LEANPLAN: OPTIMAL PLANNING WITH LLM-GENERATED HEURISTICS AND ADMISSIBILITY PROOFS

Anonymous authors
Paper under double-blind review

ABSTRACT

Frontier large language models (LLMs) can generate *heuristic functions* that guide search to achieve state-of-the-art performance in *satisficing* planning, where any plan is acceptable. However, these heuristics are not guaranteed to be *admissible* and can lead to suboptimal plans. We introduce LeanPlan, the first planning system that finds optimal plans with LLM-generated heuristics whose admissibility is machine-checked. Given a domain description and training tasks, an agentic loop uses planner feedback to iteratively improve a reusable domain-specific heuristic, its admissibility proof and the required domain assumptions. LeanPlan implements the heuristic, its proof and an efficient planner with machine-checked grounding and search in Lean 4. We evaluate LeanPlan on ten domains from the International Planning Competition and three new domains, using test tasks with up to 57 times as many objects as the training tasks. With GPT 5.6 Sol in the agentic loop, we successfully generate heuristics and admissibility proofs for all these domains. The resulting heuristics are generally more informed than the strong admissible heuristics of the state-of-the-art Scorpion planner, and with them LeanPlan solves more tasks overall than Scorpion.

1 INTRODUCTION

Classical planning asks for a sequence of actions that transforms an initial state into a state that satisfies a goal (Ghallab et al., 2004). Planning tasks come in *domains*, families of tasks with the same kinds of objects and actions, such as stacking blocks or moving a lift between floors. Domains and tasks are usually described in the Planning Domain Definition Language (PDDL) (Haslum et al., 2019). Heuristic search is one of the most successful approaches to classical planning (Bonet & Geffner, 2001; Hoffmann & Nebel, 2001). A *heuristic* estimates the cost of reaching a goal from a state, and the search uses these estimates to decide which states to explore next (Pearl, 1984). Domain-independent heuristics derive their estimates from the task description alone and therefore apply to any task. Domain-specific heuristics exploit knowledge about a particular domain and can be much more informative (Junghanns & Schaeffer, 2001), but a human expert usually has to design and implement each of them.

Large language models (LLMs) can now automate the design of such heuristics. Corrêa et al. (2025) generate heuristic programs once per domain and select them on training tasks. With greedy best-first search, their strongest heuristics solve more unseen tasks in total than strong domain-independent heuristics. Other approaches repair candidate heuristics with counterexamples to a property checked on the training tasks (Pereira et al., 2026), generate heuristics for tasks given by successor generators and goal tests (Tuisov et al., 2026) or evolve domain-independent heuristic programs (Gestrin & Seipp, 2026). LLMs can also generate generalized plans, i.e., programs that produce plans for new tasks of a domain (Silver et al., 2024; Stein et al., 2026; Murray et al., 2026).

However, these approaches focus on *satisficing planning*, where any valid plan suffices. In *optimal planning*, we require a plan of minimum cost. A* search (Hart et al., 1968) provides this guarantee when its heuristic is *admissible*, meaning that it never overestimates the remaining plan cost (Dechter & Pearl, 1985). Admissibility is hard to establish, because it must hold in every state of every task of the domain, including tasks much larger than the training tasks. Neither testing on example tasks nor learning heuristic values from solved examples (Chen et al., 2024b) can establish this property. Futuhi & Sturtevant (2026) study near-admissible neural heuristics with statistical generalization bounds,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

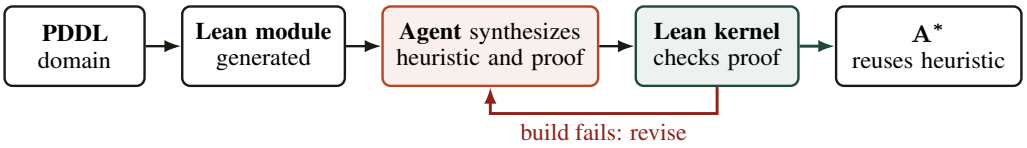


Figure 1: Proof-guided heuristic development and reuse in LeanPlan. A deterministic generator translates the domain into a Lean module, and an agent revises a heuristic and its proof using feedback from Lean. At planning time, task checks enforce the proof assumptions.

but these heuristics can still overestimate, so A* may return suboptimal plans. Phung et al. (2026) instead guarantee admissibility by construction: they evolve programs that choose task abstractions, from which pattern databases and cost partitioning derive admissible heuristics.

In this paper, we study a complementary approach: generating the heuristic program together with a proof of its admissibility. We introduce *LeanPlan*, the first planning system that finds optimal plans with LLM-generated heuristics whose admissibility is machine-checked. LeanPlan implements each domain heuristic and its proof in Lean 4 (de Moura & Ullrich, 2021), a programming language and proof assistant, so that Lean's kernel checks the proof. The heuristic can be any Lean program, for example one that combines domain-specific action counts, distance bounds and dead-end tests, as long as its admissibility can be proved. Because the proof is about the code that the planner actually runs, there is no separate implementation that could deviate from the verified one. Shared proofs, which apply to all domains, turn each admissibility proof into an optimality guarantee for the plans that the planner returns. Unlike a statistical generalization bound, the proof covers every task of the domain that satisfies its checked assumptions, however large, and not only the training tasks.

Figure 1 shows how LeanPlan develops and uses a heuristic. A deterministic generator first translates the PDDL domain into a Lean module, which fixes the vocabulary available to the heuristic and its proof. Given this module, the domain's training tasks and a shared proof library, an LLM coding agent writes the heuristic, its admissibility proof and Boolean checks of the domain assumptions that the proof relies on. The agent builds the code and runs the planner itself, uses failures as feedback, and revises the artifacts until the build succeeds or it stops making progress. Whether the proof is correct is decided by Lean's kernel, a small trusted proof checker, and not by the agent's own report of success. An accepted heuristic then serves A* on new tasks of the domain, without further LLM calls or new proofs.

We evaluate LeanPlan on 900 test tasks from ten domains of the Learning Track of the 2023 International Planning Competition (IPC) (Taitler et al., 2024) and on 270 test tasks from three new domains. The new domains were not publicly available before our experiments, which rules out exposure through public LLM training data. In IPC 2023, the hardest test tasks have 2.5 to 57 times as many objects as the largest training task of their domain, compared with at most about twice as many in the new domains. With GPT 5.6 Sol as the coding agent, LeanPlan obtains an admissible heuristic with a machine-checked proof for all thirteen domains. With the generated heuristics, LeanPlan solves 281 IPC 2023 tasks and 67 new-domain tasks, 27.1% and 31.4% more than the state-of-the-art optimal planner Scorpion (Seipp, 2024), which solves 221 and 51 tasks. A planner with machine-checked optimality guarantees can thus outperform a state-of-the-art, domain-independent optimal planner that offers no such guarantees.

2 Background

Classical Planning. We consider classical planning, where actions are deterministic and states are fully observable (Ghallab et al., 2004). The Planning Domain Definition Language (PDDL) (Haslum et al., 2019) separates a reusable *domain* from an individual *task*, usually in two files. The domain declares object types, predicates over objects and *action schemas*, i.e., actions with object parameters. The task supplies the objects, the initial state and the goal. The same domain can therefore describe many tasks with different objects, initial states and goals. *Grounding* instantiates the predicates and action schemas with the task's objects to obtain *atoms* and *ground actions*. Together with the task's initial state and goal, they form a finite *ground task*, which the planner solves.

Ground tasks use the STRIPS representation (Fikes & Nilsson, 1971). It represents a *state* by the set of atoms that are true in it. Each ground action a has a cost $c(a)$ and three sets of atoms: its preconditions, add effects and delete effects. It is applicable in a state s if all its preconditions are true in s. Applying it first removes its delete effects and then adds its add effects, leaving all other atoms unchanged. We write $s \xrightarrow{a} s'$ for the resulting transition from s to its successor state s'. The goal is a set of atoms, and a *goal state* is a state in which all of them are true. A state is *reachable* if a sequence of transitions leads to it from the initial state.

A *plan* for a state s is a sequence of actions that can be applied one after the other, starting in s, and that leads to a goal state. Its cost is the sum of its action costs, and an *optimal* plan for s has minimum cost among all plans for s. We write $h^*(s)$ for the cost of an optimal plan for s. If s has no plan, it is a *dead end* and $h^*(s) = \infty$. Optimal planning asks for an optimal plan for the initial state. In our experiments, every action has cost one, so an optimal plan also has minimum length.

Heuristic Search. A *heuristic* h estimates h^*: it maps each state s to a nonnegative number or ∞. It is *admissible* if $h(s) \leq h^*(s)$ for every state s, so an admissible heuristic returns ∞ only for dead ends. A* search (Hart et al., 1968) starts from the initial state and repeatedly *expands* a state, i.e., generates its successors. It always expands a state that minimizes the cost of the cheapest known path to it plus its heuristic value, and it prunes states whose heuristic value is ∞. With an admissible heuristic, A* returns an optimal plan for the initial state whenever one exists (Dechter & Pearl, 1985).

Admissibility is a global property, since $h^*(s)$ depends on all plans for s. We therefore prove two local properties, each concerning a single state or transition, that together imply admissibility. A heuristic is *goal-aware* if $h(s) = 0$ for every goal state s, and *consistent* if $h(s) \leq c(a) + h(s')$ for every transition $s \xrightarrow{a} s'$. Along any plan, consistency lets the heuristic value decrease by at most the cost of each action, and goal awareness makes the final value zero. The value at the start of the plan is therefore at most the plan's cost, so a goal-aware and consistent heuristic is admissible. LeanPlan's domain proofs establish both properties only for reachable states. This suffices because every state on a plan for a reachable state is itself reachable, so the argument above yields admissibility on all reachable states.

Lean Programs and Proofs. Lean 4 combines a functional programming language with an interactive theorem prover (de Moura & Ullrich, 2021). Its type system expresses both executable functions and logical statements about them. For example, a function from natural numbers to natural numbers has type `Nat -> Nat`, while the proposition `forall n : Nat, n + 0 = n` has type `Prop`, the type of propositions. Because programs and propositions share one language, a theorem can refer directly to the definition of a total function (one that terminates on every input), including its arithmetic, case distinctions and recursive calls. A proof of a proposition is a *proof term* whose type is that proposition. Lean's *kernel*, a small checker for Lean's core logic, verifies that the term has the claimed type, using only the declared definitions and axioms.

Proof terms are usually constructed with *tactics* instead of being written directly. A *proof goal* consists of a proposition to establish and the local assumptions available for proving it. A tactic transforms a proof goal into zero or more simpler goals while building the corresponding proof term, for example by unfolding definitions, applying existing lemmas, splitting cases, performing induction or solving arithmetic subgoals. The proof is complete when no goals remain. The kernel then checks the resulting term independently of how it was constructed, so trusting a proof requires trusting only the kernel and the axioms.

A theorem holds for all values of its parameters that satisfy its assumptions. A theorem about all natural numbers thus covers infinitely many values, not only those in a finite test set. Likewise, LeanPlan's domain theorems cover every task that satisfies their assumptions (Section 3), not only the training tasks. Theorems are separate declarations that the compiled planner never executes, so the proofs add no runtime cost. Figure 2 in Section 4 shows a small example of a function, a theorem about it and a tactic proof.

3 LEANPLAN

LeanPlan develops a heuristic and its admissibility proof once per domain and reuses both for every new task of that domain (Figure 1). It accepts the unit-cost STRIPS fragment of PDDL described in

Section 2, with optional typing, type hierarchies and domain constants. Appendix F shows a Miconic domain and task in this fragment. For each domain, a deterministic generator translates the PDDL action schemas into a Lean module, which fixes the vocabulary available to the heuristic and its proof.

Domain Assumptions. A heuristic may rely on properties of a domain's tasks that PDDL does not enforce. For example, a Blocksworld heuristic may assume that at most one block is directly on top of each block. This property holds in every Blocksworld task of interest, but a syntactically valid task can violate it. Following Abdulaziz et al. (2022), we make such properties explicit as *domain assumptions*, which are Boolean checks on the task's objects, initial state and goal. The agent that develops the heuristic is free to choose the domain assumptions that its proof needs, as long as all training tasks satisfy them. We call the collection of a heuristic's domain assumptions its *certificate*. The certificate also checks that the input domain has exactly the action schemas of the generated Lean module, since the proof reasons about these schemas. LeanPlan evaluates the certificate before search and refuses to use the heuristic on a task that fails any of the certificate's checks.[1]

Proof Obligations. Let d and p be a domain and task that pass LeanPlan's validators, let T be their ground task and let h be the compiled heuristic that the search evaluates. Let R be the set of states reachable in T. Assuming that the certificate holds for p, each domain theorem proves

$$h(s) = 0 \qquad \text{for every goal state } s \in R, \tag{1}$$

$$h(s) \leq c(a) + h(s') \qquad \text{for every transition } s \xrightarrow{a} s' \text{ with } s \in R. \tag{2}$$

A shared theorem then derives admissibility on R with the argument from Section 2. Appendix A gives the Lean interface and its connections to grounding and search.

Generating Heuristics and Proofs. An LLM coding agent receives the generated Lean module, the domain's training tasks, the heuristic interface and the shared proof library. It develops a heuristic and proves that the heuristic satisfies Equations 1 and 2 whenever its certificate holds. The agent builds the code and proofs and runs the planner on the training tasks, using failures as feedback for revision. The prompt specifies:

- the objective: develop a domain-specific admissible heuristic with the best training performance the agent can achieve, continuing to refine it after beating a baseline planner;
- the interface and context of the planner;
- the correctness requirements: prove goal awareness and consistency for the compiled heuristic on reachable states under its certificate, and ensure that the certificate accepts every training task;
- the baseline: measured Scorpion results, its search configuration with h^2-preprocessing, and the same per-task wall-clock limit and pinned CPU core for both planners;
- a list of useful commands;
- the prohibited shortcuts (e.g., `sorry`, `admit`, new axioms; see next paragraph);
- the stopping rule: stop when the agent judges that it cannot improve the heuristic further, leaving its best candidate with a working proof and passing certificates.

Appendix B reproduces the full template from the agentic-loop implementation.

Accepting a Candidate. Since the agent's own report of success is not evidence, an automated controller re-checks the files the agent leaves behind. It rejects the candidate if these files declare a new axiom or use one of the Lean constructs `sorry`, `admit` and `native_decide`, which skip a proof or trust unverified code. It then rebuilds the generated Lean module, the proofs and the planner, and it checks that the domain theorems depend only on Lean's standard axioms and that the certificate accepts every training task. Finally, it runs the planner on the training tasks. The controller accepts the heuristic only if all checks pass and A* with it solves at least as many training tasks as Scorpion (Seipp, 2024), while expanding strictly fewer states in total on the tasks both solve. This comparison requires a nonempty shared set with complete expansion counts, even when LeanPlan solves more tasks.

[1]These checks revealed a bug in the IPC 2023 Learning Track training set: the initial state of Floortile task `p08.pddl` contains both `(robot-at robot2 tile_2_1)` and `(clear tile_2_1)`, so another robot can move onto the occupied tile. We reported the bug to the competition organizers.

```
def passengerBound (w r : Nat) : Nat := 2 * w + r

theorem boarding_step (w r : Nat) :
    passengerBound (w + 1) r =
      1 + passengerBound w (r + 1) := by
  unfold passengerBound
  omega
```

Figure 2: A Lean example of a function and a theorem about it with a tactic proof. For a lift with w waiting and r riding passengers, `passengerBound` is a lower bound on the number of remaining boarding and leaving actions, and `boarding_step` shows that boarding reduces this bound by exactly one. The `def` declaration gives the function's parameters, return type and body, and the `theorem` declaration states an equality about the function for all natural numbers `w` and `r`. The block after `by` constructs its proof: `unfold` replaces the function by its definition, and `omega` closes the resulting linear arithmetic goal.

Grounding. To solve a new task, LeanPlan parses and validates it, grounds it and, if the task passes the certificate, runs A* with the heuristic. Grounding is similar to the algorithm by Helmert (2009), which is used in popular planners such as Fast Downward (Helmert, 2006) and Scorpion (Seipp, 2024), the state-of-the-art optimal planner. It instantiates only actions whose static preconditions, which no action can change, hold in the initial state. LeanPlan then prunes actions that h^2 mutexes rule out (Alcázar & Torralba, 2015) and removes effects and actions that cannot influence the goal.

Search. LeanPlan's A* is a standard implementation with duplicate detection, reopening and goal tests at expansion time. Like Scorpion, it breaks ties between states with the same sum of path cost and heuristic value in favor of lower heuristic values (Asai & Fukunaga, 2017). Unlike Scorpion, it then prefers the most recently added state, which tends to be more efficient (Asai & Fukunaga, 2016).

Search Guarantees. Shared theorems extend the domain guarantees from the heuristic to the whole search. Every plan that A* returns is valid for the ground task, and with an admissible, goal-aware heuristic, it is optimal whenever it costs less than a large bound B, far above the cost of any plan in our experiments. Since heuristics in Lean return natural numbers, LeanPlan represents ∞ by B and prunes every state whose heuristic value reaches it. The proofs therefore certify optimal plans but not dead ends or unsolvable tasks, and Appendix A states the theorems and explains this limitation.

4 Domain Heuristics

The generated heuristics use domain structure to count actions that every solution must take. We illustrate this with three domains that show action counting, dead-end detection and distance-based relaxations, and Appendix C summarizes the remaining heuristics and representation details.

Miconic. In Miconic, a lift transports passengers to their destination floors. The heuristic considers only passengers whose `served` goal is unmet, and it lets $W(s)$ contain those still waiting and $R(s)$ those aboard. A waiting passenger must board the lift and leave it at the destination with the domain's `depart` action, while a passenger aboard needs only to leave. Let $F(s)$ be the set of their destination floors and the origin floors of $W(s)$, excluding the current lift floor, so that a floor needed by several passengers counts once. The resulting heuristic is $2|W(s)| + |R(s)| + |F(s)|$. A move reaches at most one required floor, and boarding or leaving reduces the passenger term $2|W(s)| + |R(s)|$ by exactly one without reducing $F(s)$, since it happens at the current floor, which $F(s)$ excludes. Under the certified location and goal assumptions, each action therefore reduces the heuristic by at most one, so it is consistent, and all three sets are empty at a goal, so it is goal-aware.

Figure 2 isolates the arithmetic step for boarding as a standalone Lean example. This step alone does not establish consistency, since the domain proof must also justify the count updates from the state transition and show that boarding does not reduce the floor term under the domain invariants.

Spanner. A worker walks along a directed path to a gate with loose nuts that need to be tightened. On the path, the worker can collect spanners, each of which can tighten only one nut. The heuristic combines action counting with dead-end detection. Let $L(s)$ be the number of goal nuts that are

still loose and $C(s)$ the number of distinct usable spanners that the worker carries. Let $D(s)$ be the directed distance from the worker to the gate. Tightening the nuts requires $L(s)$ actions and collecting the missing spanners at least $\max(0, L(s) - C(s))$ pickups, and adding the movement bound gives $L(s) + \max(0, L(s) - C(s)) + D(s)$. The heuristic is zero when all goal nuts are tight. It returns the pruning threshold B if $C(s)$ plus the number of distinct usable spanners still reachable on the ground is less than $L(s)$, which detects dead ends with too few usable spanners.

Sokoban. In this Sokoban formulation, each goal requires a specific box on a designated square. The heuristic is the maximum of two lower bounds that both ignore the other boxes when moving a box. Both use the push graph, which connects two squares if a box can be pushed from one to the other while the robot stands behind the box. The first bound adds up pushes and walking. For each box b with an unmet goal, let $P_b(s)$ count the pushes on a shortest path to its square in the push graph. Since each push moves only one box, different boxes need different pushes, so the sum of these counts is a lower bound on the number of pushes. Before its first push, the robot must also walk to a square from which it can push some box, and $A(s)$ is the length of the shortest such walk. Since the first push may move a box that is already at its goal, $A(s)$ also considers such boxes. The second bound considers each box on its own, with $J_b(s)$ counting the robot moves and pushes on a shortest path that brings b to its square when only b and the robot remain. The two bounds may count the same actions, so the heuristic takes their maximum instead of their sum:

$$h(s) = \max \left\{ \sum_{b \text{ with unmet goal}} P_b(s) + A(s), \quad \max_{b \text{ with unmet goal}} J_b(s) \right\}$$

All terms are zero when no box goal is unmet, so the heuristic is goal-aware. LeanPlan computes the distance tables with breadth-first search when it compiles the heuristic. Instead of proving this search correct, it checks that each table has the properties the proof relies on, such as distances that decrease by at most one per move.

5 Experimental Results

We use the ten domains from the IPC 2023 Learning Track (Taitler et al., 2024) and three new domains, Orrery, Cisterns and Qubit Routing, described in Appendix E. The new domains and their tasks were not publicly available before the experiments, so the LLM cannot have seen them in public training data. Each domain has 90 test tasks, for 900 IPC 2023 tasks and 270 tasks in the new domains. The heuristics are developed on each domain's training tasks, 89 to 99 per IPC 2023 domain and 20 per new domain. We run GPT 5.6 Sol at its highest reasoning effort as the coding agent in the Codex harness to generate heuristics and prove their admissibility. We ran the agent in a sandbox that prevented access to test tasks and restricted file modifications to the designated workspace.

Across our runs, the agentic loop took an average of 69.8 minutes (stdev. $\pm$ 16.5 min.) per domain, excluding time spent evaluating the baseline planner. It used an average of 33 724 876 input tokens, including cached input, and 103 196 output tokens per domain; approximately 98.6% of the input tokens were cached. The agent explored 10.3 heuristic configurations per domain on average, with up to 13 attempts in some domains. The average cost was US\$17.30 (stdev. $\pm$ US\$4.62) per domain. Given the time and monetary cost of repeated runs, we prioritized evaluating a broad set of thirteen domains over running the agentic loop multiple times per domain.

We compare five planner configurations. Two use LeanPlan: *blind LeanPlan* with the blind heuristic (0 at goal states and 1 elsewhere) and *LeanPlan* with the generated heuristics. The other three use the state-of-the-art optimal planner Scorpion (Seipp, 2024): *blind Scorpion* with the blind heuristic, *LM-cut* with the domain-independent LM-cut heuristic (Helmert & Domshlak, 2009) and *SCP* with Scorpion's recommended configuration. SCP combines abstraction heuristics via saturated cost partitioning (Seipp et al., 2020), and we adjust its component budgets to our resource limit.[2] All five read the same PDDL inputs and run A* after pruning actions that h^2 mutexes rule out (Alcázar &

[2]The recommended Scorpion configuration allocates up to 100 seconds to systematic pattern collection construction and 1 000 seconds to online cost partitioning. We use 10 and 150 seconds, respectively, which gave the best coverage in earlier parameter sweeps on the test suite under our 300-second overall limit. During heuristic development, the agentic loop gives both planners 30 seconds per training task and uses budgets of 1 and 15 seconds (Appendix B).

Table 1: Tasks solved, with 90 per domain and separate sums for IPC 2023 and the new domains. Total combines both suites. The left columns use blind A* search and the right columns use A* search with heuristics. SCP is Scorpion's recommended configuration, which uses saturated cost partitioning (Seipp et al., 2020). All configurations prune tasks with h^2 mutexes (Alcázar & Torralba, 2015). Bold values and shading mark the highest coverage in each row, including ties.

		Blind A*		A* with heuristics		
	Domain	Scorpion	LeanPlan	Scorpion LM-cut	Scorpion SCP	LeanPlan
IPC 2023	Blocksworld	6	6	11	11	**20**
	Childsnack	9	9	9	9	**16**
	Ferry	10	10	18	19	**29**
	Floortile	10	10	**20**	**20**	**20**
	Miconic	30	30	36	**40**	37
	Rovers	15	12	16	17	**18**
	Satellite	12	12	20	**27**	25
	Sokoban	27	27	29	**31**	30
	Spanner	30	30	30	30	**72**
	Transport	8	8	9	**17**	14
	Sum	157	154	198	221	**281**
New	Orrery	0	0	1	18	**20**
	Cisterns	9	8	10	**26**	25
	Qubit Routing	0	0	0	7	**22**
	Sum	9	8	11	51	**67**
	Total	166	162	209	272	**348**

Torralba, 2015). We run each configuration once per task with Downward Lab (Seipp et al., 2017), limiting every run to 300 CPU seconds and 8 GiB of memory. We report *coverage*, the number of tasks solved, as well as planner time and the number of states expanded during search. The plan validator VAL (Howey & Long, 2003) accepts every returned plan, and all planners that solve a task find plans of the same cost.

5.1 Blind Search

Since the blind heuristic gives no guidance, comparing blind LeanPlan with blind Scorpion measures how efficiently the two planners implement grounding and search. Blind LeanPlan solves nearly as many tasks as blind Scorpion (Table 1). The two planners solve the same number of tasks in all domains except Rovers and Cisterns, and every task solved by blind LeanPlan is also solved by blind Scorpion. On the IPC 2023 tasks that both solve, they expand almost the same number of states in total (differences are due to tie-breaking), but LeanPlan's search takes about 1.5 times as long. The remaining gap therefore stems from slower state expansion rather than from different search behavior. Slower expansion also explains why blind LeanPlan solves fewer tasks within one second (106 compared to 116), although it nearly catches up within ten seconds (138 compared to 139). Overall, proof-checked grounding and search cost little coverage compared with Scorpion, whose Fast Downward code base has been optimized for many years.

5.2 A* Search with Heuristics

With the generated heuristics, LeanPlan solves the most IPC 2023 tasks (Table 1). Its largest gain is in Spanner, which we attribute to the dead-end test of its heuristic, but even without Spanner, it solves more tasks than SCP (209 compared to 191). Still, SCP solves the most tasks in four domains, so the generated heuristics do not dominate domain-independent ones everywhere. LeanPlan's advantage is largest on tasks beyond the training sizes. Of the 595 IPC 2023 test tasks with more objects than every training task of their domain, LeanPlan solves 53, compared to 13 for SCP, 7 for LM-cut and 0 for blind search. The generated heuristics thus remain effective far outside the training distribution.

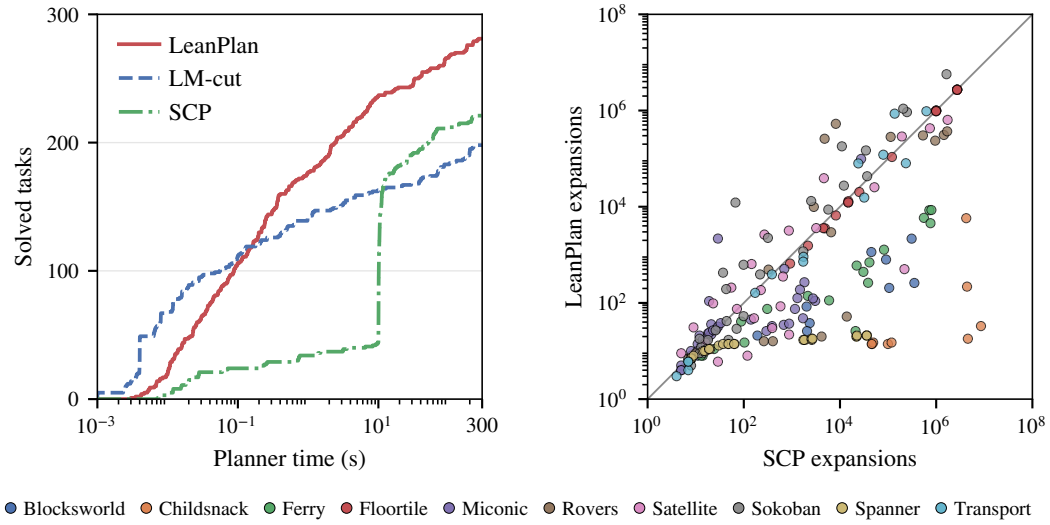


Figure 3: A* search with heuristics on IPC 2023. Left: number of tasks solved within each reported per-task time threshold, with a logarithmic horizontal axis. Right: expansions of LeanPlan and SCP on the 211 tasks that both solve, with logarithmic axes. Each point represents one task, points below the diagonal favor LeanPlan and colors identify domains.

The left panel of Figure 3 shows how quickly each configuration solves tasks. LeanPlan and LM-cut need little precomputation, so their curves rise from the start. LM-cut's curve flattens early, however, because its heuristic is expensive to evaluate: on the tasks that both solve, LM-cut's search takes five times as long as LeanPlan's. SCP first spends up to ten seconds constructing its pattern collection, so its curve rises only afterwards. Within ten seconds, LeanPlan solves more tasks than SCP within the full time limit.

The right panel of Figure 3 compares the expansions of LeanPlan and SCP on the 211 IPC 2023 tasks that both solve. LeanPlan expands fewer states on 152 of these tasks and more on 55, and the balance depends on the domain. In Spanner, where LeanPlan's heuristic prunes dead ends, LeanPlan expands fewer states on every task. In Sokoban, SCP expands fewer states on most tasks, presumably because its abstractions capture some interactions between boxes, which LeanPlan's relaxations ignore. Since the planners also differ in other components, such as tie-breaking, these counts compare complete configurations rather than heuristics alone. Compared with blind LeanPlan, the generated heuristics reduce the number of expansions on every task that both solve, by more than two orders of magnitude in total.

In the new domains, blind search solves only a few Cisterns tasks, and LM-cut solves few tasks in any domain. LeanPlan solves more tasks than SCP overall, especially in Qubit Routing, but one task fewer in Cisterns. There, the certificate refuses three test tasks, the only refusals among all 1 170 test tasks, so the proof does not cover them and LeanPlan counts them as unsolved. SCP solves one of them, which accounts for its lead. No configuration solves the other two. Since the certificate must accept all training tasks, a more varied training set would have led the controller to reject this heuristic and the agent to find one that also covers such tasks.

6 Related Work

LLMs for Planning. Early studies found that LLMs cannot reliably generate valid plans, even with chain-of-thought prompting (Valmeekam et al., 2023; Stechly et al., 2024). Recent frontier models are competitive with classical planners on newly generated tasks from the IPC domains (Corrêa et al., 2025), but their plans still require external validation. Many approaches therefore use LLMs to produce artifacts that symbolic systems execute or check (Kambhampati et al., 2024). These artifacts include PDDL models (Guan et al., 2023; Tantakoun et al., 2025), successor functions and goal tests (Katz et al., 2024) and the generalized plans and heuristic functions discussed in the introduction.

LLM-Guided Program Search. Outside of planning, LLM-guided search finds programs that score well on a fixed set of instances (Romera-Paredes et al., 2024; Liu et al., 2024; Ye et al., 2024; Novikov et al., 2025), but offers no guarantees beyond them. Counterexample-guided inductive synthesis instead checks each proposed program against a formal specification (Solar-Lezama et al., 2006; Abate et al., 2018). LeanPlan combines both ideas: search on training tasks measures how informative a heuristic is, while the Lean kernel checks its admissibility.

Learned Heuristics and Admissibility. Machine learning has long been used to obtain heuristics for search (Arfaee et al., 2011; Shen et al., 2020; Ferber et al., 2020; Chen et al., 2024a; Agostinelli et al., 2019; Chen et al., 2024b). Such heuristics are generally inadmissible, and methods that aim for heuristics that are admissible with high probability (Ernandes & Gori, 2004; Marom & Rosman, 2020) or come with statistical generalization bounds (Futuhi & Sturtevant, 2026) do not guarantee admissibility on unseen states. Domain-independent heuristics such as LM-cut (Helmert & Domshlak, 2009), pattern databases (Edelkamp, 2001; Haslum et al., 2007), merge-and-shrink (Helmert et al., 2014) and saturated cost partitioning (Seipp et al., 2020) are admissible by construction, and Phung et al. (2026) use LLMs to generate pattern generators within this family.

Verification and Certificates in Planning. Plan validators such as VAL (Howey & Long, 2003) check plans, Abdulaziz & Lammich (2018) formally verify such a validator in Isabelle/HOL (Nipkow et al., 2002) and Abdulaziz & Kurz (2023) formally verify a SAT-based planner. Certifying algorithms (McConnell et al., 2011) return a certificate that an independent checker can verify, and certifying planners provide such certificates for unsolvability (Eriksson et al., 2017; 2018) and optimality (Mugdan et al., 2023; Dold et al., 2025). Dold et al. (2025) extend A^* to emit pseudo-Boolean certificates of optimality during the search, which a verified checker then validates. These certificates are generated and checked for each task, whereas LeanPlan proves admissibility once per domain. What we call a certificate (Section 3) is therefore not a per-task proof, but a fixed set of Boolean checks that establishes the premise of the domain theorems for each task. Closest to our work, Abdulaziz et al. (2022) use Isabelle/HOL to prove that generalized potential heuristics provide descending measures of progress across tasks of a domain (Francès et al., 2019). Behnke et al. (2026) formalize A^* and standard planning heuristics in Lean, using LLM-generated proofs for several components. LeanPlan instead targets admissibility of generated domain-specific heuristics, and LLM coding agents construct their proofs without human interaction.

Theorem Proving and Verified Code Generation. LLMs increasingly construct proofs in interactive theorem provers such as Lean and Isabelle (Jiang et al., 2023; Yang et al., 2023; Xin et al., 2024; Ren et al., 2025; Hubert et al., 2026), and they can repair failed proofs using the prover's error messages (First et al., 2023). Verifiable code generation asks for programs together with proofs of their correctness (Aggarwal et al., 2025; Ye et al., 2026). For example, Li et al. (2026) first generate a joint plan for the program and its proof, then elaborate both in Lean. LeanPlan specializes this idea to heuristic search, where every heuristic has the same proof obligations and where correctness alone is easy to achieve, since the heuristic that always returns zero is admissible. The agent must therefore also make the heuristic informative, which we measure by search on training tasks.

7 Conclusions

We introduced LeanPlan, a planner in which LLM coding agents develop domain-specific heuristics together with machine-checked proofs of their admissibility. For all domains considered, the agents produced such heuristics and proofs, so LLMs can help optimal planning without giving up the optimality guarantee. Requiring proofs did not force the agents to settle for weak heuristics: the proved heuristics exploit domain structure, are generally more informed than strong domain-independent heuristics and let LeanPlan solve more tasks than the state-of-the-art optimal planner Scorpion.

Limitations. LeanPlan supports only the unit-cost STRIPS fragment of PDDL, and our evaluation uses thirteen domains and a single LLM and coding-agent harness. The proofs guarantee admissibility only on tasks that satisfy the checked domain assumptions, and these assumptions can exclude test tasks, as in Cisterns. They do not certify unsolvability. The guarantees also rely on parts that the proofs do not check: Lean's kernel, compiler and runtime, LeanPlan's PDDL parser and the Lean definitions that formalize PDDL semantics. Extending the approach to action costs and richer PDDL fragments and evaluating other LLMs, including open-weight ones, remain open directions.

AI USE STATEMENT

We used LLM coding agents to develop the heuristic programs and Lean proofs described in Section 3. We also used AI assistants to check the literature, review and revise the manuscript and prepare the analysis and figures.

ETHICS STATEMENT

This work uses public planning benchmarks and three new benchmark domains (Appendix E), and involves no human subjects or personal data. LeanPlan's guarantees refer to the PDDL model, so an inaccurate model can still yield plans that are unsafe or suboptimal for the real system it describes. Developing each heuristic and proof requires LLM computation, but LeanPlan reuses every accepted heuristic on new tasks of its domain without further LLM calls.

REPRODUCIBILITY STATEMENT

Appendix A specifies the proof interface and search conditions, and Appendix C describes the domain heuristics. Appendix D documents the run records and their analysis. We will make the source code, benchmarks and run logs publicly available upon acceptance.

REFERENCES

Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *Proc. CAV 2018, Part I*, pp. 270–288, 2018.

Mohammad Abdulaziz and Friedrich Kurz. Formally verified SAT-based AI planning. In *Proc. AAAI 2023*, pp. 15073–15081, 2023.

Mohammad Abdulaziz and Peter Lammich. A formally verified validator for classical planning problems and solutions. In *Proc. ICTAI 2018*, pp. 474–479, 2018.

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.

Pranjal Aggarwal, Bryan Parno, and Sean Welleck. AlphaVerus: Bootstrapping formally verified code generation through self-improving translation and Treefinement. In *Proc. ICML 2025*, pp. 587–615, 2025.

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1:356–363, 2019. doi: 10.1038/s42256-019-0070-z.

Vidal Alcázar and Álvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *Proc. ICAPS 2015*, pp. 2–6, 2015.

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

Masataro Asai and Alex Fukunaga. Tiebreaking strategies for a^* search: How to explore the final frontier. In *Proc. AAAI 2016*, pp. 673–679, 2016.

Masataro Asai and Alex Fukunaga. Tie-breaking strategies for cost-optimal best first search. *Journal of Artificial Intelligence Research*, 58:67–121, 2017.

Gregor Behnke, Simone Kilian, and Malvin Gattinger. A^* with $h^{\max}$ definitely finds optimal plans – formally verifying a planner based on heuristic search. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, 2026.

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

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

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

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.

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

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Proc. CADE 2021*, pp. 625–635, 2021.

Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

Simon Dold, Malte Helmert, Jakob Nordström, Gabriele Röger, and Tanja Schindler. Pseudo-boolean proof logging for optimal classical planning. In *Proc. ICAPS 2025*, pp. 54–63, 2025.

Stefan Edelkamp. Planning with pattern databases. In *Proc. ECP 2001*, pp. 84–90, 2001.

Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In *Proc. ICAPS 2017*, pp. 88–97, 2017.

Salomé Eriksson, Gabriele Röger, and Malte Helmert. A proof system for unsolvable planning tasks. In *Proc. ICAPS 2018*, pp. 65–73, 2018.

Marco Ernandes and Marco Gori. Likely-admissible and sub-symbolic heuristics. In *Proc. ECAI 2004*, pp. 613–617, 2004.

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

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

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proc. ESEC/FSE 2023*, pp. 1229–1241, 2023.

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

Ehsan Futuhi and Nathan R. Sturtevant. Learning admissible heuristics for A*: Theory and practice. In *Proc. ICLR 2026*, 2026.

Elliot Gestrin and Jendrik Seipp. LLM-evolved domain-independent heuristics for symbolic AI planning. In *ICAPS Workshop on Planning in the Era of LLMs*, 2026.

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

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*, pp. 79081–79094, 2023.

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

Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. AAAI 2007*, pp. 1007–1012, 2007.

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.

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

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, pp. 162–169, 2009.

Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM*, 61(3):16:1–63, 2014.

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

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

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *Proc. ICLR 2023*, 2023.

Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1–2):219–251, 2001.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B. Murthy. Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. In *Proc. ICML 2024*, pp. 22895–22907, 2024.

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

Zenan Li, Ziran Yang, Peiyang Song, Zhaoyu Li, and Kaiyu Yang. P^3: Joint program-and-proof planning for verified code generation. arXiv:2608.09277 [cs.AI], 2026.

Fei Liu, Tong Xialiang, 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*, pp. 32201–32223, 2024.

Ofir Marom and Benjamin Rosman. Utilising uncertainty for efficient learning of likely-admissible heuristics. In *Proc. ICAPS 2020*, pp. 560–568, 2020.

Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–162, 2011.

Esther Mugdan, Remo Christen, and Salomé Eriksson. Optimality certificates for classical planning. In *Proc. ICAPS 2023*, pp. 286–294, 2023.

Andrew Murray, Danial Dervovic, Alberto Pozanco, and Michael Cashmore. GenePlan: Evolving better generalized PDDL plans using large language models. In *Proc. ICAPS 2026*, 2026.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2002.

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.

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

André G. Pereira, Augusto B. Corrêa, and Jendrik Seipp. Property-guided LLM program synthesis for planning. arXiv:2605.16142 [cs.AI], 2026.

Windy Phung, Dominik Drexler, Arnaud Lequen, and Jendrik Seipp. LLM-evolved pattern generators for optimal classical planning. In *ICAPS Workshop on Reliability In Planning and Learning (RIPL)*, 2026.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. arXiv:2504.21801 [cs.CL], 2025.

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.

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

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

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

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

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

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pp. 404–415. ACM, 2006.

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

Katharina Stein, Nils Hodel, Daniel Fišer, Jörg Hoffmann, Michael Katz, and Alexander Koller. Improved generalized planning with LLMs through strategy refinement and reflection. In *Proc. ICAPS 2026*, 2026.

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.

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*, pp. 25167–25188. Association for Computational Linguistics, 2025.

Alexander Tuisov, Yonatan Vernik, and Alexander Shleyfman. Successor-generator planning with LLM-generated heuristics. In *Proc. ICAPS 2026*, 2026.

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

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. arXiv:2405.14333 [cs.AI], 2024.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Proc. NeurIPS 2023*, pp. 21573–21612, 2023.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Proc. NeurIPS 2024*, 2024.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking verifiable code generation. In *Proc. ICLR 2026*, 2026.

A Proof Interface and Search Conditions

The shared interface in `Proofs/Heuristic.lean` expresses the contract in Equations 1 and 2 as follows:

```
structure ImprovedProof (d : Pddl.Domain) (p : Pddl.Problem)
    (rel : Bool) (check : Bool) (h : Heuristic) : Prop where
  goalAware : check = true ->
    (ground d p rel).GoalAwareOn (Reachable (ground d p rel)) h.eval
  consistent : check = true ->
    (ground d p rel).ConsistentOn (Reachable (ground d p rel)) h.eval
```

The listing replaces Lean's implication glyph with its ASCII notation. The declaration `structure ... : Prop` defines a proposition with two proof fields, `goalAware` and `consistent`. Constructing a proof of `ImprovedProof` requires supplying a proof of each field. In each field, `check = true ->` introduces an assumption, and `h.eval` names the executable function whose property must be proved. The Boolean `check` is therefore an input to the proposition, not a proof term. At runtime, evaluating it establishes which tasks satisfy the theorem's premise without reconstructing the proof. The flag `rel` selects whether grounding uses relevance pruning, and the domain theorems cover both values. Each of the thirteen domain files exports the theorem `improved_proof_of_certificate`, whose premise `Validated d p` requires both the domain and task validators to succeed. The theorem instantiates `check` with the domain certificate and `h` with the runtime evaluator: `improved (ground d p rel)` for IPC 2023 and `Agent.heuristic (ground d p rel)` for the new domains. The theorem `ImprovedProof.admissible` derives admissibility from these two fields and closure of reachability under transitions. The theorem covers every accepted task and every state reachable in that task.

From Domain Invariants to Executable Values. The proof establishes the initial invariants from the certificate and proves that the action schemas preserve them. `reachable_abstracts_inv` in `Proofs/Reachable.lean` transfers these invariants to reachable packed states using the shared grounding lemmas. `ComputesOn` in `Proofs/LiftedHeuristic.lean` equates a lifted heuristic with its compiled runtime value on the invariant states. The domain proofs combine shared correspondence lemmas with domain-specific arguments to prove the contract for the compiled evaluator. The shared grounding proofs account for atom numbering, static preconditions and relevance pruning. Thus the proof applies to the function that search evaluates.

Grounding Proof Boundary. The theorem `ground_plan_sound` transfers a plan of the packed-state task to the atom-level operators produced by `groundedOps`, preserving goal satisfaction and cost. Separate lemmas establish schema-instantiation properties and plan preservation under relevance pruning. The search guarantees below concern the executable ground task.

Returned Plans and Completion. In `Proofs/Search/AStar.lean`, `astar_sound` proves that any returned plan executes from the initial state to a goal. The theorem `astar_optimal` compares its cost with every valid plan of cost less than $B = 2^{40}$, assuming goal awareness and admissibility on an invariant containing the initial state. Taking that invariant to be reachability gives the statement in Section 3. If a plan of cost below B exists, any returned plan is globally optimal. Otherwise the cost comparison is vacuous and does not establish optimality. Pruning at B is also why the proofs do not certify dead ends or unsolvability: a pruned state has no plan that costs less than B, but it may still have a more expensive one, even if a domain-specific dead-end test caused the pruning. When `astar` exhausts its open list, LeanPlan therefore reports that it found no plan rather than that the task is unsolvable. Because Lean can only reason about functions that provably terminate, `astar` takes an explicit iteration budget, its fuel, which is 2^{62} by default. Both theorems hold whenever the fuel suffices to return a plan, and exhausting it is reported as a failure. The separate theorem `astar_terminates` in `Proofs/Search/Fuel.lean` gives a sufficient completion bound for well-formed unit-cost tasks: fuel greater than $N(N+1)+1$, where N counts all packed states satisfying `State.WF t.numFacts`.

Implementation. LeanPlan uses Lean 4.30.0 with mathlib at revision `c5ea00351c28e24afc9f0f84379aa41082b1188f`. The executable and the proof library are separate build targets, `lake build planner` and `lake build Proofs`, both run

from `planner/`, so checking the proofs requires the second. The command-line interface calls the proved `astar` routine. The library's `Proofs/AxiomAudit.lean` prints axiom dependencies for the search theorems and domain endpoints. The permitted dependencies are Lean's standard axioms `propext`, `Classical.choice` and `Quot.sound`. Proof-target build and audit outputs are needed to confirm this property for a particular artifact revision.

B AGENT PROMPT

The template below reproduces the prompt text from the agentic-loop implementation, with angle-bracket placeholders for the fields filled by the controller. These include domain and file names, training tasks, baseline results, proof endpoints, certificate instructions and worked examples. We render the function arrow as `->` and standardize the spelling of h^2-preprocessing; the instructions are otherwise unchanged.

```
<role>
You are a highly-skilled AI planning researcher and a proficient
Lean 4 programmer.  You work in the LeanPlan repository at `<repo>`,
an optimal STRIPS planner written in Lean 4.  Write a
domain-dependent heuristic for the PDDL domain
<domain><domain-name></domain>, and a Lean proof that it is
admissible.  A* stays optimal only if the heuristic is admissible,
and Lean's kernel decides whether it is.
</role>

<objective>
Create the registry heuristic `<heuristic-name>` with the best
possible performance on the <training-count> training tasks in
`<training-dir>`. Make it as informed and domain-dependent as
possible, and prove it admissible. Test and refine it on the
training tasks. Beating the Scorpion baseline is the minimum
requirement. Once you beat it, you must continue refining the
heuristic until you cannot improve it further.

The baseline is Scorpion with h^2-preprocessing and the search
configuration below. It uses the documented Scorpion configuration
with three time limits changed: 1 second to build the Sys-SCP
pattern collection, 1 second per restart, and 15 seconds of
cumulative online order-finding during search. Your heuristic must
be goal-aware and consistent on every reachable state its
certificate admits. Goal-awareness and consistency give
admissibility, which gives optimality; the shared library derives
that step for you.

<baseline-results>

Both planners get the same budget per task, enforced the same way
with `runlim` on one pinned core: <time-limit> seconds of wall-clock
time, one CPU core.

You work alone until you finish.  Nobody will answer you, and nobody
will prompt you again. Run the checks below yourself and act on what
they say.
</objective>

<what-you-produce>
<output-file-count>

1. The runtime heuristic, `<runtime-path>`, in namespace
   `<runtime-namespace>`, exporting `<runtime-function> : Task ->
   Heuristic` whose `name` field is exactly `<heuristic-name>`.  It
   compiles against `Planner/` alone, so it cannot import mathlib or
   anything under `Proofs/`.
```

````
2. The proof, `<proof-path>`, module `<proof-module>`, declaring
   these endpoints:
<theorem-endpoints>

<certificate-instructions>

Add the registry entry for `<heuristic-name>` in
`planner/Planner/ExampleHeuristics/Registry.lean`, with the imports
it needs, so that `--heuristic <heuristic-name>` runs.  Until you
do, every check below reports an unknown heuristic.
</what-you-produce>

<success-criteria>
Success means finding the best admissible heuristic you can for the
training set. Beating the baseline is only the minimum. We expect
you to push performance as far beyond that minimum as you can
through continued refinement.

The final candidate must satisfy these correctness requirements:

1. Goal-aware: the heuristic is 0 on every goal state the
   certificate admits.
2. Consistent: `h s <= cost o + h (o s)` for every grounded operator
   `o` and every reachable state `s` the certificate admits.
3. The theorems concern the compiled heuristic the registry builds,
   which is the function A* evaluates, not an abstract surrogate.
4. The certificate accepts all <training-count> training tasks.
</success-criteria>

<how-to-check-your-own-work>
Run these from `<repo>`.

Every planner command below is wrapped in `runlim`, at the same
<time-limit>-second limit the controller uses.  Run them that way.
A command that would not finish inside the limit is a failure, and
you want to see it as one rather than wait for it.

```bash
# 1. The runtime heuristic compiles.  Seconds, and no mathlib.
(cd planner && lake build planner)

# 2. Walk reachable states and report any state where the heuristic is not
#    goal-aware or not consistent, naming the transition that breaks it.  A
#    breach here means no proof exists.
runlim --real-time-limit=<time-limit> --kill --propagate \
    <planner-binary> --heuristic <heuristic-name> \
    --audit-consistency 20000 <domain-file> <first-training-task>

# 3. The certificate accepts a task.  The controller runs this on every task
#    under the same limit, so a certificate too slow to finish fails.
runlim --real-time-limit=<time-limit> --kill --propagate \
    <planner-binary> --heuristic <heuristic-name> \
    --check-certificate <domain-file> <first-training-task>

# 4. The proof.  Minutes; this is the slow one.
(cd planner && lake build <proof-module>)

# 5. Write a scratch file importing <proof-module> that runs
#    `#print axioms` on each endpoint, then elaborate it:
(cd planner && lake env lean /tmp/audit.lean)

# 6. What the search actually does on a task.
runlim --real-time-limit=<time-limit> --kill --propagate \
    <planner-binary> --heuristic <heuristic-name> <domain-file>
  <middle-training-task>
````
````

```
# 7. The same task under Scorpion, for comparison.  Scorpion lives at
#    $SCORPION_SRC, and gets the same budget: <time-limit> s, one core.
runlim --real-time-limit=<time-limit> --kill --propagate \
    taskset --cpu-list 0 "$SCORPION_SRC/fast-downward.py" \
    --preprocess <domain-file> <middle-training-task> \
    --search "<scorpion-search>"
```
</how-to-check-your-own-work>

```
<what-not-to-do>
- Do not write `sorry`, `admit`, or `native_decide`, and do not
  declare an `axiom`.  The controller scans for all four before it
  checks anything else.
- Do not weaken a required theorem to make it easy.  Each endpoint
  states admissibility of the compiled heuristic under the
  certificate.  A theorem that is true but says less fails.
- Do not use the certificate to dodge difficulty.  It states what
  your proof assumes; the controller runs it on every training task
  and rejects a candidate that refuses one.
</what-not-to-do>

<when-to-stop>
Stop when you are confident that the heuristic is as informed as you
can make it and you cannot refine it further. Leave the best
candidate, with a working proof and passing certificates, in the
files.
</when-to-stop>

<orientation>
Read these first, in roughly this order:

<reading-list>

<worked-example-description>

The domain file is `<domain-file>`, and the <training-count>
training tasks are in `<training-dir>`, for example
<example-training-tasks>.  The registry name `<heuristic-name>` maps
to the PDDL domain `<pddl-domain-name>`.  The planner binary is
`<planner-binary>`.
</orientation>
```

C Domain Heuristic Details

Table 2 summarizes the IPC 2023 heuristics in `Planner/ExampleHeuristics/`, with one `Improved.lean` file per domain. The entries describe how the main action-count and distance bounds combine. The final domain theorems use the interface in Appendix A for all ten evaluators. Distance-based terms use the checked tables described below, including their finite caps and zero-table fallback.

Table 2: Components of the registered domain heuristics. A maximum combines bounds that may count the same actions. Additive terms count separate necessary actions under the domain invariants.

Domain	Bound composition
Blocksworld	Maximum of unmet-clear count and required grab/place count, with held-block corrections and two actions per forced detour.
Childsnack	Unserved goal children, tray-loading shortfall, sandwich-making shortfall and necessary tray moves. Shortfalls account for gluten-free demand and existing sandwiches.
Ferry	Unmet car goals: three actions per nonaboard car or one or two per carried car, plus the maximum of repositioning and waiting-location corrections.
Floortile	Unmet painted goals, missing-color changes and the maximum of farthest painting-position distance, half the unattended-tile count rounded up and distinct forced painting squares without a robot. A reachability test also detects dead ends.
Miconic	Boarding and leaving counts plus distinct required floors other than the current lift floor.
Rovers	Communication, sampling, imaging, calibration-shortfall and empty-store-shortfall counts, plus the maximum of travel and unoccupied required-waypoint counts.
Satellite	Missing images, calibration and power-on cover bounds, possible power-slot release, missing pointing goals, unpaid image directions and a possible calibration approach turn. The cover bounds take values zero, one or two.
Sokoban	Maximum of total push distances plus the walking bound before the first push and the largest one-box/robot distance.
Spanner	Tightening actions, pickup shortfall and the farthest-nut distance. A usable-spanner shortage test detects dead ends.
Transport	Package loading and unloading actions plus the maximum of the farthest package driving requirement and distinct required locations without a vehicle.

Blocksworld. For each block, compilation selects the first `on` goal naming it as the upper block and the first naming it as the lower block. These goals specify its selected support and the block it should carry, respectively. The goal-support chain follows the selected supports downwards. The heuristic marks blocks with an unmet selected `on` goal or `on-table` goal, held blocks and blocks standing on a support required to be clear or to carry a different selected block. It propagates these marks upwards through the current towers. Each marked block contributes one grab (`pickup` or `unstack`) and one placement (`putdown` or `stack`). The bound subtracts one grab if a marked block is already held and one placement if some marked block has no unmet support goal (`on` or `on-table`). The latter subtraction allows such a block to remain held in the relaxation. A marked block incurs a *detour* if its selected goal support is marked and the block is either held or standing on a block that belongs below it in the compiled goal-support chain. Such a block must be placed temporarily before it can reach its goal support, adding one grab and one placement. The heuristic takes the maximum of this action count and the number of unmet `clear` goals.

Childsnack. Let n count unserved goal children and a those with gluten allergies. Let t and f count all sandwiches and gluten-free sandwiches on trays, and let k and g be the corresponding kitchen counts. Kitchen counts exclude sandwiches already on trays. The loading bound is $L = \max(0, a - f, n - t)$, and the sandwich-making bound is $M = \max(0, a - f - g, L - k)$. The heuristic returns $n + L + M + R$ when $n > 0$ and zero otherwise. Here R counts nonkitchen places whose waiting unserved goal children outnumber the sandwiches on trays currently there, plus one if $L > 0$ and no tray is in the kitchen. A tray without a recorded location contributes no sandwiches to any place count. Serving, loading, making sandwiches and moving trays require separate actions, so their bounds are additive.

Ferry. Each unmet car goal contributes three actions if the car is not aboard, two if it is aboard away from its destination and one if it is aboard and the ferry is at its destination. We call a car with an unmet goal *waiting* if it is not aboard, even when its location is unrecorded. Let W contain the distinct recorded locations of waiting cars. The emptiness test reads the `empty-ferry` atom, with a missing atom counting as false. The repositioning correction is zero if no car is waiting, if the emptiness test succeeds at a location in W, or if the test fails and the carried car with an unmet goal has its destination in W. It is one otherwise. The waiting-location correction counts locations in W that are neither destinations of unmet car goals nor the current location of a ferry for which the emptiness test succeeds. Both corrections bound additional actions beyond the per-car counts, which can be paid by sailing or extra car handling. The heuristic adds the per-car counts and the maximum of these corrections. A waiting car with no recorded location still contributes three actions, but no waiting location. An unknown ferry location matches neither waiting locations nor destinations.

Floortile. The heuristic adds the number of unmet painted goals, the number of distinct required colors held by no robot and a movement bound. The movement bound is the maximum of three quantities. The first takes, for each unmet painted goal, the minimum static-graph distance from any robot to any square from which the tile can be painted, then selects the largest distance. The second counts *unattended* tiles with unmet goals, which have no robot at a painting position, and divides by two, rounding up. One move can put a robot in position to paint at most two such tiles. The third counts distinct forced painting squares that no robot currently occupies. A goal tile has a forced painting square below it if that square exists and the upward chain consists entirely of goal tiles and ends at a tile with nothing above it. Since painting makes a tile impassable, these goal tiles must be painted from below, starting at the top. The heuristic also searches the movement graph through clear and robot-occupied squares. It returns B if an unmet tile's forced painting square is unreachable, or, for a tile without a forced square, if none of its painting positions is reachable. Otherwise it returns the sum of the painting, color-change and movement bounds.

Miconic. The certificate requires unique passengers in `served` goals and typed, unique destination atoms covering those passengers. It checks at most one initial lift floor and at most one origin per passenger, without requiring that either exists. It also checks the expected schemas and floor types. Together with the invariants, these conditions justify filtering out passengers without goals and counting each needed floor only once. The waiting and riding sets in Section 4 partition the passengers with unmet goals, and `neededFloors` deduplicates their required floors. An absent passenger origin contributes no floor, and an absent lift location excludes no floor from this set. Figure 4 shows the heuristic's Lean code without helper definitions. The heuristic `improved` calls `compile` once per task to build passenger and floor tables and evaluates `value` in each state.

```
def value (d : Data) (s : State) : Nat :=
  2 * (waiting d s).size + (riding d s).size
    + (neededFloors d s).length

def improved (t : Task) : Heuristic :=
  let d := compile t
  { name := "miconic-improved", eval := value d }
```

Figure 4: Miconic's heuristic counts waiting and riding passengers and required floors other than the current floor. Helper definitions are omitted.

Rovers. The heuristic counts one communication per unmet data goal, one sample per missing soil or rock analysis and one image-taking action per missing image. An analysis or image is missing only if no rover already holds it. Each image-taking action consumes a calibration, so the calibration shortfall is the number of required image-taking actions minus the number of currently true `calibrated` atoms, truncated at zero. Similarly, the empty-store shortfall is the number of required samples minus the number of currently true `empty` atoms, truncated at zero. The travel bound is the largest distance to a required sampling waypoint from its nearest suitably equipped rover, using each rover's traversal-and-visibility graph. Rovers without recorded locations are omitted, and a minimum with no located suitable rover is zero. A second navigation bound counts distinct required sampling waypoints that no rover currently occupies. The heuristic adds the five action counts to the maximum of the two navigation bounds. Communication, sampling, imaging, calibration, emptying stores and navigation use separate actions.

Satellite. Each unmet `have_image` or `pointing` goal contributes one. The heuristic adds the following five bounds to these counts. An instrument is *ready* if it is both powered and calibrated. The calibration bound considers modes needed by unmet image goals that no ready instrument supports. It is zero if no mode remains, one if a single instrument supports all remaining modes and two otherwise. The power-on bound first removes modes supported by a powered instrument, then applies the same rule to the remaining modes, considering all instruments for the cover. For each unmet image, we omit its direction if an unmet pointing goal already names it. We also omit it if a satellite points there with an instrument that supports the image's mode and is either ready or has that direction as its calibration target. The image-direction bound counts distinct directions of the remaining images. For the remaining terms, a candidate is any instrument that supports a mode needed by an unmet image goal. The calibration-approach bound is zero if no calibration is needed. It is also zero if any candidate has an unknown carrier or target, its carrier is already at its target, or its target occurs among the unmet pointing-goal directions or additional image directions. Otherwise it is one. The power-slot-release bound is one if some needed mode has no powered instrument and every candidate has a known carrier with a recorded, currently false `power_avail` atom. Otherwise it is zero.

Sokoban. Each box named by a location goal has one designated goal square, and the push graph records relaxed legal pushes. The one-box product graph keeps the robot because reaching a pushing side may require walking even when a box is close to its goal. The additive push bound and the maximum product bound count some of the same actions, so the registered heuristic takes their maximum. The runtime registration selects `baseValue`. A separate function `value` adds an explicit dead-end branch, but the planner does not use this wrapper. The implementation uses finite caps for unreachable distance-table entries and checks the properties required by the proofs. For n locations, an unrecorded box position contributes n to its push bound and n^2 to its product bound when a robot position is recorded. An unrecorded robot position makes the product and walking bounds zero. The walking bound is the table cap if no recorded box location has a relaxed pushing position. All components are zero when no compiled box-location goal remains unmet, even if another dynamic goal is still unmet.

Spanner. The implementation permits several workers. Let G contain the goal nuts with a recorded initial location. The compiled heuristic omits other goal nuts because nut positions never change and an unlocated nut cannot be tightened. The movement bound is the maximum over nuts in G with unmet `tightened` goals of the minimum directed-link distance from a worker's current location. The empty maximum is zero. Workers without a recorded location are omitted from the inner minimum, which equals the distance-table cap if no recorded worker location remains.

Unreachable distances use the finite bound stored in the compiled distance table. The ground-spanner count excludes carried spanners and counts only usable objects at a location reachable from some worker. The certificate checks location uniqueness, object-type disjointness and the size bound $2|G| + D_{\text{bound}} \leq B$, where D_{bound} is the distance-table bound. These checks support the finite representation and the domain proof.

Transport. Each unmet package goal contributes two handling actions if the package is on the ground and one if it is carried. For a ground package, the driving requirement adds the shortest road distance from any vehicle to the package and the distance from there to its destination. For a carried package, it uses the carrying vehicle's distance to the destination. The first driving bound is the maximum requirement over packages, with an empty maximum equal to zero. The second counts distinct destinations and recorded ground locations of packages with unmet goals that currently contain no vehicle. The heuristic adds the handling count and the maximum of these driving bounds. A package with neither a ground location nor a carrier contributes zero handling and individual driving cost, but its destination remains in the location count. Missing vehicle locations use the distance-table cap.

C.1 Checked Task Classes

The following certificate conditions delimit the supported task classes. They apply in addition to ordinary validation and exact equality of the action-schema list with the generated domain. An *exact type* requires each object that inhabits that type through subtyping to declare that precise type. The initial-state checks establish the premises whose dynamic components the domain proofs preserve. Static-predicate checks ensure that the atoms in their static components cannot change. A dynamic goal uses a predicate that an action schema can change. These are task acceptance conditions, distinct from the zero-table fallback used by distance computations. The definitions are in each domain's `Certificate.lean` file.

Blocksworld. The certificate requires a declared `object` type, at least one object and duplicate-free goals. Each `on` goal has two arguments, but goals need not consist only of `on` atoms. Initially, each block has at most one support and supports at most one block. At most one block is held, and `arm-empty` holds exactly when none is held. A held block neither stands on nor supports a block and is neither on the table nor clear. A block standing on another is not on the table, and a clear block supports nothing. Following initial supports must terminate within the number of `on` atoms plus one. The checks establish unambiguous acyclic towers without requiring complete block locations.

Childsnack. The sandwich, tray, place, child, bread-portion and content-portion types must be declared, and sandwich, tray and place types must be exact. At least one tray and sandwich and a place named `kitchen` must exist. Goals form a nonempty, duplicate-free list of `served` atoms naming children declared with type `child`. Each goal child has a waiting place and an allergy classification. Waiting places are unique per child and typed, and allergic and nonallergic classifications are disjoint. Both classification predicates and `waiting` are static. Initially, each tray occupies at most one place, each sandwich belongs to at most one tray, and kitchen and tray membership are disjoint. Nonexistent sandwiches are in neither location and cannot be gluten-free. No sandwich is initially gluten-free. Tray-location existence is not required. These checks support distinct resource counts and preserved sandwich and tray invariants.

Ferry. Car and location types must be declared and exact, with at least one object of each. Every car has exactly one `at` goal naming a typed location. The extracted car–destination pairs and their car names are duplicate-free, but other goal predicates are not excluded. Initially, the ferry has at most one location, each car has at most one ashore location, and at most one car is aboard. An aboard car has no ashore location. An empty-ferry atom implies that no car is aboard, without requiring the converse. Initial location existence is not required. These conditions establish the capacity-one invariant used by the per-car sailing bound.

Floortile. Tile, robot and color types are exact. Goals are duplicate-free, name each painted tile once and exclude grid edges. Grid predicates are static. Initially, each robot has at most one position and color, and each tile has at most one robot. Occupied tiles are neither clear nor painted, and clear tiles are unpainted. The certificate grounds both relevance settings and checks each compiled task. Up/down edges have arity two, `up` is functional and inverse-functional, and each `down` edge reverses

an `up` edge. Compiled painted goals have known colors and unique tile entries, and present clear-atom indices match their tiles. For N compiled tile entries and distance cap D_{bound}, the certificate requires $2N + \max(D_{\text{bound}}, N) \leq B$. Neither robot-position existence nor complete rectangular topology is required.

Miconic. Floor and passenger types must be declared, with at least one object of each, and the floor type must be exact. Goals are a nonempty list of `served` atoms naming distinct passengers declared with type `passenger`. Destinations are static, unique per passenger, typed as floors and present for every goal passenger. The initial invariant checks at most one lift floor and at most one origin per passenger. As explained above, these multiplicity checks do not require initial location existence.

Spanner. Location, man and spanner types must be exact. Nut, man and spanner objects must inhabit pairwise disjoint types. Links are static, and goals are duplicate-free and contain no links. Other goal predicates are not excluded. Initially, each object has at most one location, no nut is both loose and tightened, and a carried object has no ground location. There is no requirement that every object has a location or that only one worker exists. The compiled-task bound is $2|G| + D_{\text{bound}} \leq B$ for the set G defined above.

Rovers. Rover, waypoint, store, camera, mode, lander and objective types must be declared, and rover and waypoint types must be exact. Goals are duplicate-free and dynamic, but need not be communication goals. The heuristic compiles only the three communicated-data goal predicates. Each rover initially has at most one position, without requiring one to exist. Traversal, visibility and soil/rock equipment predicates must be static. Traversal self-loops are forbidden. Every traversal edge also present in `visible` must have reverse traversal and visibility atoms. The per-rover graphs intersect traversal and visibility. There are no additional explicit type-disjointness, store/camera assignment-multiplicity or task-size conditions. Graph distances use node-count caps and checked zero-table fallback.

Satellite. Satellite, direction, instrument and mode types must be declared and exact. Goals are duplicate-free and dynamic, but need not be image or pointing goals, the only forms compiled into heuristic goal tables. Initial instrument-carrier, calibration-target and satellite-pointing relations are single-valued on their first argument, without requiring any entry to exist. Initial `pointing` and `not-pointing` atoms must not contradict each other. Every instrument assigned to a carrier initially lacks `power_on`. The `on_board`, `supports` and `calibration_target` predicates are static. The certificate does not require complete pointing complements, initial power availability or complete relation coverage, and imposes no task-size condition.

Transport. Location and vehicle types are exact, loading and unloading parameter types are declared, and no vehicle is also a package. Goals are duplicate-free, exclude `road`, and contain at most one `at` destination per object, with no such goal naming a vehicle. Other goal predicates are permitted. Initially, each object has at most one location and at most one carrier, and cannot have both. Neither location nor carrier existence is required. Road and capacity-predecessor predicates are static, without road symmetry, connectivity or capacity-multiplicity checks. Road distances use checked node-count caps and zero-table fallback. The certificate imposes no size bound relative to B.

Sokoban. Location, direction and box types must be declared, and location and box types must be exact. Goals are duplicate-free and dynamic, with at most one `at` destination per box. Other dynamic goals are permitted, and neither a goal nor an initial location is required for every box. The robot initially occupies exactly one square, which is clear. Each box initially occupies at most one square, each square contains at most one box, and box-occupied squares are not clear. Static adjacency forbids self-loops and requires reverse edges with exactly typed direction objects. For n locations, checked push and walk distances use cap n, and product distances use cap n^2, with zero-table fallback. There is no certificate bound relative to B.

Distance-Table Checks. The shared graph-distance check verifies table dimension bounds, bounded entries, a zero diagonal, edge consistency and a cap at least as large as the node count. It also checks the following property for every edge from u to v and every target t: if the entry from v to t is below the cap, then the entry from u to t is also below the cap. Sokoban's product check requires n^2 entries, zero at valid goal configurations, bounded entries, edge consistency and the same backward-propagation property. These checks connect the finite tables to the distance properties used by the domain proofs. Failure selects zero tables during compilation rather than rejecting the task.

D Experiment Records

The dataset contains all 8 550 recorded configuration–task runs, including the 5 850 runs of the five configurations reported here, with coverage, cost, search and total time, expansions, resource limits, certificate flags and whether expansions came from an interrupted run's checkpoint. The IPC 2023 snapshot contains 7 200 runs, and a separate snapshot adds 1 350 runs on the new domains, retaining their original algorithm identifiers. Every reported configuration has a run for each of the 90 test tasks per domain, and all 1 157 solved runs of the reported configurations carry expansion measurements from completed runs. We exclude checkpoint counts from the expansion comparisons.

On IPC 2023, the three-way intersection of the heuristic configurations has 198 tasks, the LeanPlan–SCP pair 211 and the pair of blind configurations 154. Figure 3 uses this suite only, while Table 1 includes both suites with separate totals. The runtime curves use `total_time` for every configuration. Only solved runs contribute, so each curve ends at full-budget coverage. We draw times below one millisecond at one millisecond on the logarithmic axis, whose values are per-task thresholds rather than a cumulative budget.

For LeanPlan, total time is wall-clock time from before PDDL reading through search, covering validation, grounding and heuristic setup, and excluding plan output and external VAL checking. Search time runs from the end of grounding until search returns, including certificate checking and heuristic setup. LeanPlan counts a state as expanded when it generates its successors, excluding stale queue entries and the popped goal. Each planner reports expansions under its own definition.

The footnote in Section 5 specifies SCP's component budgets, selected in earlier sweeps on the test suite. For LeanPlan, Section 3 gives the generation protocol, its acceptance checks and its selection rule.

E New Domains

Orrery. Orrery models the alignment of coupled wheels in a clockwork model of the solar system. Each wheel has K discrete angular positions, and a twist advances one wheel by one position while retreating a coupled wheel by one position, modulo K. The goal aligns every wheel with its reference mark using as few twists as possible. Every twist preserves the sum of wheel positions modulo K, so correcting one wheel changes another and may require routing displacement through several couplings.

Cisterns. Cisterns models water redistribution through a network of hoses. An action transfers one unit of water between connected cisterns when the source holds at least two units more than the destination. Goals prescribe exact water levels in selected cisterns, and plans minimize the number of transfers. Transfers conserve water but strictly decrease the sum of squared levels, making the state transitions irreversible. Equalizing levels too early can remove the gradients needed to reach a goal.

Qubit Routing. Qubit Routing models the placement of logical qubits on a hardware coupling grid while executing an ordered sequence of two-qubit gates. A swap exchanges the logical qubits at adjacent grid nodes; a gate can execute only when its two operands occupy adjacent nodes. The goal executes every gate in order, with no required final qubit placement. Both swaps and gate executions have unit cost, so the fixed number of gate executions makes minimizing plan length equivalent to minimizing swaps. The placement left after one gate affects the routing needed by later gates.

F PDDL Input Example

Figure 5 shows a Miconic domain excerpt and a complete task. The `board` action puts a passenger aboard when the lift reaches their origin floor. The certificate checks additional assumptions about the task before LeanPlan uses the domain heuristic.

(a) Miconic domain excerpt.

```
(define (domain miconic)
  (:requirements :strips :typing)
  (:types passenger - object
          floor - object)
  ; Predicate declarations omitted.
  (:action board
    :parameters (?f - floor
                 ?p - passenger)
    :precondition (and (lift-at ?f)
                       (origin ?p ?f))
    :effect (and (boarded ?p)
                 (not (origin ?p ?f))))
  ; Other actions omitted.
)
```

(b) Miconic task `easy-p01.pddl`.

```
(define (problem miconic-01)
  (:domain miconic)
  (:objects p1 - passenger
            f1 f2 f3 f4 - floor)
  (:init (lift-at f1)
    (origin p1 f2)
    (destin p1 f3)
    (above f1 f2) (above f1 f3)
    (above f1 f4) (above f2 f3)
    (above f2 f4) (above f3 f4))
  (:goal (and (served p1))))
```

Figure 5: PDDL from the Miconic benchmark used in our experiments. The domain excerpt shows the boarding action and omits predicate declarations and other actions. The complete task asks the lift to carry passenger `p1` from floor `f2` to `f3`.