatable, place) -> global_id
        self.in_fact_id = {}      # (crate, truck) -> global_id
        self.clear_fact_id = {}   # surface -> global_id
        self.lifting_fact_id = {} # (hoist, crate) -> global_id
        for f in model.facts:
            pred, args = self._parse_fact_name(f.name)
            if pred == "on" and len(args) == 2:
                self.on_fact_id[(args[0], args[1])] = f.global_id
            [...]
        self.goal_on = {}  # crate -> target_surface
        for gname in model.state_explicit_repr(model.goals):
            pred, args = self._parse_fact_name(gname)
            if pred == "on" and len(args) == 2:
                self.goal_on[args[0]] = args[1]
        [...]
        self.task_info = {}  # abstract_task.global_id -> (task_name, args)
        for at in model.abstract_tasks:
            [...]
        h_value = self._compute(initial_node.state, initial_node.task_network)
        return super().initialize(model, h_value)

    def _estimate_put_on_cost(self, crate, target_surface, state):
        """Estimate minimum cost for do_put_on(crate, target_surface)."""
        key = (crate, target_surface)
```

```
        if key in self.on_fact_id and (state >> self.on_fact_id[key]) & 1:
            return 0  # Already on target - nop
        cost = 0
        if not self._is_clear(target_surface, state):
            blocking = self._count_stack_above(target_surface, state)
            cost += blocking * 4  # Lift + Load + Unload + Drop per crate (conservative)
        target_place = self._get_surface_place(target_surface)
        crate_place = self._get_crate_location(crate, state)
        if crate_place is not None:
            if not self._is_clear(crate, state):
                blocking_on_crate = self._count_stack_above(crate, state)
                cost += blocking_on_crate * 4
            if crate_place == target_place:
                cost += 2   # Same place: Lift + Drop
            else:
                cost += 5   # Different place: Lift + Load + Drive + Unload + Drop
        elif self._is_crate_in_truck(crate, state):
            cost += 3       # Drive + Unload + Drop
        elif self._is_crate_being_lifted(crate, state):
            cost += 1       # Drop
        else:
            cost += 5       # Unknown state - conservative estimate
        return cost

    def _compute(self, state, task_network):
        """Compute heuristic value for a state and task network."""
        if len(task_network) == 0:
            if self.model.goals & state == self.model.goals:
                return 0
            return float('inf')
        # Term 1: Goal displacement
        goal_cost = 0
        unsatisfied_goals = 0
        for crate, target in self.goal_on.items():
            if not self._is_on_goal(crate, state):
                unsatisfied_goals += 1
                [...]   # 2 at the same place, 5 elsewhere, 3 in a truck, 1 on a hoist
        # Term 2: Task network cost estimation
        task_cost = 0
        for task in task_network:
            gid = task.global_id
            if gid >= self.model.iabt_init and gid <= self.model.iabt_end:
                task_name, args = self.task_info[gid]
                if task_name == "do_put_on" and len(args) == 2:
                    task_cost += self._estimate_put_on_cost(args[0], args[1], state)
                elif task_name == "do_clear" and len(args) == 2:
                    if not self._is_clear(args[0], state):
                        blocking = self._count_stack_above(args[0], state)
                        task_cost += max(blocking * 4, 2)
                elif task_name == "do_get_truck" and len(args) == 2:
                    at_key = (args[0], args[1])
                    if at_key in self.at_fact_id:
                        if not ((state >> self.at_fact_id[at_key]) & 1):
                            task_cost += 1  # Drive
                elif task_name == "do_lift_crate":
                    task_cost += 1
                elif task_name in ("do_load_truck", "do_unload_truck"):
                    task_cost += 2
                else:
                    task_cost += 1
            elif gid >= self.model.iop_init and gid <= self.model.iop_end:
                op = self.model.operators[gid - self.model.iop_init]
                if "nop" not in op.name:
                    task_cost += 1
        # Term 3: Clearing cost for goal surfaces not yet in the task network
        clearing_cost = 0
        for crate, target in self.goal_on.items():
            if not self._is_on_goal(crate, state) and not self._is_clear(target, state):
                top_crate = self._get_crate_on_surface(target, state)
                if top_crate is not None and top_crate != crate:
                    if self.goal_on.get(top_crate) != target:
                        clearing_cost += 2  # At least Lift+Load to move it
        return max(goal_cost, task_cost, unsatisfied_goals + clearing_cost)

    def __call__(self, parent_node: HTNNode, node: HTNNode) -> int:
        h_value = self._compute(node.state, node.task_network)
        super().update_info(h_value)
        return h_value
```

E.2 Weak Heuristic: Gemini 2.5 Pro on Depots

The heuristic precomputes a cost for every method by a fixed point over the action names in its subtasks, which is a reasonable plan, and then evaluates a node by looking up the method that produced it. It reads that method from an attribute `node.method` that the PYTRICH search node does not define, so the guard on the first line of `_calculate_node_heuristic` fires on every node and the heuristic returns zero throughout the search, which is then blind. The record shows the consequence. Forty-four Depots candidates, all 35 from Gemini 2.0 Flash, all 6 from Gemini 2.5 Pro and 3 of the 4 from ChatGPT-4o, expand exactly the same number of nodes as one another on every instance, and solve the same 17. Most of them read the same missing attribute; the rest never read the state at all. This is the failure mode behind anti-pattern A1 of the prompt (Appendix A), and the selection step cannot catch it, since a blind search solves the selection instance.

Listing 3: Weak heuristic for Depots (Gemini 2.5 Pro, candidate 3), abridged.

```
class DepotGemini25ProHeuristic3(Heuristic):
    def __init__(self, depot_gemini25pro_heuristic_3=None, drive_penalty=3,
                 manipulation_penalty=1, clear_penalty=0.5):
        [...]
        self.method_costs = {}
        self.task_costs = {}

    def initialize(self, model: Model, initial_node: HTNNode):
        self.model = model
        self._precompute_costs()
        h_value = 0
        return super().initialize(model, h_value)

    def _precompute_costs(self):
        """Pre-computes the heuristic cost for each decomposition method and
        abstract task using a fixed-point iteration approach."""
        for task in self.model.abstract_tasks:
            self.task_costs[task.name] = float('inf')
        changed = True
        while changed and iterations < max_iterations:
            changed = False
            for method in self.model.decompositions:
                current_method_cost = 0
                for subtask in method.task_network:
                    if isinstance(subtask, Operator):
                        action_name = subtask.name.lower().split(' ')[0]
                        if 'drive' in action_name:
                            current_method_cost += self.drive_penalty
                        elif any(op in action_name
                                 for op in ['lift', 'drop', 'load', 'unload']):
                            current_method_cost += self.manipulation_penalty
                    elif isinstance(subtask, AbstractTask):
                        [...]
                self.method_costs[method.name] = current_method_cost
            [...]   # task cost = min over its methods

    def _calculate_node_heuristic(self, node: HTNNode) -> float:
        """The value is based on the pre-computed cost of the method used for
        decomposition."""
        if not hasattr(node, 'method') or not node.method:
            return 0.0
        return float(self.method_costs.get(node.method.name, 1.0))

    def __call__(self, parent_node: HTNNode, node: HTNNode) -> float:
        h_value = self._calculate_node_heuristic(node)
        super().update_info(h_value)
        return h_value
```

The two listings differ on the dimension that Section 6 identifies. The first prices each pending task from the state and so sees when a task is already achieved or nearly so, while the second computes a static cost per method and then fails to attach it to the search node, so its estimate carries no information at all.

F Additional Results and Figures

This appendix contains the complete set of comparison figures for the four baselines. The main paper includes the within-PYTRICH scatter plots and the algorithm-comparison chart; the cross-planner

figures against PANDA RC^{FF} appear here. All figures use the LLM virtual best over the five main models and three search algorithms.

F.1 PANDA RC^{FF}

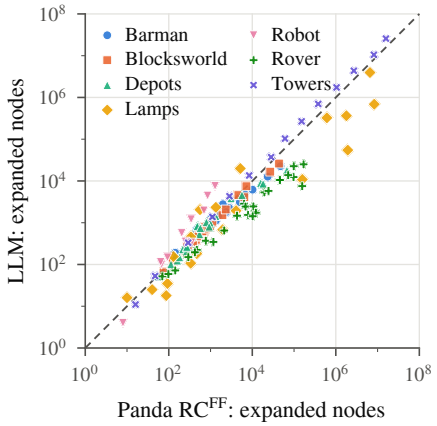
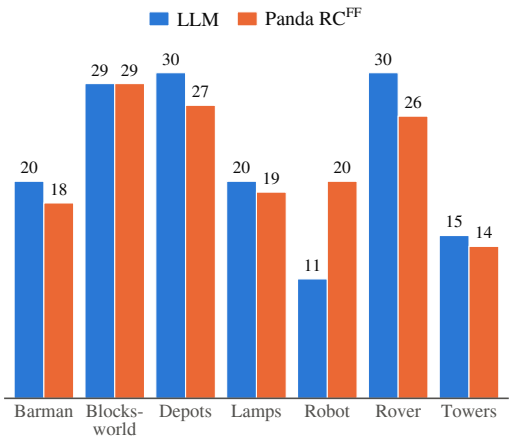


Figure 3: Cross-planner search effort: per-problem expanded nodes of LLM against PANDA RC^{FF}; points below the diagonal indicate an LLM advantage. Node counts are planner-specific, since PANDA and PYTRICH differ in per-node bookkeeping, and should be read with care across systems. On the 143 shared instances LLM expands fewer nodes on 92 (64%), median ratio 0.88, carried by Rover, Blocksworld and Lamps; Robot and Towers are losses. Table 11 gives the margins per domain.

Table 11: Per-domain margins on expanded nodes, LLM against PANDA RC^{FF}, on the instances both solve: how many instances each side expands fewer nodes on, the median per-instance node ratio, and the mean reduction in nodes on the instances LLM wins.

Domain	Shared	LLM fewer	RC^{FF} fewer	Median ratio	Mean reduction on wins (%)
Barman	18	10	8	0.94	27
Blocksworld	28	26	2	0.89	15
Depots	27	14	13	0.98	24
Robot	11	2	9	1.80	27
Rover	26	26	0	0.26	69
Towers	14	1	13	1.56	31
Lamps	19	13	6	0.47	67
All	143	92	51	0.88	41

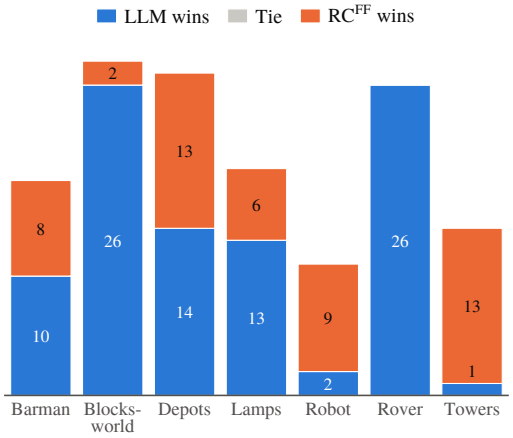


(a) Coverage by domain. LLM matches or exceeds PANDA RC^{FF} on six of seven domains; Robot is the exception, where every PYTRICH configuration stops at the eleven smallest instances within the time limit.

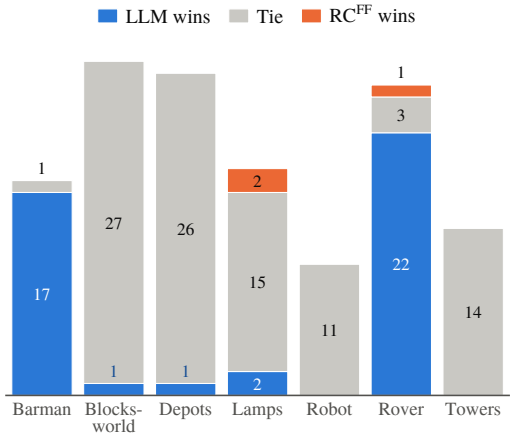
	Barman	Blocks-world	Depots	Lamps	Robot	Rover	Towers
Panda RC^{FF}	90	97	90	95	100	87	93
Claude Opus 4.6	100	93	100	70	55	100	100
Claude Sonnet 4.5	0	0	77	0	0	0	0
GPT-4o	0	0	77	0	0	0	0
GPT-5	0	0	0	0	0	0	100
GPT-5.2	0	0	0	0	0	0	100
Gemini 2.0 Flash	95	67	57	0	55	100	100
Gemini 2.5 Pro	25	47	57	65	55	23	100
Gemini 3 Flash	90	93	87	100	55	100	100
Gemini 3 Pro	100	80	0	0	55	100	100

(b) Coverage heatmap across models and domains. Claude Opus 4.6 and Gemini 3 Flash solve 148 instances each; GPT-4o, Claude Sonnet 4.5, GPT-5 and GPT-5.2 have results in one domain. Cells give the percentage of the instances in that domain solved by LLM or RC^{FF}.

Figure 4: Coverage results for LLM vs. PANDA RC^{FF}: per-domain bar chart (a) and per-model heatmap (b).

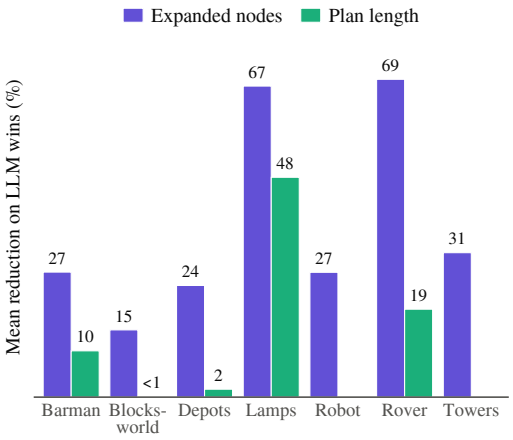


(a) Win/loss on expanded nodes. LLM expands fewer nodes on 92 of 143 shared instances, every one in Rover and 26 of 28 in Blocksworld; RC^{FF} wins Robot (9 of 11) and Towers (13 of 14), and Barman and Depots split about evenly.

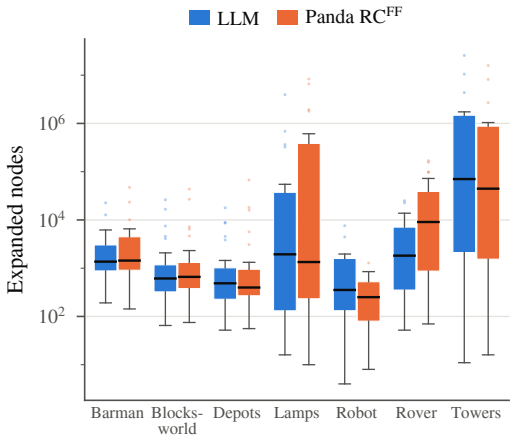


(b) Win/loss on solution size. Shortest plan over all runs on each side: LLM is shorter on 43 of 143 shared instances, equal on 97 and longer on 3; the wins are Barman (17 of 18) and Rover (22 of 26), with one instance each in Blocksworld and Depots and two in Lamps.

Figure 5: Win/loss breakdown by domain: LLM vs. PANDA RC^{FF} on expanded nodes (a) and solution size (b). Each bar segment counts shared instances where LLM expands fewer (win), equal (tie), or more (loss) nodes or actions.

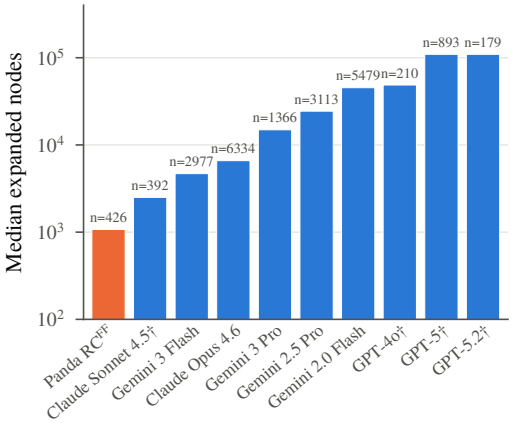


(a) Average percentage improvement in expanded nodes on instances where LLM wins. Rover shows the largest mean reduction (69%) and Lamps the next (67%); Blocksworld the smallest (15%).

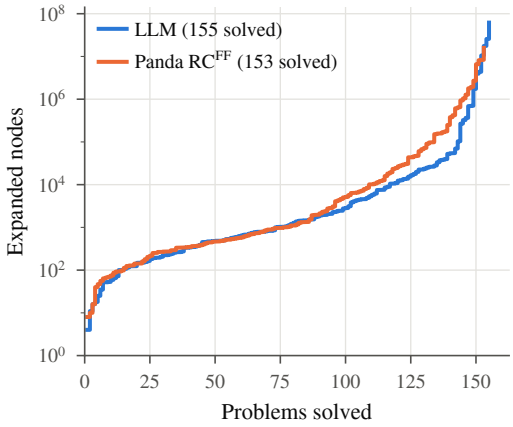


(b) Boxplot of expanded nodes per domain on the problems solved by both systems, with the best run of each side on each instance. Rover shows the clearest LLM advantage; Robot and Towers favour RC^{FF}.

Figure 6: Search efficiency detail: LLM vs. PANDA RC^{FF}. Panel (a) shows mean improvement on LLM wins; panel (b) shows the distribution of best node counts per domain on shared instances.



(a) Median expanded nodes per model on problems each model solved. No LLM model matches the PANDA medians (1,073 nodes for RC^{FF} and 757 for RC^{LMC} over their solved runs, against 4,686 for Gemini 3 Flash and 6,599 for Claude Opus 4.6); the counts are planner-specific and are not comparable across the two systems. Models with results in only one domain are shown with a dagger.

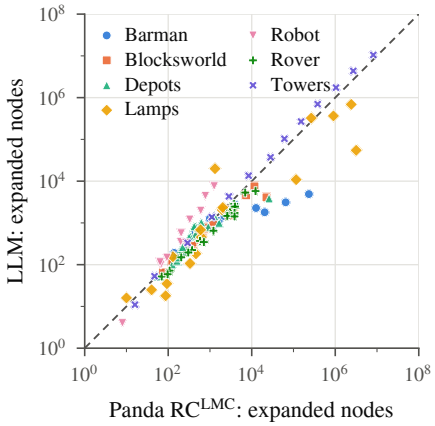


(b) Cactus plot: problems solved within a given node-expansion budget. The two curves track each other up to a budget of ten thousand nodes and separate at the tail, where LLM solves the Rover and Barman instances that RC^{FF} solves late or not at all; within ten thousand nodes LLM is ahead in Rover (20 against 13) and behind in Lamps and Towers by one.

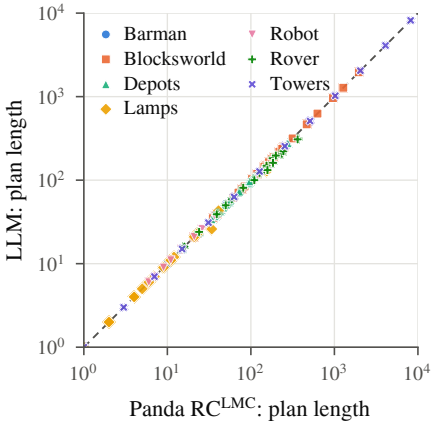
Figure 7: Model ranking and cumulative coverage: LLM vs. PANDA RC^{FF}. Panel (a) compares median search effort per LLM model against the two PANDA variants; panel (b) shows how coverage accumulates as the node budget grows.

F.2 PANDA RC^{LMC}

This subsection mirrors the RC^{FF} comparison for completeness. Since PANDA RC^{LMC} solves fewer problems than PANDA RC^{FF}, the shared instance set is smaller in several domains.

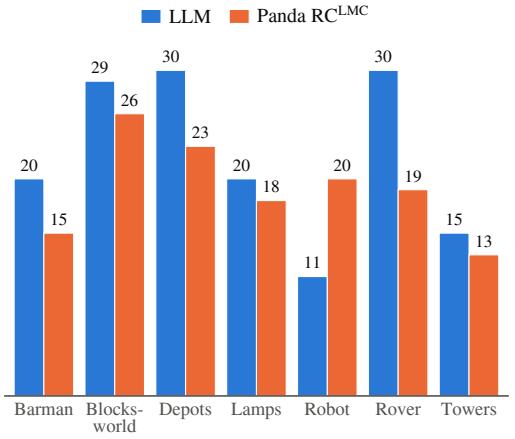


(a) Expanded nodes: LLM vs. PANDA RC^{LMC}. PANDA RC^{LMC} has the lowest median node count of all baselines (757 over its solved runs), and LLM expands fewer nodes on 76 of the 125 shared instances, median ratio 0.88, winning every Rover instance and 25 of 26 in Blocksworld while losing Robot (10 of 11), Towers (12 of 13) and most of Depots.

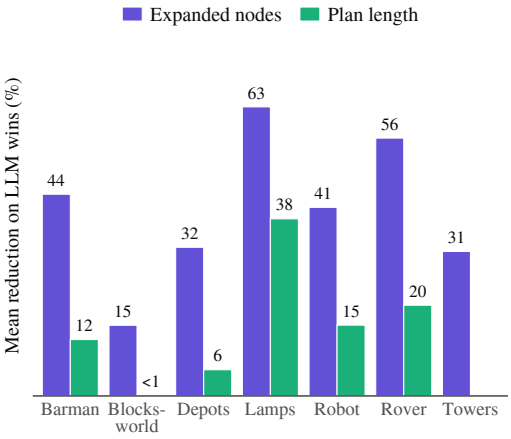


(b) Solution size: LLM vs. PANDA RC^{LMC}. Points sit on the diagonal except in Barman and Rover: shortest plan over all runs, LLM is shorter on 18 of 125 shared instances, equal on 105 and longer on 2.

Figure 8: Per-problem scatter plots: LLM vs. PANDA RC^{LMC} on expanded nodes (a) and solution size (b). Points below the diagonal indicate LLM advantage.



(a) Coverage by domain. PANDA RC^{LMC} solves 134 of 180 problems, against 155 for LLM; the gap is largest in Rover (19 against 30) and Depots (23 against 30).



(b) Average percentage improvement in expanded nodes on instances where LLM wins. On the instances LLM wins, the mean reduction is largest in Lamps (63%) and Barman (44%) and smallest in Blocksworld (12%).

Figure 9: Coverage and search-efficiency summary: LLM vs. PANDA RC^{LMC}.

F.3 Plan Length by Domain

Table 12 reports median plan length restricted to the problems solved by all seven systems, with each entry being the virtual best over the three search algorithms. Every system returns the same median length in Blocksworld, Depots, Robot, Towers and Lamps. The systems differ only in Barman and

Table 12: Median plan length (primitive actions) per domain, restricted to the problems solved by all seven systems. Each entry is the virtual best over the three search algorithms (GBFS, A*, WA*).

		PANDA			PYTRICH			
Domain	n	RC^{FF}	RC^{LMC}	LMCO^{P}	TDG	LMCO	Opus 4.6	LLM
Barman-BDI	15	44	38	38	43	40	38	38
Blocksworld-GTOHP	20	146	146	146	146	146	146	146
Depots	23	57	57	57	57	57	57	57
Robot	11	11	11	11	11	11	11	11
Rover-GTOHP	16	101	96	146	92	90	90	90
Towers	13	127	127	127	127	127	127	127
Lamps	12	4	4	4	4	4	4	4
Median		57	57	57	57	57	57	57

Rover: in Barman RC^{FF} returns plans six actions longer than the LLM heuristics in the median, and in Rover RC^{FF} and LMCO^{P} return plans 11 and 56 actions longer. Yousefi et al. [27] establish that A^* is incomplete for total-order HTN planning, so differences in plan length across planners with different internal search strategies are unsurprising (see also Section 5.2).

F.4 Median Expanded Nodes by Model and Algorithm

Table 13 reports median expanded nodes broken down by model, algorithm, and domain. Dashes indicate that the model produced no working heuristics for that domain. Claude Opus 4.6 rows are highlighted in bold; the overall median pools all problems solved by each model under each algorithm.

Table 13: Median expanded nodes over solved runs by model, algorithm and domain. LLM pools the five main models; the PYTRICH baseline rows pool their runs under the three algorithms, which the record does not label.

Model	Algo	Barman	Blocks.	Depots	Robot	Rover	Towers	Lamps	Overall
LLM	GBFS	47,074	1,358	3,861	1,419	13,785	110,252	4,229	8,761
LLM	WA*	76,541	1,904	6,571	1,495	15,570	110,208	8,298	11,934
LLM	A*	377,048	25,352	23,502	1,495	16,425	110,375	7,260	21,602
GPT-4o	GBFS	—	—	8,614	—	—	—	—	8,614
	WA*	—	—	74,097	—	—	—	—	74,097
	A*	—	—	74,097	—	—	—	—	74,097
Opus 4.6	**GBFS**	**15,523**	**540**	**953**	**1,185**	**29,136**	**109,274**	**10,982**	**3,273**
	WA*	**26,788**	**647**	**1,167**	**1,117**	**36,430**	**109,378**	**10,306**	**4,972**
	A*	**515,107**	**47,166**	**7,776**	**1,116**	**17,805**	**109,484**	**8,319**	**20,835**
Claude Sonnet 4.5	GBFS	—	—	1,128	—	—	—	—	1,128
	WA*	—	—	1,860	—	—	—	—	1,860
	A*	—	—	8,436	—	—	—	—	8,436
GPT-5	GBFS	—	—	—	—	—	109,274	—	109,274
	WA*	—	—	—	—	—	109,378	—	109,378
	A*	—	—	—	—	—	109,484	—	109,484
GPT-5.2	GBFS	—	—	—	—	—	109,274	—	109,274
	WA*	—	—	—	—	—	109,378	—	109,378
	A*	—	—	—	—	—	109,484	—	109,484
Gemini 2.0 Flash	GBFS	909,166	39,582	18,379	1,498	80,128	111,058	—	20,795
	WA*	3,244,057	64,696	74,097	1,582	71,108	111,162	—	48,631
	A*	3,244,057	66,868	74,097	1,582	71,108	111,162	—	48,631
Gemini 2.5 Pro	GBFS	3,376,203	62,760	18,379	1,498	80,128	111,058	4,229	31,074
	WA*	3,244,057	69,069	74,097	1,582	71,108	111,162	11,527	32,970
	A*	3,244,057	69,069	74,097	1,582	71,108	111,162	8,319	23,502
Gemini 3 Flash	GBFS	80,200	1,511	1,102	797	5,415	106,154	3,253	3,765
	WA*	71,556	1,702	1,113	788	5,451	105,722	2,479	3,942
	A*	110,384	4,522	4,464	789	15,418	108,112	5,035	6,554
Gemini 3 Pro	GBFS	10,558	1,357	—	877	2,624	110,638	—	14,639
	WA*	12,714	1,519	—	884	2,720	110,468	—	14,705
	A*	15,784	3,933	—	957	4,224	110,671	—	23,425
LMCO^{P}	GBFS	1,596	789	412	596	19,535	44,468	2,365	1,196
	WA*	16,604	984	482	621	61,166	44,468	1,226	1,378
	A*	357,790	2,667	14,232	158	126,745	44,468	1,289	3,597
PANDA RC^{Add}	GBFS	1,438	692	456	1,060	2,766	44,468	1,014	1,010
	WA*	1,546	692	506	1,076	2,807	44,468	1,010	1,074
	A*	1,452	692	691	825	3,612	44,468	643	1,090
PANDA RC^{FF}	GBFS	1,438	682	398	1,067	10,106	44,468	1,346	1,024
	WA*	1,992	737	482	966	11,394	44,468	2,300	1,079
	A*	3,169	636	870	361	21,919	44,468	671	1,107
PANDA RC^{LMC}	GBFS	1,173	629	334	1,019	2,595	28,236	850	721
	WA*	1,947	655	342	622	2,660	28,236	789	727
	A*	7,910	618	6,106	392	701	28,236	648	1,005
LMCO	all	24,944	1,457	1,414	1,450	17,389	111,162	6,462	4,936
TDG	all	84,196	1,468	3,297	891	3,512	109,972	1,512	4,730

F.5 PYTRICH-TDG

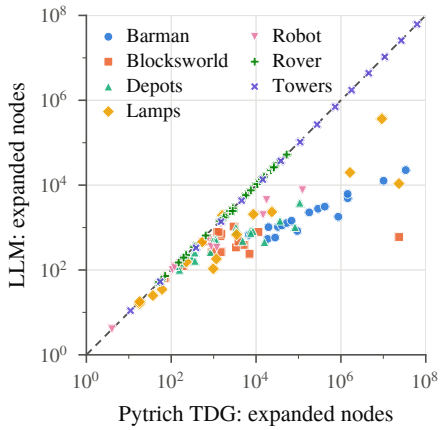
This subsection compares LLM against PYTRICH guided by TDG. The figures report the same metrics as above: coverage, expanded nodes, and plan length on shared solved instances.

Why the margin over TDG varies by domain. TDG gives every method a weight of one and every operator its cost, estimates each pending task by its cheapest method, and sums over the task network, so it never reads the state. Wherever a task has a method with no subtasks whose precondition is that the task is already achieved, TDG assumes that method, and its estimate reduces to the number of pending tasks whatever the state. In Barman every `Achieve` task has such a method, for instance `AddIngredientToShotNull` when the shot already contains the ingredient, while the true cost depends on whether the shot is clean, the shaker empty and a hand free; TDG expands 1,460 nodes per plan action at the median against 29 for the best Claude Opus 4.6 heuristic, and the

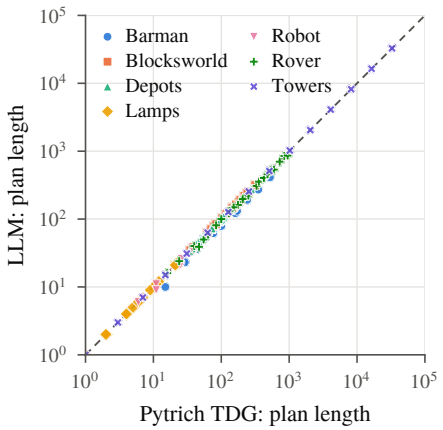
LLM virtual best expands 98% fewer nodes. Blocksworld and Depots have the same shape, with `do_put_on` and `do_clear` decomposing to `nop` when the block or crate is already in place, while the true cost is the height of the stack above it; the LLM virtual best expands 55% and 59% fewer nodes. In Lamps `play` and each `propagate` task end with an empty method, while the cost is the number of lamps in the row that still share the flipped status; 76% fewer nodes. In Rover `get_soil_data` decomposes into navigate, empty the store, sample and send, each with a fixed cheapest cost, so TDG estimates about four actions per sample goal and six per image goal and only the navigation distance depends on the state; TDG and the best LLM heuristic both expand about 12 nodes per plan action, and the two are at parity. In Towers the preconditions of the methods are mutually exclusive, `on ?r ?t1` against `on ?r ?r1` and `exchangeLR` against `exchangeRL`, so at most one method applies in any state; every system, PANDA included, expands the decomposition tree of about four hundred nodes per action and returns the same plan, and the heuristic cannot matter. Robot is the exception. `achieve-goals` has the empty method `finished`, so TDG is as uninformative as in Barman, yet the LLM heuristics expand only 23% fewer nodes and no PYTRICH heuristic solves more than the eleven smallest instances; the recursive `achieve-goals`, `release` and `move_abstract` tasks place every state one decomposition step from any other, so the difficulty lies in the branching rather than in the estimate. Table 14 gives the structure of each lifted domain beside the median node ratio of LLM against TDG, ordered by that ratio. The margin tracks the decomposition branching factor only loosely. Rover and Towers, the two domains with the fewest methods per task, are the two where TDG is at parity, and the domains with two or more methods per task show the largest gains, but what separates Barman and Lamps from Blocksworld and Depots is not in the table. It is which of those methods is the empty one, as the paragraph above describes.

Table 14: Structure of the seven domains, measured on the lifted HDDL domain file, and the median node ratio of LLM against TDG on shared instances. $|A|$, $|C|$ and $|M|$ count primitive action, compound task and method schemas; methods per task and subtasks per method are mean (max); depth is the longest chain in the condensation of the task decomposition graph.

Domain	$\|A\|$	$\|C\|$	$\|M\|$	Methods/task	Subtasks/method	Depth	Recursive	Node ratio vs TDG
Barman	11	10	22	2.20 (3)	1.82 (6)	6	no	0.02
Lamps	1	6	15	2.50 (3)	1.00 (5)	2	yes	0.24
Depots	6	6	12	2.00 (4)	2.42 (4)	3	yes	0.41
Blocksworld	5	4	8	2.00 (2)	2.00 (4)	1	yes	0.45
Robot	4	6	11	1.83 (4)	1.45 (2)	1	yes	0.77
Towers	1	5	8	1.60 (3)	1.25 (2)	3	yes	0.96
Rover	14	10	16	1.60 (3)	2.12 (4)	3	yes	1.00

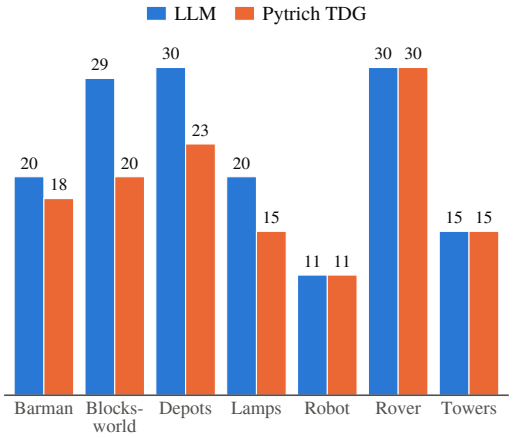


(a) Expanded nodes: LLM vs. TDG. LLM expands fewer nodes on 107 of the 132 shared instances, ties on 24, most of them in Rover, and loses one, in Lamps. The 48 instances TDG does not solve are absent from this plot.

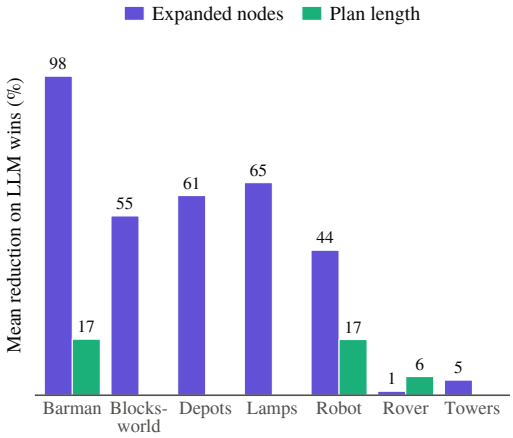


(b) Solution size: LLM vs. TDG. Shortest plan over all runs on each side: LLM is shorter on 43 of 132 shared instances and equal on the rest, with the wins in Barman (all 18), Rover (23 of 30) and two Robot instances.

Figure 10: Per-problem scatter plots: LLM vs. TDG on expanded nodes (a) and solution size (b). Points below the diagonal indicate LLM advantage.



(a) Coverage by domain. LLM solves 155 problems and TDG 132, and LLM solves every instance that TDG solves.

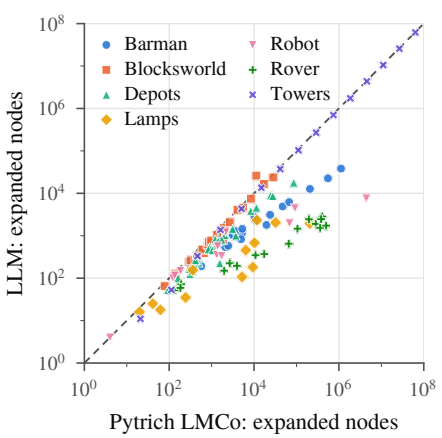


(b) Average percentage improvement in expanded nodes on shared instances where LLM wins. Barman shows the largest mean reduction (98%) and Lamps the next (65%); Towers (5%) and Rover (1%) the smallest, where the method library fixes most of the cost.

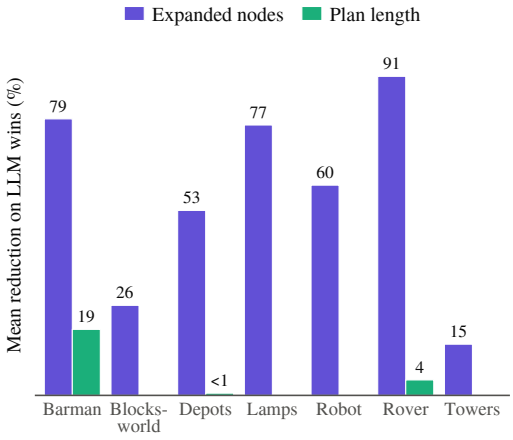
Figure 11: Coverage and search-efficiency summary: LLM vs. TDG.

F.6 PYTRICH-LMCO

This subsection compares LLM against PYTRICH guided by LMCO, the bidirectional landmark-count heuristic.



(a) Expanded nodes: LLM vs. LMCo. LLM expands fewer nodes on 126 of the 129 shared instances, median ratio 0.46, and loses two, in Blocksworld; every domain is won outright except Blocksworld (27 of 29). Points below the diagonal indicate an LLM advantage.



(b) Average percentage improvement in expanded nodes on shared instances where LLM wins. The median node ratio per domain is 0.02 in Rover, 0.11 in Lamps, 0.22 in Barman, 0.41 in Robot, 0.47 in Depots, 0.78 in Blocksworld and 0.92 in Towers; plan lengths tie on every shared instance outside Barman and Rover.

Figure 12: Search efficiency within PYTRICH: LLM vs. LMCo on per-problem expanded nodes (a) and average improvement per domain (b).

G Heuristic API Details

This appendix describes the implementation-level interface used for all LLM-generated heuristics.

Heuristic class. Each heuristic subclasses `Heuristic` and implements two methods:

- `initialize(self, model: Model, initial_node: HTNNode)`: called once before search for preprocessing.
- `__call__(self, parent_node: HTNNode, node: HTNNode) -> int`: called at each node expansion; returns a non-negative integer estimate of remaining solution cost.

Model interface. The `Model` object exposes grounded planning objects including `facts`, `operators`, `abstract_tasks`, `decompositions`, and the goal bitset `goals`.

Node interface. Each `HTNNode` provides:

- `node.state`: integer bitset encoding the current state;
- `node.task_network`: ordered remaining tasks (each tagged primitive or abstract);
- `node.decomposition`: the method applied to reach this node, or `None` at the root and after a primitive action.

Bitset access. Fact truth checks use constant-time bit operations, `(node.state >> fact.global_id) & 1`, and operator preconditions/effects are also represented as bitsets. This allows efficient reachability-style computations without string parsing.

Grounded fact naming. Grounded facts follow runtime names such as `+predicate[arg1,arg2]` and `-predicate[arg1,arg2]`, which differ from raw HDDL syntax. Prompts include this convention explicitly to avoid generation errors.

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 abstract and Section 1 state three contributions, the pipeline in PYTRICH, the evaluation of nine models on seven domains against domain-independent heuristics and PANDA, and the analysis of search effort, domain structure, models and algorithms, each supported by Section 5 and Section 6.

   Guidelines:

   - The answer [N/A] 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 [N/A] 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: Section 8 states four limitations: the evaluation covers seven total-order HTN domains and leaves partial-order, numerical and temporal planning open; the virtual best is an upper bound and the fixed configuration's single selection instance filters without ranking; generation is stochastic and the prompts contain two evaluated instances per domain, which Appendix D.5 recomputes without; and the heuristics cannot run inside PANDA, so that comparison rests on coverage.

   Guidelines:

   - The answer [N/A] 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: [N/A]

   Justification: The paper presents no theoretical results; all contributions are empirical.

   Guidelines:

   - The answer [N/A] 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: Section 3 describes the generate-evaluate-select pipeline; Appendix A reproduces the full prompt text; Appendix G documents the heuristic interface; and Appendix D provides model versions, generation parameters, and resource limits. Both baseline planners are publicly available.

   Guidelines:

   - The answer [N/A] 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: The code, the generated heuristics, the result files and the scripts that produce every table and figure are released as a repository archived on Zenodo [19], with a README that gives the exact commands to rebuild the PANDA tools at pinned commits, ground the benchmarks, submit the job arrays and compare the outcome with the released result files. The IPC benchmark domains are publicly available, and the full prompt and heuristic API are reproduced in Appendix A and Appendix G.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.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://neurips.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) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4 specifies the benchmark domains, LLM models, search algorithms, resource limits (1 CPU core, 8 GB RAM, 30-minute wall time), and evaluation metrics. Appendix D provides model versions, generation parameters, and further infrastructure details.

   Guidelines:

   - The answer [N/A] 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: [N/A]

   Justification: Every planner run is deterministic given the heuristic, the instance and the search algorithm, so coverage and expanded nodes carry no sampling error at run time; the stochastic step is candidate generation, which Section 8 lists as a limitation, and Appendix C reports that two draws of twenty candidates from the same prompt can differ by a factor of 2.7 in median nodes. Error bars over repeated generation would require regenerating every model and domain and are not reported.

   Guidelines:

   - The answer [N/A] 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 states the per-problem resource allocation (1 CPU core, 8 GB RAM, 30-minute wall-clock limit). Appendix D provides cluster details and additional execution information for both PYTRICH and PANDA runs.

   Guidelines:

   - The answer [N/A] 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: The research uses publicly available LLM APIs and IPC benchmark domains, involves no human subjects or sensitive data, and raises no ethical concerns identified in the NeurIPS Code of Ethics.

Guidelines:

- The answer [N/A] 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: [N/A]

Justification: This work is foundational algorithmic research on HTN search heuristics and has no immediate direct societal impact within the scope of this paper.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] 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., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release pre-trained models, scraped datasets, or other assets with high misuse risk. The generated heuristics are domain-specific Python functions for planning search.

Guidelines:

- The answer [N/A] 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 are cited with their original papers: PYTRICH [22], which is based on the Pyperplan planner (GPL 3 license); PANDA [2, 15] (BSD 3 license); the IPC 2020 HTN benchmark domains [1] and the IPC 2023 Lamps domain [23]. LLM APIs are accessed under their standard published terms of use.

    Guidelines:

    - The answer [N/A] 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 generated heuristics and the result files are part of the released repository (see the code question above). The heuristic interface they implement is documented in Appendix G and the generation procedure is described in Section 3.

    Guidelines:

    - The answer [N/A] 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: [N/A]

Justification: The paper involves no crowdsourcing or research with human subjects; all experiments are automated planner evaluations on benchmark problems.

Guidelines:

- The answer [N/A] 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: [N/A]

Justification: The paper involves no human subjects research and requires no IRB approval or equivalent review.

Guidelines:

- The answer [N/A] 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 rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are the core method of this work. Section 3 describes how LLMs are prompted to generate Python heuristic functions, Section 4 lists the nine models evaluated and their providers, and Appendix D gives the generation settings and the candidate counts per model and domain.

Guidelines:

- The answer [N/A] 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 in the NeurIPS handbook for what should or should not be described.