LLM-Evolved Pattern Generators for Optimal Classical Planning

Windy Phung, Dominik Drexler, Arnaud Lequen, Jendrik Seipp

Linköping University, Sweden
firstname.lastname@liu.se

Abstract

Learned heuristics have recently become a competitive alternative to traditional domain-independent heuristics for satisficing planning. Existing approaches, however, focus on improving search guidance rather than guaranteeing admissibility, which makes them unsuitable for optimal classical planning. We present the first method for learning domain-dependent heuristics that are admissible by design and thus preserve the optimality guarantees of A^* search. Instead of learning a direct mapping from states to heuristic values, we learn to construct abstractions that induce admissible heuristics. We use an LLM-driven evolutionary program-synthesis framework to obtain, for each domain, a program that produces a pattern collection for any task in that domain, and we combine the resulting patterns admissibly via saturated cost partitioning. Empirically, the learned programs encode interpretable domain-specific insights, run with negligible overhead at test time and yield heuristics that match the coverage of state-of-the-art domain-independent baselines on several domains while evaluating each state substantially faster.

1 Introduction

Classical planning aims to find a plan, a sequence of actions whose execution leads from the initial state to the goal, in a deterministic and fully observable environment. We are interested in *optimal* plans, i.e., plans of minimum cost. A^* search with an admissible heuristic is one of the most successful approaches to finding optimal plans (Hart, Nilsson, and Raphael 1968), and its effectiveness depends on the quality of the heuristic. The strongest admissible heuristics today are *domain-independent*: they derive estimates from a task in isolation, without prior knowledge of its domain (Helmert and Domshlak 2009; Seipp, Keller, and Helmert 2020). Tasks from the same domain, however, share regularities that a *domain-dependent* heuristic could exploit. Existing methods do learn such heuristics from solved instances of a domain (Toyer et al. 2018; Ståhlberg, Bonet, and Geffner 2022; Chen, Thiébaux, and Trevizan 2024), but they sacrifice admissibility and thus the optimality guarantees of A^*. We close this gap by learning domain-dependent heuristics that are admissible by construction.

Pattern-database heuristics (PDBs) are a core ingredient in the strongest admissible heuristics (Edelkamp 2001; Haslum et al. 2007; Sievers, Ortlieb, and Helmert 2012). Each pattern projects the task onto a small subset of state variables and abstracts away the rest. Goal distances in the resulting abstract state space are admissible estimates for the original task. The strongest admissible PDB heuristics combine the estimates of a *collection* of patterns (Pommerening, Röger, and Helmert 2013; Seipp 2019). Larger patterns yield more informative estimates, but the number of abstract states they induce grows exponentially with their size, so each pattern must trade information against the cost of computing and querying it.

In this paper, we learn domain-specific *pattern generators* from example tasks. A generator is a Python program that maps a task description to a pattern collection. We synthesize such generators with OpenEvolve (Sharma 2025), an LLM-driven evolutionary framework, using planning performance on small training tasks as the optimization signal. At test time, we apply the learned generator directly to new tasks from the same domain, producing pattern collections with negligible overhead.

To combine the resulting PDB heuristics admissibly, we use *saturated cost partitioning* (SCP) (Seipp, Keller, and Helmert 2020), one of the strongest methods for this purpose. SCP distributes action costs across the patterns in a collection so that the sum of their estimates remains admissible. Existing pattern generators search the space of patterns separately for each task and cannot reuse the patterns that worked on one task when faced with another from the same domain. We instead learn generators that capture the domain structure observed across example tasks.

We evaluate our learned generators against five baselines on seven domains from the optimal Autoscale benchmark set (Torralba, Seipp, and Sievers 2021). The synthesized generators match or exceed the best baseline in five domains, achieve the highest coverage in two and evaluate each state substantially faster than the strongest baselines. These results show that learning domain-specific pattern generators is a promising direction for improving the performance of optimal planners, and that LLM-driven program synthesis is a powerful tool for this purpose.

2 Background

We introduce transition systems, optimal classical planning, heuristics, projections, saturated cost partitioning and OpenEvolve. Throughout, $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$ denotes the extended real numbers.

Transition Systems. A *transition system* $\mathcal{T}$ consists of a set of states $S(\mathcal{T})$, a set of labels $L(\mathcal{T})$, a set of transitions $T(\mathcal{T})$ of the form $\langle s, \ell, s' \rangle$ with $s, s' \in S(\mathcal{T})$ and $\ell \in L(\mathcal{T})$, an initial state $s_0(\mathcal{T}) \in S(\mathcal{T})$ and a set of goal states $S_*(\mathcal{T}) \subseteq S(\mathcal{T})$. An s-path in a transition system $\mathcal{T}$ is a sequence $\pi = \langle s_1, \ell_1, s_2, \ldots, s_n, \ell_n, s_{n+1} \rangle$ with $s = s_1$ and $\langle s_i, \ell_i, s_{i+1} \rangle \in T(\mathcal{T})$ for all $i = 1, \ldots, n$. It is an s-goal-path if $s_{n+1} \in S_*(\mathcal{T})$.

A *weighted transition system* is a tuple $\langle \mathcal{T}, \mathit{cost} \rangle$, where $\mathcal{T}$ is a transition system and $\mathit{cost} : L(\mathcal{T}) \to \overline{\mathbb{R}}$ is a cost function. We write $\mathcal{C}(L(\mathcal{T}))$ for the set of all such cost functions. The cost of a path $\pi = \langle s_1, \ell_1, s_2, \ldots, s_n, \ell_n, s_{n+1} \rangle$ is $\mathit{cost}(\pi) = \sum_{i=1}^{n} \mathit{cost}(\ell_i)$. The goal distance $h^\star_{\mathcal{T}}(\mathit{cost}, s)$ in $\langle \mathcal{T}, \mathit{cost} \rangle$ is the minimum cost of any s-goal-path.

Optimal Classical Planning. A lifted planning task consists of a domain and a problem instance specified in first-order logic (McDermott et al. 1998; Haslum et al. 2019). The domain defines predicate symbols and action schemas, and the problem instance defines the objects, the initial state and the goal. Instantiating the schemas over the objects yields ground atoms and actions, and standard PDDL normalization and translation (Helmert 2009) turn the result into a finite-domain ground planning task. We work with collections of lifted tasks that share a single domain; predicates and action schemas are therefore common to all tasks in a collection, while the initial state and the goal vary.

A (ground) *classical planning task* is a tuple $\Pi = \langle V, A, \mathit{cost}, I, \gamma \rangle$, defined as follows. V is a set of binary variables v, each with domain $D_v = \{\text{true}, \text{false}\}$. We use binary rather than finite-domain variables so that each variable corresponds to a single propositional fact. Our hypothesis is that this makes it easier for the LLM to learn generators that generalize across tasks. A *partial variable assignment* is a function $\sigma : V' \to \bigcup_{v \in V} D_v$ with $V' \subseteq V$ that satisfies $\sigma(v) \in D_v$ for all $v \in V'$; it is *complete* if V' equals V. A is a set of actions, each of the form $\langle \mathit{pre}, \mathit{eff} \rangle$ where pre and eff are partial variable assignments, and $\mathit{cost} : A \to \mathbb{R}_0^+$ assigns each action a real-valued, non-negative cost. A *state* is a complete variable assignment; I is the initial state and γ is a partial variable assignment describing the goal. An action $\langle \mathit{pre}, \mathit{eff} \rangle$ is *applicable* in a state s iff $s[v] = \mathit{pre}(v)$ for every v on which pre is defined. Applying a to s then yields the successor state $s[\![a]\!] = s'$, with $s'[v] = \mathit{eff}(v)$ if $\mathit{eff}(v)$ is defined and $s'[v] = s[v]$ otherwise.

A task $\Pi = \langle V, A, \mathit{cost}, I, \gamma \rangle$ induces the *weighted transition system* $\langle \mathcal{T}, \mathit{cost} \rangle$, where $S(\mathcal{T})$ is the set of all states over V, $L(\mathcal{T}) = A$ and $\langle s, a, s' \rangle \in T(\mathcal{T})$ iff $s' = s[\![a]\!]$. The initial state is $s_0(\mathcal{T}) = I$, and $s \in S_*(\mathcal{T})$ iff $\gamma(v) = s[v]$ for every v on which γ is defined. The objective of optimal classical planning is to find an s_0-goal-path of minimum cost in $\langle \mathcal{T}, \mathit{cost} \rangle$.

Heuristics. A *heuristic* is a function $h_{\mathcal{T}} : \mathcal{C}(L(\mathcal{T})) \times S(\mathcal{T}) \to \overline{\mathbb{R}}$ that estimates the cost of a minimum s-goal-path for a state $s \in S(\mathcal{T})$ under a given cost function. A heuristic $h_{\mathcal{T}}$ is *admissible* if $h_{\mathcal{T}}(\mathit{cost}, s) \leq h^\star_{\mathcal{T}}(\mathit{cost}, s)$ for all $\mathit{cost} \in \mathcal{C}(L(\mathcal{T}))$ and all $s \in S(\mathcal{T})$. The s-goal-path returned by A* search with an admissible heuristic has minimum cost (Hart, Nilsson, and Raphael 1968). We therefore aim to construct accurate admissible heuristics for optimal classical planning.

Projections. Projections that simplify the task are a natural source of admissible heuristics (Culberson and Schaeffer 1996; Edelkamp 2001). A *pattern* $P \subseteq V$ is a subset of the task's variables. Let S denote the set of states over V and S' the set of states over the variables in P. The pattern induces a *projection* $\alpha : S \to S'$ with $\alpha(s) = s'$ iff $s[v] = s'[v]$ for all $v \in P$. This projection in turn induces an abstract transition system $\mathcal{T}'$ of the transition system $\mathcal{T}$ of Π, defined by (1) $\alpha(s_0(\mathcal{T})) = s_0(\mathcal{T}')$, (2) $\langle \alpha(s), \ell, \alpha(s') \rangle \in T(\mathcal{T}')$ whenever $\langle s, \ell, s' \rangle \in T(\mathcal{T})$ and (3) $\alpha(s) \in S_*(\mathcal{T}')$ whenever $s \in S_*(\mathcal{T})$. Setting $h_{\mathcal{T}}(\mathit{cost}, s)$ to the minimum cost of an $\alpha(s)$-goal-path in $\langle \mathcal{T}', \mathit{cost} \rangle$ yields an admissible heuristic. This heuristic is uninformative when the pattern is too small and intractable to compute when it is too large. The standard remedy is to construct a *collection* of patterns and combine their estimates admissibly with cost partitioning.

Saturated Cost Partitioning. A *cost partitioning* (Katz and Domshlak 2010) of a cost function $\mathit{cost} \in \mathcal{C}(L(\mathcal{T}))$ is a tuple $\langle \mathit{cost}_1, \ldots, \mathit{cost}_n \rangle \in \mathcal{C}(L(\mathcal{T}))^n$ such that $\sum_{i=1}^{n} \mathit{cost}_i(\ell) \leq \mathit{cost}(\ell)$ for all $\ell \in L(\mathcal{T})$. Given a collection of abstraction heuristics $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$, the corresponding *cost partitioning heuristic* is $h^{\mathrm{CP}}(\mathit{cost}, s) = \sum_{i=1}^{n} h^\star_{\mathcal{T}_i}(\mathit{cost}_i, s)$, where $\mathcal{T}_i$ is the abstract transition system underlying h_i.

Saturated cost partitioning (SCP) (Seipp, Keller, and Helmert 2020) is a greedy method for computing such cost partitionings. Given a sequence of heuristics $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$, it constructs a cost partitioning $\langle \mathit{cost}_1, \ldots, \mathit{cost}_n \rangle \in \mathcal{C}(L(\mathcal{T}))^n$ via the recurrence

$$
\begin{aligned}
&\mathit{rem}_0 = \mathit{cost} \\
&\mathit{cost}_i = \mathit{saturate}(h_i, \mathit{rem}_{i-1}) && \text{for all } 1 \leq i \leq n \\
&\mathit{rem}_i = \mathit{rem}_{i-1} - \mathit{cost}_i && \text{for all } 1 \leq i \leq n,
\end{aligned}
$$

where infinite remaining costs are sticky ($\mathit{rem}_{i-1}(\ell) = \infty$ implies $\mathit{rem}_i(\ell) = \infty$), and $\mathit{saturate}$ returns the *minimum saturated cost function* mscf_i for heuristic h_i with abstract transition system $\mathcal{T}_i$ under remaining costs rem_{i-1}:

$$
\mathit{mscf}_i(\ell) = \sup_{\substack{\langle s, \ell, s' \rangle \in T(\mathcal{T}_i) \\ \text{s.t. } h^\star_{\mathcal{T}_i}(\mathit{rem}_{i-1}, s) < \infty}} \left(h^\star_{\mathcal{T}_i}(\mathit{rem}_{i-1}, s) - h^\star_{\mathcal{T}_i}(\mathit{rem}_{i-1}, s') \right)
$$

Standard enhancements include using multiple diverse orders (Seipp, Keller, and Helmert 2017b) and saturating over only a subset of states (Seipp and Helmert 2019). The exact details are not crucial here because we use the same state-of-the-art configuration of SCP throughout.

OpenEvolve. OpenEvolve (Sharma 2025) is an open-source framework for evolving programs with large language models (LLMs), modeled after DeepMind's AlphaEvolve (Novikov et al. 2025). A *program* p is executed to produce outputs; a quality function $q(p) \in \mathbb{R}$ scores those outputs, and a feature extractor returns a b-dimensional *feature*

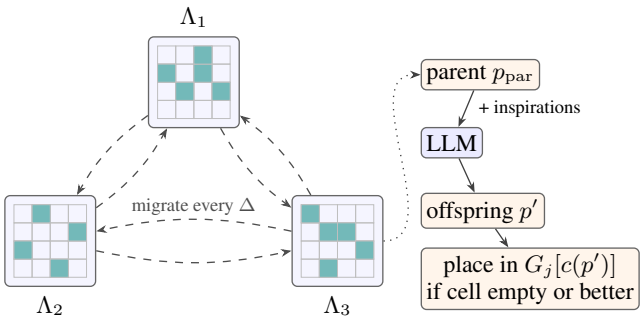

Figure 1: One iteration of OpenEvolve. Each island Λ_j holds a MAP-Elites grid; filled cells contain elite programs. A parent is sampled from the current island, the LLM is prompted with the parent together with the top programs of Λ_j and *inspirations* drawn from cells near $c(p_{\text{par}})$, and the resulting offspring replaces the occupant of its grid cell if it scores higher. Every Δ iterations, the top ρ-fraction of each island migrates to its two ring neighbors (dashed arrows).

vector $\phi(p) \in \mathbb{R}^b$ that describes the program's *behavior*, measurable characteristics that tell programs apart, such as the number of node expansions or the search time, separately from its quality.

The main loop of OpenEvolve is illustrated in Figure 1. At its core, OpenEvolve runs MAP-Elites (Mouret and Clune 2015), a *quality-diversity* algorithm that, rather than converging on a single best program, retains a diverse set of high-quality programs by keeping the best one found for each region of behavior space. To this end, it discretizes the b-dimensional behavior space into a grid G of cells, each holding at most one program, the cell's *elite*. Each program p is assigned to a cell $c(p) \in \mathbb{Z}^b$ determined by its feature vector $\phi(p)$: the raw features are first scaled to $[0, 1]$ via online min–max normalization, then mapped to discrete bin indices. A newly generated program p' competes only with the elite of $G[c(p')]$ and replaces it whenever the cell is empty or $q(p') > q(G[c(p')])$.

To prevent premature convergence and broaden exploration, OpenEvolve adopts the *island model* from evolutionary computation, splitting the population into k semi-isolated subpopulations, the *islands* $\Lambda_1, \ldots, \Lambda_k$, each evolving on its own MAP-Elites grid G_j. The islands form a ring: every Δ iterations, the top fraction ρ of each island's programs migrate to its two neighbors $G_{(j-1) \bmod k}$ and $G_{(j+1) \bmod k}$, so that strong programs spread across islands while each island still explores a different region of program space.

Within the chosen island Λ_j, a parent p_{par} is drawn by a mixed strategy: with probability $\Pr_{\text{explore}}$ uniformly at random from the island (exploration), with probability $\Pr_{\text{exploit}}$ from a global archive of elite programs (exploitation) and otherwise by fitness-weighted sampling. The LLM is then prompted with p_{par}'s code together with the top-scoring programs of Λ_j. It returns an offspring p' that builds on the strengths of these examples while preserving population diversity. Algorithm 1 summarizes the full procedure.

Algorithm 1: The OpenEvolve evolutionary loop maintains k islands and selects, mutates and replaces programs across them.

Require: LLM, quality function q, feature extractor ϕ, number of islands k, migration interval Δ, migration fraction ρ, iterations N
Ensure: Best program p^*

```
Initialize grids G_1, ..., G_k; seed G_1[c(p_0)] ← p_0
for i = 1, ..., N do
    j ← island with free capacity
    Select parent p_par from G_j
    Query LLM with p_par, top(G_j) and inspirations near c(p_par) in G_j to get offspring p'
    Execute p'; compute c(p') and q(p')
    if G_j[c(p')] = ∅ or q(p') > q(G_j[c(p')]) then
        G_j[c(p')] ← p'
    if i mod Δ = 0 then
        for all islands j' ∈ {1, ..., k} do
            Migrate top ρ-fraction of G_{j'} into G_{(j'-1) mod k} and G_{(j'+1) mod k}
return p* = arg max_{p ∈ ⋃_j G_j} q(p)
```

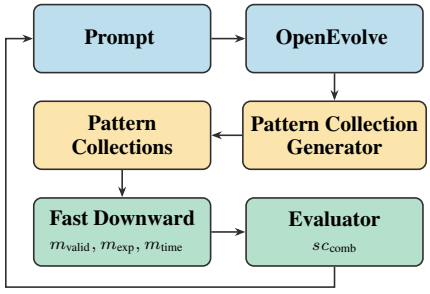

Figure 2: The evolutionary pipeline. The engine (blue) creates artifacts (orange) that are assessed by the evaluation components (green).

3 Learning to Generate Pattern Collections

For each domain, we learn one *pattern collection generator*: a Python function that maps a task from the domain to a pattern collection for that task. Framing the output as a program rather than a fixed collection lets a single generator handle tasks of any size within its domain at test time. We use the OpenEvolve framework to iteratively improve candidate generators for the domain via LLM-guided mutation, with a scoring function that measures planning performance on a small set of training tasks from the domain. Figure 2 gives an overview of the resulting pipeline; we describe its components below.

Pattern Collection Generator. The pattern collection generator takes *task information* as input and returns a list of patterns. The task information provides a structured view of the planning task: static atoms describe its time-invariant properties, fluent atoms describe its dynamic properties, and the initial and goal conditions are given as the atoms that hold in the respective (partial) states. In effect, it conveys

the entire problem instance through its ground atoms; only the grounded actions are omitted. We leave them out to keep token usage low: each grounded action is an instantiation of a lifted action schema with the task's objects, and both ingredients are already provided to the LLM.

Scoring. To evaluate a generator, we use the combined score sc_{comb} as the quality function q for Algorithm 1. It is the average of per-problem scores across the set of training tasks $\mathcal{D}$:

$$sc_{\text{comb}} = \frac{1}{|\mathcal{D}|} \sum_{t \in \mathcal{D}} sc_{\text{problem}}(t)$$

The per-problem score combines a validity indicator with two efficiency terms:

$$sc_{\text{problem}} = sc_{\text{valid}} \cdot (1 + w_{\text{exp}} \cdot sc_{\text{exp}} + w_{\text{time}} \cdot sc_{\text{time}}).$$

sc_{valid} is 1 if and only if the generated collection is valid, which means that it stays within the per-pattern and collection-size limits that keep PDB construction inside the memory budget (Section 4). Otherwise, sc_{valid} is 0. An invalid collection thus scores 0, below any valid one. A valid collection scores at least the base value of 1, to which the efficiency terms sc_{exp} and sc_{time} add a bonus for solving the task with fewer node expansions and less search time. A valid collection that does not solve the task within the limits contributes $sc_{\text{exp}} = sc_{\text{time}} = 0$ and gets the base score of 1.

We turn each raw metric m_{metric}, the node expansions or the search time, into a score by log-scaling it between a lower bound lo and an upper bound hi:

$$sc_{\text{metric}} = \frac{\log m_{\text{metric}} - \log hi}{\log lo - \log hi}.$$

The score equals 1 when the metric reaches the optimistic target lo and 0 when it reaches hi. We accordingly set hi to the resource limit, so a task solved only at the edge of the time or expansion budget scores near 0, and lo to a value a strong solver can realistically reach. We scale logarithmically because expansions and times span several orders of magnitude: the logarithm rewards each order-of-magnitude improvement equally, whereas a linear scale would let the largest tasks dominate the score. In our experiments we bound expansions to $[100, 10^6]$ and search time to $[1, 180]$ seconds, the upper time bound coinciding with the 180-second evaluation limit (Section 4). We pick these by hand rather than tuning them for each domain; the logarithm keeps the score stable under moderate changes, so their precise values are not critical.

The weights w_{exp} and w_{time} balance the two efficiency terms. We set both to 1, valuing expansion- and time-efficiency equally, and do not tune them further. Their values affect only the bonus among collections that already solve the task: whatever the weights, an invalid collection scores 0, a valid but unsolved one scores 1 and a solved one scores more than 1. The weights thus cannot reorder these three cases; they only adjust how the score trades off fewer node expansions against shorter search time.

Pipeline. We seed OpenEvolve with a single prompt that gives the LLM (i) the domain definition in PDDL form, (ii) the required function signature `generate_pattern_collection(task_info: TaskInformation) -> list[Pattern]`, (iii) task information for three tasks from the domain's training set (the easiest, the median and the hardest by difficulty) and (iv) a naive baseline implementation that places each goal atom into its own pattern. Listing 1 declares the data structures available to the generator: `Objects`, `Predicates` and `GroundAtoms` describe the planning task; a `TaskInformation` bundles the static, initial-state and goal atoms together with all fluent atoms of the task; and a `Pattern` is a list of ground atoms whose joint projection defines the state space of the corresponding PDB heuristic.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Object:
    name : str

@dataclass(frozen=True)
class Predicate:
    name : str
    arity : int

@dataclass(frozen=True)
class GroundAtom:
    predicate: Predicate
    binding: tuple[Object]

    def __post_init__(self):
        assert len(self.binding) ==
    self.predicate.arity

@dataclass(frozen=True)
class TaskInformation:
    static_ground_atoms : tuple[GroundAtom]
    fluent_initial_state_atoms :
    tuple[GroundAtom]
    fluent_goal_atoms : tuple[GroundAtom]
    all_fluent_atoms : tuple[GroundAtom]

@dataclass
class Pattern:
    pattern : list[GroundAtom]

def generate_pattern_collection(
    task_info : TaskInformation
) -> list[Pattern]:
    pass
```

Listing 1: Classes provided in the prompt to the LLM.

From this prompt, OpenEvolve evolves candidate generators. To evaluate a candidate, we apply it to each training task in order of increasing difficulty and pass the resulting pattern collection to Fast Downward (Helmert 2006), which returns the metrics needed to compute the score. Before each

run, the collection is validated against size and memory constraints (detailed in Section 4). A task is marked unsolved if the collection violates these constraints, contains a syntactic error or exhausts the time or memory limit, at which point evaluation terminates and the remaining tasks receive a default score. This early termination quickly discards generators that fail on easy instances, focusing evolutionary effort on promising candidates.

4 Experiments

Benchmark Sets. We train one generator per domain on a subset of the benchmarks from the Learning track of the International Planning Competition (IPC) 2023 (Taitler et al. 2024), and we use the Autoscale benchmark set (Torralba, Seipp, and Sievers 2021) as the test set, which provides instances of increasing size. We restrict ourselves to the domains that appear in both sets: Blocksworld, Childsnack, Floortile, Miconic, Rovers, Satellite and Transport. We exclude Sokoban because its PDDL encoding differs between the two sets.

For training, we use every task in the "easy" training set and every third task from the "easy", "medium" and "hard" test sets, for a total of 129 tasks per domain. This is large enough to capture domain diversity, yet small enough to keep the evolutionary loop fast.

OpenEvolve Configuration. The evolutionary algorithm runs on 3 islands for 100 iterations using Kimi K2.5 (Kimi Team 2026) as the underlying LLM, a capable open-weight model available at the time of our experiments. Each candidate is scored with the function from Section 3 ($w_{\text{exp}} = w_{\text{time}} = 1$; bounds $[100, 10^6]$ on expansions and $[1, 180]$ seconds on search time). Island placement uses a 3-dimensional grid over average expansions before the last f-layer, average search time and coverage.

Evaluation. We first validate each generated collection: it must contain at most 20 patterns, and each pattern's state space must not exceed $5 \cdot 10^6$ states, which keeps PDB computation within a 2 GiB memory budget. We then evaluate validated candidates with the Scorpion planner (Seipp 2024) under a 3-minute time limit and 4 GiB of memory per task. All heuristics in our experiments share the same state-of-the-art SCP configuration: a diverse set of greedy orders (Seipp, Keller, and Helmert 2017a) computed online during search (Seipp 2021) using the perim* saturator (Seipp and Helmert 2019), with a 10-second budget for order generation and a new order computed every 1000 expansions. For each domain, we evaluate the final best generator on 30 test tasks and compare the resulting heuristic, h^{evo}, against five baselines.

The first four baselines rely on systematic enumeration of *interesting* patterns, i.e., patterns that cannot be replaced by a set of smaller patterns that are equally informative. For $n \in \{1, 2, 3\}$, SYS-n generates all interesting patterns of size up to n (Pommerening, Röger, and Helmert 2013) and yields the heuristic h^{SYS}_n. SYS-SCP exhaustively generates interesting patterns and keeps only those that can increase the estimate of the saturated cost partitioning heuristic (Seipp 2019); we denote the resulting heuristic $h^{\text{SYS}}_{\text{SCP}}$.

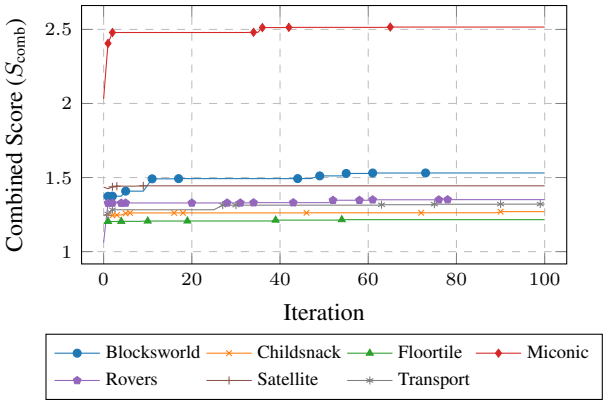

Figure 3: Combined score per iteration for all domains. Markers indicate iterations where a new best score was achieved. In most domains much of the improvement happens early, but refinement continues throughout the 100 iterations. For each domain, the point at iteration 0 shows the score of the patterns generated by SYS-1.

The fifth baseline takes the patterns produced by h^{evo} and, in each pattern, replaces every non-goal atom with a uniformly random non-goal fluent atom of the task; goal atoms and pattern sizes are preserved. This isolates the contribution of the evolutionary search: if the atom selection in our patterns were no better than random, the resulting heuristic h^{rand} would perform comparably to h^{evo}.

4.1 Evolutionary Process

Figure 3 shows how the combined score of the best program evolves over 100 iterations for each domain. In most domains, a large share of the improvement occurs within the first few iterations, as the LLM quickly converges from the initial singleton baseline to a domain-tailored strategy; in others, such as Blocksworld and Transport, the gains accrue more gradually. Evolution, however, keeps refining the program beyond this initial phase. Satellite stands out: its best generator appears already at iteration 9, suggesting that effective pattern structure is easy to discover here. For every other domain, the best generator appears at or after iteration 50, with the latest breakthroughs in Childsnack and Transport at iterations 90 and 99, respectively. These domains require sustained exploration to refine their pattern structure.

Overall, the absolute score improvements range from $+0.009$ in Satellite to $+0.484$ in Miconic, confirming that evolution contributes meaningfully in most domains, even when diminishing returns set in early for simpler ones.

4.2 Results

Coverage Table 1 shows per-domain coverage. h^{evo} matches or exceeds the best systematic baseline in five of the seven domains: it is uniquely best on Childsnack and Transport, and tied with the strongest baseline on Blocksworld, Rovers and Satellite. The two exceptions are Floortile and Miconic, where systematic enumeration of small patterns dominates and our approach fails to discover comparably

Domain	Baselines				Ours	
	h_1^{SYS}	h_2^{SYS}	h_3^{SYS}	$h_{\text{SCP}}^{\text{SYS}}$	h^{rand}	h^{evo}
Blocksworld (30)	12	**16**	**16**	**16**	12	**16**
Childsnack (30)	4	4	4	4	4	**5**
Floortile (30)	4	5	5	**7**	5	5
Miconic (30)	3	6	**20**	19	3	4
Rovers (30)	**2**	**2**	**2**	**2**	**2**	**2**
Satellite (30)	5	15	**16**	15	5	**16**
Transport (30)	10	10	12	11	10	**13**
Sum (210)	40	58	**75**	74	41	61

Table 1: Per-domain and overall coverage scores for each heuristic with the best scores highlighted in bold.

effective larger patterns. Pairwise per-domain, h^{evo} ties or beats h_3^{SYS} on six of seven domains (two wins, four ties, one loss) and $h_{\text{SCP}}^{\text{SYS}}$ on five (three wins, two ties, two losses); the losses concentrate in Floortile and Miconic.

Aggregating across domains, h^{evo} solves 61 of 210 tasks against 75 for h_3^{SYS} and 74 for $h_{\text{SCP}}^{\text{SYS}}$, but the entire gap stems from Miconic, where h^{evo} solves only 4 tasks while the two strongest baselines solve 20 and 19. Excluding Miconic, the totals reverse to 57 tasks for h^{evo} against 55 for both h_3^{SYS} and $h_{\text{SCP}}^{\text{SYS}}$.

The random ablation h^{rand} solves 41 tasks: it ties h^{evo} only on Floortile and Rovers, loses on every other domain, and never beats it. This confirms that the evolutionary process identifies meaningful structural patterns rather than merely selecting atoms at random.

Expansions Before the Last f-Layer Figure 4 shows guidance quality, measured in expansions before the last f-layer. h^{evo} dominates h_1^{SYS} on all tasks solved by both heuristics, and needs fewer expansions than h_2^{SYS} on 34 tasks out of 44. The balance shifts in favor of the systematic baselines at larger pattern sizes: against h_3^{SYS} and $h_{\text{SCP}}^{\text{SYS}}$, h^{evo} performs fewer expansions on only 15 tasks out of 55 and 11 tasks out of 59, respectively. This reveals the core trade-off of our synthesized pattern generators: exhaustive systematic enumeration at size 3 yields stronger per-state guidance than our domain-specific collections, but our generators compensate through faster evaluation.

Total Search Time Figure 5 shows total search time, which includes the computation of all PDBs and the time spent in search. Although h^{evo} provides weaker per-state guidance than the strongest baselines, it is faster overall in most comparisons. It beats h_1^{SYS} on 52 tasks out of 58, as the singleton patterns of h_1^{SYS} are efficient to evaluate but provide little guidance. Against h_3^{SYS}, h^{evo} is faster on 54 tasks out of 76, and against $h_{\text{SCP}}^{\text{SYS}}$, on 60 tasks out of 78.

$h_{\text{SCP}}^{\text{SYS}}$ spends up to 100 s per task on pattern generation, which is its bottleneck. h^{evo} shifts that cost to a one-time offline evolution; at test time, the learned generator produces a pattern collection in negligible time.

Time per Evaluation Figure 6 isolates per-state evaluation cost and reveals the mechanism behind the total-time

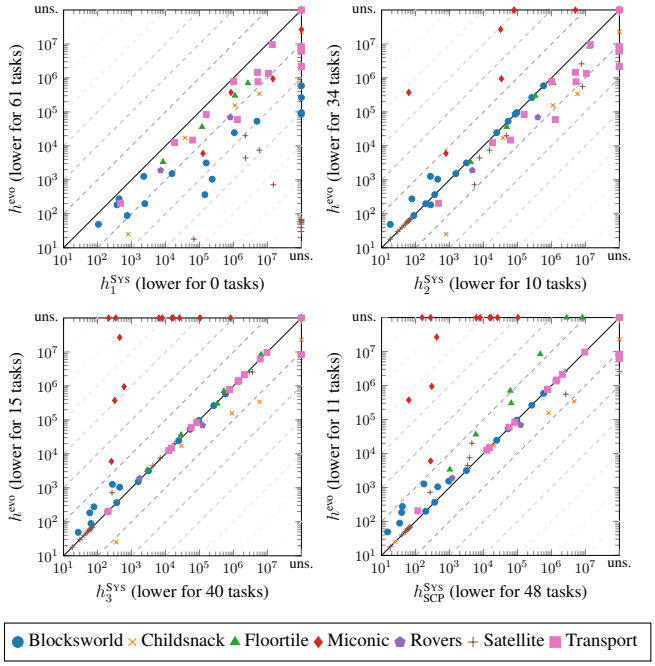

Figure 4: Expansions until last f-layer: h^{evo} vs. each baseline. Points below the diagonal indicate tasks where h^{evo} requires fewer expansions.

advantage. h^{evo} is faster than h_3^{SYS} on 56 tasks out of 75, and faster than $h_{\text{SCP}}^{\text{SYS}}$ on 50 tasks out of 75. Although the patterns in h^{evo} are individually larger, the collection generally contains fewer patterns, which reduces the number of lookups per state evaluation. Across the 40 tasks solved by all five approaches, h^{evo} produces 1230 patterns in total, compared to 5998 for $h_{\text{SCP}}^{\text{SYS}}$, 49 157 for h_3^{SYS}, 2028 for h_2^{SYS} and 267 for h_1^{SYS}. This per-state efficiency is the primary reason h^{evo} matches or exceeds stronger heuristics in most domains, despite being less informative and requiring more expansions on average, and it is precisely what gives our patterns an edge over the strongest systematic baselines. The exception is again Miconic, where the per-state evaluation cost of h_3^{SYS} and $h_{\text{SCP}}^{\text{SYS}}$ is more than offset by the information their heuristics provide.

4.3 Example Pattern Collection Generators

In this section, we describe pattern collection generators synthesized for three representative domains and analyze their empirical behavior in more depth.

Childsnack. In Childsnack, sandwiches must be assembled and delivered to children waiting at tables, some of whom require gluten-free sandwiches. The pattern generator found by our approach produces seven pattern types: *goal singletons* (1 atom); *tray location mutexes* (2–4 atoms); *kitchen resource pools* grouping all ingredients in the kitchen (2–20 atoms); *sandwich life-cycle patterns* tracking one sandwich through its states together with its gluten status (5–13 atoms); *gluten-allergy constraints* aggregating

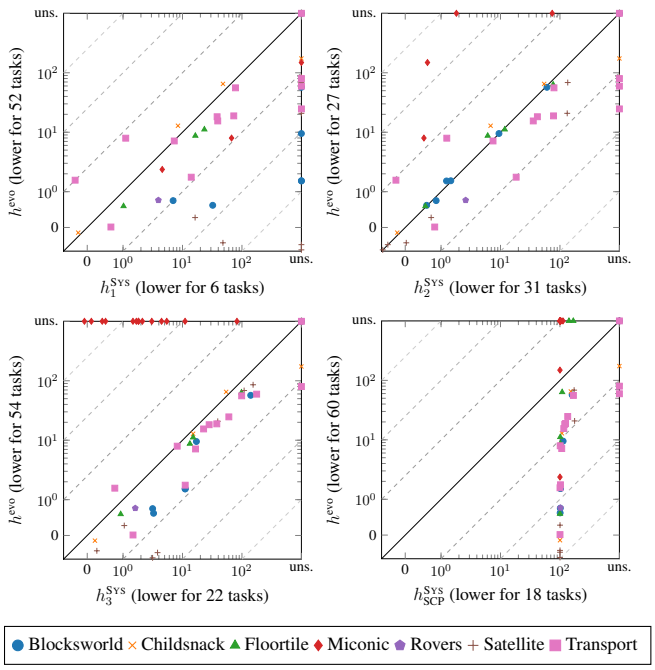

Figure 5: Total search time: h^{evo} vs. each baseline. Points below the diagonal indicate tasks where h^{evo} is faster.

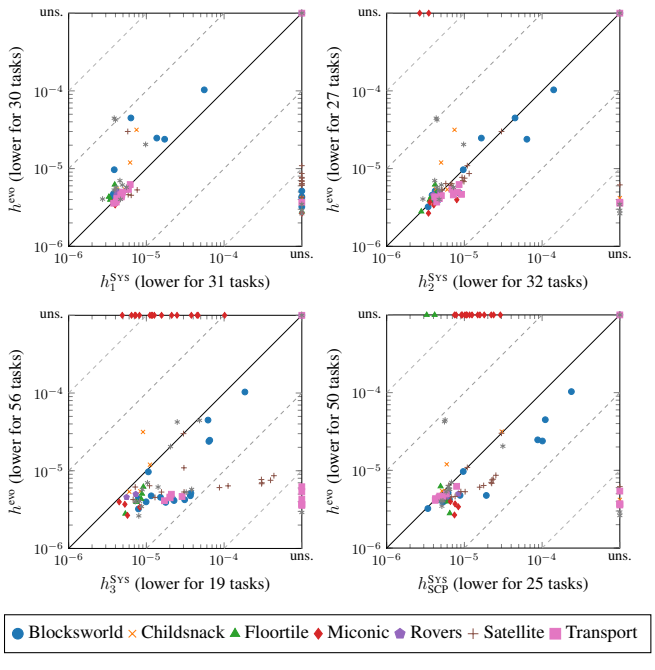

Figure 6: Time per heuristic evaluation: h^{evo} vs. each baseline. Points below the diagonal indicate tasks where h^{evo} evaluates each state more cheaply.

served-goals of allergic children with gluten-free sandwich atoms (2–20 atoms); *place-based delivery patterns* collecting all trays, sandwiches and served-goals at one delivery location (2–49 atoms); and *goal-delivery context patterns* pairing one child's served-goal with the trays, sandwiches and gluten constraints at that child's location (4–64 atoms).

These patterns are substantially larger than those found by systematic methods: on the solved tasks, h^{evo} builds patterns of up to size 20, while $h^{\text{Sys}}_{\text{SCP}}$ uses patterns of at most size 8. On every solved task, h^{evo} therefore needs fewer expansions before the last f-layer than any systematic baseline—a gap of roughly one order of magnitude—and it achieves the highest coverage on the domain (5 vs. 4 for all baselines).

Miconic. Miconic is an elevator scheduling domain where a single lift must board and deliver passengers to their destination floors. Our generator produces three pattern types: *singleton state atoms* (`boarded` or `served`, 1 atom each); *lift floor patterns* grouping `lift-at` atoms for adjacent or globally relevant floor pairs (1–19 atoms); and *per-passenger journey patterns* capturing the full life cycle of one passenger together with the lift positions at the passenger's origin and destination, optionally extended to passengers sharing a floor (5–19 atoms).

Miconic is the domain where our approach struggles the most. Despite using patterns of up to size 8, h^{evo} solves only 4 of 30 tasks, compared to 20 for h^{Sys}_3. Figure 4 confirms that h^{Sys}_3 and $h^{\text{Sys}}_{\text{SCP}}$ need far fewer expansions on nearly every solved task (16–20 out of 20), so small systematic patterns provide stronger guidance here. Miconic's structure makes the domain particularly amenable to exhaustive pattern enumeration: its uniform passenger–floor structure means that patterns of size 3 already capture the key interactions, leaving little room for the larger but less comprehensive patterns produced by our generator.

Transport. Transport is a logistics domain where vehicles with limited capacity move packages between locations on a road network. Our pattern generator produces four pattern types: *per-package delivery patterns* combining a package's initial and goal locations with all possible in-vehicle states (3–20 atoms); *per-vehicle cargo patterns* collecting all packages loadable into one vehicle together with capacity atoms (3–20 atoms); *per-vehicle movement patterns* grouping all locations a vehicle can occupy with its capacity levels (4–20 atoms); and *location clusters* grouping all objects at initial or goal locations of any package, with uncovered goal atoms added as singletons (1–20 atoms).

Transport is one of the strongest domains for h^{evo}, which achieves the best coverage (13 solved tasks vs. 12 for h^{Sys}_3). Guidance quality, measured in expansions before the last f-layer, is comparable to h^{Sys}_2 and h^{Sys}_3 on solved tasks. The main advantage of our patterns is their lower evaluation cost: h^{evo} is faster to evaluate than h^{Sys}_3 on 11 tasks out of 12, and faster than $h^{\text{Sys}}_{\text{SCP}}$ on 10 tasks out of 12. This per-state efficiency translates directly into lower total search time and higher coverage on hard instances.

5 Related Work

Pattern Database Heuristics. Pattern databases (PDBs) (Culberson and Schaeffer 1998) are one of several admissi-

ble heuristic families for optimal classical planning, alongside merge-and-shrink abstractions (Helmert et al. 2014; Sievers and Helmert 2021), Cartesian abstractions (Seipp and Helmert 2018), LP-based operator-counting heuristics (Pommerening et al. 2014) and landmark-based heuristics such as LM-cut (Helmert and Domshlak 2009). Domain-independent generators for PDBs include the hill-climbing procedure of Haslum et al. (2007) and counterexample-guided pattern selection (Rovner, Sievers, and Helmert 2019). Franco et al. (2017) construct *complementary* pattern collections iteratively: at each step they propose a new collection with a bin-packing-based generator and accept it only if a sampling-based estimator predicts that adding it reduces A^* search effort, biasing new patterns toward states the current heuristic underestimates. Systematic pattern generation (Pommerening, Röger, and Helmert 2013) identifies *interesting* patterns by exploiting causal-graph structure, and Seipp (2019) extends this approach to the current state of the art by only selecting those systematic patterns that saturated cost partitioning deems useful. Closest in spirit to our work, Edelkamp (2006) uses a genetic algorithm to evolve pattern collections directly, but the search operates over a fixed encoding of patterns rather than synthesizing a generator program. All these methods are domain-independent and treat each task in isolation, so any structure shared across tasks of the same domain has to be rediscovered every time. We lift this restriction by learning, per domain, a generator whose code captures regularities the LLM distills from a handful of training tasks.

Learning Heuristics for Planning. A growing line of work learns heuristics for satisficing planning from experience (e.g., Toyer et al. 2018; Ståhlberg, Bonet, and Geffner 2022), with Chen, Thiébaux, and Trevizan (2024) extending this to heuristics that transfer across domains. Chen, Trevizan, and Thiébaux (2024) further show that classical machine-learning models can match neural heuristics at substantially lower inference cost, and Karia and Srivastava (2021) learn relational heuristics that transfer across object counts. Francès et al. (2019) take a symbolic route, synthesizing per-domain generalized potential heuristics over a fixed concept-feature language. These heuristics are typically inadmissible, and the neural variants often require a GPU at inference time. Núñez-Molina et al. (2024) bound learned values by admissible estimates during training to mitigate the second issue, but their final heuristic remains inadmissible. Admissible learned heuristics are rarer. Closest in spirit, Futuhi and Sturtevant (2026) train a general-purpose neural admissible heuristic with a cross-entropy admissibility loss, but only enforce admissibility on the training distribution rather than by construction.

Evolutionary and LLM-Guided Heuristic Search. Automatic heuristic discovery predates LLMs. Aler, Borrajo, and Isasi (2001) use genetic programming over a fixed language to evolve domain-specific control heuristics, and Fukunaga (2008) evolves composite SAT heuristics from a fixed compositional grammar. Since FunSearch (Romera-Paredes et al. 2024), the field has moved away from such restricted languages and uses LLMs to mutate raw code instead, with strong results across diverse task families (e.g., Liu et al. 2024; Ye et al. 2024; Zheng et al. 2025; Novikov et al. 2025). We follow this trend and build on the open-source OpenEvolve framework (Sharma 2025). To our knowledge, ours is the first application of LLM-driven program synthesis to admissible heuristics for optimal planning.

LLM-Generated Domain-Specific Solvers. Several recent methods use LLMs to produce domain-specific solvers. Corrêa, Pereira, and Seipp (2025) and Tuisov, Vernik, and Shleyfman (2026) generate inadmissible heuristics that let otherwise weak planners compete with strong ones. Our generators are domain-specific in the same sense, but the evolutionary process is identical across domains and the generators produce *admissible* heuristics that plug directly into a state-of-the-art planner. A separate strand generates domain-specific policies (e.g., Chen et al. 2025; Stein et al. 2026). Closest in spirit, Murray et al. (2026) use LLM-driven evolution to produce Python functions that emit plans directly, a form of generalized planning competitive with state-of-the-art planners on their evaluation set. These policy-style approaches and ours are complementary: a learned policy can solve target tasks quickly and often without search but cannot scale beyond tasks that admit simple strategies, whereas a learned heuristic preserves the search guarantees that make symbolic planners broadly applicable.

6 Conclusions

We presented a method for learning admissible heuristics for classical planning by evolving domain-specific pattern collection generators with an LLM-guided evolutionary framework. The evolved generators are interpretable Python programs that compute a pattern collection for any task in the domain at test time and yield admissible heuristics by construction. Across seven domains, the evolved heuristic matches or exceeds the best systematic baseline in five, and is uniquely best in two of them. On a majority of tasks it is also substantially faster to evaluate per state than the strongest systematic baselines, often by orders of magnitude, thanks to the task-focused structure of the evolved patterns. The two domains where our approach falls behind, Floortile and Miconic, are precisely the ones in which small systematic patterns already provide strong guidance and leave little room for the larger domain-specific ones our generators compute.

Several directions stand out for future work. First, exposing the evolutionary algorithm to richer domain information, such as the causal graph or the domain transition graph, may improve pattern quality. Second, we trained on a small set of tasks per domain to keep token usage manageable, and larger training sets may uncover stronger generators. Third, the slight mismatch between our training score and the final evaluation metric (coverage) suggests that refining the scoring function could yield further gains. Finally, obfuscating domain names would clarify how much our approach relies on structural reasoning rather than on the LLM's prior exposure to these benchmarks.

Acknowledgments

This work is supported by the Swedish Research Council under grant number 2024-05403, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council.

References

Aler, R.; Borrajo, D.; and Isasi, P. 2001. Learning to Solve Planning Problems Efficiently by Means of Genetic Programming. *Evolutionary Computation*, 9(4): 387–420.

Chen, D. Z.; Thiébaux, S.; and Trevizan, F. 2024. Learning Domain-Independent Heuristics for Grounded and Lifted Planning. In Dy, J.; and Natarajan, S., eds., *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024)*, 20078–20086. AAAI Press.

Chen, D. Z.; Trevizan, F.; and Thiébaux, S. 2024. Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning. In Bernardini, S.; and Muise, C., eds., *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*, 68–76. AAAI Press.

Chen, D. Z.; Zenn, J.; Cinquin, T.; and McIlraith, S. A. 2025. Language Models for Generalised PDDL Planning: Synthesising Sound and Programmatic Policies. In *ICAPS 2025 Workshop on Planning in the Era of LLMs (LM4Plan)*.

Corrêa, A. B.; Pereira, A. G.; and Seipp, J. 2025. Classical Planning with LLM-Generated Heuristics: Challenging the State of the Art with Python Code. In *Proceedings of the Thirty-Ninth Annual Conference on Neural Information Processing Systems (NeurIPS 2025)*.

Culberson, J. C.; and Schaeffer, J. 1996. Searching with Pattern Databases. In McCalla, G. I., ed., *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI 1996)*, volume 1081 of *Lecture Notes in Computer Science*, 402–416. Springer-Verlag.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence*, 14(3): 318–334.

Edelkamp, S. 2001. Planning with Pattern Databases. In Cesta, A.; and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 84–90. AAAI Press.

Edelkamp, S. 2006. Automated Creation of Pattern Database Search Heuristics. In Edelkamp, S.; and Lomuscio, A., eds., *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, 35–50.

Francès, G.; Corrêa, A. B.; Geissmann, C.; and Pommerening, F. 2019. Generalized Potential Heuristics for Classical Planning. In Kraus, S., ed., *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 5554–5561. IJCAI.

Franco, S.; Torralba, Á.; Lelis, L. H. S.; and Barley, M. 2017. On Creating Complementary Pattern Databases. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4302–4309. IJCAI.

Fukunaga, A. S. 2008. Automated Discovery of Local Search Heuristics for Satisfiability Testing. *Evolutionary Computation*, 16(1): 31–61.

Futuhi, E.; and Sturtevant, N. R. 2026. Learning Admissible Heuristics for A*: Theory and Practice. In *Proceedings of the Fourteenth International Conference on Learning Representations (ICLR 2026)*. OpenReview.net.

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

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*, volume 13 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173: 503–535.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3): 16:1–63.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In Leyton-Brown, K.; and Mausam, eds., *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021)*, 8064–8073. AAAI Press.

Katz, M.; and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence*, 174(12–13): 767–798.

Kimi Team. 2026. Kimi K2.5: Visual Agentic Intelligence. arXiv:2602.02276.

Liu, F.; Xialiang, T.; Yuan, M.; Lin, X.; Luo, F.; Wang, Z.; Lu, Z.; and Zhang, Q. 2024. Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, 32201–32223. JMLR.org.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

Mouret, J.-B.; and Clune, J. 2015. Illuminating search spaces by mapping elites. arXiv:1504.04909.

Murray, A.; Dervovic, D.; Pozanco, A.; and Cashmore, M. 2026. GenePlan: Evolving Better Generalized PDDL Plans using Large Language Models. In *Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)*. AAAI Press.

Novikov, A.; Vũ, N.; Eisenberger, M.; Dupont, E.; Huang, P.-S.; Wagner, A. Z.; Shirobokov, S.; Kozlovskii, B.; Ruiz, F. J. R.; Mehrabian, A.; Kumar, M. P.; See, A.; Chaudhuri, S.; Holland, G.; Davies, A.; Nowozin, S.; Kohli, P.; and Balog, M. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131.

Núñez-Molina, C.; Asai, M.; Mesejo, P.; and Fernández-Olivares, J. 2024. On Using Admissible Bounds for Learning Forward Search Heuristics. In Larson, K., ed., *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI 2024)*, 6761–6769. IJCAI.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the Most Out of Pattern Databases for Classical Planning. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2357–2364. AAAI Press.

Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. LP-based Heuristics for Cost-optimal Planning. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 226–234. AAAI Press.

Romera-Paredes, B.; Barekatain, M.; Novikov, A.; Balog, M.; Kumar, M. P.; Dupont, E.; Ruiz, F. J. R.; Ellenberg, J. S.; Wang, P.; Fawzi, O.; Kohli, P.; and Fawzi, A. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625: 468–475.

Rovner, A.; Sievers, S.; and Helmert, M. 2019. Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 362–367. AAAI Press.

Seipp, J. 2019. Pattern Selection for Optimal Classical Planning with Saturated Cost Partitioning. In Kraus, S., ed., *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 5621–5627. IJCAI.

Seipp, J. 2021. Online Saturated Cost Partitioning for Classical Planning. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 317–321. AAAI Press.

Seipp, J. 2024. Dissecting Scorpion: Ablation Study of an Optimal Classical Planner. In Endriss, U.; and Melo, F. S., eds., *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI 2024)*, 39–42. IOS Press.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research*, 62: 535–577.

Seipp, J.; and Helmert, M. 2019. Subset-Saturated Cost Partitioning for Optimal Classical Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 391–400. AAAI Press.

Seipp, J.; Keller, T.; and Helmert, M. 2017a. A Comparison of Cost Partitioning Algorithms for Optimal Classical Planning. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 259–268. AAAI Press.

Seipp, J.; Keller, T.; and Helmert, M. 2017b. Narrowing the Gap Between Saturated and Optimal Cost Partitioning for Classical Planning. In Singh, S.; and Markovitch, S., eds., *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017)*, 3651–3657. AAAI Press.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *Journal of Artificial Intelligence Research*, 67: 129–167.

Sharma, A. 2025. OpenEvolve: An Open-Source Evolutionary Coding Agent. https://github.com/algorithmicsuperintelligence/openevolve.

Sievers, S.; and Helmert, M. 2021. Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems. *Journal of Artificial Intelligence Research*, 71: 781–883.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient Implementation of Pattern Database Heuristics for Classical Planning. In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 105–111. AAAI Press.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 629–637. AAAI Press.

Stein, K.; Hodel, N.; Fišer, D.; Hoffmann, J.; Katz, M.; and Koller, A. 2026. Improved Generalized Planning with LLMs through Strategy Refinement and Reflection. In *Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)*. AAAI Press.

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

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In Goldman, R. P.;

Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 376–384. AAAI Press.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 6294–6301. AAAI Press.

Tuisov, A.; Vernik, Y.; and Shleyfman, A. 2026. Successor-Generator Planning with LLM-Generated Heuristics. In *Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)*. AAAI Press.

Ye, H.; Wang, J.; Cao, Z.; Berto, F.; Hua, C.; Kim, H.; Park, J.; and Song, G. 2024. ReEvo: Large Language Models as Hyper-Heuristics with Reflective Evolution. In *Proceedings of the Thirty-Eighth Annual Conference on Neural Information Processing Systems (NeurIPS 2024)*.

Zheng, Z.; Xie, Z.; Wang, Z.; and Hooi, B. 2025. Monte Carlo Tree Search for Comprehensive Exploration in LLM-Based Automatic Heuristic Design. In *Proceedings of the 42nd International Conference on Machine Learning (ICML 2025)*. JMLR.org.

A Synthesized Pattern Generators

Listing 2: Blocksworld generator code.

```
def generate_pattern_collection(task_info: TaskInformation) -> list[Pattern]:
    patterns = []
    goal_atoms = list(task_info.fluent_goal_atoms)

    if not goal_atoms:
        if task_info.fluent_initial_state_atoms:
            patterns.append(Pattern(pattern=[task_info.fluent_initial_state_atoms[0]]))
        return patterns

    obj_to_atoms = {}
    for atom in goal_atoms:
        for obj in atom.binding:
            if obj not in obj_to_atoms:
                obj_to_atoms[obj] = []
            obj_to_atoms[obj].append(atom)

    visited = set()

    def dfs(atom, component):
        atom_id = id(atom)
        if atom_id in visited:
            return
        visited.add(atom_id)
        component.append(atom)
        for obj in atom.binding:
            for other in obj_to_atoms.get(obj, []):
                dfs(other, component)

    for atom in goal_atoms:
        if id(atom) not in visited:
            component = []
            dfs(atom, component)
            if len(component) > 1:
                patterns.append(Pattern(pattern=component))

    clear_map = {}
    holding_map = {}
    ontable_map = {}
    on_map = {}
    arm_empty_atom = None

    for atom in task_info.all_fluent_atoms:
        pname = atom.predicate.name
        if pname == 'clear' and len(atom.binding) == 1:
            clear_map[atom.binding[0]] = atom
        elif pname == 'holding' and len(atom.binding) == 1:
            holding_map[atom.binding[0]] = atom
        elif pname == 'on-table' and len(atom.binding) == 1:
            ontable_map[atom.binding[0]] = atom
        elif pname == 'on' and len(atom.binding) == 2:
            on_map[atom.binding] = atom
        elif pname == 'arm-empty':
            arm_empty_atom = atom

    goal_objs = set()
    for atom in goal_atoms:
        for obj in atom.binding:
            goal_objs.add(obj)

    for atom in goal_atoms:
```

```
        pattern_atoms = [atom]
        objs = set(atom.binding)

        for fluent_atom in task_info.all_fluent_atoms:
            if len(pattern_atoms) >= 6:
                break
            if fluent_atom.predicate.name == 'clear':
                if fluent_atom.binding[0] in objs:
                    pattern_atoms.append(fluent_atom)

        if arm_empty_atom and len(pattern_atoms) < 8:
            pattern_atoms.append(arm_empty_atom)

        patterns.append(Pattern(pattern=pattern_atoms))

    for obj in goal_objs:
        if obj in holding_map:
            patterns.append(Pattern(pattern=[holding_map[obj]]))

    initial_on = {a for a in task_info.fluent_initial_state_atoms if a.predicate.name == 'on'
    }
    goal_on = {a for a in goal_atoms if a.predicate.name == 'on'}
    initial_support = {a.binding[0]: a.binding[1] for a in initial_on if len(a.binding) == 2}
    goal_support = {a.binding[0]: a.binding[1] for a in goal_on if len(a.binding) == 2}

    blocks_to_unstack = set()
    blocks_to_pickup = set()
    blocks_to_putdown = set()

    for block in goal_objs:
        if block in initial_support:
            if block not in goal_support or initial_support[block] != goal_support[block]:
                blocks_to_unstack.add(block)

    for atom in goal_atoms:
        if atom.predicate.name == 'on' and len(atom.binding) == 2:
            block = atom.binding[0]
            if block in ontable_map and block in clear_map:
                blocks_to_pickup.add(block)

    for atom in goal_atoms:
        if atom.predicate.name == 'on-table' and len(atom.binding) == 1:
            block = atom.binding[0]
            if block in holding_map:
                blocks_to_putdown.add(block)

    for atom in goal_atoms:
        if atom.predicate.name == 'on' and len(atom.binding) == 2:
            x, y = atom.binding[0], atom.binding[1]
            if x in holding_map and y in clear_map:
                stack_pattern = [atom, holding_map[x], clear_map[y]]
                if arm_empty_atom:
                    stack_pattern.append(arm_empty_atom)
                if len(stack_pattern) <= 15:
                    patterns.append(Pattern(pattern=stack_pattern))

    for block in blocks_to_unstack:
        if block in initial_support and block in clear_map and arm_empty_atom:
            y = initial_support[block]
            if (block, y) in on_map:
                unstack_pattern = [on_map[(block, y)], clear_map[block], arm_empty_atom]
                if len(unstack_pattern) <= 12:
                    patterns.append(Pattern(pattern=unstack_pattern))

    for block in blocks_to_pickup:
```

```
        if block in ontable_map and block in clear_map and arm_empty_atom:
            pickup_pattern = [ontable_map[block], clear_map[block], arm_empty_atom]
            if len(pickup_pattern) <= 10:
                patterns.append(Pattern(pattern=pickup_pattern))

    for block in blocks_to_putdown:
        if block in holding_map and block in ontable_map:
            putdown_pattern = [ontable_map[block], holding_map[block]]
            if len(putdown_pattern) <= 10:
                patterns.append(Pattern(pattern=putdown_pattern))

    initial_above = {}
    for a in initial_on:
        if len(a.binding) == 2:
            initial_above[a.binding[1]] = a.binding[0]

    for atom in goal_atoms:
        if atom.predicate.name == 'on' and len(atom.binding) == 2:
            y = atom.binding[1]
            if y in initial_above:
                z = initial_above[y]
                if z in clear_map and arm_empty_atom and (z, y) in on_map:
                    obstacle_pattern = [atom, on_map[(z, y)], clear_map[z], arm_empty_atom]
                    if len(obstacle_pattern) <= 12:
                        patterns.append(Pattern(pattern=obstacle_pattern))

    for atom in goal_atoms:
        patterns.append(Pattern(pattern=[atom]))

    for atom in goal_atoms:
        patterns.append(Pattern(pattern=[atom]))

    seen = set()
    unique_patterns = []
    for p in patterns:
        key = tuple(sorted(id(atom) for atom in p.pattern))
        if key not in seen and len(p.pattern) <= 20:
            seen.add(key)
            unique_patterns.append(p)

    goal_atom_set = set(goal_atoms)

    def pattern_value(p):
        coverage = len(set(p.pattern) & goal_atom_set)
        size = len(p.pattern)
        cost = 2 ** size

        if coverage == 0:
            return -cost * 0.1 if size <= 4 else -cost

        value = (coverage * 1000.0) / cost
        if size >= 12:
            value *= 0.1
        elif size >= 8:
            value *= 0.4
        elif size >= 6:
            value *= 0.7

        return value

    unique_patterns.sort(key=pattern_value, reverse=True)

    selected_patterns = []
    covered_goals = set()
    total_states = 0
```

```
    for p in unique_patterns:
        cost = 2 ** len(p.pattern)
        if total_states + cost <= 500000000:
            selected_patterns.append(p)
            total_states += cost
            covered_goals.update(set(p.pattern) & goal_atom_set)

    for goal in goal_atom_set - covered_goals:
        if total_states + 2 <= 500000000:
            selected_patterns.append(Pattern(pattern=[goal]))
            total_states += 2

    return selected_patterns
```

Listing 3: Childsnack generator code.

```
def generate_pattern_collection(task_info: TaskInformation) -> list[Pattern]:
    patterns = []

    def count_vars(atom_list):
        return len(set(o for atom in atom_list for o in atom.binding))

    fluent_by_pred = {}
    for atom in task_info.all_fluent_atoms:
        fluent_by_pred.setdefault(atom.predicate.name, []).append(atom)

    static_by_pred = {}
    for atom in task_info.static_ground_atoms:
        static_by_pred.setdefault(atom.predicate.name, []).append(atom)

    for goal in task_info.fluent_goal_atoms:
        patterns.append(Pattern(pattern=[goal]))

    at_atoms = fluent_by_pred.get('at', [])
    trays = {}
    for atom in at_atoms:
        trays.setdefault(atom.binding[0], []).append(atom)

    for tray_atoms in trays.values():
        if count_vars(tray_atoms) <= 20:
            patterns.append(Pattern(pattern=tray_atoms))

    notexist = {a.binding[0]: a for a in fluent_by_pred.get('notexist', [])}
    at_kitchen = {a.binding[0]: a for a in fluent_by_pred.get('at_kitchen_sandwich', [])}
    no_gluten = {a.binding[0]: a for a in fluent_by_pred.get('no_gluten_sandwich', [])}
    ontray_by_sw = {}
    for atom in fluent_by_pred.get('ontray', []):
        ontray_by_sw.setdefault(atom.binding[0], []).append(atom)

    tray_to_loc = {}
    for atom in at_atoms:
        tray_to_loc[atom.binding[0]] = atom

    for sw in set(notexist.keys()) | set(at_kitchen.keys()) | set(ontray_by_sw.keys()):
        atoms = []
        if sw in notexist: atoms.append(notexist[sw])
        if sw in at_kitchen: atoms.append(at_kitchen[sw])
        if sw in ontray_by_sw:
            atoms.extend(ontray_by_sw[sw])
            for ontray_atom in ontray_by_sw[sw]:
                tray = ontray_atom.binding[1]
                if tray in tray_to_loc:
                    atoms.append(tray_to_loc[tray])
```

```
        if sw in no_gluten: atoms.append(no_gluten[sw])
        if 1 < count_vars(atoms) <= 20:
            patterns.append(Pattern(pattern=atoms))

    bread = fluent_by_pred.get('at_kitchen_bread', [])
    content = fluent_by_pred.get('at_kitchen_content', [])
    if bread or content:
        atoms = bread + content
        if 1 < count_vars(atoms) <= 20:
            patterns.append(Pattern(pattern=atoms))

    waiting = {a.binding[0]: a.binding[1] for a in static_by_pred.get('waiting', [])}
    allergic = {a.binding[0] for a in static_by_pred.get('allergic_gluten', [])}
    no_gluten_set = set(no_gluten.keys())
    ontray_atoms = fluent_by_pred.get('ontray', [])

    for goal in task_info.fluent_goal_atoms:
        if goal.predicate.name != 'served':
            continue
        child = goal.binding[0]
        if child not in waiting:
            continue

        place = waiting[child]
        atoms = [goal]

        trays_here = []
        for atom in at_atoms:
            if atom.binding[1] == place:
                atoms.append(atom)
                trays_here.append(atom.binding[0])

        valid_sandwiches = set()
        for atom in ontray_atoms:
            if atom.binding[1] in trays_here:
                if child not in allergic or atom.binding[0] in no_gluten_set:
                    atoms.append(atom)
                    valid_sandwiches.add(atom.binding[0])

        if child in allergic:
            for sw_id in valid_sandwiches:
                if sw_id in no_gluten:
                    atoms.append(no_gluten[sw_id])

        if 1 < count_vars(atoms) <= 20:
            patterns.append(Pattern(pattern=atoms))

    if allergic:
        atoms = [g for g in task_info.fluent_goal_atoms
                if g.predicate.name == 'served' and g.binding[0] in allergic]
        for atom in fluent_by_pred.get('no_gluten_sandwich', []):
            if count_vars(atoms + [atom]) <= 20:
                atoms.append(atom)
        if len(atoms) > 1 and count_vars(atoms) <= 20:
            patterns.append(Pattern(pattern=atoms))

    places = set()
    for atom in at_atoms:
        places.add(atom.binding[1])

    for place in places:
        place_atoms = []
        trays_here = []

        for atom in at_atoms:
```

```
            if atom.binding[1] == place:
                place_atoms.append(atom)
                trays_here.append(atom.binding[0])

        for atom in ontray_atoms:
            if atom.binding[1] in trays_here:
                place_atoms.append(atom)

        for goal in task_info.fluent_goal_atoms:
            if goal.predicate.name == 'served':
                child = goal.binding[0]
                if child in waiting and waiting[child] == place:
                    place_atoms.append(goal)

        if 1 < count_vars(place_atoms) <= 20:
            patterns.append(Pattern(pattern=place_atoms))

    seen = set()
    unique = []
    for p in patterns:
        key = frozenset((a.predicate.name, a.binding) for a in p.pattern)
        if key not in seen:
            seen.add(key)
            unique.append(p)

    return unique
```

Listing 4: Floortile generator code.

```
def generate_pattern_collection(task_info: TaskInformation) -> list[Pattern]:
    patterns = []

    def get_predicate_name(atom):
        return atom.predicate.name

    paint_adjacency = {}
    move_adjacency = {}

    for atom in task_info.static_ground_atoms:
        pname = get_predicate_name(atom)
        if pname == 'up':
            target, source = atom.binding  # target is up from source
            if target not in paint_adjacency:
                paint_adjacency[target] = []
            paint_adjacency[target].append(source)
            if source not in move_adjacency:
                move_adjacency[source] = []
            move_adjacency[source].append(target)
            if target not in move_adjacency:
                move_adjacency[target] = []
            move_adjacency[target].append(source)
        elif pname == 'down':
            target, source = atom.binding  # target is down from source
            if target not in paint_adjacency:
                paint_adjacency[target] = []
            paint_adjacency[target].append(source)
            if source not in move_adjacency:
                move_adjacency[source] = []
            move_adjacency[source].append(target)
            if target not in move_adjacency:
                move_adjacency[target] = []
            move_adjacency[target].append(source)
        elif pname == 'left':
            target, source = atom.binding
```

```
            if source not in move_adjacency:
                move_adjacency[source] = []
            move_adjacency[source].append(target)
            if target not in move_adjacency:
                move_adjacency[target] = []
            move_adjacency[target].append(source)
        elif pname == 'right':
            target, source = atom.binding
            if source not in move_adjacency:
                move_adjacency[source] = []
            move_adjacency[source].append(target)
            if target not in move_adjacency:
                move_adjacency[target] = []
            move_adjacency[target].append(source)

    goal_tiles = {}  # tile -> color
    robot_at_atoms = []
    robot_has_atoms = []
    painted_atoms = []
    clear_atoms = []
    tile_to_clear_atom = {}

    for atom in task_info.fluent_goal_atoms:
        if get_predicate_name(atom) == 'painted':
            tile, color = atom.binding
            goal_tiles[tile] = color

    for atom in task_info.all_fluent_atoms:
        pname = get_predicate_name(atom)
        if pname == 'robot-at':
            robot_at_atoms.append(atom)
        elif pname == 'robot-has':
            robot_has_atoms.append(atom)
        elif pname == 'painted':
            painted_atoms.append(atom)
        elif pname == 'clear':
            clear_atoms.append(atom)
            tile_to_clear_atom[atom.binding[0]] = atom

    init_robot_at = None
    for atom in task_info.fluent_initial_state_atoms:
        if get_predicate_name(atom) == 'robot-at':
            init_robot_at = atom
            break

    goal_tiles_set = set(goal_tiles.keys())

    for goal_atom in task_info.fluent_goal_atoms:
        if get_predicate_name(goal_atom) != 'painted':
            continue

        pattern_atoms = [goal_atom]
        tile, color = goal_atom.binding

        if init_robot_at is not None:
            patterns.append(Pattern(pattern=[goal_atom, init_robot_at]))

        for rha in robot_has_atoms:
            if rha.binding[1] == color:
                pattern_atoms.append(rha)
                break

        if tile in paint_adjacency:
            for adj_tile in paint_adjacency[tile]:
                for raa in robot_at_atoms:
```

```
                    if raa.binding[1] == adj_tile:
                        pattern_atoms.append(raa)
                        break

        if tile in tile_to_clear_atom:
            pattern_atoms.append(tile_to_clear_atom[tile])
        if tile in paint_adjacency:
            for adj_tile in paint_adjacency[tile]:
                if adj_tile in tile_to_clear_atom:
                    pattern_atoms.append(tile_to_clear_atom[adj_tile])

        unique_atoms = []
        seen = set()
        for a in pattern_atoms:
            key = (a.predicate.name, tuple(o.name for o in a.binding))
            if key not in seen:
                seen.add(key)
                unique_atoms.append(a)

        if len(unique_atoms) <= 20:
            patterns.append(Pattern(pattern=unique_atoms))
        else:
            patterns.append(Pattern(pattern=[goal_atom]))

    robot_at_by_tile = {}
    for ra in robot_at_atoms:
        robot_at_by_tile.setdefault(ra.binding[1], []).append(ra)

    for atom in task_info.static_ground_atoms:
        if atom.predicate.name not in ['up', 'down']:
            continue
        target, source = atom.binding[0], atom.binding[1]
        if target not in goal_tiles_set:
            continue
        for ratom in robot_at_by_tile.get(source, []):
            robot = ratom.binding[0]
            for catom in clear_atoms:
                if catom.binding[0] == target:
                    for hasatom in robot_has_atoms:
                        if hasatom.binding[0] == robot:
                            precond_pattern = [ratom, hasatom, catom]
                            if len(precond_pattern) <= 15:
                                patterns.append(Pattern(pattern=precond_pattern))

    if robot_at_atoms and robot_has_atoms:
        robot_pattern = list(dict.fromkeys(robot_at_atoms + robot_has_atoms))
        relevant_clears = set()
        for tile in goal_tiles:
            if tile in paint_adjacency:
                for adj_tile in paint_adjacency[tile]:
                    if adj_tile in tile_to_clear_atom:
                        relevant_clears.add(tile_to_clear_atom[adj_tile])
        robot_pattern.extend(list(relevant_clears)[:15])
        robot_pattern = list(dict.fromkeys(robot_pattern))
        if len(robot_pattern) <= 20:
            patterns.append(Pattern(pattern=robot_pattern))

    clear_by_tile = {a.binding[0]: a for a in clear_atoms}
    tile_groups = {}

    for tile in clear_by_tile:
        parts = tile.name.split('_')
        if len(parts) >= 3:
            row, col = parts[1], parts[2]
            tile_groups.setdefault(('r', row), []).append(clear_by_tile[tile])
```

```
        tile_groups.setdefault(('c', col), []).append(clear_by_tile[tile])

for group in tile_groups.values():
    if 2 <= len(group) <= 8:
        patterns.append(Pattern(pattern=group))

robots = set()
for atom in robot_at_atoms:
    robots.add(atom.binding[0])
for robot in robots:
    has_atoms = [a for a in robot_has_atoms if a.binding[0] == robot]
    if 1 < len(has_atoms) <= 20:
        patterns.append(Pattern(pattern=has_atoms))

all_tiles = set()
for atom in clear_atoms + painted_atoms:
    all_tiles.add(atom.binding[0])
for tile in all_tiles:
    tile_atoms = ([a for a in clear_atoms if a.binding[0] == tile] +
                  [a for a in painted_atoms if a.binding[0] == tile])
    if 1 < len(tile_atoms) <= 20:
        patterns.append(Pattern(pattern=tile_atoms))

color_to_goals = {}
for tile, color in goal_tiles.items():
    if color not in color_to_goals:
        color_to_goals[color] = []
    for pa in painted_atoms:
        if pa.binding[0] == tile and pa.binding[1] == color:
            color_to_goals[color].append(pa)
            break

for color, goal_atoms in color_to_goals.items():
    if goal_atoms:
        color_pattern = list(dict.fromkeys(goal_atoms + robot_has_atoms))
        color_pattern.extend(robot_at_atoms[:2])
        if len(color_pattern) <= 20:
            patterns.append(Pattern(pattern=color_pattern))

seen = set()
unique_patterns = []
for p in patterns:
    key = tuple(sorted([(a.predicate.name, tuple(o.name for o in a.binding)) for a in p.
pattern]))
    if key not in seen and len(p.pattern) > 0:
        seen.add(key)
        unique_patterns.append(p)

def pattern_priority(p):
    covers_goal = any(atom in task_info.fluent_goal_atoms for atom in p.pattern)
    return (0 if covers_goal else 1, len(p.pattern))

unique_patterns.sort(key=pattern_priority)

selected = []
total_states = 0
budget = 400000000  # 400M limit (under 500M)

for p in unique_patterns:
    n_atoms = len(p.pattern)
    if n_atoms > 20:  # Enforce per-pattern atom limit
        continue
    states = 2 ** n_atoms
    if total_states + states <= budget:
        selected.append(p)
```

```
            total_states += states

    covered_goals = set()
    for p in selected:
        for atom in p.pattern:
            if atom in task_info.fluent_goal_atoms:
                covered_goals.add(atom)

    for goal in task_info.fluent_goal_atoms:
        if goal not in covered_goals:
            selected.append(Pattern(pattern=[goal]))
            total_states += 2  # 2^1 = 2 states

    if not selected and task_info.fluent_goal_atoms:
        for goal in task_info.fluent_goal_atoms:
            selected.append(Pattern(pattern=[goal]))

    return selected
```

Listing 5: Miconic generator code.

```
def generate_pattern_collection(task_info: TaskInformation) -> list[Pattern]:

    patterns = []
    existing_patterns = set()  # Track to avoid duplicates

    atom_index = {}
    for atom in task_info.all_fluent_atoms:
        key = (atom.predicate.name, atom.binding)
        atom_index[key] = atom

    passenger_destin = {}
    for atom in task_info.static_ground_atoms:
        if atom.predicate.name == 'destin':
            p, f = atom.binding[0], atom.binding[1]
            passenger_destin[p] = f

    passenger_origin = {}
    for atom in task_info.fluent_initial_state_atoms:
        if atom.predicate.name == 'origin':
            p, f = atom.binding[0], atom.binding[1]
            passenger_origin[p] = f

    passenger_to_atoms = {}
    for atom in task_info.all_fluent_atoms:
        if atom.predicate.name in ('boarded', 'served', 'origin'):
            passenger = atom.binding[0]
            if passenger not in passenger_to_atoms:
                passenger_to_atoms[passenger] = []
            passenger_to_atoms[passenger].append(atom)

    lift_atoms = [atom for atom in task_info.all_fluent_atoms if atom.predicate.name == 'lift
    -at']

    for p, f_orig in passenger_origin.items():
        if p not in passenger_destin:
            continue

        f_dest = passenger_destin[p]
        pattern_atoms = []

        origin_key = ('origin', (p, f_orig))
        if origin_key in atom_index:
            pattern_atoms.append(atom_index[origin_key])
```

```
        boarded_key = ('boarded', (p,))
        if boarded_key in atom_index:
            pattern_atoms.append(atom_index[boarded_key])

        served_key = ('served', (p,))
        if served_key in atom_index:
            pattern_atoms.append(atom_index[served_key])

        lift_orig_key = ('lift-at', (f_orig,))
        if lift_orig_key in atom_index:
            pattern_atoms.append(atom_index[lift_orig_key])

        if f_dest != f_orig:  # Avoid duplicates if origin == destin
            lift_dest_key = ('lift-at', (f_dest,))
            if lift_dest_key in atom_index:
                pattern_atoms.append(atom_index[lift_dest_key])

        if len(pattern_atoms) >= 2:
            frozen_pat = frozenset(pattern_atoms)
            if frozen_pat not in existing_patterns:
                patterns.append(Pattern(pattern=pattern_atoms))
                existing_patterns.add(frozen_pat)

        for atom in pattern_atoms:
            if atom.predicate.name in ('boarded', 'served'):
                singleton = [atom]
                frozen_single = frozenset(singleton)
                if frozen_single not in existing_patterns:
                    patterns.append(Pattern(pattern=singleton))
                    existing_patterns.add(frozen_single)

    relevant_floors = set(passenger_origin.values()) | set(passenger_destin.values())
    relevant_lift_atoms = [atom for atom in lift_atoms if atom.binding[0] in relevant_floors]

    if relevant_lift_atoms:
        unique_floors = set(atom.binding[0] for atom in relevant_lift_atoms)
        if len(unique_floors) <= 20:
            frozen_pat = frozenset(relevant_lift_atoms)
            if frozen_pat not in existing_patterns:
                patterns.append(Pattern(pattern=relevant_lift_atoms))
                existing_patterns.add(frozen_pat)
        else:
            for atom in relevant_lift_atoms:
                singleton = [atom]
                frozen_single = frozenset(singleton)
                if frozen_single not in existing_patterns:
                    patterns.append(Pattern(pattern=singleton))
                    existing_patterns.add(frozen_single)

    floor_to_passengers = {}
    for passenger in passenger_to_atoms:
        if passenger in passenger_origin:
            floor = passenger_origin[passenger]
            floor_to_passengers.setdefault(floor, []).append(passenger)
        if passenger in passenger_destin:
            floor = passenger_destin[passenger]
            floor_to_passengers.setdefault(floor, []).append(passenger)

    for floor, passengers in floor_to_passengers.items():
        if len(passengers) <= 1:
            continue
        pattern_atoms = []
        for p in passengers[:6]:  # 6 passengers * ~3 atoms + 1 lift = ~19 variables
            pattern_atoms.extend(passenger_to_atoms[p])
```

```
        for atom in lift_atoms:
            if atom.binding[0] == floor:
                pattern_atoms.append(atom)
                break
        unique_vars = set()
        for atom in pattern_atoms:
            unique_vars.update(atom.binding)
        if len(unique_vars) <= 20:
            frozen_pat = frozenset(pattern_atoms)
            if frozen_pat not in existing_patterns:
                patterns.append(Pattern(pattern=pattern_atoms))
                existing_patterns.add(frozen_pat)

    if len(lift_atoms) > 1:
        floor_to_atom = {}
        floor_keys = []

        for atom in lift_atoms:
            floor_obj = atom.binding[0]
            try:
                floor_num = int(floor_obj.name[1:])
                floor_to_atom[floor_num] = atom
                floor_keys.append(floor_num)
            except (ValueError, IndexError):
                floor_to_atom[floor_obj.name] = atom
                floor_keys.append(floor_obj.name)

        try:
            sorted_floors = sorted(floor_keys)
        except TypeError:
            sorted_floors = sorted(floor_keys, key=str)

        for i in range(len(sorted_floors) - 1):
            f_curr, f_next = sorted_floors[i], sorted_floors[i + 1]
            atom_curr = floor_to_atom[f_curr]
            atom_next = floor_to_atom[f_next]

            adj_pattern = [atom_curr, atom_next]
            frozen_pat = frozenset(adj_pattern)
            if frozen_pat not in existing_patterns:
                patterns.append(Pattern(pattern=adj_pattern))
                existing_patterns.add(frozen_pat)

    goal_singletons_added = set()
    for pattern in patterns:
        if len(pattern.pattern) == 1:
            goal_singletons_added.add(pattern.pattern[0])

    for goal_atom in task_info.fluent_goal_atoms:
        if goal_atom not in goal_singletons_added:
            patterns.append(Pattern(pattern=[goal_atom]))

    if not patterns:
        for goal_atom in task_info.fluent_goal_atoms:
            patterns.append(Pattern(pattern=[goal_atom]))

    return patterns
```

Listing 6: Rovers generator code.

```
def generate_pattern_collection(task_info: TaskInformation) -> list[Pattern]:
    patterns = []
    estimated_states = 0
    MAX_TOTAL_STATES = 500_000_000
```

```
    processed_sigs = set()

    def count_vars(atom_list):
        objs = set()
        for atom in atom_list:
            objs.update(atom.binding)
        return len(objs)

    def add_pattern(atom_list):
        nonlocal estimated_states
        if not atom_list:
            return

        n_vars = count_vars(atom_list)
        if n_vars > 20:
            return

        states = 2 ** n_vars
        if estimated_states + states > MAX_TOTAL_STATES:
            return

        key = frozenset((a.predicate.name, a.binding) for a in atom_list)
        if key in processed_sigs:
            return
        processed_sigs.add(key)

        patterns.append(Pattern(pattern=list(atom_list)))
        estimated_states += states

    fluent_atoms = list(task_info.all_fluent_atoms)

    atoms_by_pred = {}
    for atom in fluent_atoms:
        pred_name = atom.predicate.name
        if pred_name not in atoms_by_pred:
            atoms_by_pred[pred_name] = []
        atoms_by_pred[pred_name].append(atom)

    store_to_rover = {}
    camera_to_rover = {}
    obj_visible_from = {}  # objective -> set of waypoints where visible
    lander_location = None
    visible_waypoints = set()  # waypoints visible from lander

    for atom in task_info.static_ground_atoms:
        if atom.predicate.name == 'store_of':
            store_to_rover[atom.binding[0]] = atom.binding[1]
        elif atom.predicate.name == 'on_board':
            camera_to_rover[atom.binding[0]] = atom.binding[1]
        elif atom.predicate.name == 'visible_from':
            obj, wp = atom.binding[0], atom.binding[1]
            if obj not in obj_visible_from:
                obj_visible_from[obj] = set()
            obj_visible_from[obj].add(wp)
        elif atom.predicate.name == 'at_lander':
            lander_location = atom.binding[1]

    if lander_location:
        for atom in task_info.static_ground_atoms:
            if atom.predicate.name == 'visible':
                wp1, wp2 = atom.binding[0], atom.binding[1]
                if wp1 == lander_location:
                    visible_waypoints.add(wp2)
                if wp2 == lander_location:
                    visible_waypoints.add(wp1)
```

```
rover_capabilities = {}
for atom in task_info.static_ground_atoms:
    if atom.predicate.name == 'equipped_for_soil_analysis':
        rover = atom.binding[0]
        if rover not in rover_capabilities:
            rover_capabilities[rover] = set()
        rover_capabilities[rover].add('soil')
    elif atom.predicate.name == 'equipped_for_rock_analysis':
        rover = atom.binding[0]
        if rover not in rover_capabilities:
            rover_capabilities[rover] = set()
        rover_capabilities[rover].add('rock')
    elif atom.predicate.name == 'equipped_for_imaging':
        rover = atom.binding[0]
        if rover not in rover_capabilities:
            rover_capabilities[rover] = set()
        rover_capabilities[rover].add('imaging')

image_goals_by_obj = {}
soil_goals = []
rock_goals = []

for goal in task_info.fluent_goal_atoms:
    if goal.predicate.name == 'communicated_image_data':
        obj, mode = goal.binding[0], goal.binding[1]
        if obj not in image_goals_by_obj:
            image_goals_by_obj[obj] = []
        image_goals_by_obj[obj].append((goal, mode))
    elif goal.predicate.name == 'communicated_soil_data':
        soil_goals.append((goal, goal.binding[0]))
    elif goal.predicate.name == 'communicated_rock_data':
        rock_goals.append((goal, goal.binding[0]))

for obj, goal_modes in image_goals_by_obj.items():
    have_images = []
    relevant_rovers = set()
    modes = []
    for goal, mode in goal_modes:
        modes.append(mode)
        for have_img in atoms_by_pred.get('have_image', []):
            if have_img.binding[1] == obj and have_img.binding[2] == mode:
                have_images.append(have_img)
                relevant_rovers.add(have_img.binding[0])

    for rover in relevant_rovers:
        if 'imaging' not in rover_capabilities.get(rover, set()):
            continue

        pattern_atoms = [gm[0] for gm in goal_modes]
        for have_img in have_images:
            if have_img.binding[0] == rover and have_img not in pattern_atoms:
                pattern_atoms.append(have_img)

        relevant_cameras = set()
        for static_atom in task_info.static_ground_atoms:
            if static_atom.predicate.name == 'supports':
                cam, cam_mode = static_atom.binding[0], static_atom.binding[1]
                if cam_mode in modes and camera_to_rover.get(cam) == rover:
                    relevant_cameras.add(cam)

        for cal in atoms_by_pred.get('calibrated', []):
            if cal.binding[1] == rover and cal.binding[0] in relevant_cameras:
                pattern_atoms.append(cal)
```

```
        relevant_wps = obj_visible_from.get(obj, set())
        priority_wps = relevant_wps & visible_waypoints
        secondary_wps = (relevant_wps | visible_waypoints) - priority_wps

        for at in atoms_by_pred.get('at', []):
            if at.binding[0] == rover and at.binding[1] in priority_wps:
                pattern_atoms.append(at)
        for at in atoms_by_pred.get('at', []):
            if at.binding[0] == rover and at.binding[1] in secondary_wps and at not in
pattern_atoms:
                pattern_atoms.append(at)

        add_pattern(pattern_atoms)

        for at in atoms_by_pred.get('at', []):
            if at.binding[0] == rover and at.binding[1] in visible_waypoints:
                comm_pattern = [gm[0] for gm in goal_modes] + [at]
                for have_img in have_images:
                    if have_img.binding[0] == rover:
                        comm_pattern.append(have_img)
                add_pattern(comm_pattern)
                break

for goal, waypoint in soil_goals:
    pattern_atoms = [goal]
    for have_soil in atoms_by_pred.get('have_soil_analysis', []):
        if have_soil.binding[1] == waypoint:
            rover = have_soil.binding[0]
            if 'soil' not in rover_capabilities.get(rover, set()):
                continue
            pattern_atoms.append(have_soil)
            for at in atoms_by_pred.get('at', []):
                if at.binding[0] == rover and at.binding[1] == waypoint:
                    pattern_atoms.append(at)
            for at in atoms_by_pred.get('at', []):
                if at.binding[0] == rover and at.binding[1] in visible_waypoints and at
not in pattern_atoms:
                    pattern_atoms.append(at)
            for empty_atom in atoms_by_pred.get('empty', []):
                store = empty_atom.binding[0]
                if store_to_rover.get(store) == rover:
                    pattern_atoms.append(empty_atom)
            for full_atom in atoms_by_pred.get('full', []):
                store = full_atom.binding[0]
                if store_to_rover.get(store) == rover:
                    pattern_atoms.append(full_atom)

    add_pattern(pattern_atoms)

    for rover in rover_capabilities:
        if 'soil' not in rover_capabilities[rover]:
            continue
        for at in atoms_by_pred.get('at', []):
            if at.binding[0] == rover and at.binding[1] in visible_waypoints:
                comm_pat = [goal]
                for have_soil in atoms_by_pred.get('have_soil_analysis', []):
                    if have_soil.binding[0] == rover and have_soil.binding[1] == waypoint
:
                        comm_pat.append(have_soil)
                        break
                if len(comm_pat) < 20:
                    comm_pat.append(at)
                add_pattern(comm_pat)
                break
```

```
    for goal, waypoint in rock_goals:
        pattern_atoms = [goal]
        for have_rock in atoms_by_pred.get('have_rock_analysis', []):
            if have_rock.binding[1] == waypoint:
                rover = have_rock.binding[0]
                if 'rock' not in rover_capabilities.get(rover, set()):
                    continue
                pattern_atoms.append(have_rock)
                for at in atoms_by_pred.get('at', []):
                    if at.binding[0] == rover and at.binding[1] == waypoint:
                        pattern_atoms.append(at)
                for at in atoms_by_pred.get('at', []):
                    if at.binding[0] == rover and at.binding[1] in visible_waypoints and at
    not in pattern_atoms:
                        pattern_atoms.append(at)
                for empty_atom in atoms_by_pred.get('empty', []):
                    store = empty_atom.binding[0]
                    if store_to_rover.get(store) == rover:
                        pattern_atoms.append(empty_atom)
                for full_atom in atoms_by_pred.get('full', []):
                    store = full_atom.binding[0]
                    if store_to_rover.get(store) == rover:
                        pattern_atoms.append(full_atom)

        add_pattern(pattern_atoms)

        for rover in rover_capabilities:
            if 'rock' not in rover_capabilities[rover]:
                continue
            for at in atoms_by_pred.get('at', []):
                if at.binding[0] == rover and at.binding[1] in visible_waypoints:
                    comm_pat = [goal]
                    for have_rock in atoms_by_pred.get('have_rock_analysis', []):
                        if have_rock.binding[0] == rover and have_rock.binding[1] == waypoint
    :
                            comm_pat.append(have_rock)
                            break
                    if len(comm_pat) < 20:
                        comm_pat.append(at)
                    add_pattern(comm_pat)
                    break

    rover_locs = {}
    for atom in atoms_by_pred.get('at', []):
        rover = atom.binding[0]
        if rover not in rover_locs:
            rover_locs[rover] = []
        rover_locs[rover].append(atom)

    for rover, locs in rover_locs.items():
        if 1 < len(locs):
            add_pattern(locs)

    rover_stores = {}
    for atom in atoms_by_pred.get('empty', []):
        store = atom.binding[0]
        if store in store_to_rover:
            rover = store_to_rover[store]
            if rover not in rover_stores:
                rover_stores[rover] = []
            rover_stores[rover].append(atom)

    for atom in atoms_by_pred.get('full', []):
        store = atom.binding[0]
        if store in store_to_rover:
```

```
            rover = store_to_rover[store]
            if rover not in rover_stores:
                rover_stores[rover] = []
            rover_stores[rover].append(atom)

    for rover, store_atoms in rover_stores.items():
        if 1 < len(store_atoms):
            add_pattern(store_atoms)

            for atom in store_atoms:
                if atom.predicate.name == 'full':
                    store = atom.binding[0]
                    r = store_to_rover.get(store)
                    if r:
                        for have_soil in atoms_by_pred.get('have_soil_analysis', []):
                            if have_soil.binding[0] == r:
                                add_pattern([atom, have_soil])
                        for have_rock in atoms_by_pred.get('have_rock_analysis', []):
                            if have_rock.binding[0] == r:
                                add_pattern([atom, have_rock])

    rover_cals = {}
    for atom in atoms_by_pred.get('calibrated', []):
        rover = atom.binding[1]
        if rover not in rover_cals:
            rover_cals[rover] = []
        rover_cals[rover].append(atom)

    for rover, cals in rover_cals.items():
        if 1 < len(cals):
            add_pattern(cals)

            for cal in cals:
                cam = cal.binding[0]
                for static_atom in task_info.static_ground_atoms:
                    if static_atom.predicate.name == 'calibration_target' and static_atom.
    binding[0] == cam:
                        target_obj = static_atom.binding[1]
                        for vis_atom in task_info.static_ground_atoms:
                            if vis_atom.predicate.name == 'visible_from' and vis_atom.binding
    [0] == target_obj:
                                cal_wp = vis_atom.binding[1]
                                for at in atoms_by_pred.get('at', []):
                                    if at.binding[0] == rover and at.binding[1] == cal_wp:
                                        add_pattern([cal, at])
                                        break
                        break

    covered_goals = set()
    for p in patterns:
        for atom in p.pattern:
            if atom in task_info.fluent_goal_atoms:
                covered_goals.add(atom)

    for goal in task_info.fluent_goal_atoms:
        if goal not in covered_goals:
            add_pattern([goal])

    if not patterns and task_info.fluent_initial_state_atoms:
        add_pattern([task_info.fluent_initial_state_atoms[0]])

    return patterns
```

Listing 7: Satellite generator code.

```
def generate_pattern_collection(task_info: TaskInformation) -> list[Pattern]:
    patterns = []

    supports_map = {}  # mode -> list of instruments
    on_board_map = {}  # instrument -> satellite
    calib_target_map = {}  # instrument -> calibration direction

    for atom in task_info.static_ground_atoms:
        if atom.predicate.name == "supports":
            inst, mode = atom.binding[0], atom.binding[1]
            if mode not in supports_map:
                supports_map[mode] = []
            supports_map[mode].append(inst)
        elif atom.predicate.name == "on_board":
            inst, sat = atom.binding[0], atom.binding[1]
            on_board_map[inst] = sat
        elif atom.predicate.name == "calibration_target":
            inst, dir = atom.binding[0], atom.binding[1]
            calib_target_map[inst] = dir

    fluent_index = {}
    for atom in task_info.all_fluent_atoms:
        pred = atom.predicate.name
        if pred not in fluent_index:
            fluent_index[pred] = []
        fluent_index[pred].append(atom)

    def get_atom(pred_name, condition):
        for atom in fluent_index.get(pred_name, []):
            if condition(atom):
                return atom
        return None

    def count_variables(atom_list):
        vars = set()
        for atom in atom_list:
            for obj in atom.binding:
                vars.add(obj)
        return len(vars)

    def deduplicate_atoms(atom_list):
        seen = set()
        result = []
        for atom in atom_list:
            if atom not in seen:
                seen.add(atom)
                result.append(atom)
        return result

    pointing_by_sat = {}
    for atom in task_info.all_fluent_atoms:
        if atom.predicate.name == "pointing" and len(atom.binding) >= 2:
            sat = atom.binding[0]
            if sat not in pointing_by_sat:
                pointing_by_sat[sat] = []
            pointing_by_sat[sat].append(atom)

    for sat, point_atoms in pointing_by_sat.items():
        if len(point_atoms) > 1:
            deduped = deduplicate_atoms(point_atoms)
            if count_variables(deduped) <= 20:
                patterns.append(Pattern(pattern=deduped))

    processed_instruments = set()
```

```
    for goal in task_info.fluent_goal_atoms:
        if goal.predicate.name != "have_image":
            patterns.append(Pattern(pattern=[goal]))
            continue

        direction, mode = goal.binding[0], goal.binding[1]

        candidate_instruments = supports_map.get(mode, [])[:3]

        for inst in candidate_instruments:
            sat = on_board_map.get(inst)
            if not sat:
                continue

            pattern_atoms = [goal]

            calib = get_atom("calibrated", lambda a: a.binding[0] == inst)
            if calib:
                pattern_atoms.append(calib)

            power = get_atom("power_on", lambda a: a.binding[0] == inst)
            if power:
                pattern_atoms.append(power)

            point = get_atom("pointing", lambda a: a.binding[0] == sat and a.binding[1] ==
    direction)
            if point:
                pattern_atoms.append(point)

            calib_dir = calib_target_map.get(inst)
            if calib_dir and calib_dir != direction:
                point_calib = get_atom("pointing", lambda a: a.binding[0] == sat and a.
    binding[1] == calib_dir)
                if point_calib and count_variables(pattern_atoms + [point_calib]) <= 20:
                    pattern_atoms.append(point_calib)

            pavail = get_atom("power_avail", lambda a: a.binding[0] == sat)
            if pavail and count_variables(pattern_atoms + [pavail]) <= 20:
                pattern_atoms.append(pavail)

            final_atoms = deduplicate_atoms(pattern_atoms)
            if len(final_atoms) > 1 and count_variables(final_atoms) <= 20:
                patterns.append(Pattern(pattern=final_atoms))

            if inst not in processed_instruments:
                processed_instruments.add(inst)
                calib_pattern = []

                if calib:
                    calib_pattern.append(calib)
                if power:
                    calib_pattern.append(power)
                if pavail:
                    calib_pattern.append(pavail)
                if calib_dir:
                    point_calib = get_atom("pointing", lambda a: a.binding[0] == sat and a.
    binding[1] == calib_dir)
                    if point_calib:
                        calib_pattern.append(point_calib)

                final_calib = deduplicate_atoms(calib_pattern)
                if len(final_calib) >= 2 and count_variables(final_calib) <= 20:
                    patterns.append(Pattern(pattern=final_calib))
```

```
    sat_to_instruments = {}
    for inst, sat in on_board_map.items():
        if sat not in sat_to_instruments:
            sat_to_instruments[sat] = []
        sat_to_instruments[sat].append(inst)

    for sat, insts in sat_to_instruments.items():
        power_pattern = []
        pavail = get_atom("power_avail", lambda a: a.binding[0] == sat)
        if pavail:
            power_pattern.append(pavail)
        for inst in insts:
            pon = get_atom("power_on", lambda a: a.binding[0] == inst)
            if pon:
                power_pattern.append(pon)
        if len(power_pattern) > 1 and count_variables(power_pattern) <= 20:
            patterns.append(Pattern(pattern=deduplicate_atoms(power_pattern)))

    goals_by_sat = {}
    for goal in task_info.fluent_goal_atoms:
        if goal.predicate.name == "have_image":
            direction, mode = goal.binding[0], goal.binding[1]
            for inst in supports_map.get(mode, []):
                sat = on_board_map.get(inst)
                if sat:
                    if sat not in goals_by_sat:
                        goals_by_sat[sat] = []
                    goals_by_sat[sat].append((goal, inst, direction))

    for sat, entries in goals_by_sat.items():
        if len(entries) > 1:
            combined = []
            pavail = get_atom("power_avail", lambda a: a.binding[0] == sat)
            if pavail:
                combined.append(pavail)

            for i, (goal, inst, direction) in enumerate(entries[:2]):
                if count_variables(combined + [goal]) > 20:
                    break
                combined.append(goal)

                calib = get_atom("calibrated", lambda a: a.binding[0] == inst)
                if calib and count_variables(combined + [calib]) <= 20:
                    combined.append(calib)

                power = get_atom("power_on", lambda a: a.binding[0] == inst)
                if power and count_variables(combined + [power]) <= 20:
                    combined.append(power)

                point = get_atom("pointing", lambda a: a.binding[0] == sat and a.binding[1]
    == direction)
                if point and count_variables(combined + [point]) <= 20:
                    combined.append(point)

            if len(combined) > 1:
                final_combined = deduplicate_atoms(combined)
                if count_variables(final_combined) <= 20 and len(final_combined) > 1:
                    patterns.append(Pattern(pattern=final_combined))

    covered_goals = set()
    for p in patterns:
        for atom in p.pattern:
            if atom in task_info.fluent_goal_atoms:
                covered_goals.add(atom)
```

```
    for goal in task_info.fluent_goal_atoms:
        if goal not in covered_goals:
            patterns.append(Pattern(pattern=[goal]))

    if not patterns and task_info.fluent_initial_state_atoms:
        patterns.append(Pattern(pattern=[task_info.fluent_initial_state_atoms[0]]))

    return patterns
```

Listing 8: Transport generator code.

```
def generate_pattern_collection(task_info: TaskInformation) -> list[Pattern]:
    patterns = []

    packages = set()
    vehicles = set()

    for atom in task_info.fluent_goal_atoms:
        if atom.predicate.name == 'at':
            packages.add(atom.binding[0])

    for atom in task_info.all_fluent_atoms:
        if atom.predicate.name == 'in':
            packages.add(atom.binding[0])  # package
            vehicles.add(atom.binding[1])  # vehicle
        elif atom.predicate.name == 'capacity':
            vehicles.add(atom.binding[0])

    for pkg in packages:
        pkg_atoms = []

        for atom in task_info.fluent_initial_state_atoms:
            if atom.predicate.name == 'at' and atom.binding[0] == pkg:
                pkg_atoms.append(atom)
                break

        for atom in task_info.fluent_goal_atoms:
            if atom.predicate.name == 'at' and atom.binding[0] == pkg:
                pkg_atoms.append(atom)
                break

        for atom in task_info.all_fluent_atoms:
            if atom.predicate.name == 'in' and atom.binding[0] == pkg:
                pkg_atoms.append(atom)

        if pkg_atoms and len(pkg_atoms) <= 20:
            patterns.append(Pattern(pattern=pkg_atoms))

    for veh in vehicles:
        veh_atoms = []
        for atom in task_info.all_fluent_atoms:
            if atom.predicate.name == 'at' and atom.binding[0] == veh:
                veh_atoms.append(atom)
            elif atom.predicate.name == 'capacity' and atom.binding[0] == veh:
                veh_atoms.append(atom)

        if veh_atoms and len(veh_atoms) <= 20:
            patterns.append(Pattern(pattern=veh_atoms))

    for veh in vehicles:
        cargo_atoms = []
        for atom in task_info.all_fluent_atoms:
            if atom.predicate.name == 'in' and atom.binding[1] == veh:
                cargo_atoms.append(atom)
```

```
            elif atom.predicate.name == 'capacity' and atom.binding[0] == veh:
                cargo_atoms.append(atom)

        if cargo_atoms:
            cargo_atoms = cargo_atoms[:20]
            patterns.append(Pattern(pattern=cargo_atoms))

    critical_locations = set()

    for atom in task_info.fluent_initial_state_atoms:
        if atom.predicate.name == 'at' and atom.binding[0] in packages:
            critical_locations.add(atom.binding[1])

    for atom in task_info.fluent_goal_atoms:
        if atom.predicate.name == 'at' and atom.binding[0] in packages:
            critical_locations.add(atom.binding[1])

    for loc in critical_locations:
        loc_atoms = [atom for atom in task_info.all_fluent_atoms
                     if atom.predicate.name == 'at' and atom.binding[1] == loc]

        if len(loc_atoms) > 20:
            priority_atoms = []
            for atom in loc_atoms:
                if atom in task_info.fluent_goal_atoms or atom in task_info.
    fluent_initial_state_atoms:
                    priority_atoms.append(atom)
            loc_atoms = priority_atoms[:20]

        if loc_atoms and len(loc_atoms) <= 20:
            patterns.append(Pattern(pattern=loc_atoms))

    covered_atoms = set()
    for pat in patterns:
        for atom in pat.pattern:
            covered_atoms.add(atom)

    for goal_atom in task_info.fluent_goal_atoms:
        if goal_atom not in covered_atoms:
            patterns.append(Pattern(pattern=[goal_atom]))

    if not patterns and task_info.fluent_initial_state_atoms:
        patterns.append(Pattern(pattern=[task_info.fluent_initial_state_atoms[0]]))

    return patterns
```