Property-Guided LLM Program Synthesis for Planning

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

Augusto B. Corrêa
University of Oxford
United Kingdom

Jendrik Seipp
Linköping University
Sweden

Abstract

LLMs have shown impressive success in program synthesis, discovering programs that surpass previously known solutions. However, these approaches rely on simple numeric scores to signal program quality, such as the value of the solution or the number of passed tests. Because such a score offers no guidance on *why* a program failed, the system must generate and evaluate many candidates per problem in the hope that some succeed, increasing both LLM inference and evaluation costs. We study a different approach: property-guided LLM program synthesis. Instead of scoring programs after evaluation, we check whether a candidate satisfies a formally defined property. When the property is violated, we stop the evaluation early and provide the LLM with a concrete counterexample showing exactly how the program failed. In our experiments, this feedback reduces both the number of program generations and the evaluation cost, and guides the LLM to stronger programs. We evaluate this approach on PDDL planning domains, asking the LLM to synthesize *direct* heuristic functions: every non-goal state reachable from the initial state by strictly improving transitions has a strictly improving successor. A heuristic with this property leads the hill-climbing algorithm directly to a goal state. A counterexample-guided repair loop generates one candidate program, checks the property over a training set, and returns the first case that violates the property. On the ten IPC 2023 Learning Track domains, our approach with Gemini 3.1 Pro generates seven times fewer programs than the best prior generation method, spends about $40\times$ less time evaluating candidates on training tasks, and solves more out-of-distribution test tasks using only hill climbing. Although the property is checked only on small training tasks, hill climbing with the synthesized heuristics gets stuck in only one of $2{,}700$ test runs. The coverage advantage also holds with GPT-5.5 and the small open-weight Gemma 4 31B. On three newly created domains that were not publicly available when Gemini 3.1 Pro was trained, the synthesized heuristics expand about $3.5\times$ fewer states than the best prior generation method, and hill climbing never encounters a property violation on the test tasks. Our results show that property-guided LLM program synthesis can reduce the synthesis and evaluation cost of heuristic functions while improving their quality.

1 Introduction

Using large language models (LLMs) for *program synthesis* yields programs that surpass previously known solutions on problems ranging from competitive programming [39] to open mathematical questions [55] and algorithmic discovery [45, 42, 67, 18, 12]. These approaches use simple numeric scores to rank candidate programs, such as the average value of the solution or the number of passed tests. Because such a score offers no guidance on *why* a program failed, the system must generate and evaluate many candidates per problem in the hope that some succeed, increasing both LLM inference and evaluation costs.

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

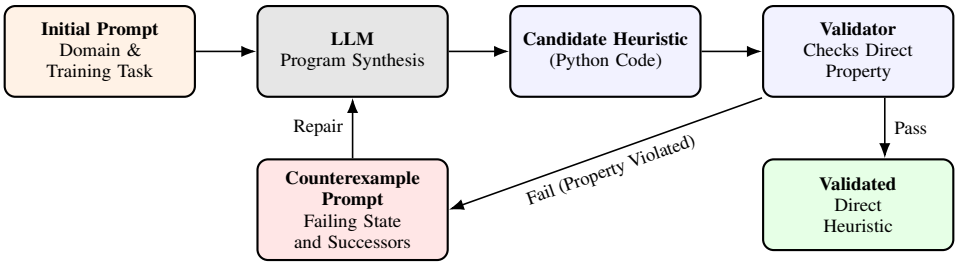


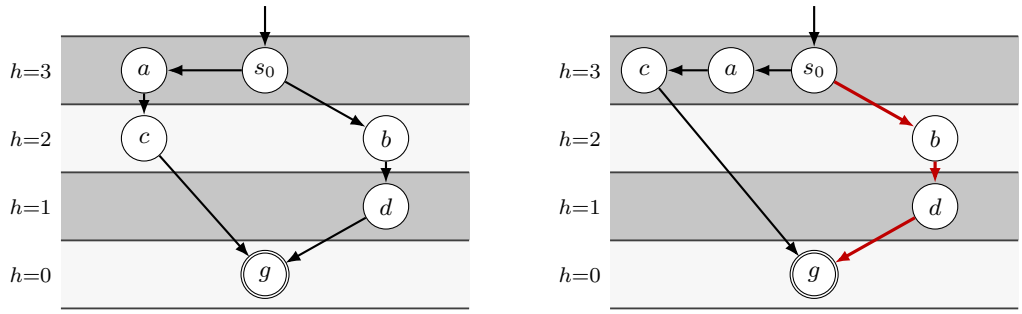
Figure 1: Counterexample-driven repair loop for property-guided heuristic function synthesis. Given a planning domain and a task from the training set, the LLM proposes a candidate heuristic function in Python. The validator evaluates this heuristic on the training set, verifying the *direct* property. If the property is satisfied, the valid heuristic is output. If the property is violated, evaluation stops early, and a concrete counterexample is returned.

There is an idea from formal methods that replaces this fitness score with something more informative: *counterexample-guided inductive synthesis* (CEGIS) [62, 38, 1]. A verifier checks each candidate against a formal specification, and on failure it returns a concrete input on which the candidate is wrong; the synthesizer can then repair or discard the candidate using that witness instead of a single number. CEGIS underpins much of classical program synthesis and has recently been combined with LLMs [47]. We use an LLM as the synthesizer in a CEGIS-style loop with a property checker as the verifier. On every failed candidate, we hand the LLM a concrete counterexample of where and how the program went wrong, which lets us stop evaluation as soon as the property is violated. This can reduce the number of candidates the LLM has to generate and steer it toward stronger programs.

We apply this idea to *classical planning* [28], a problem known to be challenging for LLMs [69, 68, 64]. Given a planning domain, we ask the LLM to synthesize a *heuristic function* as Python code, which is then automatically injected into a planner. Ideally, such a heuristic would guide hill climbing from the initial state to a goal without ever getting stuck, solving tasks without any combinatorial search. To capture this requirement formally, we target the *direct* property [20]: a heuristic is direct if every non-goal state reachable from the initial state via strictly improving transitions has a strictly improving successor. This property is a natural fit for property-guided synthesis, since any violation immediately yields a concrete counterexample. Synthesizing a direct heuristic is challenging both in theory [35, 20] and in practice [e.g., 25]: it requires reasoning about every state on every strictly improving path, and a single state without an improving successor is enough to break the property. In fact, synthesizing a direct heuristic for a given task is as hard as solving the task itself.

Our counterexample-driven repair loop (Figure 1) takes a PDDL (Planning Domain Definition Language) domain together with a training set of PDDL tasks from this domain as input, and validates the direct property over the training set. Validation terminates as soon as a counterexample is found. In this case, we prompt the LLM with the failing task and the counterexample: the state where the property fails, its heuristic value, and the successor states with their heuristic values. The LLM is asked to provide a new heuristic that is direct for this task. This loop repeats until the heuristic passes the full training set. Although validation is confined to the training tasks, we aim for the synthesized heuristic to generalize, remaining direct on the much harder, out-of-distribution test tasks. To our knowledge, our work is the first to use LLMs to synthesize heuristics via property-guided repair.

On the ten planning domains from the Learning Track of the International Planning Competition (IPC) 2023 [65], evaluated on an out-of-distribution test set, our approach with Gemini 3.1 Pro generates seven times fewer programs per domain on average compared to the best prior generation method, sample-and-select [18], spends about $40\times$ less time evaluating candidates on training tasks, and solves more tasks via hill climbing without any combinatorial search. Hill climbing with the synthesized heuristics gets stuck in only one of $2{,}700$ out-of-distribution test runs, indicating that the property generalizes well beyond the training set. The coverage advantage over sample-and-select also holds with GPT-5.5 and the small open-weight Gemma 4 31B. We also evaluate our approach on three newly created domains that were not publicly available when Gemini 3.1 Pro was trained. The repair loop converges in all nine synthesis runs, and no direct-property violation is observed on any test task. This shows that the model can synthesize effectively direct heuristics for domains it had not



(a) Descending heuristic: every alive state has at least one strictly improving successor, including state a which will never be expanded by GBFS or HC.

(b) Direct heuristic: the same state space with a different heuristic admits a strictly improving path ($s_0 \to b \to d \to g$), but state a is a plateau ($h(a) = h(s_0)$).

Figure 2: Example descending and direct heuristics for the same state space. Nodes are placed on horizontal heuristic layers; downward transitions strictly decrease the heuristic value h. State s_0 is the initial state and g is the goal state. Both heuristics are dead-end avoiding since all states are solvable. The left heuristic is descending and thus also direct; the right heuristic is direct but not descending.

encountered during training, with coverage comparable to sample-and-select and fewer candidates and expanded states overall.

2 Background

Classical planning asks for a sequence of deterministic actions, a *plan*, that transforms a given initial state into a state satisfying a goal in a fully observable, discrete environment with a single agent. Planning tasks are commonly described in the Planning Domain Definition Language (PDDL) [43, 33], and our work directly uses this representation.

A PDDL task is defined by a set of *objects* and *predicates*. Together they form *ground atoms*, i.e., predicates instantiated with concrete objects, which describe the current state. *Actions* have preconditions that must hold for the action to be applicable, and effects that add or remove ground atoms from the state. The task also specifies an *initial state* s_0 and a *goal*. PDDL separates the *domain* (shared actions and predicates) from the *task* (specific objects, initial state, and goal), typically stored in two separate files.

We consider *satisficing planning*, where any plan is acceptable regardless of its length, and we assume all actions have unit cost. Most satisficing planners are based on *greedy best-first search* (GBFS) [52, 8, 36, 34, 54, 40], a state-space search algorithm guided by a *heuristic function* h that maps each state s to a value $h(s) \in \mathbb{R} \cup \{\infty\}$ estimating the cost to reach a goal from s [50]. Infinite values indicate that the goal is unreachable from s. GBFS starts from s_0 and maintains a priority queue (the *open list*) of generated states ordered by their heuristic value. At each step, it *expands* the most promising state s, i.e., the one with the smallest heuristic value $h(s)$, generating its successors $succ(s)$ and adding them to the open list. This process continues until a goal state is reached. GBFS is complete: if a solution exists, it will eventually find one, regardless of the heuristic, as long as all generated states are eventually expanded.

In contrast to the global nature of GBFS, *hill climbing* (HC) is a simple local-search algorithm: starting from the initial state, it greedily moves to a successor with a strictly smaller heuristic value and terminates when it reaches a goal state. Because HC keeps no open list, it is very memory efficient. However, HC is incomplete: it can fail at *plateaus* (where no successor strictly improves the heuristic) or *local minima* (where every successor has a larger heuristic value), even when the goal is reachable. Completeness of HC therefore depends on the heuristic.

A *descending and dead-end avoiding* (DDA) heuristic [59] eliminates both failure modes. Following Seipp et al. [59], we call a state *alive* if it is reachable, solvable, and not a goal.

Definition 1 (Descending and Dead-End Avoiding Heuristics [59, 35])**.** *A heuristic h is* descending *for a planning task Π if every alive state s has at least one successor $s' \in succ(s)$ with $h(s') < h(s)$, and it is* dead-end avoiding *if every such improving successor is solvable.*

Proposition 1 (Seipp et al. [59])**.** *Let h be a descending, dead-end avoiding heuristic for a solvable planning task Π. Then hill climbing guided by h reaches the goal from s_0.*

The proof is straightforward: every alive state has a solvable improving successor, so heuristic values strictly decrease at each step. Strict decrease prevents revisiting a state, and since the reachable state space is finite, the search must reach a goal in finitely many steps.

The DDA property is, however, unnecessarily strong. It enforces that all alive states have improving successors, *even when these states would never be expanded* by a planner. For example, the heuristic h in Figure 2a is DDA because all alive states have improving successors. But this includes a, which is never expanded by GBFS or HC using h, because $h(b) < h(a)$, and b already has a strictly improving path to the goal (i.e., h strictly decreases at every step). To address this issue, Dold and Helmert [20] introduce a relaxation that only considers states reachable from s_0 by repeatedly taking strictly improving transitions. They call this property *wet descending and dead-end avoiding* (WDDA). For brevity, we refer to a heuristic with this property as a *direct* heuristic throughout the rest of the paper.

Definition 2 (Direct Heuristic [20])**.** *Let $S_\downarrow$ be the set of alive states reachable from s_0 via strictly improving transitions. A heuristic h is* direct *if s_0 is solvable, every $s \in S_\downarrow$ has at least one successor $s' \in succ(s)$ with $h(s') < h(s)$, and every such improving successor is either a goal state or itself in $S_\downarrow$.*

For instance, state a in Figure 2b is alive but has $h(a) = h(s_0)$, so $a \notin S_\downarrow$ and the direct property places no requirement on it, whereas DDA would still demand an improving successor. This seemingly minor relaxation can make the property substantially easier to satisfy and to synthesize [20]. DDA implies directness, but not the other way round. HC guided by a direct heuristic also reaches a goal state without ever getting stuck.

Synthesizing a direct heuristic for a single PDDL task is also theoretically hard. A direct heuristic exists for a task Π if and only if Π is solvable: any solvable task has a direct heuristic (e.g., the perfect heuristic), and any direct heuristic guides hill climbing to a goal. Deciding whether a direct heuristic exists is therefore equivalent to deciding plan existence, which is **EXPSPACE**-hard for PDDL planning (a.k.a. "lifted planning") [23, 15]. We sidestep this per-task cost by synthesizing a single *domain-general* direct heuristic from a small training set, amortizing the synthesis effort over the potentially infinite distribution of tasks in the domain.

3 Property-Guided LLM Heuristic Function Synthesis

We use an LLM to synthesize a domain-dependent heuristic function, implemented as Python code, for a given PDDL domain. Our approach uses a single counterexample-driven repair loop: the LLM generates one candidate, we validate it on a set of training tasks, and if it fails we return a concrete counterexample to the LLM and ask for a new candidate. This loop repeats until the heuristic passes the full training set or a generation budget is exhausted. Below, we discuss the target property (Section 3.1), the prompts (Section 3.2), and validation (Section 3.3).

3.1 Targeting Direct Heuristics

The DDA property is a strict global requirement: it must hold at every alive state, and a single violating state is enough to fail, even at states that hill climbing would never reach. We therefore target heuristics with the *direct* property: every non-goal state reachable from s_0 by strictly improving transitions must have a strictly improving successor. A direct heuristic still guarantees that hill climbing reaches a goal, but it focuses only on the subgraph that hill climbing can actually traverse.

Crucially for our method, every direct-property failure yields a concrete, local counterexample. The first type of failure occurs at a state s on some strictly improving path from s_0 whose successors s' all satisfy $h(s') \geq h(s)$; the counterexample reports s, its heuristic value $h(s)$, and the list of successors together with their heuristic values. In domains with dead ends, a second type of failure can occur when the DFS reaches a non-goal state s with no applicable action via a strictly improving transition from a parent s'; the counterexample reports s, its heuristic value $h(s)$, and the parent's heuristic value $h(s')$, together with the actionable suggestion that s should be assigned a value $\geq h(s')$ so that hill climbing never enters this dead end. This information is directly actionable and gives the LLM precise feedback on how to improve the heuristic. Although validation is confined to the

training tasks, we aim for the synthesized heuristic to generalize and remain direct on the much harder, out-of-distribution test tasks.

3.2 Counterexample-Driven Repair Loop

The repair loop maintains a history of all previously generated heuristics and their failure feedback. It starts with an *initial prompt* that requests the first candidate and uses a *repair prompt* after each validation failure.

Initial Prompt. We use the same prompt as Corrêa et al. [18] to make a direct comparison easier. It provides the PDDL domain, the smallest and largest training PDDL tasks from this domain, heuristic Python code examples for two domains disjoint from the evaluated domains, relevant planner code, and a checklist. Note that the prompt does not instruct the LLM to generate a direct heuristic.

Repair Prompt. When a candidate fails validation on a training task, the repair prompt asks the LLM to propose a heuristic that is direct on this task as well, and provides the definition of the direct property. It also includes:

1. the PDDL domain file;
2. the PDDL task on which validation failed;
3. the failure feedback: the failing state, its heuristic value, and the successor states with their heuristic values;
4. all previously generated heuristic functions, together with their failure feedback;
5. a checklist similar to the one in the initial prompt.

Items 2 and 3 give the LLM the exact nature of the failure. Item 4 lets the LLM reason about what has already been tried and avoid repeating the same mistakes (see Appendix D for an example prompt).

3.3 Validation

To check whether a candidate heuristic h satisfies the direct property on a task Π, we run a depth-first search (DFS) from s_0. At each expanded non-goal state s, we check whether it has at least one successor s' with $h(s') < h(s)$. If not, the direct property fails; we record s, its heuristic value, and all its successor states with their heuristic values as the counterexample. Otherwise, the DFS pushes onto the stack only the successors s' with $h(s') < h(s)$.

We apply this check to each task in the training set. As soon as a failure is found, we return the counterexample and stop, without checking the remaining tasks. We consider the heuristic a success if all expanded states have a strictly improving successor across all training tasks.

Even modestly sized planning tasks can admit exponentially many strictly improving paths to a goal, so validation cannot be expected to finish within a reasonable time on every task. We therefore impose a per-task time limit during validation. If this limit is reached before a counterexample is found, we treat the heuristic as direct on the task and continue with the next task.

4 Experimental Results

Experimental Setting. We run all experiments using Downward Lab [60] on AMD EPYC 7742 processors running at 2.25 GHz. We use Pyperplan [3] with PyPy 7.3.9 as our planning framework for both training and testing. We generate heuristics with two frontier models, Gemini 3.1 Pro [26] and GPT-5.5 [46], and with Gemma 4 31B [27], an open-weight model. We use default API parameters for Gemini 3.1 Pro and GPT-5.5, and default inference parameters for Gemma 4 31B. Our main benchmark uses the ten domains together with their training and test tasks from the IPC 2023 Learning Track [65]; we summarize each domain in Appendix A.[1] The IPC test set contains 90 tasks per domain (900 in total) and is out-of-distribution with respect to the training set: test tasks are generally much larger and may differ structurally from training tasks. Figure 3 shows the range of the main domain parameter on the IPC training and testing tasks. During training, each validation run is limited

[1]Two domains, Ferry and Satellite, use negative preconditions not supported by Pyperplan. We convert them to equivalent domains via a transformation that eliminates negative preconditions.

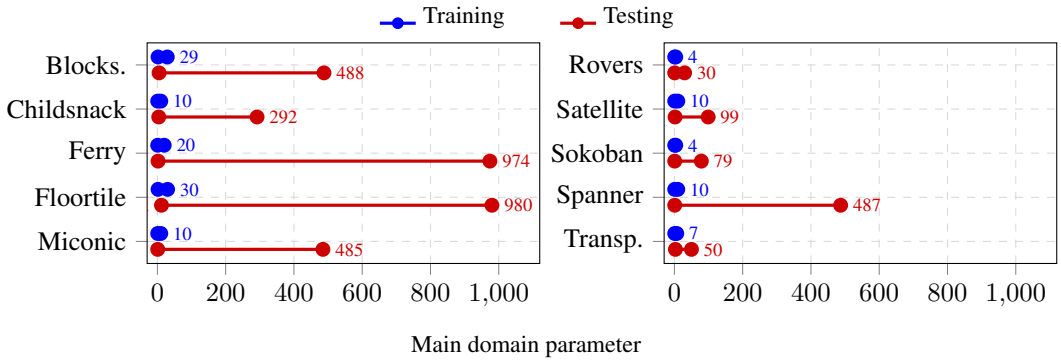


Figure 3: Size of training and test tasks measured by the main domain parameter. Each line spans the range of values for the corresponding set; maximum endpoints are annotated with their numeric values. The parameter counts blocks (Blocksworld), children (Childsnack), cars (Ferry), tiles (Floortile), passengers (Miconic), rovers (Rovers), satellites (Satellite), boxes (Sokoban), spanners (Spanner), and vehicles (Transport). The test set is out-of-distribution: in most domains the testing range extends well beyond the training range, and test tasks may also differ structurally from training tasks.

to 30 seconds and 8 GiB per task, and the repair loop runs for at most 10 repair iterations, i.e., at most 11 LLM calls including the initial one. During testing, each run is limited to 5 minutes and 8 GiB. For our methods, we report averages over three runs. The source code, prompts, generated heuristics, and experimental data are publicly available at [51].

We study the LLM as a *program synthesizer*: the repair loop generates one heuristic per domain, which is then used to solve all tasks in that domain, including unseen test tasks. In *end-to-end* planning, by contrast, the LLM is invoked separately for each training and test task, incurring much higher costs and losing all soundness guarantees on the resulting plans. For end-to-end planning performance across LLM families, sizes, and reasoning models, see Corrêa et al. [18]; for frontier LLMs in this setting, see Corrêa et al. [17].

Baselines. The *FF* heuristic [36] is one of the most commonly used heuristics for satisficing planning [e.g., 10, 16, 29]. It is a delete-relaxation heuristic computed in polynomial time by solving a relaxed planning problem that ignores *delete effects*; it is not designed to be *direct*. Corrêa et al. [18] introduce the current state-of-the-art method for LLM-based heuristic function generation for planning: it uses an LLM to independently generate a fixed budget of 25 candidate heuristics per domain and selects the one with the highest GBFS coverage on the training set and runs it with GBFS on the test set; it is also not designed to be direct. We refer to this method as *sample-and-select* throughout (abbreviated S&S in tables). We re-run this method with each model used by our approach to ensure a fair comparison.

Synthesis and Evaluation Costs of Candidate Heuristics. For Gemini 3.1 Pro, our repair loop is much more efficient than fixed-budget sample-and-select regarding the number of candidates the LLM must produce and the time spent evaluating them. Across the ten IPC domains and three independent runs per domain (30 runs in total), our method generates 101 candidate heuristics, on average 3.4 per run with standard deviation 3.0. Eight of the ten domains converge (i.e., generate a heuristic that is direct on the full training set) within at most 5 candidates per run, and four converge within 2 candidates on average. Sokoban and Floortile are the clear outliers: Sokoban averages 9 candidates per run with a maximum of 11 (initial attempt plus 10 repairs), and Floortile averages 8 with a maximum of 9. By comparison, the sample-and-select baseline of Corrêa et al. [18] fixes a budget of 25 candidates per domain regardless of difficulty, so the repair loop makes roughly $7.4\times$ fewer LLM calls per domain. Table 4 in Appendix C reports the per-domain breakdown.

Evaluation cost shrinks even more dramatically. In sample-and-select [18], each domain evaluates a fixed pool of candidates with GBFS, so evaluation cost scales with the number of candidates. In our repair loop, we validate one candidate at a time and stop at the first counterexample, which makes failed candidates cheap to reject. Summed over all candidates, our training-task validation takes 33.11 minutes per run on average. In our sample-and-select rerun, the measured serial sum of

Table 1: Per-domain coverage (number of solved tasks) on the 900 IPC 2023 test tasks in the Gemini 3.1 Pro experiment. **FF** is the classical delete-relaxation heuristic; **S&S** is our re-run of the sample-and-select baseline [18] with Gemini 3.1 Pro. Both baselines are run using GBFS and HC. **Direct** is our proposed method (direct heuristics run with HC). **Cov.** is the ablation that replaces the direct-property check with numeric coverage feedback (see "Ablation" paragraph), run using GBFS and HC. All proposed and ablation values are averages over three independent runs. Bold marks the best value per domain (ties are co-bolded). The full table with standard deviations is in Appendix B (Table 3).

	Baselines				**Ablation**		**Ours**
	FF		**S&S**		**Cov.**		**Direct**
Domain	**GBFS**	**HC**	**GBFS**	**HC**	**GBFS**	**HC**	**HC**
Blocksworld (90)	23	0	84	**90**	81.0	49.7	77.3
Childsnack (90)	12	0	58	70	58.0	46.7	**71.7**
Ferry (90)	45	0	67	71	64.7	49.3	**72.3**
Floortile (90)	11	0	30	0	39.0	15.7	**40.7**
Miconic (90)	61	1	83	7	82.0	34.0	**90.0**
Rovers (90)	29	2	**44**	12	36.7	20.7	42.7
Satellite (90)	35	17	57	**60**	56.0	57.3	55.7
Sokoban (90)	29	1	**30**	0	26.7	12.7	**30.0**
Spanner (90)	30	6	66	**79**	66.0	**79.0**	75.0
Transport (90)	23	3	54	41	62.7	47.3	**68.0**
Total (900)	298	30	573	430	572.7	412.3	**623.3**

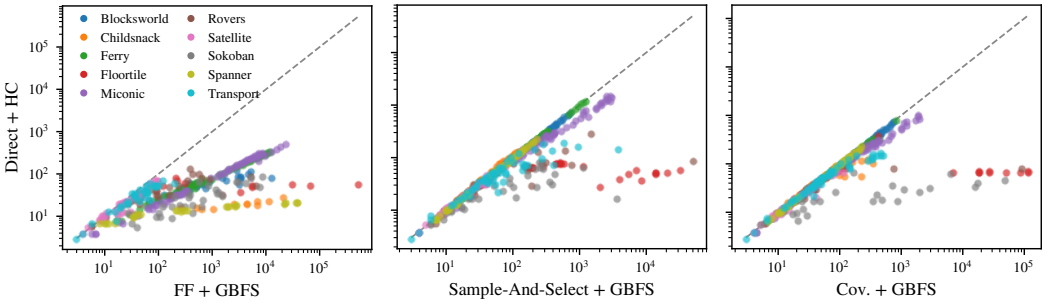


Figure 4: State expansions in the Gemini 3.1 Pro experiment, comparing direct + HC with (a) FF, (b) sample-and-select, and (c) Cov., each run with GBFS, on test tasks solved by both methods. Each point is one task, colored by domain. Points below the diagonal favor direct + HC. Note that restricting to tasks solved by both methods, in general, favors the method that solves fewer tasks.

training-task evaluations is 1,339.8 minutes per domain on average. Thus, the repair loop spends about 40× less time evaluating candidates on the training tasks.

Direct Heuristics vs. State of the Art. We compare the direct heuristics to the two baselines: the FF heuristic [36] and the sample-and-select method by Corrêa et al. [18]. Both baselines use GBFS, which may backtrack extensively before finding a solution. Our method uses HC, which follows a single greedy path and reaches the goal without backtracking. Table 1 reports results for the baselines under both GBFS and HC.

For Gemini 3.1 Pro, the main result is that our direct heuristics with HC solve 623.3 tasks by going directly to the goal, while sample-and-select with GBFS solves 573 tasks, and FF with GBFS solves only 298. That is, we solve more tasks than the best prior LLM method, and we do so without backtracking. Running the sample-and-select heuristics under HC solves only 430 tasks, confirming that the gain comes from the property our repair loop trains for, not merely from the choice of search algorithm. FF under HC collapses to 30 solved tasks. It is not a descending heuristic and offers no guarantee of progress at every state, so HC fails at the first plateau or local minimum it encounters.

Figures 4a and 4b make the efficiency gap concrete: on tasks solved by both methods, our heuristics with HC expand far fewer states than FF and sample-and-select, sometimes by several orders of magnitude. Together with $7.4\times$ fewer LLM calls and about $40\times$ less training-task evaluation time reported above, these results show that property-guided synthesis is generally better than sample-and-select: it solves more tasks, reaches the goal with fewer expansions, and costs far less to synthesize.

Generalization of the Direct Property. We validate the direct property only on small training tasks, and there is no guarantee that a heuristic that passes this check will remain direct on the much larger, out-of-distribution test tasks. At test time, hill climbing fails as soon as it reaches a state s with no successor s' satisfying $h(s') < h(s)$, which is exactly a violation of the direct property. Across the three independent Gemini 3.1 Pro runs over all 900 test tasks (2,700 evaluations in total), only a single evaluation stops for this reason (in Floortile). The remaining evaluations either find a plan or hit the time limit. The synthesized heuristics are therefore effectively direct on the out-of-distribution test set as well. This is especially striking in Sokoban, a **PSPACE**-complete domain [19], where no test evaluation across any of the three runs ever fails because of a property violation. The heuristics we generate from small training tasks therefore generalize robustly to much harder test tasks.

Ablation. To isolate the contribution of the direct property, we run the same repair loop but replace the property check with a numeric coverage score: the repair prompt reports which training task the heuristic failed to solve within the time limit and how many states were expanded before timing out, and asks for a new heuristic that solves more tasks with GBFS. We refer to this ablation as *Cov.* With Gemini 3.1 Pro, Cov. under GBFS solves 572.7 ($\pm$16.0) tasks, roughly matching sample-and-select (573). Under HC, however, Cov. degrades sharply to 412.3 tasks, well below our direct heuristics with HC (623.3, $\pm$6.7). Coverage feedback alone is therefore not sufficient to obtain heuristics that guide HC. Figure 4c reinforces this point: on tasks solved by both methods, direct heuristics consistently require fewer expansions.

Results with Other LLMs. Table 2 reports total test coverage with GPT-5.5 and Gemma 4 31B. With GPT-5.5, Direct with HC solves 594.7 tasks, compared with 550.7 for Cov. with GBFS and 532 for S&S with GBFS. With Gemma 4 31B, Cov. with GBFS solves 393.3 tasks, below the 463 solved by S&S with GBFS. Thus, the shorter repair loop guided by numeric coverage feedback does not preserve S&S coverage in this experiment. Direct with HC instead solves 494.7 tasks while generating 6.57 candidates per run on average, compared with the fixed budget of 25 used by S&S. Property-guided iterative repair therefore improves coverage with fewer candidates even when using the open-weight model.

Table 2: Total coverage on the 900 IPC 2023 test tasks with GPT-5.5 and Gemma 4 31B. The evaluation limits, methods, and reporting conventions are the same as in Table 1.

Model	S&S GBFS	Cov. GBFS	Direct HC
GPT-5.5	532	550.7	**594.7**
Gemma 4 31B	463	393.3	**494.7**

Results on Newly Created Domains. To test whether an LLM can synthesize direct heuristics for domains absent from its training data, we created Cisterns, Orrery, and Qubit Routing. These domains were not publicly available when Gemini 3.1 Pro was trained. Each domain has 99 training and 90 test tasks. In *Cisterns*, water is distributed unevenly across cisterns connected by siphons. Transfers are permitted only under level constraints, and the goal specifies the final levels of selected cisterns. In *Orrery*, coupled wheels each display one of K marks. An action advances one wheel while retreating another, so the total position is preserved, and the goal returns every wheel to its reference mark. In *Qubit Routing*, logical qubits occupy positions on a hardware connectivity graph. Swaps move them until the operands of each gate are adjacent, allowing the gates to be executed in order.

With Gemini 3.1 Pro, the repair loop converges in all nine synthesis runs, and no direct-property violation is observed across 810 hill-climbing test runs. FF with GBFS, S&S with GBFS, and Direct with HC solve 31, 27, and 26.7 tasks in Cisterns, 58, 90, and 90 in Orrery, and 61, 86, and 85.7 in Qubit Routing, respectively. Their total coverage is 150, 203, and 202.3 of 270 tasks, respectively. The repair loop generates 2.44 candidates per run on average, compared with 25 for S&S. On test tasks solved by S&S and all three Direct runs, the geometric mean of the three per-domain geometric mean expansion counts is about 112 for S&S with GBFS and 32 for Direct with HC. These results show that the model can synthesize effectively direct heuristics for domains it had not encountered

during training, with similar coverage and about $3.5\times$ fewer expanded states overall than S&S with GBFS.

Qualitative Analysis. Inspecting what the repair loop changes in the Gemini 3.1 Pro heuristics reveals why the direct property is a useful synthesis target. Two domains illustrate the outcomes.

Blocksworld. Blocksworld has a single arm that rearranges blocks into goal stacks by picking up, putting down, stacking, and unstacking one block at a time. The initial heuristic counts 2 actions per misplaced block (one to pick it up, one to put it down), subtracts 1 if a block is currently held, and adds 2 for each deadlock it detects. It uses two different deadlock conditions: for blocks on the table, the penalty activates only when the target block is itself misplaced; for a held block, the penalty activates whenever its target is not ready, meaning the target is either misplaced or covered by another block. Consider the state where b_1 is on b_2 and b_3 is on b_4, with the goal of swapping them: b_1 should go on b_4 and b_3 on b_2, while b_2 and b_4 are already correctly on the table. Both b_1 and b_3 are misplaced, giving $h = 4$. Since b_2 and b_4 are correctly on the table, the first condition does not activate and no penalty is added. After unstacking b_3, the arm holds b_3 and needs to place it on b_2, but b_1 still covers b_2, so b_2 is not ready: the second condition activates and adds $+2$, giving $h = 2 \times 2 - 1 + 2 = 5$. The same happens when unstacking b_1. The inconsistency is that the arm-empty state and the holding state use different conditions, so the mutual blocking between b_1 and b_3 is invisible in the arm-empty state but penalized in every successor, creating a local minimum. The final heuristic fixes this by building a dependency graph among misplaced blocks. It adds an edge from b_3 to b_1 (because b_3 covers b_4, the target of b_1) and from b_1 to b_3 (because b_1 covers b_2, the target of b_3), detects the cycle, and adds $+2$ in the arm-empty state itself, giving $h = 6$. The successors then give $h = 5$, restoring descent. The final heuristic is effectively direct on all test tasks.

Satellite. Satellite requires a satellite to switch on an instrument, turn to one of its calibration targets, calibrate it, and then point to the image direction before taking the image. The initial heuristic estimates each missing image goal independently by adding costs for switching on an instrument, calibrating it, turning to the needed directions, and taking the image. This gives an estimate of goal distance, but it ignores the ordering constraints created by calibration and by the single power slot on each satellite. An action may temporarily move the satellite away from the image goal direction, and switching from one instrument to another can force recalibration. The final heuristic fixes this by reasoning separately about each satellite's remaining work as a short action sequence. For each satellite, it groups the images that still need to be taken, considers which instrument should achieve them, and counts the actions still needed to finish them in order: switching instruments, calibrating, turning, and taking the images. It no longer treats the remaining requirements as an unordered set of subgoals, but as a sequence whose order matters. Because of this, each improving move removes a real remaining step in that sequence, which is why the final heuristic becomes direct.

5 Related Work

Counterexample-Guided Inductive Synthesis. Counterexample-guided inductive synthesis (CEGIS) [62, 38, 1] alternates between a synthesizer that proposes candidate programs and a verifier that checks them against a formal specification. When a candidate fails, the verifier returns a concrete counterexample, and the synthesizer uses it to rule out or repair the candidate. Several works refine LLM outputs with external feedback: CRITIC [30] iteratively revises model outputs using LLM-generated critiques grounded in external tools, Self-Debugging [13] repairs generated code using model-generated explanations and, when available, unit-test results, and Lemur [70] uses LLMs to propose invariants that automated reasoners check to verify a given program, which itself remains unchanged. Closer to CEGIS, Orvalho et al. [47] and our work use an LLM to repair programs guided by counterexamples. We apply CEGIS-style counterexample feedback to the LLM-based synthesis of heuristic functions for planning.

LLM-Based Program Synthesis. Most LLM-based program synthesis evaluates candidates with numeric scores such as test-pass rates or the value of the produced solutions, and uses these scores to rank or select among many candidates [39, 42, 67, 12, 55, 45]. In general, these approaches can combine multiple scores and use specific prompt strategies to guide the LLM toward better candidates, but they do not provide concrete feedback on how to improve a candidate.

LLM-Based Heuristic Generation. Our closest precursor is Corrêa et al. [18], who sample a fixed budget of heuristic candidates with an LLM and select the one that solves the most training tasks.

Other recent work that generates heuristics with LLMs includes Tuisov et al. [67], who focus on a different planning formalism, and Chen et al. [12], who focus on generalized policies.

Descending Heuristics. The descending and dead-end avoiding (DDA) property was coined by Seipp et al. [59]. Helmert et al. [35] showed that verifying or synthesizing DDA potential heuristics [53] is **PSPACE**-complete for SAS^+ planning tasks, with structural relaxations dropping the complexity to lower levels of the polynomial hierarchy. Dold and Helmert [20] introduce the direct (orig. "wet DDA") variant, which only requires descent along states reachable from s_0 by strictly improving transitions. These properties are useful to estimate the hardness of planning tasks based on the simplest form of their DDA/direct heuristics [59, 20].

Learning Search Control. Francès et al. [25] use a mixed-integer program to learn generalized potential heuristics, i.e., weighted sums over description-logic features [5], that are DDA on the states of small training tasks, and refine them on states where the property fails. They also prove by hand that such heuristics exist for several domains. Unlike our Python programs, these heuristics are restricted to a fixed feature class. Several lines of work synthesize or learn search control mechanisms for planning domains [9, 21, 22]. There is also a long history of approaches that learn heuristics for planning [58, 14, 57, 4]. These include methods that learn for specific tasks [24, 49, 6] and methods that learn for entire domains [61, 63, 11, 32].

LLMs for Planning. In addition, there is a large body of recent work that uses LLMs for planning in several settings such as end-to-end plan generation [7, 64, 37], PDDL domain or task construction [66, 31, 48, 41], and generalized planning [56, 44].

6 Limitations

Our method depends on the existence of a property that can be efficiently checked and turned into actionable feedback. In planning, the direct property provides exactly this, but a direct heuristic exists only for solvable tasks, so all training tasks must be solvable. For problems where no such informative property is available, or where the training set is too small to expose meaningful counterexamples, the repair loop loses its main advantage and degenerates to score-based ranking.

A second limitation concerns the scope of verification. We check the direct property only on a small training set under bounded resources, so passing this check does not certify that the heuristic is direct on unseen tasks. Our Gemini 3.1 Pro experiments show that the property typically generalizes well to much larger out-of-distribution tasks, but counterexamples on test tasks remain possible in principle.

A third limitation is that we synthesize unconstrained Python code. This flexibility is useful, but it makes the resulting heuristics harder to interpret and verify than structured representations such as fixed feature sets or linear potentials. A promising direction for future work is to combine our repair loop with strategies that automatically prove the synthesized heuristic to be direct on an entire domain, as has been done in restricted settings [2], possibly using theorem provers powered by LLMs.

7 Conclusions

In this paper, we studied property-guided LLM program synthesis for planning and showed that a counterexample-driven repair loop can synthesize direct heuristic functions efficiently. Across the ten IPC 2023 Learning Track domains, and compared to the previous best heuristic generation method, our method with Gemini 3.1 Pro requires about $7.4\times$ fewer LLM calls for heuristic generation, spends about $40\times$ less time evaluating candidates on training tasks, and solves more out-of-distribution test tasks by hill climbing directly to the goal. The coverage advantage also holds with GPT-5.5 and Gemma 4 31B, showing that iterative repair does not require a frontier model. On three newly created domains that were not publicly available when Gemini 3.1 Pro was trained, the repair loop converges in all nine synthesis runs and matches the coverage of sample-and-select with far fewer candidates, and hill climbing never encounters a direct-property violation in 810 test runs. The direct property is effective both as a synthesis target and as a feedback mechanism: when violated, it provides concrete counterexamples that guide repair, and when satisfied on the training tasks, it often generalizes to much larger unseen tasks. Our results show that, for heuristic functions, property-guided LLM repair matches or outperforms score-based candidate ranking at a fraction of its synthesis and evaluation cost.

Acknowledgments

This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. André G. Pereira acknowledges support from FAPERGS with 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] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*, 2018.

[2] Mohammad Abdulaziz, Florian Pommerening, and Augusto B. Corrêa. Mechanically proving guarantees of generalized heuristics: First results and ongoing work. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, 2022.

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

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

[5] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler, editors. *An Introduction to Description Logic*. Cambridge University Press, 2017.

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

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

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

[9] Blai Bonet, Guillem Francès, and Héctor Geffner. Learning features and abstract actions for computing generalized plans. In *Proc. AAAI 2019*, pages 2703–2710, 2019.

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

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

[12] Dillon Ze Chen, Johannes Zenn, Tristan Cinquin, and Sheila A. McIlraith. Language models for generalised PDDL planning: Synthesising sound and programmatic policies. In *ICAPS Workshop on Planning in the Era of LLMs*, 2025.

[13] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.

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

[15] Augusto B. Corrêa and Giuseppe De Giacomo. Lifted planning: Recent advances in planning using first-order representations. In *Proc. IJCAI 2024*, pages 8010–8019, 2024.

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

[17] Augusto B. Corrêa, André G. Pereira, and Jendrik Seipp. Frontier large language models rival state-of-the-art planners. arXiv:2511.09378 [cs.AI], 2025.

[18] Augusto B. Corrêa, André G. Pereira, and Jendrik Seipp. Classical planning with LLM-generated heuristics: Challenging the state of the art with Python code. In *Proc. NeurIPS 2025*, pages 42070–42108, 2025.

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

[20] Simon Dold and Malte Helmert. Novelty vs. potential heuristics: A comparison of hardness measures for satisficing planning. In *Proc. AAAI 2024*, pages 20692–20699, 2024.

[21] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Learning sketches for decomposing planning problems into subproblems of bounded width. In *Proc. ICAPS 2022*, pages 62–70, 2022.

[22] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Expressing and exploiting subgoal structure in classical planning using sketches. *Journal of Artificial Intelligence Research*, 80: 171–208, 2024.

[23] Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):75–88, 1995.

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

[25] Guillem Francès, Augusto B. Corrêa, Cedric Geissmann, and Florian Pommerening. Generalized potential heuristics for classical planning. In *Proc. IJCAI 2019*, pages 5554–5561, 2019.

[26] Gemini Team Google. Gemini 3.1 Pro model card. Google DeepMind, 2026.

[27] Gemma Team. Gemma 4 technical report. `https://arxiv.org/abs/2607.02770`, 2026.

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

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

[30] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations (ICLR)*, 2024.

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

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

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

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

[35] Malte Helmert, Silvan Sievers, Alexander Rovner, and Augusto B. Corrêa. On the complexity of heuristic synthesis for satisficing classical planning: Potential heuristics and beyond. In *Proc. ICAPS 2022*, pages 124–133, 2022.

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

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

[38] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ACM/IEEE International Conference on Software Engineering*, 2010.

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

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

[41] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering large language models with optimal planning proficiency. arXiv:2304.11477, 2023.

[42] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proc. ICML 2024*, pages 32201–32223, 2024.

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

[44] Andrew Murray, Danial Dervovic, Alberto Pozanco, and Michael Cashmore. GenePlan: Evolving better generalized PDDL plans using large language models. In *Proc. ICAPS 2026*, pages 667–676, 2026.

[45] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131, 2025.

[46] OpenAI. GPT-5.5 system card. `https://openai.com/index/gpt-5-5-system-card/`, 2026.

[47] Pedro Orvalho, Mikolás Janota, and Vasco M. Manquinho. Counterexample guided program repair using zero-shot learning and MaxSAT-based fault localization. In *AAAI Conference on Artificial Intelligence*, 2025.

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

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

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

[51] André G. Pereira, Augusto B. Corrêa, and Jendrik Seipp. Code and experiment data from the NeurIPS 2026 paper "Property-Guided LLM Program Synthesis for Planning". `https://doi.org/10.5281/zenodo.24012455`, 2026.

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

[53] Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From non-negative to general operator cost partitioning. In *Proc. AAAI 2015*, pages 3335–3341, 2015.

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

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

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

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

[58] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[59] Jendrik Seipp, Florian Pommerening, Gabriele Röger, and Malte Helmert. Correlation complexity of classical planning domains. In *Proc. IJCAI 2016*, pages 3242–3250, 2016.

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

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

[62] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In Vivek Sarkar and Mary W. Hall, editors, *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

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

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

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

[66] Marcus Tantakoun, Christian Muise, and Xiaodan Zhu. LLMs as planning formalizers: A survey for leveraging large language models to construct automated planning models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 25167–25188. Association for Computational Linguistics, 2025.

[67] Alexander Tuisov, Yonatan Vernik, and Alexander Shleyfman. Successor-generator planning with LLM-generated heuristics. In *Proc. ICAPS 2026*, pages 332–341, 2026.

[68] Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. PlanBench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In *Proc. NeurIPS 2023*, pages 38975–38987, 2023.

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

[70] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *International Conference on Learning Representations (ICLR)*, 2024.

A Benchmark Domains

Our main benchmark uses the ten domains of the Learning Track of the IPC 2023 [65]. For each domain, synthesis and validation use small training tasks, which define the in-distribution setting, while final evaluation uses 90 held-out competition test tasks. The test set evaluates out-of-distribution generalization to generally larger tasks with different structural configurations. Figure 3 shows the ranges of the main size parameters. Below, we describe each IPC domain based on the PDDL definitions distributed by the competition and explain the structural differences between its training and test tasks. The three newly created domains are described in the main text.

Blocksworld. A single arm rearranges distinguishable blocks into a goal configuration of stacks. The arm holds at most one block at a time and supports four actions: pick a clear block up from the table, put a held block down on the table, stack a held block onto another clear block, and unstack a clear block from the block below it. Goals specify the desired stacking via `on` and `on-table` atoms. Test tasks have different initial and goal tower structures from training tasks, changing the depth and interaction of support dependencies.

Childsnack. An agent must prepare sandwiches in a kitchen and serve them to children seated at tables. Each child is either gluten-allergic or not; gluten-free sandwiches require both gluten-free bread and gluten-free content, both of which are scarce. Sandwiches are placed on trays that move between the kitchen and the tables. The goal is to have every child served a sandwich compatible with their allergy. Compared with training tasks, test tasks have different ratios of allergic children to gluten-free ingredients and different tray and table distributions, changing how scarce resources must be allocated.

Ferry. A single ferry transports cars between locations on a sea. The ferry carries at most one car at a time and alternates between sailing (empty or loaded), boarding, and disembarking. The goal specifies the target location of each car. Test tasks differ from training tasks in the distributions of car origins and destinations across locations, changing the sequence of ferry trips required.

Floortile. Robots paint tiles on a rectangular grid in two colors. Each robot can move onto an adjacent unpainted tile, change the color it currently holds, or paint the tile immediately above or below itself. A robot cannot step onto a painted tile, so the planner must reason about painting order and about positioning robots so they do not block each other. Test tasks have different grid dimensions, aspect ratios, and robot placements from training tasks, changing painting order and robot interference.

Miconic. A single elevator picks up passengers at their origin floors and delivers them to their destination floors. The four actions are board, depart, drive up, and drive down; passenger capacity is unbounded. The goal is to have every passenger served at their destination. Test tasks differ from training tasks in the overlaps and directions of passenger trips across floors, changing which passengers can be served together.

Rovers. A team of rovers performs scientific operations on a network of waypoints and communicates results to a lander. Operations include sampling soil and rock at the current waypoint, calibrating an on-board camera against a calibration target, taking images of objectives, and communicating data via line of sight to the lander. Rovers differ in what they can traverse and which instruments they carry, so plans must allocate tasks across heterogeneous agents. Compared with training tasks, test tasks have different waypoint graphs, traversability relations, visibility relations, equipment assignments, and communication links, changing how scientific tasks must be allocated among rovers.

Satellite. A fleet of satellites takes calibrated images of celestial directions in particular imaging modes. Before an image can be acquired, the relevant instrument must be switched on, calibrated against a calibration target, and the satellite must be pointing at the target direction. Each satellite has a single power slot, so instruments must be switched on one at a time. Test tasks have different instrument-to-satellite and instrument-to-mode capability graphs and calibration dependencies from training tasks, changing which satellites and instruments can achieve each image goal.

Sokoban. An agent on a grid pushes boxes onto designated goal cells. The agent can move into a clear adjacent cell or, if a box sits in the adjacent cell, push it one cell further in the same direction, provided the destination cell is clear. Boxes cannot be pulled, so a box pushed against a wall or into a corner in the wrong orientation creates an unrecoverable dead end. Compared with training tasks, test

tasks use different and substantially larger mazes, with unseen wall layouts, corridors, bottlenecks, dead ends, and box-to-goal configurations.

Spanner. A man walks along a directed chain of locations to a workshop, where he must tighten a number of loose nuts. Spanners are scattered along the way; each spanner can be used to tighten exactly one nut and the chain cannot be retraversed. The man must therefore pick up enough usable spanners before reaching the workshop. Test tasks differ from training tasks in corridor lengths, tool locations, and ratios of spanners to nuts, changing how many mistakes can be recovered from before reaching the workshop.

Transport. Trucks with bounded discrete capacity move packages along a road network. Pick-up and drop actions consume and free one unit of capacity, encoded as a chain of size objects related by a predecessor predicate. The plan must coordinate truck routes and remaining capacity to deliver every package to its goal location. Compared with training tasks, test tasks have different road graphs, package routes, vehicle placements, and capacity distributions, changing the routing and allocation decisions required to deliver all packages.

B Full Coverage Results with Standard Deviations

Table 3 reports the same per-domain coverage numbers as Table 1 in the main text, but additionally includes standard deviations across the three independent runs for the **Cov.** ablation (GBFS and HC) and the **Direct** proposed method (HC). The baseline columns (**FF** and **S&S** under GBFS and HC) are reproduced without standard deviations because FF is deterministic and S&S selects a single heuristic per domain.

Table 3: Per-domain coverage (number of solved tasks) on the 900 IPC 2023 test tasks in the Gemini 3.1 Pro experiment. This is the full version of Table 1 in the main text, including standard deviations across three independent runs for the ablation and proposed method. **FF** is the classical delete-relaxation heuristic; **S&S** is our re-run of the sample-and-select baseline [18] with Gemini 3.1 Pro. Both baselines are run using GBFS and HC. **Direct** is our proposed method (direct heuristics run with HC). **Cov.** is the ablation that replaces the direct-property check with numeric coverage feedback, using both GBFS and HC. Bold marks the best value per domain (ties are co-bolded).

	Baseline				**Ablation**		**Our Method**
	FF		**S&S**		**Cov.**		**Direct**
Domain	**GBFS**	**HC**	**GBFS**	**HC**	**GBFS**	**HC**	**HC**
Blocksworld (90)	23	0	84	**90**	81.00 ± 2.00	49.67 ± 3.79	77.3 ± 11.5
Childsnack (90)	12	0	58	70	58.00 ± 0.00	46.67 ± 40.41	**71.7** ± 0.6
Ferry (90)	45	0	67	71	64.67 ± 4.04	49.33 ± 22.48	**72.3** ± 1.2
Floortile (90)	11	0	30	0	39.00 ± 11.36	15.67 ± 16.01	**40.7** ± 9.0
Miconic (90)	61	1	83	7	82.00 ± 8.00	34.00 ± 48.59	**90.0** ± 0.0
Rovers (90)	29	2	**44**	12	36.67 ± 5.69	20.67 ± 14.15	42.7 ± 13.6
Satellite (90)	35	17	57	**60**	56.00 ± 1.73	57.33 ± 5.51	55.7 ± 14.5
Sokoban (90)	29	1	**30**	0	26.67 ± 1.15	12.67 ± 6.43	**30.0** ± 0.0
Spanner (90)	30	6	66	**79**	66.00 ± 0.00	**79.00** ± 0.00	75.0 ± 0.0
Transport (90)	23	3	54	41	62.67 ± 4.04	47.33 ± 28.54	**68.0** ± 0.0
Total (900)	298	30	573	430	572.67 ± 15.95	412.33 ± 41.93	**623.3** ± 6.7

C Synthesis Statistics

Table 4 summarizes how many candidate heuristics the repair loop generates before it terminates. We report per-domain statistics over the three independent runs: the minimum, mean (with standard deviation), and maximum number of candidates generated in a single run. We also report the separately measured mean synthesis wall time, including LLM latency. The loop produces a heuristic accepted by the training-set check in 28 of the 30 runs. Two Sokoban runs exhaust the 11-candidate budget.

Table 4: Number of candidate heuristics generated by the repair loop with Gemini 3.1 Pro per synthesis run, summarized across the three independent runs per domain. n is the number of runs. Min, Mean $\pm$ SD, and Max refer to candidate counts. Time is the separately measured mean synthesis wall time in minutes, including LLM latency.

Domain	n	**Min**	**Mean $\pm$ SD**	**Max**	**Time (min)**
Blocksworld	3	1	2.33 ± 2.31	5	29
Childsnack	3	1	2.00 ± 1.00	3	48
Ferry	3	1	2.00 ± 1.00	3	27
Floortile	3	7	8.33 ± 1.15	9	91
Miconic	3	1	1.67 ± 0.58	2	20
Rovers	3	2	2.33 ± 0.58	3	44
Satellite	3	2	2.33 ± 0.58	3	42
Sokoban	3	5	9.00 ± 3.46	11	108
Spanner	3	1	1.33 ± 0.58	2	13
Transport	3	2	2.33 ± 0.58	3	50
Overall	30	1	3.37 ± 2.99	11	47

D Example Repair Prompt

Below we show the repair prompt used in the repair loop, instantiated for the Blocksworld domain at iteration 3. We edit the prompt to highlight its main structural components and match the nomenclature used in the paper.

```
<role>
You are a highly-skilled AI planning researcher and a proficient Python programmer
    creating a domain-dependent heuristic function for the PDDL domain blocksworld.
</role>

<objective>
Produce a revised heuristic where every non-goal state reachable by Depth-First Search,
    which only expands strictly improving successors, has at least one improving
    successor, regardless of the order in which operators are applied.
</objective>

<evaluation-method>
The heuristic is validated using a Depth-First Search (DFS) that enforces the direct
    property at every step:
- Only successors with h(s') < h(s) are added to the DFS stack.
- A property violation occurs when DFS expands a non-goal state and finds no successor
    with a strictly lower heuristic value.
...
</evaluation-method>

<success-criteria>
1. Alive State: reachable from the initial state, solvable, and not a goal state.
2. Improving Successor: a successor s' such that h(s') < h(s).
3. Direct: every alive state reachable from the initial state via a sequence of strictly
    decreasing h values has at least one improving successor. All states that DFS can
    reach must contain an improving successor.
4. Dead-End Avoiding: no improving successor of an alive state reachable by DFS is itself
     unsolvable (h = infinity).
</success-criteria>

<previous-iterations>
Iteration 0:
- Prompt Type: Init
- Solved 5 instances (Average expanded nodes: 2.4)
- Direct validated on 5 tasks (last index: 4)
- Failure Analysis:
  Failure Type: property
  Failing Task: p06.pddl
  Property Failure Kind: direct
  State: ['(clear b1)', '(clear b2)', '(holding b3)', '(on-table b1)', '(on-table b2)']
  Heuristic Value: 3
  Reason: State has h=3 but no successor has h < 3.
  Successors:
    1. action=(stack b3 b2), h=6, added=['(clear b3)', '(on b3 b2)', '(arm-empty)'],
        deleted=['(holding b3)', '(clear b2)']
```

```
    2. action=(stack b3 b1), h=6, added=['(clear b3)', '(on b3 b1)', '(arm-empty)'],
         deleted=['(holding b3)', '(clear b1)']
    3. action=(putdown b3), h=4, added=['(clear b3)', '(on-table b3)', '(arm-empty)'],
         deleted=['(holding b3)']
<previous-heuristic-code>
...
</previous-heuristic-code>

Iteration 1:
- Prompt Type: Property
- Main Idea: ...
- Solved 16 instances (Average expanded nodes: 7.0)
- Direct validated on 16 tasks (last index: 15)
- Failure Analysis:
  Failure Type: property
  Failing Task: p17.pddl
  Property Failure Kind: direct
  State: ['(arm-empty)', '(clear b1)', '(clear b3)', '(clear b4)', '(on b2 b5)', '(on b3
       b2)', '(on-table b1)', '(on-table b4)', '(on-table b5)']
  Heuristic Value: 10
  Reason: State has h=10 but no successor has h < 10.
  Successors:
    1. action=(unstack b3 b2), h=11, added=['(holding b3)', '(clear b2)'], deleted=['(
         clear b3)', '(on b3 b2)', '(arm-empty)']
    2. action=(pickup b4), h=11, added=['(holding b4)'], deleted=['(on-table b4)', '(
         clear b4)', '(arm-empty)']
    3. action=(pickup b1), h=11, added=['(holding b1)'], deleted=['(clear b1)', '(on-
         table b1)', '(arm-empty)']
<previous-heuristic-code>
...
</previous-heuristic-code>

Iteration 2:
- Prompt Type: Property
- Main Idea: ...
- Solved 17 instances (Average expanded nodes: 7.8)
- Direct validated on 17 tasks (last index: 16)
- Failure Analysis:
  Failure Type: property
  Failing Task: p18.pddl
  Property Failure Kind: direct
  State: ['(arm-empty)', '(clear b1)', '(clear b3)', '(clear b5)', '(on b2 b4)', '(on b5
       b2)', '(on-table b1)', '(on-table b3)', '(on-table b4)']
  Heuristic Value: 8
  Reason: State has h=8 but no successor has h < 8.
  Successors:
    1. action=(unstack b5 b2), h=9, added=['(holding b5)', '(clear b2)'], deleted=['(on
         b5 b2)', '(clear b5)', '(arm-empty)']
    2. action=(pickup b1), h=9, added=['(holding b1)'], deleted=['(clear b1)', '(on-table
          b1)', '(arm-empty)']
    3. action=(pickup b3), h=9, added=['(holding b3)'], deleted=['(clear b3)', '(on-table
          b3)', '(arm-empty)']
<previous-heuristic-code>
...
</previous-heuristic-code>
</previous-iterations>

<current-status>
Domain Progress: solving 99 instances of increasing difficulty.
Validation Time Limit: 30 seconds per task.
Current Result: The heuristic was direct on 73 instances, but failed on p74.pddl.
</current-status>

<failure-analysis>
Failure Type: property
Property Failure Kind: direct
State: ['(arm-empty)', '(clear b10)', '(clear b11)', '(clear b15)', '(clear b17)', '(
     clear b2)', '(clear b21)', '(clear b22)', '(clear b3)', '(clear b4)', '(clear b5)',
     '(clear b6)', '(clear b8)', '(clear b9)', '(on b1 b14)', '(on b12 b18)', '(on b14
     b13)', '(on b15 b12)', '(on b18 b7)', '(on b21 b1)', '(on b6 b20)', '(on b7 b19)',
     '(on b9 b16)', '(on-table b10)', '(on-table b11)', '(on-table b13)', '(on-table b16)
     ', '(on-table b17)', '(on-table b19)', '(on-table b2)', '(on-table b20)', '(on-table
      b22)', '(on-table b3)', '(on-table b4)', '(on-table b5)', '(on-table b8)']
Heuristic Value: 38
Reason: State has h=38 but no successor has h < 38.
Successor Analysis:
1. action=(unstack b15 b12), h=39, added=['(holding b15)', '(clear b12)'], deleted=['(
     clear b15)', '(on b15 b12)', '(arm-empty)']
2. action=(unstack b9 b16), h=39, added=['(holding b9)', '(clear b16)'], deleted=['(on b9
      b16)', '(clear b9)', '(arm-empty)']
```

```
3. action=(pickup b11), h=39, added=['(holding b11)'], deleted=['(on-table b11)', '(clear
     b11)', '(arm-empty)']
4. action=(pickup b8), h=39, added=['(holding b8)'], deleted=['(arm-empty)', '(clear b8)
    ', '(on-table b8)']
5. action=(pickup b3), h=39, added=['(holding b3)'], deleted=['(clear b3)', '(on-table b3
    )', '(arm-empty)']
6. action=(pickup b4), h=39, added=['(holding b4)'], deleted=['(on-table b4)', '(clear b4
    )', '(arm-empty)']
7. action=(pickup b17), h=39, added=['(holding b17)'], deleted=['(clear b17)', '(on-table
     b17)', '(arm-empty)']
8. action=(unstack b6 b20), h=39, added=['(holding b6)', '(clear b20)'], deleted=['(clear
     b6)', '(on b6 b20)', '(arm-empty)']
9. action=(pickup b10), h=39, added=['(holding b10)'], deleted=['(on-table b10)', '(clear
     b10)', '(arm-empty)']
10. action=(pickup b22), h=39, added=['(holding b22)'], deleted=['(on-table b22)', '(
    clear b22)', '(arm-empty)']
11. action=(pickup b5), h=39, added=['(holding b5)'], deleted=['(on-table b5)', '(clear
    b5)', '(arm-empty)']
12. action=(unstack b21 b1), h=39, added=['(clear b1)', '(holding b21)'], deleted=['(
    clear b21)', '(on b21 b1)', '(arm-empty)']
13. action=(pickup b2), h=39, added=['(holding b2)'], deleted=['(clear b2)', '(on-table
    b2)', '(arm-empty)']
</failure-analysis>

This is the current Python code of the heuristic function:
<heuristic-code-file>
...
</heuristic-code-file>

This is the PDDL domain file of the blocksworld domain:
<domain-file>
...
</domain-file>

This is the first PDDL instance file of the blocksworld domain where the heuristic is
    failing the direct property:
<task-file>
...
</task-file>

Here is a checklist to help you with your code:
<checklist>
  1) All used modules are imported.
  2) The information from static facts is extracted into suitable data structures in the
       constructor.
  3) Use the failure analysis and iteration history to preserve what worked and fix what
       failed.
  4) The name of the heuristic should be BlocksworldHeuristic.
</checklist>

<final-instruction>
  Return a new heuristic function code that satisfies the direct property in the instance
       in <task-file> as well.

  Output your answer in the following XML format:
  <generated-main-idea>
    A short paragraph describing only the idea of the new heuristic you are returning. Do
         not include any other information.
  </generated-main-idea>
  <generated-heuristic-code>
    Provide the Python code of the domain-dependent heuristic for the blocksworld.
  </generated-heuristic-code>
</final-instruction>
```

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 introduction state the three concrete claims we make: roughly $7\times$ fewer LLM calls than the state of the art, about $40\times$ less training-task evaluation time, and more solved tasks via hill climbing. All three are backed up by the experiments in Section 4.

   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: There is a dedicated Limitations section that covers (i) the dependence on having a checkable property, solvable training tasks, and a useful training set, (ii) the fact that we only verify the property on training tasks under bounded resources, and (iii) the trade-off of synthesizing unconstrained Python rather than a structured representation.

   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: We do not prove new theorems. The paper restates one known proposition with a proof sketch and refers to the literature for all other formal results.

   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: The experimental setting paragraph names the planner (Pyperplan with PyPy 7.3.9), the LLMs (Gemini 3.1 Pro and GPT-5.5 with default API parameters, and Gemma 4 31B with default inference parameters), the benchmark (IPC 2023 Learning Track), the iteration budget, and the per-task time and memory limits. Code, prompts, generated heuristics, and experimental data 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: Our code, prompts, generated heuristics, and experimental data, together with instructions for reproducing the training and testing runs, are publicly available.

   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: The experimental setting paragraph in Section 4 gives the train/test split, the iteration and time budgets used during repair, the search and time limits used at test time, and the API and inference settings for the LLMs. We do not tune any hyperparameters.

   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: [Yes]

   Justification: We run our method (and the coverage ablation) three times per domain with independent LLM samples and report the mean. For the main Gemini 3.1 Pro experiment, the full appendix table gives the standard deviation across the three runs for every per-domain coverage number.

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: All planner runs ran on AMD EPYC 7742 CPUs at 2.25 GHz with 8 GiB memory per task, with a 30 s validation limit during training and a 5 min limit at test time. Section 4 also reports the total evaluation cost per domain, both for our method and for the sample-and-select baseline.

   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: We have read the Code of Ethics and our research adheres to it. The work uses only synthetic planning benchmarks and has no human subjects.

   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 is foundational work on synthesizing heuristic functions for classical planning benchmarks. We do not see a direct path from it to either positive or negative societal applications.

    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: We do not release any pretrained models, generative datasets, or scraped data. The only released artifacts are domain-specific Python heuristic functions for classical planning, which carry no misuse risk.

    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: We cite Pyperplan, Downward Lab, the IPC 2023 Learning Track benchmarks, and the Gemini 3.1 Pro, GPT-5.5, and Gemma 4 31B models at their points of first use. All are used within their respective licenses and terms of use (Pyperplan and Downward Lab are GPL; the IPC benchmarks are publicly distributed by the competition).

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: Our code, prompts, and generated heuristics, together with a README that explains how to run training and testing, are publicly available.

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: No crowdsourcing or human-subjects research is involved in this work.

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 does not involve crowdsourcing nor research with human subjects.

    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: An LLM is the core synthesizer in our method: we use Gemini 3.1 Pro, GPT-5.5, and Gemma 4 31B to generate the candidate heuristic functions, with the prompt and counterexample-driven repair loop described in Section 3.

    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.