LLM-EVOLVED DOMAIN-INDEPENDENT HEURISTICS FOR SYMBOLIC AI PLANNING

Anonymous authors
Paper under double-blind review

ABSTRACT

AI planners rely on heuristic functions that estimate the cost from a state to the goal. The strongest heuristics have been hand-engineered over decades, and while large language models can now synthesize heuristics for a single domain, one heuristic that works on *any* symbolic planning task has remained out of reach. We address this with an evolutionary search over C++ heuristics: LLMs propose code mutations, a MAP-Elites archive keyed by informedness and evaluation speed maintains diversity, and fitness blends coverage with solving time. Benchmarked against nineteen hand-engineered heuristics, our evolved heuristics are both the most informative and among the fastest. On held-out domains they are competitive with the strongest baselines, and our best one outperforms all of them. Further, seeding evolution from the trivial `blind` heuristic yields a similar mean but a higher spread and ceiling compared to seeding from the state-of-the-art `FF` heuristic, and even the `FF` variants it creates beat those evolved from `FF` itself. The evolved programs are ordinary C++ that inherit the soundness and completeness of the underlying search and can be integrated into existing planners. Combining two evolved heuristics inside LAMA solves 50 more held-out tasks, yields a $1.11\times$ speedup to the first plan and improves its aggregate plan-quality score.

1 INTRODUCTION

Symbolic AI planning underpins applications such as logistics, scheduling, robotics and formal verification, where an agent must compose long action sequences to reach a goal with hard correctness requirements. Given a symbolic world model, an initial state and a goal description, the task is to find a sequence of actions that transforms the initial state into one satisfying the goal. This is PSPACE-complete (Bylander, 1994), and nearly all competitive solvers cast it as a heuristic search problem (e.g., Helmert, 2006). The heuristics powering these planners are cost-to-go estimates that guide search toward promising states, and the choice of heuristic largely determines what is solvable in practice. Designing strong heuristics is therefore central to the field. Foundational examples such as `FF` (Hoffmann & Nebel, 2001), `LM-cut` (Helmert & Domshlak, 2009) and `merge-and-shrink` (Helmert et al., 2014) each took years of theoretical work and hand-tuning.

We aim to partly automate this discovery process, and two recent lines of work motivate ours. First, large language models (LLMs) can generate strong heuristics for individual problems or *domains* (families of problems sharing a common structure) (e.g., Corrêa et al., 2025; Tuisov et al., 2026). Second, LLMs have served as mutation operators in evolutionary search over programs, typically targeting heuristics for a single domain (e.g., Romera-Paredes et al., 2024; Novikov et al., 2025). To our knowledge, we are the first to show that LLMs can produce *domain-independent* heuristics that match the hand-engineered state of the art on diverse held-out domains.

We cast heuristic discovery as an evolutionary search over C++ code for the Scorpion planner (Seipp et al., 2020), an extension of the widely used Fast Downward planning system (Helmert, 2006). Mutations are LLM-driven code rewrites, and fitness blends coverage with per-task runtime over a calibrated training set drawn from Autoscale (Torralba et al., 2021). We implement this on top of OpenEvolve (Sharma, 2025), an open-source evolutionary coding agent comparable to AlphaEvolve (Novikov et al., 2025), and evaluate on two datasets: the 2023 IPC Learning Track (Taitler et al., 2024) and held-out portions of Autoscale. In contrast to approaches that invoke an LLM at solve time to produce plans directly (e.g., Corrêa et al., 2025), our artifacts are deterministic C++

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

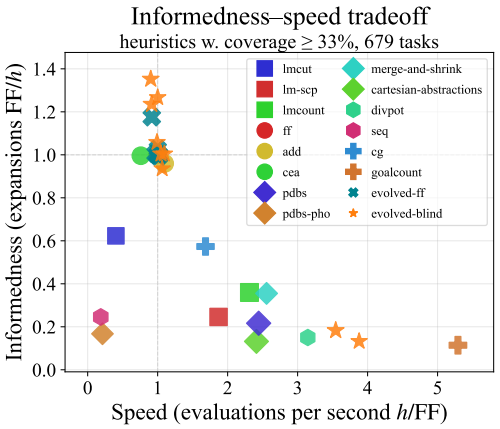


Figure 1: Informedness–speed tradeoff across heuristics; higher is better on each. Scores are calculated across the 679 test tasks solved by all heuristics, and each task is normalized by the performance of FF before a geometric mean is taken. To increase the number of commonly solved tasks, we only include heuristics that solve at least a third of the tasks. Our evolved heuristics are found at both edges of the frontier: they are the only ones more informative than FF, and they reach speeds only `goalcount` exceeds.

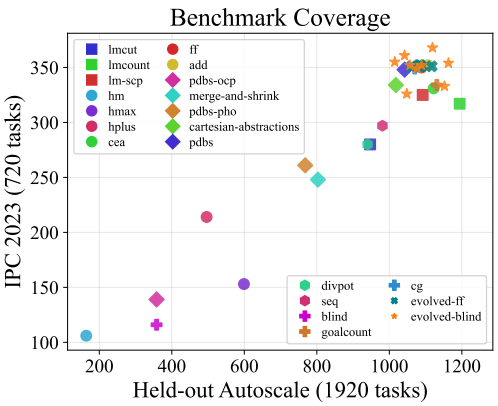


Figure 2: Coverage across two different benchmarks. For high-performing heuristics, coverage in one dataset transfers poorly to the other, with a Spearman coefficient of +0.05 for the top 50% of heuristics by IPC. Nevertheless, our evolved heuristics crowd in the best corner, with `lmcount` as the only baseline that outperforms all evolved heuristics in either benchmark, and it is then beaten by all evolved heuristics on the other.

programs that inherit the soundness and completeness guarantees of the underlying search. This sidesteps the hallucination and reliability concerns that limit LLM-as-planner methods, and keeps the approach usable in the offline, low-resource and high-assurance settings that motivate symbolic planning. Because the heuristics are domain-independent, they apply to any compatible planning task, amortizing the computational and environmental cost of evolution, and may yield insights into symbolic planning more broadly.

Our experiments vary two axes. First, we seed evolution from either the `blind` heuristic, the least informed heuristic possible, or FF, a state-of-the-art heuristic. Second, we vary LLM reasoning effort.

Figures 1 and 2 show our main results. More informed heuristics typically cost more per state, but can save time overall by avoiding unpromising expansions. We benchmark this tradeoff and are, to our knowledge, the first to compare heuristics directly on this plane. Within this space, our evolved heuristics cover both ends of the Pareto frontier: they include the most informed heuristics and some of the fastest. Further, we see that strength on one benchmark frequently does not transfer to the other. On IPC, the four best heuristics are all evolved. On Autoscale, only `lmcount` outperforms all evolved heuristic, and evolved heuristics rank second and third, yet every evolved heuristic outperforms `lmcount` on IPC. Summed over both benchmarks, the highest-coverage heuristic overall is evolved, see Section 4.2. These benefits extend to a complete state-of-the-art planner: combining `evolved-blind-medium-1` and `evolved-blind-none-3` inside LAMA solves 50 more held-out tasks, yields a $1.11\times$ speedup to the first plan and improves its aggregate plan-quality score (Table 2).

Two further findings extend beyond these numbers. First, seeding from `blind` does not raise average performance over seeding from FF, but it does produce a wider spread of heuristics and therefore a higher ceiling. Since we can select the best evolved heuristics, this spread is desirable. Notably, `blind`-seeded runs that converge to FF variants still outperform those seeded with FF. Second, reasoning effort has a modest impact, with higher effort mostly reducing compilation failures.

Our contributions are: (i) the first LLM-driven evolutionary framework to produce *domain-independent* heuristics; (ii) a tunable fitness function combining coverage with the *agile* score to reward speed; (iii) a benchmark exposing the informedness–speed tradeoff across nineteen diverse heuristics; and (iv) evolved heuristics that directly improve on state-of-the-art planning systems.

2 BACKGROUND

Planning Tasks. Planning problems are conventionally specified in the Planning Domain Definition Language (PDDL) (McDermott et al., 1998). Modern planners translate PDDL into the SAS$^+$ formalism (Bäckström & Nebel, 1995), a finite-domain representation in which the state is a vector of multi-valued variables (Helmert, 2006). All tasks we consider are *SAS*$^+$ tasks with action costs, tuples $\Pi = \langle V, A, s_0, s_\star \rangle$. V is a finite set of variables v, each with a finite domain $\mathrm{dom}(v)$. A *state* is a total assignment that maps every $v \in V$ to a value in $\mathrm{dom}(v)$, and S denotes the set of all states. The *initial state* is $s_0 \in S$ and the *goal* $s_\star$ is a partial assignment to V. A state s *satisfies the goal* when $s_\star$ agrees with s on every variable it assigns. An *action* $a \in A$ is a triple $\langle \mathrm{pre}(a), \mathrm{eff}(a), c(a) \rangle$ in which $\mathrm{pre}(a)$ and $\mathrm{eff}(a)$ are partial assignments to V and $c(a) \in \mathbb{R}_{\geq 0}$ is its *cost*. Action a is *applicable* in s when $\mathrm{pre}(a)$ agrees with s, and applying it yields the successor that agrees with $\mathrm{eff}(a)$ on every variable assigned there and with s on the rest. A *plan* is a sequence $a_1, \ldots, a_n \in A$ applicable in turn from s_0 and ending in a state that satisfies the goal, and its *cost* is $\sum_{i=1}^{n} c(a_i)$.

Heuristic Search. A *heuristic* $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$ estimates the distance to the goal from a state s, with ∞ marking a dead-end state. Search algorithms such as A* and greedy best-first search (GBFS) use h to order state expansions. We focus on the latter, which repeatedly expands the state that h deems closest to the goal. An expansion evaluates h on every successor and queues those for future expansion. Most planners combine several heuristics with quite involved search algorithms, and we evaluate some of our evolved heuristics in the state-of-the-art LAMA system (Richter et al., 2011).

`blind` and `FF` Heuristics. We use the `blind` and `FF` (Hoffmann & Nebel, 2001) heuristics as seeds for evolution. The `blind` heuristic returns 0 at goal states and the minimum action cost otherwise, providing no guidance but maximal speed. `FF` solves the task from the given state while ignoring delete effects, finding a so-called *relaxed plan*, and uses the sum of action costs in said relaxed plan as the estimate, giving strong informedness at moderate cost.

Evolutionary Algorithms. An *evolutionary algorithm* maintains a population of candidate solutions and iteratively produces new candidates by sampling parents, applying mutation or crossover, evaluating the offspring against a fitness function and replacing weaker individuals (Eiben & Smith, 2015). *Quality-diversity* variants such as MAP-Elites (Mouret & Clune, 2015) replace the single-objective archive with a grid keyed by behavioral features, retaining the best individual per cell. The grid protects locally optimal niches from being overwritten by globally fitter but behaviorally homogeneous offspring, and supplies a structured set of parents for the next generation. We use MAP-Elites because the heuristics we want to discover trade off two distinct quantities, informedness and per-state evaluation speed, and a single scalar fitness collapses that tradeoff prematurely.

3 EVOLUTIONARY FRAMEWORK

Our framework builds on OpenEvolve (Sharma, 2025) and runs a five-step loop:

1. **Generate.** Select an island, then sample a parent from its MAP-Elites archive and an LLM from the generation pool. Prompt the LLM with the parent source, the top individuals on the island and a description of the heuristic API to produce a child heuristic.
2. **Check.** Attempt to compile and call the heuristic on a minimal problem.
3. **Repair.** On a compilation or runtime error, sample an LLM from the repair pool and prompt it with the diagnostics, retrying up to a fixed number of times.
4. **Evaluate.** Run the heuristic under greedy best-first search across the training problem set with a calibrated per-task time budget to obtain a fitness score.
5. **Store.** Insert the individual into the MAP-Elites grid keyed by informedness and speed. Each cell retains the highest-scoring individual seen.

After a fixed number of such iterations, we return the highest-fitness individual.

MAP-Elites, Islands and LLM Pools. Combining ideas from diverse heuristics can yield stronger hybrids than refining any single lineage. We promote diversity through three mechanisms. First, we use MAP-Elites to maintain an archive that is diverse along the informedness–speed axes, so

that only behaviorally similar heuristics compete directly. Two heuristics in the same cell produce a single survivor (the higher-scoring one), while a more informed but slower variant occupies its own cell, preserving both points on the tradeoff. We space cell boundaries evenly between the lowest and highest feature values seen so far, letting the grid adapt to the distribution of heuristics under evolution. Second, we maintain three independent *islands*, each a separate MAP-Elites grid. Parents are sampled round-robin across islands and uniformly within the chosen island, and the LLM is further conditioned on the top individuals from that same island. Periodically, the best programs migrate between islands, letting strong ideas spread without homogenizing the population. Third, we draw from a pool of LLMs for mutation and a separate pool for repair, sampling uniformly each time. Different models bring different inductive biases, so the choice of mutation operator itself becomes a source of diversity.

Prompting and Repairs. We invoke LLMs in two roles, mutation and repair. The repair stage is necessary because the Scorpion heuristic API is largely absent from public training data, so even strong models often fail to produce compilable and executable C++ on the first attempt. Both prompts contain the API description and a description of the LLM's role. The mutation prompt additionally contains the parent source and the top three individuals on the same island, together with their scores and features. The repair prompt instead contains the failed heuristic and any compiler diagnostics. Appendix B lists the full prompts.

Evaluation. We score each generated heuristic on a training set of escalating difficulty drawn from diverse domains. On each task p, we run greedy best-first search under a per-task time limit of $T_p = \max(30\,\text{s}, 1.3 \cdot t^p_{\text{FF}})$ and a memory limit of 8 GB, where t^p_{FF} is the wall-clock time taken by FF on the same task and 8 GB is the standard planning competition limit. The 30-second floor accommodates heuristics with heavy precomputation and dampens timing noise on otherwise short runs. If a heuristic fails on a task, we abort the remainder of that domain and treat the skipped tasks as failures, conserving wall-clock budget for promising candidates.

Fitness Function. *Coverage*, the number of solved tasks, is the most common metric in planning but a coarse signal during evolution. Solving a new task often requires a large jump in heuristic quality, and the gradient flattens between such jumps. We therefore combine coverage with the *agile* score, a continuous metric that decays smoothly from 1 to 0 as solving time approaches the per-task budget. The combined signal rewards both solving more tasks and solving them faster, with a user-tunable constant α interpolating between the two. Formally, the score of a heuristic h is

$$\text{score}(h) \coloneqq \frac{1}{|P|} \sum_{p \in P} \left(\alpha \left\{ \begin{array}{ll} 1, & t^p_h \le T_p \\ 0, & t^p_h > T_p \end{array} \right\} + (1-\alpha) \left\{ \begin{array}{ll} 1, & t^p_h \le 1 \\ 1 - \dfrac{\log(t^p_h)}{\log(T_p)}, & 1 < t^p_h \le T_p \\ 0, & t^p_h > T_p \end{array} \right\} \right)$$

where P is the training set, t^p_h is the time for h to solve p and T_p is the per-task time limit.

Features. The MAP-Elites features should capture the informedness–speed tradeoff and remain well-defined when the heuristic fails to solve some tasks. For *informedness* (evaluated states), a naive average across the training set is unsuitable. Task difficulty grows exponentially across the training set, so the average is dominated by the hardest instance solved, and there is no principled value to assign on failure. We therefore normalize the number of evaluated states per task against FF on the same problem before averaging. This yields a stable scale and an interpretable failure sentinel ($10\times$ FF evaluations). For *speed* (evaluations per wall-clock second), the same concern does not apply: per-state evaluation speed is not size-dependent in the same way, and larger instances arguably give a more reliable estimate as one-time precomputation amortizes over many evaluations. We therefore use raw evaluations-per-second averaged across the training set, with 0 as the failure value. Formally,

$$\text{evals}(h) = \tfrac{1}{|P|} \textstyle\sum_{p \in P} \left\{ \begin{array}{ll} \dfrac{e^p_h}{e^p_{\text{FF}}}, & t^p_h \le T_p \\ 10, & t^p_h > T_p \end{array} \right\} \quad \text{speed}(h) = \tfrac{1}{|P|} \textstyle\sum_{p \in P} \left\{ \begin{array}{ll} \dfrac{e^p_h}{t^p_h}, & t^p_h \le T_p \\ 0, & t^p_h > T_p \end{array} \right\},$$

where e^p_h is the number of evaluations GBFS with h needs to solve p and t^p_h is the corresponding wall-clock time. Lower $\text{evals}(h)$ thus indicates a more informed heuristic. In the rest of the paper, we report its inverse, informedness, consistent with Figure 1.

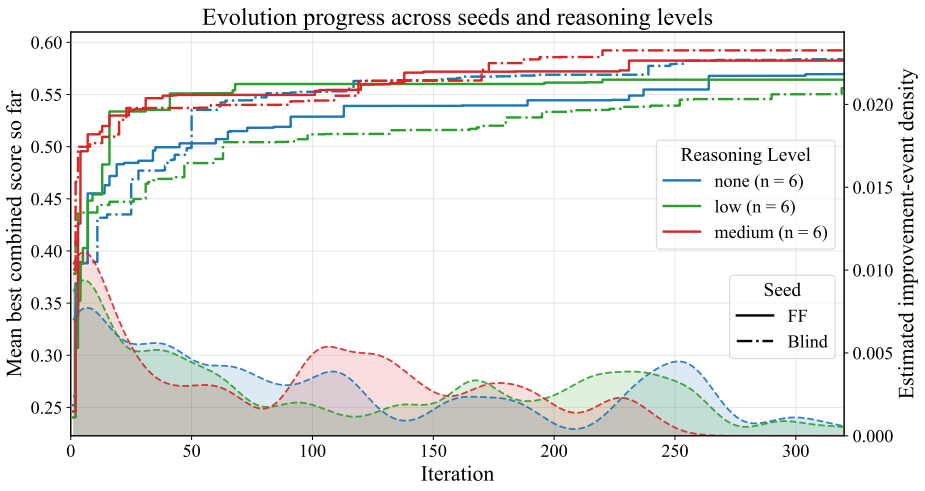


Figure 3: Average trajectories of the best-so-far training score (left axis) and a kernel-density estimate of improvement events (right axis), grouped by reasoning effort. The pronounced late-iteration drop-off at `medium` (and to a lesser extent `low`) is partly the credit-exhaustion artifact discussed in Section 4.1. Improvement density thins rather than vanishes toward the end of the budget, indicating that evolution remains productive. Runs approximate logistic improvement toward scores just above 0.6.

4 Evolved Heuristics

Having specified the framework in Section 3, we now run it. We sweep over two seeds (`blind` and `FF`) and three reasoning-effort levels (`none`, `low` and `medium`), with three runs per configuration. We then identify patterns in how evolution proceeds and highlight notable heuristics that emerge. We access LLMs via OpenRouter and use the same pool for mutation and repairs: GLM-5.1 (Z.ai, 2026), Kimi-K2.6 (Moonshot AI, 2026) and MiMo-v2.5-pro (Xiaomi, 2026). These were selected as the strongest performers on the public Artificial Analysis benchmark (Artificial Analysis, 2026) relative to their cost. Each run executes 320 iterations of the loop in Section 3 across its three islands. Further configuration details and results are available in Appendix A.

4.1 Evolution Dynamics

Three patterns hold across runs. First, the best-so-far training score climbs steadily with iteration count, with improvement events thinning toward the end of the budget rather than vanishing (Figure 3). At `medium` (and to a lesser extent `low`), this late thinning is partly an artifact: after LLM API credits ran out mid-experiment, later iterations ran on new compute nodes under heavier load, which depressed the wall-clock-sensitive agile scores. It therefore does not necessarily indicate that evolution stalled.

Second, the LLM pool emits a working heuristic on the first attempt 79.1% of the time, and only 2.4% of iterations exhaust the four-attempt repair budget (Appendix C.2). Higher reasoning effort raises first-attempt success from 73.0% (`none`) to 83.4% (`medium`), but leaves the score distribution of working heuristics essentially unchanged: reasoning affects whether a heuristic works, not how good it is. Moreover, all three models produce fitness-improving children and appear in the lineages of the final-best heuristics, suggesting that none of them is individually indispensable.

Third, seeding from `blind` rather than `FF` leaves average outcomes unchanged but widens their spread and appears to raise the ceiling. This is an advantage, since only the best heuristic found is deployed. Training scores for `blind`-seeded runs span 0.528 to 0.606, while `FF`-seeded scores span 0.543 to 0.586, and three of the nine `blind`-seeded runs exceed every `FF`-seeded run. However, exact two-sided permutation tests over all 48 620 relabelings of the 18 runs find no significant difference in training-score means ($p = 0.60$), group maxima ($p = 0.21$) or dispersion (Brown–Forsythe, $p = 0.17$). Test coverage on IPC and the held-out Autoscale set reinforces this pattern. Summing

across both datasets, the means remain indistinguishable (1434.6 for `blind` against 1437.3 for `FF`, $p = 0.89$), but `blind`-seeded runs are far more dispersed (standard deviation 52.1 against 14.2, $p = 0.0044$) and the three runs with the highest coverage are all `blind`-seeded. The same patterns hold for each dataset separately. Per-domain coverage is in Appendix D.1. Both seeding strategies tend to produce `FF` variants. This may partly be because `FF` is a well-known heuristic, but it may also reflect our fitness function, which uses tasks on which `FF`'s solve times are reasonably spread. Notably, the highest-fitness `blind`-seeded heuristic (discussed below) is an `FF` variant that achieves higher IPC and Autoscale coverage than every `FF`-seeded heuristic. This suggests that rediscovering `FF` from diverse inspirations can outperform refining it directly.

4.2 Top Evolved Heuristics

We highlight three of the top training scorers: the best `blind`-seeded (`evolved-blind-medium-1`, 0.6055, tied #1 overall), the best `FF`-seeded (`evolved-ff-medium-3`, 0.5863, #4) and the best heuristic that is not an `FF` variant (`evolved-blind-none-3`, 0.6050, #3). We evaluate them in Section 5 and their code is provided in Appendix F.2.

`evolved-blind-medium-1`, Algorithm 1. This heuristic is an `FF` variant that evolved from the `blind` seed, even though the mutating LLMs were never shown an implementation of `FF`. On top of `FF`, it first scans all goal variables and the actions that affect them, marking as *sinks* the non-goal values that no action can change and that thus can never be left once reached. During search, any state containing such a value is a dead end. This can be viewed as a restricted form of domain transition graph (Helmert, 2006) reachability check. If a state is neither a goal nor such a dead end, a relaxed plan is calculated as for `FF`. The implementation ignores unary precondition-free actions, which may yield worse relaxed plans, inflate the estimate or falsely report dead ends. Such actions are rare, however, and occur in neither of our benchmark sets. To increase efficiency, the plan extraction also uses generation counters to avoid clearing certain arrays. Once a plan is found, its length is returned as a heuristic value rather than its cost. Such unit-cost approximations have been shown to help in satisficing search for systems such as LAMA (Richter et al., 2011), but this is usually a search-level choice rather than part of the heuristic itself.

`evolved-ff-medium-3`, Algorithm 2. This heuristic computes plain `FF`. It differs only in implementation details, such as checking for goal states, caching action costs and collecting the actions of the relaxed plan in a list instead of scanning all actions.

`evolved-blind-none-3`, Algorithm 3. This heuristic ignores relaxed planning in favor of domain transition graphs (DTGs). At setup it computes backward Dijkstra distances D_v to the goal value in the DTG of each goal variable v. It then adds surcharges for cross-variable requirements to the edge weights and recomputes the distances, for at most three passes and, due to an early exit in the code, not necessarily until a fixed point. A requirement on another goal variable adds the distance from the required value back to that variable's goal value, and a requirement on a non-goal variable adds the cost of its cheapest value-changing transition, or zero if there is none. Each goal variable v also gets a *level* ℓ_v, which is zero if no other goal variable depends on v in the causal graph (CG) and otherwise one more than the highest level among those that do. The levels are propagated for at most $|V|$ rounds, so they need not converge if the CG has cycles, and are then scaled down to $[0, 3]$. Algorithm 3 combines these values with the weights $w_v = |\text{preds}_{\text{CG}}(v)| + \min(\lfloor n_v/4 \rfloor, 5)$, where n_v is the number of transitions in v's DTG, and $b_v = \max(1, |\text{preds}_{\text{CG}}(v)|) + 2\ell_v + 2$. Distances thus count more for goal variables with more CG predecessors, larger DTGs or higher levels, and the b_v terms add a weighted count of the unsatisfied goals. It is most similar to our baseline `cg`, the causal graph heuristic (Helmert, 2004), which also sums DTG distances of goal variables with transition costs that include the costs of conditions on other variables, but computes these costs from the current state in every evaluation instead of once before search.

These heuristics illustrate two modes of discovery. `evolved-blind-medium-1` and `evolved-ff-medium-3` refine `FF` with implementation-level optimizations and small algorithmic additions, such as a precomputed dead-end check and unit-cost plan lengths. `evolved-blind-none-3` instead abandons relaxed planning altogether, combining DTG distances, causal-graph structure and goal ordering into a heuristic that scores within 0.0005 of the best training score. Its weighting is hard to interpret, but it shows that evolution is not confined to refining known designs.

Table 1: Solved tasks on the 2023 IPC Learning Track (720 tasks) and the Autoscale test set (1920 tasks, held out during training) with GBFS, 30 minutes, 8 GB. Per-domain coverage and all evolved heuristics are in Appendix D.1. Our evolved heuristics are competitive with all baselines, of which `lmcount` is the strongest due to its Autoscale performance. Asterisks mark admissible heuristics.

	Landmarks			Relaxation						Abstractions					Op. count.		Other			Evolved		
	lm-scp*	lmcount	lmcut*	add	cea	ff	hm*	hmax*	hplus*	cartesian*	m&s*	pdbs*	pdbs-ocp*	pdbs-pho*	divpot*	seq*	blind*	cg	goalcount	bl-m-1	ff-m-3	bl-n-3
IPC	325	317	280	352	331	350	106	153	214	334	248	348	139	261	280	297	116	349	334	**354**	351	333
Autoscale	1092	**1194**	947	1107	1122	1089	164	599	496	1018	803	1042	358	768	940	981	358	1070	1131	1163	1087	1152
Total	1417	1511	1227	1459	1453	1439	270	752	710	1352	1051	1390	497	1029	1220	1278	474	1419	1465	**1517**	1438	1485

Algorithm 1: `evolved-blind-medium-1`

if $\mathrm{Goal}(s)$ **then return** 0;
if *s contains a sink* **then return** DEADEND;
$\pi^+ \leftarrow \mathrm{RelaxedPlan}(s)$;
if $\pi^+ = \mathrm{FAILURE}$ **then return** DEADEND;
return $\max(1, |\pi^+|)$;

Algorithm 2: `evolved-ff-medium-3` (= `ff`)

if $\mathrm{Goal}(s)$ **then return** 0;
$\pi^+ \leftarrow \mathrm{RelaxedPlan}(s)$;
if $\pi^+ = \mathrm{FAILURE}$ **then return** DEADEND;
return $\sum_{a \in \pi^+} c(a)$;

Algorithm 3: `evolved-blind-none-3`

if $\mathrm{Goal}(s)$ **then return** 0;
$U \leftarrow$ goal variables v with $s[v] \neq s_\star[v]$;
$d_v \leftarrow D_v[s[v]]$ for all $v \in U$;
if *some* $d_v = \infty$ **then return** DEADEND;
$h \leftarrow \sum_{v \in U} \big(\lfloor d_v(1 + \ell_v)(1 + w_v/4) \rfloor + b_v \big)$;
$h \leftarrow h + \max_{v \in U} d_v + 2 \max_{v \in U} \ell_v$;
return $\max(1, h)$;

Table 2: LAMA on IPC and held-out Autoscale (2640 tasks), 5 minutes, 8 GB. The top `ff` row is default LAMA. Quality is the IPC score and speed-up the geometric mean of default LAMA's time to first plan over the variant's on tasks both solve (Appendix E).

	Coverage	Quality	Speed-up
`ff`	1936	1842.2	1.00×
`bl-m-1`	1940	1854.4	1.05×
`ff-m-3`	1934	1838.9	1.01×
`bl-n-3`	1691	1622.6	0.62×
Landmark heuristic replaced by `bl-n-3`			
`ff`	1982	1901.1	1.09×
`bl-m-1`	**1986**	**1911.2**	**1.11**×
`ff-m-3`	1972	1889.3	1.09×

5 STATE-OF-THE-ART EVALUATION

We perform two experiments on the performance of our evolved heuristics, partly we compare them all to a wide suite of heuristics within GBFS, thereby focusing on their individual performance. Following this, we evaluate the three heuristics from Section 4.2 inside the state-of-the-art LAMA planning system. Configuration details and further information are available in Appendix A and E.

5.1 GBFS

We compare against 19 common domain-independent heuristics implemented in Scorpion (Seipp et al., 2020), described in Appendix A.4.

Coverage. Table 1 gives the total coverage of all baselines and the three evolved heuristics of Section 4.2 on both test sets. On the IPC, the strongest baseline, `add`, solves 352 tasks while the highest-training-score evolved heuristic, `evolved-blind-medium-1`, solves 354. More broadly, four of the nine `blind`-seeded runs exceed `add` and two match it, with the best overall solving 368, whereas all `FF`-seeded runs solve 351 or 352 tasks. On Autoscale, `lmcount` leads with 1194 tasks, followed by `evolved-blind-medium-1` at 1163, the best evolved heuristic. The strongest

heuristic thus depends on the benchmark (Figure 2): `evolved-blind-medium-1` has the highest pooled coverage at 1517, narrowly ahead of `lmcount` at 1511. Their close aggregate performance and differing domain strengths support competitiveness across benchmarks rather than a uniform advantage. Compared to `FF` at 1439, most of `evolved-blind-medium-1`'s gain comes from `openstacks` (+34), `elevators` (+26) and `parcprinter` (+24), all domains with action costs (Appendix D.1). It leads on `parcprinter` and ties for best on `openstacks`, while `evolved-blind-none-3` leads on `elevators` with 60 tasks against its 54. These gains are consistent with a benefit from using relaxed-plan lengths instead of costs. We see for all heuristics, that the performance between datasets varies heavily. As such, it is not possible to claim that there is a universally best heuristic, but our evolved heuristics are competitive across both datasets.

The evolved heuristics also complement each other: `evolved-blind-medium-1` and `evolved-blind-none-3` jointly solve 1712 distinct tasks, contributing 227 and 195 tasks unique to each within this pair. This exceeds the strongest baseline pair, `cea` and `goalcount` (1709), while the strongest pair overall is `evolved-blind-none-3` and `evolved-blind-medium-2` (1741), followed by `evolved-blind-medium-1` and `lmcount` (1738). These are oracle unions of separate runs under their full budgets, rather than portfolio coverage under a shared budget.

Time and quality. We compare search times on the same 431 tasks across both test sets, solved by all 19 heuristics in the top half by pooled coverage and requiring at least 1 s for one of them. Geometric-mean speedups over `FF` are $1.27\times$ for `evolved-blind-medium-1` ($p = 2.3 \cdot 10^{-41}$), $1.39\times$ for `evolved-blind-none-3` ($p = 0.0084$) and $1.01\times$ for `evolved-ff-medium-3` ($p = 0.42$), compared to $2.04\times$ for `lmcount` ($p = 3.9 \cdot 10^{-6}$). The p-values are from one-sided paired Wilcoxon tests on log time ratios. Appendix D.3 reports all heuristics and Holm-adjusted p-values, under which the evidence for `evolved-blind-none-3`'s advantage no longer reaches the 0.05 threshold ($p_{\text{Holm}} = 0.194$). On the IPC, `evolved-blind-medium-1` and `evolved-ff-medium-3` return plans of the same cost as `FF` on every task that both solve. Over both test sets, `evolved-blind-medium-1` has an IPC quality score of 0.498, second only to `lmcount` (0.512) and higher than `FF` (0.481).

5.2 LAMA

We first replace LAMA's `FF` heuristic, retaining landmarks and its anytime search schedule. With `evolved-blind-medium-1`, LAMA solves four more tasks, finds first plans $1.05\times$ faster and raises its quality score from 1842.2 to 1854.4 (Table 2). The speedup is small but consistent across tasks and domains (Appendix E); `evolved-ff-medium-3` matches default LAMA.

As an FF replacement, `evolved-blind-none-3` supplies no *preferred operators* and loses coverage overall, but solves 56 tasks that default LAMA does not. Replacing LAMA's landmarks with it instead adds 38 to 46 tasks, with preferred operators supplied by the FF-like heuristic. Combining `evolved-blind-medium-1` and `evolved-blind-none-3` gives the highest coverage and quality: 1986 tasks and 1911.2, versus 1936 and 1842.2 for default LAMA, with a $1.11\times$ first-plan speedup. The coverage gain is significant at task level, but concentrated in a few domains (Appendix E).

6 Related Work

Evolved Heuristics. Automatic heuristic discovery predates LLMs. Aler et al. (2001) use genetic programming over a fixed language to evolve domain-specific control heuristics, and Fukunaga (2008) similarly evolves composite heuristics for SAT from a fixed grammar of compositions. Since FunSearch (Romera-Paredes et al., 2024), more recent work has moved away from restricted languages and instead uses LLMs to mutate raw code, with strong results across distinct task families (e.g., Liu et al., 2024; Novikov et al., 2025). ReEvo (Ye et al., 2024) frames this as LLM-driven hyper-heuristic search, but each evolved heuristic is still tied to the input encoding of a single combinatorial problem, and generalization is only evaluated across instance sizes of that problem. In concurrent work, Phung et al. (2026) also build on OpenEvolve, but they produce domain-*specific* pattern collection functions, that in turn induce admissible heuristics. We follow this general trend, but are, to our knowledge, the first to LLM-generated evolution to domain-*independent* heuristics.

Learned Neural Heuristics. Deep-learning approaches train neural heuristics on data from solved tasks and generalize within a single domain (e.g., Ståhlberg et al., 2022; Fritzsche et al., 2026). Chen

et al. (2024) extend this to inter-domain generalization, but such multi-domain coverage remains rare. Neural heuristics are also inherently black-box and typically require a GPU at inference time, whereas our evolved heuristics are white-box C++ that runs on CPU.

LLMs as Planners. Corrêa et al. (2025) show that frontier LLMs can solve challenging planning tasks directly at inference time. Our setting is complementary: LLM-as-planner suits one-off tasks where formal guarantees are less critical, while our evolved artifacts are deterministic C++ inheriting the soundness and completeness of the underlying symbolic search.

LLM-Generated Solvers. Several recent methods use LLMs to produce *domain-specific* solvers. Corrêa et al. (2025) and Tuisov et al. (2026) generate domain-specific heuristics that let otherwise weak planners compete with strong ones. We instead target *domain-independent* heuristics, produce artifacts that plug into a state-of-the-art planner unchanged and search for them with a quality-diversity evolutionary framework. A separate strand generates domain-specific policies (e.g., Chen et al., 2025; Stein et al., 2026). Closest in spirit to our work, Murray et al. (2026) use LLM-driven evolution to produce Python functions that directly emit plans, a form of generalized planning competitive with state-of-the-art planners on their evaluation set. These approaches are complementary to ours: generalized planners and policies solve their target tasks quickly and often without search, but cannot extend beyond tasks admitting simple strategies, whereas our heuristics retain the search properties that make symbolic planners broadly applicable.

7 Limitations

Our evaluation is restricted to typed STRIPS with negation and action costs (McDermott et al., 1998), though the framework should extend unchanged to richer formalisms such as ADL or numeric planning (Fox & Long, 2003). Apart from the LAMA experiment, we use a single greedy best-first search configuration, which disadvantages the admissible baselines. Evolving other planner components and using the evolved heuristics in portfolios are natural next steps.

Our evolved heuristics carry no admissibility guarantees, making them unsuitable for optimal planning. Admissibility could be prompted for and checked empirically in the loop, e.g., against optimal costs on small tasks, but a guarantee would still require manual validation, which white-box code makes possible: for instance, the dead-end detection of `evolved-blind-none-3` is sound, since unreachability in a projection implies unreachability in the task.

Due to API and cluster cost, we conduct only three runs per configuration, so comparisons between configurations are suggestive rather than conclusive. An evolution run takes on average 50.7 wall clock hours and 811.2 CPU hours, and a GBFS evaluation 1509 CPU hours. We have not ablated the MAP-Elites grid resolution or the LLM pool, and our reasoning-effort finding covers only three models.

8 Conclusions

We have shown that LLM-driven evolution can produce domain-*independent* heuristics for symbolic AI planning that match the hand-engineered state of the art in satisficing search on diverse held-out IPC domains. Selected by training score alone, our evolved heuristics match the strongest baseline, while our best evolved heuristic solves 368 of 720 test tasks against 352 for the strongest baseline. The broader suite spans the Pareto frontier of informedness against speed. Unlike LLM-as-planner approaches, the evolved artifacts are deterministic C++ that inherit the soundness and completeness of the underlying search. Two findings extend beyond pure numbers. First, seeding from the `blind` heuristic does not shift average outcomes but yields a wider spread and a higher ceiling than seeding from `FF`, even when the result is itself an `FF` variant. Second, at least for the models we use, reasoning effort primarily determines whether a working heuristic is produced rather than its quality once working. Together, these findings suggest that diversity across seeds, models and the MAP-Elites archive can matter more than any single strong starting point. This opens avenues for evolving other planning components, such as the search algorithms themselves.

Ethics Statement

Our work involves no human subjects or personal data, and its main ethical consideration is the energy use, and resulting emissions, of the LLM queries and computation it requires. Evolution is not cheap: our 18 runs comprise 5760 iterations, each issuing at least one LLM call, and together take roughly 14,600 CPU hours (Section 7). However, this cost is paid once. The output is a set of deterministic C++ heuristics that run on commodity CPUs without any further model queries, and since they are domain-independent, the cost of evolution is amortized over every task they are later applied to. In contrast, approaches that query an LLM per domain or per task, whether to write a heuristic, a policy or the plan itself, incur an inference cost and environmental footprint that grows with every new problem solved. We therefore consider the re-use of results from a single bounded search preferable to continuous LLM usage at solve time. To support such re-use, we release all evolved heuristics so that others can deploy them without repeating the evolution, and we encourage future work on LLM-driven program discovery to likewise report its compute and release its artifacts.

Reproducibility Statement

The supplementary material contains the full evolutionary framework, the source code of every evolved heuristic, the training-set selection scripts and the evaluation harness, and we will release all of it upon acceptance. Appendix A.2 lists the exact training tasks, Appendix A.3 the test sets, Appendix B the complete prompts and Appendices F.2 and F.1 the verbatim source of the discussed evolved heuristics and both seeds. Section 4 and Appendix A specify all hyperparameters, models, hardware and time and memory limits. The released heuristics and their evaluation are fully reproducible, since the heuristics are deterministic C++ run inside a public planner. The evolutionary trajectories are not, because the hosted models carry no versioning or determinism guarantee.

AI Use Statement

LLMs are the object of study in this work. GLM-5.1, Kimi-K2.6 and MiMo-v2.5-pro act as mutation and repair operators inside the evolutionary framework (Section 3) and every non-baseline heuristic we report on was written by them. Separately, the authors used LLM-based coding assistants to help implement the evolutionary framework, evaluation harness and analysis and plotting scripts and LLM assistance to minor edits and to condense the manuscript text. We did not use generative AI to formulate the research questions or design the experiments. We take full responsibility for the content of this paper, including all text, claims and artifacts produced with the aid of generative AI.

References

Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Learning to solve planning problems efficiently by means of genetic programming. *Evolutionary Computation*, 9(4):387–420, 2001.

Artificial Analysis. Artificial Analysis LLM leaderboard. `https://artificialanalysis.ai/`, 2026. Accessed April 2026.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Blai Bonet. An admissible heuristic for SAS^+ planning obtained from the state equation. In Francesca Rossi (ed.), *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pp. 2268–2274. AAAI Press, 2013.

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

Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

Dillon Z. Chen, Sylvie Thiébaux, and Felipe Trevizan. Learning domain-independent heuristics for grounded and lifted planning. In Jennifer Dy and Sriraam Natarajan (eds.), *Proceedings of the*

Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024), pp. 20078–20086. AAAI Press, 2024.

Dillon Ze Chen, Johannes Zenn, Tristan Cinquin, and Sheila A. McIlraith. Language models for generalised PDDL planning: Synthesising sound and programmatic policies. In *ICAPS 2025 Workshop on Planning in the Era of LLMs (LM4Plan)*, 2025.

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

Augusto B. Corrêa, André Grahl Pereira, and Jendrik Seipp. Classical planning with LLM-generated heuristics: Challenging the state of the art with Python code. In *Proceedings of the Thirty-Ninth Annual Conference on Neural Information Processing Systems (NeurIPS 2025)*, pp. 42070–42108, 2025.

Stefan Edelkamp. Symbolic pattern databases in heuristic search planning. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso (eds.), *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, pp. 274–283. AAAI Press, 2002.

A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2nd edition, 2015.

Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

Markus Fritzsche, Elliot Gestrin, and Jendrik Seipp. Symmetry-aware transformer training for automated planning. In Chad Jenkins and Matthew Taylor (eds.), *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence (AAAI 2026)*, pp. 36236–36244. AAAI Press, 2026.

Alex S. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61, 2008.

Patrik Haslum and Héctor Geffner. Admissible heuristics for optimal planning. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock (eds.), *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pp. 140–149. AAAI Press, 2000.

Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, pp. 1007–1012. AAAI Press, 2007.

Malte Helmert. A planning heuristic based on causal graph analysis. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig (eds.), *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pp. 161–170. AAAI Press, 2004.

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

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis (eds.), *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pp. 162–169. AAAI Press, 2009.

Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen (eds.), *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pp. 140–147. AAAI Press, 2008.

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

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

Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In Craig Boutilier (ed.), *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pp. 1728–1733. AAAI Press, 2009.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, pp. 32201–32223. JMLR.org, 2024.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 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, 1998.

Moonshot AI. Kimi-K2.6. `https://www.kimi.com/ai-models/kimi-k2-6`, 2026.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. arXiv:1504.04909, 2015.

Andrew Murray, Danial Dervovic, Alberto Pozanco, and Michael Cashmore. 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, 2026.

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

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

Florian Pommerening, Gabriele Röger, and Malte Helmert. Getting the most out of pattern databases for classical planning. In Francesca Rossi (ed.), *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pp. 2357–2364. AAAI Press, 2013.

Florian Pommerening, Gabriele Röger, Malte Helmert, and Blai Bonet. LP-based heuristics for cost-optimal planning. In Steve Chien, Alan Fern, Wheeler Ruml, and Minh Do (eds.), *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, pp. 226–234. AAAI Press, 2014.

Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From non-negative to general operator cost partitioning. In Blai Bonet and Sven Koenig (eds.), *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, pp. 3335–3341. AAAI Press, 2015.

Masood Feyzbakhsh Rankooh and Jussi Rintanen. Efficient computation and informative estimation of h^+ by integer and linear programming. In Sylvie Thiébaux and William Yeoh (eds.), *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, pp. 71–79. AAAI Press, 2022.

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

Silvia Richter, Matthias Westphal, and Malte Helmert. LAMA 2008 and 2011 (planner abstract). In *IPC 2011 Planner Abstracts*, pp. 50–54, 2011.

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

Jendrik Seipp and Malte Helmert. Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577, 2018.

Jendrik Seipp, Florian Pommerening, and Malte Helmert. New optimization functions for potential heuristics. In Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein (eds.), *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pp. 193–201. AAAI Press, 2015.

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

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. `https://github.com/algorithmicsuperintelligence/openevolve`, 2025.

Silvan Sievers and Malte Helmert. Merge-and-shrink: A compositional theory of transformations of factored transition systems. *Journal of Artificial Intelligence Research*, 71:781–883, 2021.

Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In Sylvie Thiébaux and William Yeoh (eds.), *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, pp. 629–637. AAAI Press, 2022.

Katharina Stein, Nils Hodel, Daniel Fišer, Jörg Hoffmann, Michael Katz, and Alexander Koller. 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, 2026.

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

Álvaro Torralba, Jendrik Seipp, and Silvan Sievers. Automatic instance generation for classical planning. In Robert P. Goldman, Susanne Biundo, and Michael Katz (eds.), *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, pp. 376–384. AAAI Press, 2021.

Alexander Tuisov, Yonatan Vernik, and Alexander Shleyfman. Successor-generator planning with LLM-generated heuristics. In *Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)*. AAAI Press, 2026.

Xiaomi. MiMo-v2.5-pro. `https://mimo.xiaomi.com/mimo-v2-5-pro/`, 2026.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Proceedings of the Thirty-Eighth Annual Conference on Neural Information Processing Systems (NeurIPS 2024)*, 2024.

Z.ai. GLM-5.1. `https://z.ai/blog/glm-5.1`, 2026.

APPENDIX OVERVIEW

The appendix follows the order of the main paper.

- A *Experimental setup:* evolution hyperparameters and hardware, the training set, the two test sets and the baselines.
- B *LLM prompts:* the system prompt, the Scorpion C++ heuristic API reference, the mutation prompt and the repair prompt.
- C *Evolution dynamics:* score distributions over generations, per-run trajectories and LLM reliability, complementing Section 4.1.
- D *Additional GBFS results:* per-domain coverage, further coverage plots and speedups with paired statistical tests, complementing Section 5.
- E *LAMA evaluation:* configuration, statistical details and ablations.
- F *Source code:* the two seeds and the three top evolved heuristics.

A Experimental Setup

A.1 Evolution Configuration

All 18 runs share a single hyperparameter configuration: a 4×4 MAP-Elites grid per island, three islands with migration every ten iterations per island, 320 iterations per run, an agile-blend constant $\alpha = 0.25$ and the training set of Appendix A.2. The mutation prompt includes the top three programs of the parent's island, and a program that fails to compile receives up to four repair attempts. Each run executes on 16 CPU cores of an Intel Xeon Gold 6130 (2.1 GHz) node. The API cost of all 18 runs is $438, or on average $16, $24 and $33 per run at `none`, `low` and `medium` reasoning effort.

Grid Resolution. We deliberately keep the MAP-Elites grid coarse. Most of the informedness–speed plane is infeasible, since a program cannot be both maximally informed and maximally fast. A finer grid would spread the same few programs over more cells and weaken selection pressure, in the limit giving every program its own cell and making it a candidate parent regardless of quality. The 4×4 grid does not saturate either. Across runs, the median run occupies 15 of the 48 cells on its three islands (9 to 33), and the last previously empty cell is filled between iterations 216 and 319, with a median around 310.

Seeds and Reasoning Effort. The `blind` seed is simple and uninformed, leaving room for diverse improvements, whereas `FF` already solves the training set and concentrates evolutionary pressure on solving time through either informedness or speed. Appendix F.1 lists both seeds. Per-call response timeouts scale with reasoning effort (300, 600 and 900 s for `none`, `low` and `medium`) to give the models room to produce longer outputs.

A.2 Training Set

The training set comprises 10 domains $\times$ 10 problems $=$ 100 problem instances drawn from the Autoscale benchmark suite (Torralba et al., 2021), which ships per-domain generators that scale instance difficulty parametrically. We draw from both its `21.11-optimal-strips` and `21.11-agile-strips` sub-suites.

Domain Selection. The ten domains were chosen for structural diversity across the Autoscale catalogue: `tpp`, `barman`, `freecell`, `gripper`, `hiking`, `snake`, `sokoban`, `termes`, `tidybot` and `transport`. All other Autoscale domains form the held-out Autoscale test set, and the two that also appear in the 2023 IPC Learning Track (`sokoban` and `transport`) are dropped from the IPC test set (Appendix A.3).

Per-Domain Problem Selection. Within each domain we select 10 problems ordered by FF runtime, supplying an escalating difficulty curve from instances solved in under a second to ones taking over two minutes. This serves two purposes: it prevents the training signal from being dominated by either extreme, and it supplies the per-task FF-runtime baseline t^p_{FF} that calibrates the per-task time limit T_p in Section 3. Per-instance runtimes span 0.65 s to 147.63 s across the full set. We aimed to select even spreads across all domains, but the distribution of FF runtimes varies widely between them, so some domains have more tightly clustered selections than others. Table 3 lists the exact 100 problems used.

Table 3: Exact training problems used. Within each domain the 10 problems are ordered shortest-first by FF runtime. Each cell lists the Autoscale sub-suite (`o` = `21.11-optimal-strips`, `s` = `21.11-agile-strips`) and problem index, followed by the FF runtime in seconds.

Domain	Problem (sub-suite + index / FF runtime [s])									
`tpp`	o5/0.65	s4/1.12	o6/2.11	s5/5.29	o8/8.76	o7/12.69	o11/35.23	s7/46.52	o9/60.40	s8/109.70
`barman`	s1/1.89	o11/3.70	o18/4.49	o25/10.59	o26/17.55	o19/23.05	o27/33.76	o17/41.38	o28/58.22	s5/133.86
`freecell`	o13/1.00	s14/3.07	s16/5.21	s17/9.86	s18/17.14	o28/42.11	s19/59.06	s20/92.68	o29/128.59	s24/147.63
`gripper`	s9/1.34	s10/1.95	s11/2.89	s13/3.97	s16/8.97	s19/13.91	s21/20.32	s23/28.13	s26/39.88	s30/69.45
`hiking`	o12/2.23	o13/5.16	o14/12.20	o17/18.79	o8/27.14	o19/47.28	o20/59.08	s5/74.11	o21/92.28	o23/143.43
`snake`	o6/1.01	o8/1.62	o19/1.91	o21/3.50	o7/4.37	o23/5.29	o10/8.67	o13/21.15	o11/33.01	o15/65.12
`sokoban`	o2/1.18	o6/4.36	o26/6.36	o13/10.15	o3/20.35	o27/38.15	o29/56.66	o7/77.82	o12/119.20	o21/134.40
`termes`	s2/1.02	s1/1.40	o27/2.75	s18/4.95	s21/8.69	s3/14.57	s20/25.14	s8/63.45	s23/102.81	s25/138.16
`tidybot`	o11/1.93	s3/2.15	o26/4.08	s4/7.18	o9/10.11	s7/13.00	s11/21.21	s14/41.67	o15/75.40	o12/98.39
`transport`	o27/1.69	o10/1.72	o25/2.62	s1/4.63	o29/5.54	o22/5.83	s2/7.45	o17/7.78	o19/25.63	s3/42.33

A.3 TEST SETS

We evaluate on two test sets, 2640 tasks in total.

IPC Learning Track. The first test set is drawn from the 2023 International Planning Competition (IPC) Learning Track (Taitler et al., 2024), the most recent IPC track built explicitly to challenge general-purpose solvers, which ships 10 domains with 90 test problems each. We evaluate on the eight domains that do not appear in our training set: `blocksworld`, `childsnack`, `ferry`, `floortile`, `miconic`, `rovers`, `satellite` and `spanner`. The two domains shared with training, `sokoban` and `transport`, are excluded. Although Autoscale and the IPC Learning Track use independent problem generators, dropping these domains removes any residual risk of cross-contamination between training and test. This yields $8 \times 90 = 720$ tasks.

Held-Out Autoscale. The second test set contains every Autoscale domain other than the ten training domains, 32 in total. For each domain we include all 30 tasks of `21.11-optimal-strips` and all 30 of `21.11-agile-strips`, giving $32 \times 60 = 1920$ tasks. Six of these domains (`blocksworld`, `childsnack`, `floortile`, `miconic`, `rovers` and `satellite`) also appear in the IPC test set, but with instances from different generators, so the pooled test sets cover 34 distinct domains.

Configuration. Every heuristic runs in greedy best-first search (`eager_greedy`) under the standard IPC budget of 30 minutes and 8 GB of memory, using one core per task. This keeps the comparison attributable to the heuristic itself rather than to a configurable solver pipeline, but is an unfavorable configuration for the admissible baselines, which are designed for optimal rather than satisficing search. Baselines use Scorpion's release build. The evolved heuristics are compiled as a plugin in the build used during evolution, which has the same optimization level but keeps assertions enabled, so this difference can only slow the evolved heuristics down.

A.4 BASELINES

We compare against 19 common domain-independent heuristics implemented in Scorpion (Seipp et al., 2020). They represent the major families used in modern planners: critical paths (`hm` (Haslum & Geffner, 2000)), delete-relaxation (`ff` (Hoffmann & Nebel, 2001), `add`/`hmax` (Bonet & Geffner, 2001), `cea` (Helmert & Geffner, 2008), `hplus` (Rankooh & Rintanen, 2022)), causal graph analysis (`cg` (Helmert, 2004)), landmarks (`lmcut` (Helmert & Domshlak, 2009), `lmcount` (Richter & Westphal, 2010), `lm-scp` (Karpas & Domshlak, 2009; Seipp et al., 2020)), abstractions (`pdbs` (Edelkamp, 2002; Haslum et al., 2007; Pommerening et al., 2013), `cartesian-abstractions` (Seipp & Helmert, 2018), `pdbs-ocp` (Pommerening et al., 2013; 2015), `merge-and-shrink` (Sievers & Helmert, 2021), `pdbs-pho` (Pommerening et al., 2013)), operator-counting (`seq` (Bonet, 2013; Pommerening et al., 2014), `divpot` (Seipp et al., 2015)) and the simple baselines `goalcount` and `blind`. Six of these heuristics are designed for satisficing search (`add`, `ff`, `cg`, `goalcount`, `cea` and `lmcount`) and form the fairest comparison under GBFS. We include the admissible heuristics as further reference points and note that several remain competitive. While the performance of individual heuristics can vary heavily between domains, `FF` is frequently considered state of the art.

B LLM PROMPTS

The system message is the concatenation of a task framing (Appendix B.1) and the Scorpion C++ heuristic API reference (Appendix B.2). Both mutation and repair calls use this system message and differ only in the user message: mutations use the diff-based mutation prompt (Appendix B.3) and repairs the repair prompt (Appendix B.5). Placeholders in braces (`{fitness_score}`, `{current_program}`, etc.) are filled by OpenEvolve at call time from the parent program, the island's population and prior evaluation artifacts. All prompts are reproduced verbatim, up to line breaks and, for the API reference, replacing a few non-ASCII characters and one table with ASCII equivalents for typesetting.

B.1 SYSTEM PROMPT: TASK FRAMING

```
# Task
You are an expert in AI planning and C++ programming competing to design the
best possible heuristic function for the Scorpion planner (a Fast Downward
fork) for classical PDDL planning.

The goal is to beat state-of-the-art planning heuristics. Playing it safe is
not expected. This is a search over a large space of ideas, and bold,
unconventional approaches are exactly what is needed. Many attempts will
follow, each building on the best results so far, so take risks: try novel
combinations, unusual data structures, or heuristic ideas that haven't been
tried before.

Each submission is a modified version of the C++ heuristic. The following
must be preserved:
- The class name `MyHeuristic`
- The plugin key `TypedFeature("my_h")`
- The fixed `MyHeuristicFeature` class and `_plugin` registration at the end
  of the file

Design considerations:
- Do heavy preprocessing in the constructor (runs once at search start)
  rather than in `compute_heuristic` (called for every evaluated state)
- Return `DEAD_END` for provably unsolvable states
- Return `0` for goal states
- The search is greedy best-first, so optimise for good state ordering, not
  admissibility
- The heuristic need not be admissible
```

B.2 SYSTEM PROMPT: SCORPION C++ HEURISTIC API

The system message appends the following Scorpion C++ heuristic API reference so that the LLM has the type signatures and idioms it needs to write valid plugin code.

````
# Scorpion Heuristic API Reference

When writing a heuristic, your class inherits from `Heuristic` and receives `task_proxy` (type
`TaskProxy`).
Inside `compute_heuristic`, always call `State state = convert_ancestor_state(ancestor_state)`
first.

---

## Checking goals and applicability

```cpp
task_properties::is_goal_state(task_proxy, state)   // -> bool
task_properties::is_applicable(op, state)           // -> bool
```

---

## Reading state values

```cpp
````
````

```
state[var_id]        // -> FactProxy  (by integer index)
state[var_proxy]     // -> FactProxy  (by VariableProxy)
state.size()         // -> size_t     (number of variables)
for (FactProxy f : state) { ... }   // iterate all facts

FactProxy fact = state[i];
int var   = fact.get_variable().get_id();
int value = fact.get_value();
fact.is_mutex(other_fact)               // -> bool
```

Variables

```cpp
VariablesProxy vars = task_proxy.get_variables();
vars.size()             // number of state variables
vars[i]                 // -> VariableProxy

VariableProxy var = vars[i];
var.get_id()            // -> int
var.get_domain_size()   // -> int  (number of possible values)
var.get_fact(value)     // -> FactProxy for (var, value)

// All facts across all variables:
for (FactProxy f : vars.get_facts()) { ... }
```

Goals

```cpp
GoalsProxy goals = task_proxy.get_goals();
goals.size()    // number of goal facts
goals[i]        // -> FactProxy
for (FactProxy g : goals) { ... }
```

Operators

```cpp
OperatorsProxy ops = task_proxy.get_operators();
ops.size()      // number of operators
ops.empty()     // bool
ops[i]          // -> OperatorProxy  (by index)
ops[op_id]      // -> OperatorProxy  (by OperatorID)
for (OperatorProxy op : ops) { ... }

OperatorProxy op = ops[i];
op.get_id()           // -> int
op.get_cost()         // -> int

// Preconditions
PreconditionsProxy prec = op.get_preconditions();
prec.size(), prec.empty(), prec[i]   // -> FactProxy
for (FactProxy p : prec) { ... }

// Effects
EffectsProxy effs = op.get_effects();
effs.size(), effs[i]   // -> EffectProxy
for (EffectProxy eff : effs) {
    FactProxy fact = eff.get_fact();   // what this effect sets
}
```

Generating successors

```cpp
// Check applicability first, then generate successor (unregistered, unpacked):
if (task_properties::is_applicable(op, state)) {
    State succ = state.get_unregistered_successor(op);
    // succ[i], iterate, etc.
}
```

````
## Task-level utilities

```cpp
task_properties::get_min_operator_cost(task_proxy)     // -> int
task_properties::get_average_operator_cost(task_proxy) // -> double
task_properties::get_operator_costs(task_proxy)        // -> vector<int>
task_properties::get_num_facts(task_proxy)             // -> int  (total facts)
task_properties::get_num_total_effects(task_proxy)     // -> int
task_properties::is_unit_cost(task_proxy)              // -> bool

// Initial state (unregistered):
State init = task_proxy.get_initial_state();

// Causal graph (see section below):
const causal_graph::CausalGraph &cg = task_proxy.get_causal_graph();
```

---

## Causal Graph

The causal graph encodes dependencies between variables. All returned lists are sorted and
exclude the queried variable itself.

```cpp
#include "../task_utils/causal_graph.h"

const causal_graph::CausalGraph &cg = task_proxy.get_causal_graph();

cg.get_successors(var)    // -> const vector<int> &  vars that var influences (pre->eff + eff->eff arcs)
cg.get_predecessors(var)  // -> const vector<int> &  vars that influence var

// Finer-grained arc types:
cg.get_pre_to_eff(var)    // vars where var appears as precondition and they appear as effect
cg.get_eff_to_pre(var)    // vars where var appears as effect and they appear as precondition
cg.get_eff_to_eff(var)    // vars that share an effect with var (eff->eff arcs)
```

Useful for ordering subgoals, detecting independent subproblems, or weighting goal facts by
their position in the dependency structure.

---

## Domain Transition Graphs (DTGs)

A DTG for variable `var` is a directed graph whose nodes are the possible values of that
variable and whose edges are the operators that change it. Each edge carries the operator's
cost and the local preconditions that variable's transition requires. DTGs are the basis for
the CG heuristic and provide richer cost estimates than the CG topology alone.

```cpp
#include "domain_transition_graph.h"   // path: heuristics/domain_transition_graph.h
#include "../algorithms/priority_queues.h"

using namespace domain_transition_graph;
```

### Building DTGs

```cpp
// pruning_condition(dtg_var, cond_var) -> true means: prune transitions of dtg_var
// whose preconditions involve cond_var (used to avoid infinite recursion).
// Pass [](int,int){return false;} for no pruning (all transitions included).
std::function<bool(int, int)> pruning = [](int, int) { return false; };
DTGFactory factory(task_proxy, /*collect_side_effects=*/false, pruning);
std::vector<std::unique_ptr<DomainTransitionGraph>> dtgs = factory.build_dtgs();
// dtgs[var] is the DTG for variable var; one DTG per state variable.
```

### DTG structure

```cpp
DomainTransitionGraph *dtg = dtgs[var].get();
dtg->var                          // int -- the variable this DTG belongs to
dtg->nodes                        // vector<ValueNode> -- one node per value (indexed by value)
dtg->local_to_global_child[i]     // int -- maps local precondition index -> global variable ID
```

### ValueNode -- a single value in a DTG

```cpp
````
````

```
ValueNode &node = dtg->nodes[value];
node.value                          // int -- value this node represents
node.transitions                    // vector<ValueTransition> -- outgoing edges
```

ValueTransition -- one target value reachable from this node

```cpp
ValueTransition &trans = node.transitions[i];
trans.target                        // ValueNode* -- destination node
trans.labels                        // vector<ValueTransitionLabel> -- one label per operator achieving this transition
```

ValueTransitionLabel -- one operator achieving a transition

```cpp
ValueTransitionLabel &label = trans.labels[j];
label.op_id                         // int -- operator index
label.is_axiom                      // bool -- if true, index into task_proxy.get_axioms(); else get_operators()
label.precond                       // vector<LocalAssignment> -- local preconditions that must hold
label.effect                        // vector<LocalAssignment> -- side effects on other local variables
```

LocalAssignment -- a condition/effect on a local variable

```cpp
LocalAssignment &assign = label.precond[k];
assign.local_var                    // short -- local index; convert with dtg->local_to_global_child[local_var]
assign.value                        // short -- required/assigned value
```

FactPair (lightweight data struct)

```cpp
FactPair fp = fact.get_pair();
fp.var    // int -- variable ID
fp.value  // int -- value
FactPair::no_fact  // sentinel for "not found"
```

Return values from compute_heuristic

```cpp
return 0;          // goal state
return N;          // estimated distance (any positive integer)
return DEAD_END;   // provably unsolvable (constant from Heuristic)
```

Subclassing RelaxationHeuristic (for relaxed-graph heuristics)

Instead of inheriting from `Heuristic` directly, inherit from `RelaxationHeuristic` to get a
preprocessed relaxation graph built once at construction. It converts the full task into:

- `vector<Proposition> propositions` -- one per domain fact, indexed by PropID (use `get_prop_id(var, value)` to convert)
- `vector<UnaryOperator> unary_operators` -- one per (operator x effect) pair

All types (`Proposition`, `UnaryOperator`, `PropID`, `OpID`) and the sentinel `NO_OP = -1` live
in
the `relaxation_heuristic` namespace -- add `using namespace relaxation_heuristic;` at file
scope.
Use `NO_OP` when initialising or checking `prop.reached_by`.

The constructor takes an extra first argument `tasks::AxiomHandlingType axioms`, and the
feature
class uses `add_relaxation_heuristic_options_to_feature` /
`get_relaxation_heuristic_arguments_from_options`
instead of the plain heuristic variants:

```cpp
using namespace relaxation_heuristic;

class MyHeuristic : public RelaxationHeuristic {
public:
    MyHeuristic(tasks::AxiomHandlingType axioms,
                const shared_ptr<AbstractTask> &transform, bool cache_estimates,
```

```
                const string &description, utils::Verbosity verbosity)
        : RelaxationHeuristic(axioms, transform, cache_estimates, description, verbosity) {}
};
```

Key structs:

```cpp
struct Proposition {
    int cost;                           // h-value in current exploration
    OpID reached_by;                    // which unary op last achieved this (30-bit); NO_OP = -1 when unset
    unsigned is_goal : 1;
    unsigned marked : 1;                // for preferred operator marking
    int num_precondition_occurences;
    array_pool::ArrayPoolIndex precondition_of;  // ops that need this as precondition
};

struct UnaryOperator {
    int cost;                           // base_cost + sum/max of precondition costs
    int base_cost;                      // original operator cost
    int unsatisfied_preconditions;      // countdown during exploration
    PropID effect;                      // the fact this op produces
    int num_preconditions;
    array_pool::ArrayPoolIndex preconditions;
    int operator_no;                    // index into task operators
};
```

Useful inherited members and methods:

```cpp
// Data
vector<Proposition> propositions;
vector<UnaryOperator> unary_operators;
vector<PropID> goal_propositions;
array_pool::ArrayPool preconditions_pool;     // backing store for operator precondition lists
array_pool::ArrayPool precondition_of_pool;   // backing store for proposition->operator lists

// Convert a state fact to PropID:
PropID get_prop_id(int var, int value) const;
PropID get_prop_id(const FactProxy &fact) const;

// Convert PropID back to FactPair:
FactPair get_fact(PropID id) const;

// Convert struct reference to index (pointer arithmetic):
PropID get_prop_id(const Proposition &prop) const;
OpID   get_op_id(const UnaryOperator &op) const;

// Pointer access by index:
Proposition    *get_proposition(PropID id);
UnaryOperator  *get_operator(OpID id);
Proposition    *get_proposition(int var, int value);

// Iterate preconditions of a unary operator (returns an iterable ArrayPoolSlice):
array_pool::ArrayPoolSlice get_preconditions(OpID op_id) const;
// Usage: for (PropID prec : get_preconditions(op_id)) { ... }

// Iterate operators that have a given proposition as a precondition:
// for (OpID op_id : precondition_of_pool.get_slice(
//          prop->precondition_of, prop->num_precondition_occurences)) { ... }
```

Include: `#include "relaxation_heuristic.h"`

AdaptiveQueue (for Dijkstra-style exploration)

`AdaptiveQueue<Value>` is a priority queue that automatically switches between a bucket queue (fast for small integer keys) and a binary heap (for large key ranges).

```cpp
#include "../algorithms/priority_queues.h"

priority_queues::AdaptiveQueue<PropID> queue;  // or any value type

queue.push(int key, Value value);   // insert with priority = key (lower = higher priority)
queue.pop();                        // -> pair<int, Value>  (key, value) of minimum-key element
queue.empty();                      // -> bool
queue.clear();                      // reset for reuse

// Typical exploration loop:
```

```
queue.push(0, start_prop);
while (!queue.empty()) {
    auto [cost, prop_id] = queue.pop();
    // process...
    queue.push(new_cost, next_prop);
}
```

ArrayPool (variable-length lists in flat storage)

`ArrayPool` stores many variable-length arrays in a single flat `vector`, avoiding per-list heap allocations. Used by `RelaxationHeuristic` for precondition lists.

```cpp
#include "array_pool.h"   // path: heuristics/array_pool.h

array_pool::ArrayPool<int> pool;

// Build phase -- append items and record the index:
ArrayPoolIndex idx = pool.append(items.begin(), items.end(), items.size());

// Access phase -- get an iterable slice:
array_pool::ArrayPoolSlice<int> slice = pool.get_slice(idx, num_items);
for (int item : slice) { ... }
// or:
slice.begin(), slice.end(), slice.size()
```

Only useful if building your own relaxation-style graph with many per-node lists.

Preprocessing pattern

The standard idiom for non-trivial heuristics is to do all heavy work in the constructor (once per search), and keep `compute_heuristic` lightweight:

```cpp
MyHeuristic(const shared_ptr<AbstractTask> &transform, ...) : Heuristic(...) {
    // build internal graph / tables from task_proxy here
    // e.g. iterate operators, build reverse-index arrays, etc.
}

virtual int compute_heuristic(const State &ancestor_state) override {
    State state = convert_ancestor_state(ancestor_state);
    if (task_properties::is_goal_state(task_proxy, state)) return 0;
    // reset per-state data, run exploration, return result
}
```

Constructor has access to `task_proxy` (inherited). Typical constructor work:
- Iterate `task_proxy.get_variables()` to size arrays
- Iterate `task_proxy.get_operators()` to build relaxed graph
- Precompute reverse indices (e.g. which operators are affected by each fact)

However, you do not have to follow this pattern, you can do more work in `compute_heuristic` if needed. Nevertheless, be mindful of performance since `compute_heuristic` is called for every evaluated state.

Landmarks

A landmark is a fact (or disjunction of facts) that must be true at some point on every solution path. `LandmarkFactory` computes a `LandmarkGraph` -- a set of landmarks with ordering relations -- given a planning task.

Includes:
```cpp
#include "../landmarks/landmark.h"
#include "../landmarks/landmark_graph.h"
#include "../landmarks/landmark_factory.h"
```

Computing the graph in the constructor:
```cpp
MyHeuristic::MyHeuristic(
    const shared_ptr<landmarks::LandmarkFactory> &lm_factory,
    const shared_ptr<AbstractTask> &transform, ...)
```

````
    : Heuristic(transform, ...) {
    // task is an inherited shared_ptr<AbstractTask> member (not task_proxy)
    lm_graph = lm_factory->compute_landmark_graph(task);
}
shared_ptr<landmarks::LandmarkGraph> lm_graph;  // store as member
```

`compute_landmark_graph` is expensive (comparable to an FF computation).

**Iterating the graph:**
```cpp
for (const auto &node : *lm_graph) {
    const landmarks::Landmark &lm = node->get_landmark();
    // lm.atoms              -- vector<FactPair>  facts that form this landmark
    // lm.is_true_in_goal    -- bool
    // lm.first_achievers    -- unordered_set<int> operator IDs that first achieve it
    // lm.possible_achievers -- unordered_set<int> all operator IDs that can achieve it
    // node->get_id()        -- int  stable index into the graph
    // node->parents         -- unordered_map<LandmarkNode*, OrderingType>
    // node->children        -- unordered_map<LandmarkNode*, OrderingType>
}
int n   = lm_graph->get_num_landmarks();
bool has = lm_graph->contains_atomic_landmark(FactPair{var, val});
```

**Available factory algorithms** (specify as `lm_factory` plugin option):

- `lm_rhw` -- RPG/SASP (Richter-Helmert-Westphal). Backchains through the
  relaxed planning graph; fast and the standard choice.
- `lm_exhaust` -- Exhaustive RPG. Systematically disproves each candidate
  fact; finds more landmarks than `lm_rhw` but slower.
- `lm_zg` -- Zhu-Givan. Graph-based; tends to find fewer landmarks than
  RPG methods.
- `lm_hm` -- hm-landmarks. Finds landmarks via hm-reachability; can find
  more landmarks for m>1 but cost is exponential in m.
- `lm_merged(lm1=X(), lm2=Y())` -- Union of two factories. Merges two
  landmark graphs to get more landmarks than either alone.

**Adding the factory option to your plugin:**
```cpp
// In Feature constructor:
add_option<shared_ptr<landmarks::LandmarkFactory>>(
    "lm_factory",
    "landmark factory",
    "lm_rhw()");

// In create_component:
opts.get<shared_ptr<landmarks::LandmarkFactory>>("lm_factory")
```

---

## Required includes

```cpp
#include "../heuristic.h"
#include "../plugins/plugin.h"
#include "../task_utils/task_properties.h"
#include "../utils/logging.h"
// if using the causal graph:
#include "../task_utils/causal_graph.h"
// if subclassing RelaxationHeuristic:
#include "relaxation_heuristic.h"
#include "array_pool.h"
// if using AdaptiveQueue:
#include "../algorithms/priority_queues.h"
// if using landmarks:
#include "../landmarks/landmark.h"
#include "../landmarks/landmark_graph.h"
#include "../landmarks/landmark_factory.h"
```
````

B.3 User Prompt: Diff-Based Mutation

`{artifacts}` contains the parent's last execution output, such as a smoke-test error, and is empty when there is none. `{evolution_history}` is filled from the sub-template in Appendix B.4. The sentence "Do not generally rewrite the entire program ..." fills a `{scope_directive}` placeholder with a single variation, so it appears in every prompt.
````

```
# Current Program Information
- Fitness: {fitness_score}
- Feature coordinates: {feature_coords}
- Focus areas: {improvement_areas}

{artifacts}

# Program Evolution History
{evolution_history}

# Current Program
```{language}
{current_program}
```

Task
Suggest improvements to the program that will improve its FITNESS SCORE.
The system maintains diversity across these dimensions: {feature_dimensions}
Different solutions with similar fitness but different features are valuable.

Do not generally rewrite the entire program, focus on targeted and iterative improvements.

You MUST use the exact SEARCH/REPLACE diff format shown below to indicate
changes:

<<<<<<< SEARCH
Original code to find and replace (must match exactly)
=======
New replacement code
>>>>>>> REPLACE

Example of valid diff format:
<<<<<<< SEARCH
for i in range(m):
    for j in range(p):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
=======
# Reorder loops for better memory access pattern
for i in range(m):
    for k in range(n):
        for j in range(p):
            C[i, j] += A[i, k] * B[k, j]
>>>>>>> REPLACE

You can suggest multiple changes. Each SEARCH section must exactly match
code in the current program.
Be thoughtful about your changes and explain your reasoning thoroughly.
```

B.4 Evolution-History Block

The `{evolution_history}` placeholder above is filled from the following sub-template, where `{top_programs}` expands to the three highest-scoring programs on the parent's island, each with its score, feature values and full source code.

```
## Top Performing Programs

{top_programs}
```

B.5 User Prompt: Repair on Compile Failure

When a candidate program fails to compile, the repair step (Section 3) uses the following user-message template instead of the mutation prompt, together with the same system message. `{error_message}`

is the compiler output, {repair_context} repeats the Scorpion API reference followed by the compiler output and {broken_code} is the failing program, which on later attempts is the output of the previous failed repair. Programs that compile but fail the smoke test are not repaired. They receive a score of 0, and the error reaches later mutations through {artifacts}.

```
The following {language} program failed to evaluate due to the error below.
Repair the program using minimal SEARCH/REPLACE diffs.
Preserve all structure, class names, plugin keys, and invariants that are unrelated to the error.

# Error

{error_message}

# Additional context

{repair_context}

# Broken Program

‘‘‘{language}
{broken_code}
‘‘‘

Use the exact SEARCH/REPLACE diff format shown below.
Each SEARCH block must match the broken program exactly (including whitespace).
Multiple diff blocks are allowed.

<<<<<<< SEARCH
# exact lines to replace
=======
# corrected replacement
>>>>>>> REPLACE
```

C Evolution Dynamics

This appendix complements Section 4.1. Appendix C.1 shows how training scores evolve, and Appendix C.2 how reliably the LLMs produce working heuristics and how much each model contributes.

C.1 Score Trajectories

Figure 4 complements the best-so-far trajectories of Figure 3 by reporting the distribution of training scores per generation across all evolved heuristics, and Figure 5 shows the individual runs behind Figure 3.

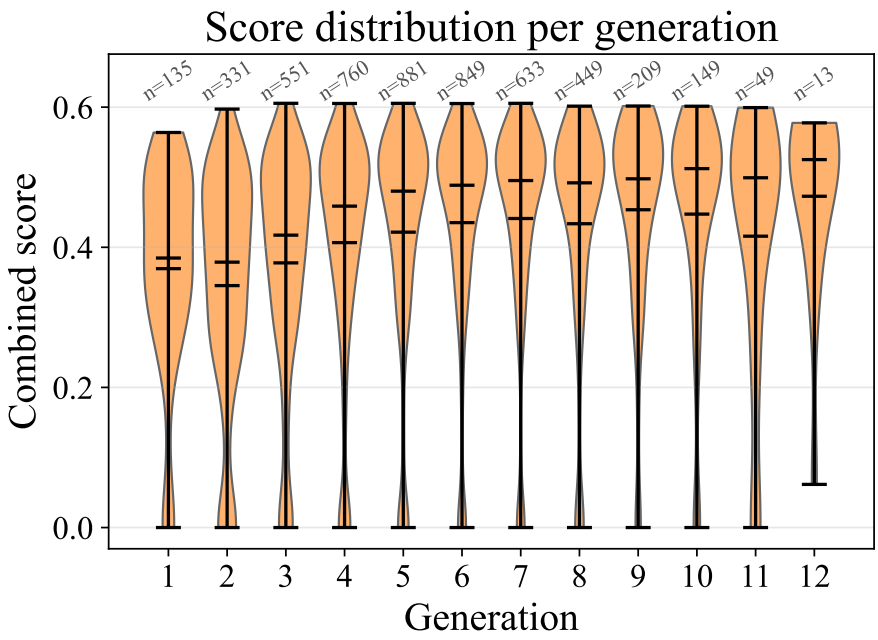


Figure 4: Distribution of training scores at each generation across all evolved runs. Later generations trend toward higher scores, aligning with the best-so-far improvement curves of Figure 3 and indicating that the gains in per-run maxima reflect a broader population-level shift rather than isolated lucky children.

C.2 LLM Reliability and Contributions

Table 4 counts how many repair attempts each generated heuristic needs before it compiles, Figure 6 shows how reasoning effort shifts the distribution of training scores and Table 5 breaks calls, fitness-improving children and contributions to final-best lineages down by model.

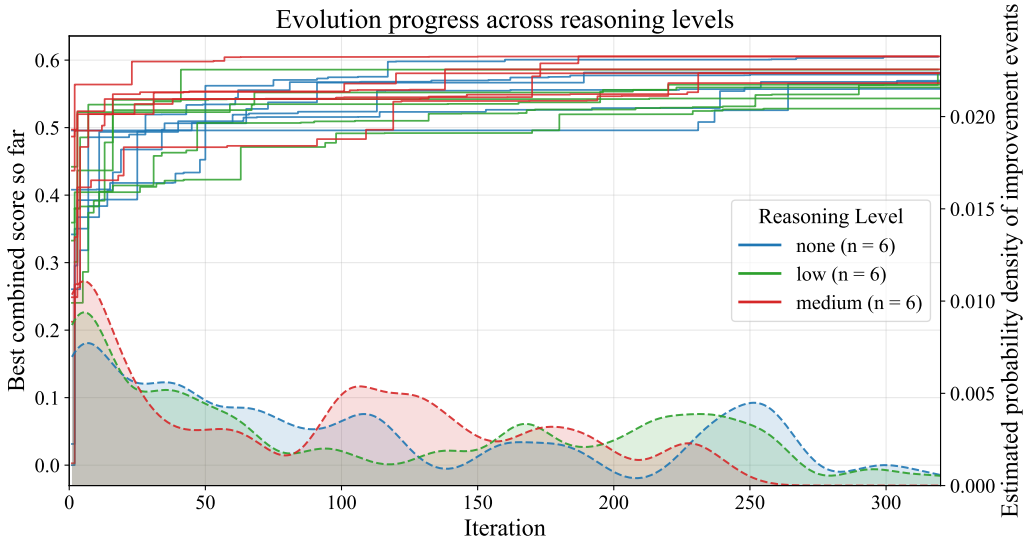


Figure 5: Per-run trajectories of the best-so-far training score (left axis) alongside a kernel-density estimate of improvement events over iterations (right axis), grouped by reasoning effort. This is the per-run counterpart of Figure 3. Trajectories cluster more by run than by reasoning level, and improvement density thins rather than vanishes toward the end of the budget.

Table 4: Repair-attempt outcomes per reasoning effort. Most heuristics compile on the first attempt or recover within one repair, and only 1.3% to 4.2% fail all four repair attempts. Rows sum to less than the 1920 iterations per effort level because iterations without a parseable program, for example when the model exhausts its response timeout while reasoning, never reach compilation. This is most frequent at `low`.

	No repair	1	2	3	4	Failed
none	1303	266	77	38	27	75
low	1247	212	37	11	3	24
medium	1411	189	52	15	2	22
Total	3961	667	166	64	32	121

Table 5: LLM performance. `mimo`, `glm` and `kimi` abbreviate MiMo-v2.5-pro, GLM-5.1 and Kimi-K2.6. 'Improv.' counts fitness-improving children, 'In best' counts contributions to final-best lineages.

Model	Calls	Improv.	In best	$/call
`mimo`	2109	584	36	$0.0415
`glm`	2723	691	36	$0.0636
`kimi`	2150	599	31	$0.0824

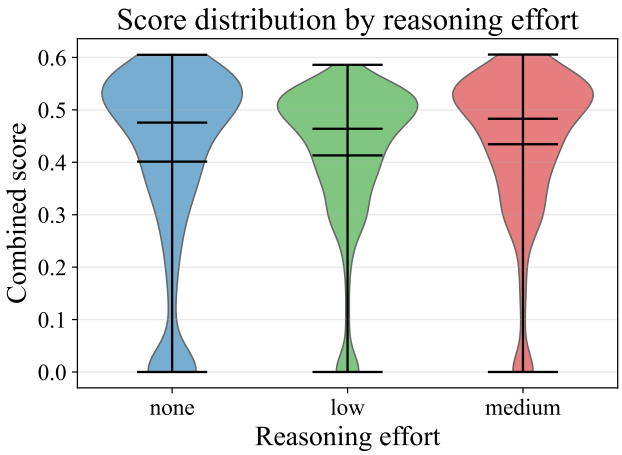


Figure 6: Distribution of training scores grouped by reasoning effort. The primary effect is to reduce the number of 0-scoring heuristics, consistent with Table 4. The distribution of scores above 0 is essentially unchanged, suggesting that reasoning effort primarily affects the likelihood of producing a working heuristic rather than the quality of those that work.

D ADDITIONAL GBFS RESULTS

This appendix complements Section 5 with results for all 37 heuristics: the 19 baselines of Appendix A.4 and all 18 evolved runs (nine FF-seeded and nine `blind`-seeded).

D.1 PER-DOMAIN COVERAGE

Table 6 breaks coverage down by domain for the IPC test set and Table 7 for the held-out Autoscale test set.

Table 6: Per-domain coverage (solved tasks out of 90 per domain, 720 in total) on the eight IPC Learning Track test domains for all baselines and evolved runs. Bold marks the best value per column.

	blocksworld	childsnack	ferry	floortile	miconic	rovers	satellite	spanner	Σ
Landmarks									
lm-scp	30	13	73	3	**90**	39	47	**30**	325
lmcount	39	15	51	4	**90**	41	47	**30**	317
lmcut	29	19	60	11	71	30	30	**30**	280
Relaxation									
add	**44**	29	43	14	**90**	33	69	**30**	352
cea	40	29	41	10	79	36	66	**30**	331
ff	28	25	67	12	**90**	34	64	**30**	350
hm	7	6	9	5	27	11	11	**30**	106
hmax	16	11	11	**21**	30	19	15	**30**	153
hplus	15	12	41	8	49	29	30	**30**	214
Abstractions									
cartesian-abstractions	26	18	65	3	**90**	**50**	52	**30**	334
merge-and-shrink	16	16	22	10	77	30	47	**30**	248
pdbs	29	25	**74**	10	**90**	43	47	**30**	348
pdbs-ocp	6	13	15	1	30	24	20	**30**	139
pdbs-pho	22	18	54	4	70	32	31	**30**	261
Op. count.									
divpot	23	18	32	9	**90**	38	40	**30**	280
seq	26	18	61	5	**90**	32	35	**30**	297
Other									
blind	8	9	10	2	30	15	12	**30**	116
cg	34	29	41	7	**90**	39	**79**	**30**	349
goalcount	32	25	69	3	**90**	38	47	**30**	334
Evolved (FF seed)									
evolved-ff-none-1	28	25	67	12	**90**	34	65	**30**	351
evolved-ff-none-2	28	25	68	12	**90**	34	65	**30**	352
evolved-ff-none-3	28	25	68	12	**90**	34	65	**30**	352
evolved-ff-low-1	28	25	67	12	**90**	34	65	**30**	351
evolved-ff-low-2	28	25	67	12	**90**	34	65	**30**	351
evolved-ff-low-3	28	25	68	12	**90**	34	65	**30**	352
evolved-ff-medium-1	28	25	67	12	**90**	34	65	**30**	351
evolved-ff-medium-2	28	25	67	12	**90**	34	65	**30**	351
evolved-ff-medium-3	28	25	67	12	**90**	34	65	**30**	351
Evolved (blind seed)									
evolved-blind-none-1	34	**30**	67	11	**90**	32	67	**30**	361
evolved-blind-none-2	24	25	69	3	**90**	38	47	**30**	326
evolved-blind-none-3	31	25	69	3	**90**	38	47	**30**	333
evolved-blind-low-1	35	27	67	13	**90**	32	61	**30**	355
evolved-blind-low-2	28	26	67	12	**90**	34	65	**30**	352
evolved-blind-low-3	**44**	**30**	43	11	**90**	32	69	**30**	349
evolved-blind-medium-1	28	26	69	12	**90**	34	65	**30**	354
evolved-blind-medium-2	39	**30**	64	13	**90**	34	68	**30**	**368**
evolved-blind-medium-3	28	26	67	12	**90**	34	65	**30**	352

Table 7: Per-domain coverage (solved tasks out of 60 per domain, 1920 in total) on the 32 held-out Autoscale test domains for the same heuristics as Table 6. The domains are split over two pages, and the total appears on the second one. Bold marks the best value per column.

	agricola	airport	blocksworld	childsnack	data-network	depots	driverlog	elevators	floortile	ged	grid	logistics	miconic	mprime	nomystery	openstacks
Landmarks																
lm-scp	57	22	33	13	**36**	46	49	14	6	27	35	32	**60**	26	**27**	2
lmcount	**60**	24	35	14	35	**48**	52	14	6	27	36	32	**60**	27	26	2
lmcut	38	18	36	15	15	28	33	25	9	27	20	33	**60**	17	11	2
Relaxation																
add	59	34	**38**	23	18	32	40	35	**11**	29	33	30	**60**	34	7	2
cea	57	**52**	**38**	23	18	30	40	36	10	27	34	42	**60**	**37**	6	2
ff	50	32	35	20	20	32	37	28	9	29	36	35	**60**	30	15	2
hm	0	0	9	5	3	2	2	2	8	5	3	10	2	0	4	2
hmax	**60**	22	24	8	16	19	6	17	**11**	27	12	20	5	10	7	2
hplus	0	2	24	7	8	12	29	5	4	3	17	25	32	5	8	2
Abstractions																
cartesian-abstractions	53	28	35	12	24	36	39	19	6	29	31	33	**60**	27	19	2
merge-and-shrink	27	0	28	14	21	31	39	11	8	27	26	33	56	25	15	2
pdbs	16	26	33	20	34	39	47	14	6	28	33	**44**	42	32	19	2
pdbs-ocp	0	4	10	8	23	3	11	3	3	1	6	16	7	6	5	2
pdbs-pho	0	22	30	13	31	33	35	8	4	27	14	32	58	11	10	2
Op. count.																
divpot	59	20	34	11	18	32	25	11	9	27	33	19	12	34	25	2
seq	48	16	33	7	26	28	36	6	8	17	32	33	**60**	29	24	2
Other																
blind	6	10	13	5	15	12	2	11	5	27	10	11	3	5	4	2
cg	20	21	36	25	24	31	38	39	7	29	34	43	**60**	**37**	6	2
goalcount	6	32	33	20	32	34	**53**	40	6	**60**	29	26	**60**	29	10	25
Evolved (FF seed)																
evolved-ff-none-1	50	32	35	20	18	32	37	31	9	29	36	35	**60**	30	15	2
evolved-ff-none-2	50	32	35	20	20	32	37	28	9	29	35	35	**60**	30	15	2
evolved-ff-none-3	50	32	35	20	20	32	37	28	9	29	36	35	**60**	30	15	2
evolved-ff-low-1	50	32	35	20	20	32	37	28	9	29	36	35	**60**	30	15	2
evolved-ff-low-2	50	32	35	20	20	32	37	28	9	29	36	35	**60**	30	15	2
evolved-ff-low-3	50	32	35	20	18	32	37	34	9	29	36	35	**60**	30	15	2
evolved-ff-medium-1	50	32	35	20	18	33	37	34	9	29	36	35	**60**	30	15	**36**
evolved-ff-medium-2	50	32	35	20	20	32	37	28	9	29	36	35	**60**	30	15	2
evolved-ff-medium-3	50	32	35	20	20	32	37	28	9	29	36	35	**60**	30	15	2
Evolved (blind seed)																
evolved-blind-none-1	42	32	**38**	25	19	33	36	31	9	29	33	30	**60**	26	11	2
evolved-blind-none-2	16	22	33	20	32	27	42	9	6	32	33	42	**60**	35	26	2
evolved-blind-none-3	16	20	33	20	32	34	45	**60**	6	41	33	42	**60**	35	26	25
evolved-blind-low-1	51	36	**38**	22	19	33	37	30	10	29	33	30	**60**	27	12	2
evolved-blind-low-2	50	32	35	20	20	33	37	28	9	29	36	35	**60**	30	17	2
evolved-blind-low-3	51	34	**38**	**28**	17	31	36	30	9	29	34	30	**60**	24	12	18
evolved-blind-medium-1	50	32	35	20	26	33	37	54	10	29	36	35	**60**	30	16	**36**
evolved-blind-medium-2	50	44	37	**28**	21	32	37	36	10	29	**37**	30	**60**	32	9	7
evolved-blind-medium-3	39	32	35	20	20	33	37	28	9	29	36	35	**60**	30	17	2

Table 7: Per-domain coverage on the held-out Autoscale test set (continued).

	organic-synthesis-split	parcprinter	parking	pathways	pegsol	pipesworld-notankage	pipesworld-tankage	rovers	satellite	scanalyzer	storage	tetris	thoughtful	visitall	woodworking	zenotravel	Σ
Landmarks																	
lm-scp	20	55	26	42	**60**	44	28	**60**	33	32	31	35	27	**60**	19	35	1092
lmcount	23	43	**60**	38	**60**	**50**	**37**	**60**	34	**54**	31	42	45	**60**	21	38	**1194**
lmcut	37	57	35	39	**60**	31	28	39	34	25	**33**	24	25	36	25	32	947
Relaxation																	
add	35	43	38	41	**60**	31	27	40	44	48	**33**	33	34	38	34	43	1107
cea	25	48	37	40	**60**	28	29	40	44	47	**33**	32	33	37	31	46	1122
ff	**50**	36	34	40	**60**	37	31	51	40	41	**33**	30	34	35	27	40	1089
hm	1	11	3	11	45	6	3	2	0	6	3	6	0	1	6	3	164
hmax	37	25	9	19	**60**	23	19	9	9	12	5	24	**50**	6	13	13	599
hplus	1	51	10	35	37	10	8	54	19	9	12	7	0	18	16	26	496
Abstractions																	
cartesian-abstractions	17	44	37	**46**	**60**	42	27	**60**	35	14	32	25	42	35	19	30	1018
merge-and-shrink	1	50	31	33	**60**	26	19	41	34	23	32	5	5	31	18	31	803
pdbs	21	54	46	35	**60**	47	22	47	34	42	32	25	10	56	27	49	1042
pdbs-ocp	1	57	3	15	56	12	2	31	2	5	9	6	5	31	6	9	358
pdbs-pho	5	49	30	15	53	38	10	41	33	28	31	15	5	33	18	34	768
Op. count.																	
divpot	16	53	58	14	58	32	27	59	33	22	**33**	24	36	44	30	30	940
seq	22	56	36	15	**60**	15	14	53	33	27	31	**46**	45	**60**	**38**	25	981
Other																	
blind	13	12	8	13	56	20	21	5	6	10	6	26	5	5	6	5	358
cg	14	48	40	45	**60**	38	21	40	**53**	**54**	**33**	40	13	38	31	**50**	1070
goalcount	17	36	**60**	35	**60**	**50**	32	**60**	34	**54**	31	40	7	**60**	22	38	1131
Evolved (FF seed)																	
evolved-ff-none-1	49	16	34	40	**60**	37	31	51	40	34	**33**	30	34	35	26	40	1061
evolved-ff-none-2	49	36	33	40	**60**	37	31	51	40	40	**33**	29	34	35	27	40	1084
evolved-ff-none-3	49	36	33	40	**60**	37	31	51	40	40	**33**	30	34	35	27	40	1086
evolved-ff-low-1	49	36	34	40	**60**	37	31	51	41	41	**33**	29	34	35	27	40	1088
evolved-ff-low-2	47	36	34	40	**60**	37	31	51	40	41	**33**	29	34	35	27	40	1085
evolved-ff-low-3	47	36	34	40	**60**	37	31	51	40	34	**33**	29	34	35	26	40	1081
evolved-ff-medium-1	48	36	34	40	**60**	37	31	51	40	34	**33**	29	34	35	26	40	1117
evolved-ff-medium-2	49	36	33	40	**60**	37	31	51	40	40	**33**	29	34	35	27	40	1085
evolved-ff-medium-3	49	36	34	40	**60**	37	31	51	40	41	**33**	29	34	35	27	40	1087
Evolved (blind seed)																	
evolved-blind-none-1	31	27	33	43	**60**	33	32	43	40	41	**33**	30	35	38	26	41	1042
evolved-blind-none-2	17	42	**60**	35	**60**	**50**	23	**60**	34	**54**	31	29	7	**60**	20	29	1048
evolved-blind-none-3	17	42	**60**	35	**60**	**50**	26	**60**	34	**54**	32	29	7	**60**	27	31	1152
evolved-blind-low-1	39	16	32	37	**60**	35	32	39	37	42	32	24	28	35	27	31	1015
evolved-blind-low-2	47	36	34	40	**60**	37	31	51	40	41	**33**	30	34	35	27	40	1089
evolved-blind-low-3	37	16	37	35	**60**	32	27	42	42	47	32	33	34	47	33	41	1076
evolved-blind-medium-1	41	**60**	34	40	**60**	37	31	51	40	34	**33**	31	34	35	23	40	1163
evolved-blind-medium-2	45	44	31	43	**60**	44	33	40	43	34	**33**	29	31	38	29	43	1119
evolved-blind-medium-3	35	27	34	40	**60**	37	31	51	40	41	**33**	30	34	35	27	40	1057

D.2 Coverage Plots

Four plots complement the coverage results of Section 5, all pooling both test sets (2640 tasks):

- Figure 7 shows the unfiltered informedness–speed scatter across all heuristics, including the weak baselines excluded from Figure 1 to obtain a larger set of shared tasks.
- Figure 8 shows the outcomes for each heuristic and task, including all evolved runs.
- Figure 9 reports, for each ordered pair of heuristics, the share of tasks which the former solves and the latter does not.
- Figure 10 reports the Jaccard similarity between the per-heuristic sets of solved tasks.

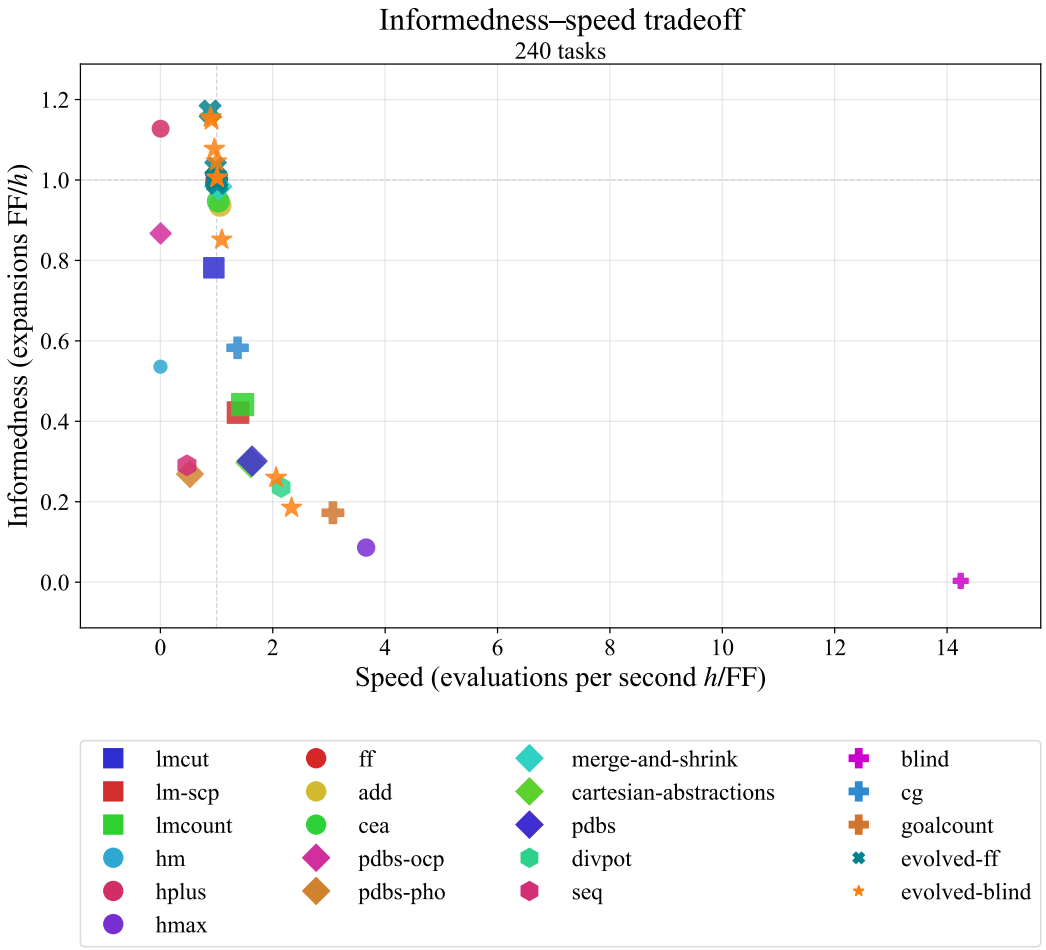


Figure 7: Unfiltered informedness–speed scatter across all heuristics, including the weak baselines that were excluded from Figure 1 to obtain a larger set of shared tasks. The two plots use different task sets and are therefore not directly comparable. Some heuristics shift, since this plot is dominated by easier tasks.

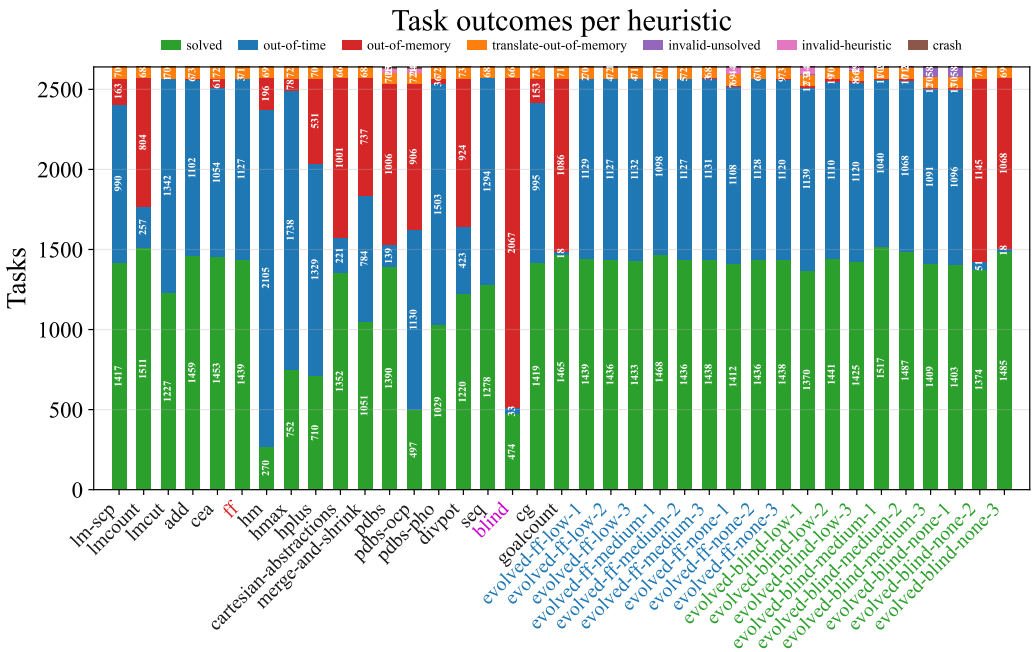


Figure 8: Stacked task outcomes per heuristic on the pooled IPC Learning Track and held-out Autoscale test sets (2640 tasks): solved, out of time, out of memory, translator out of memory, invalid heuristic value and crash. Every evolved run appears as its own bar.

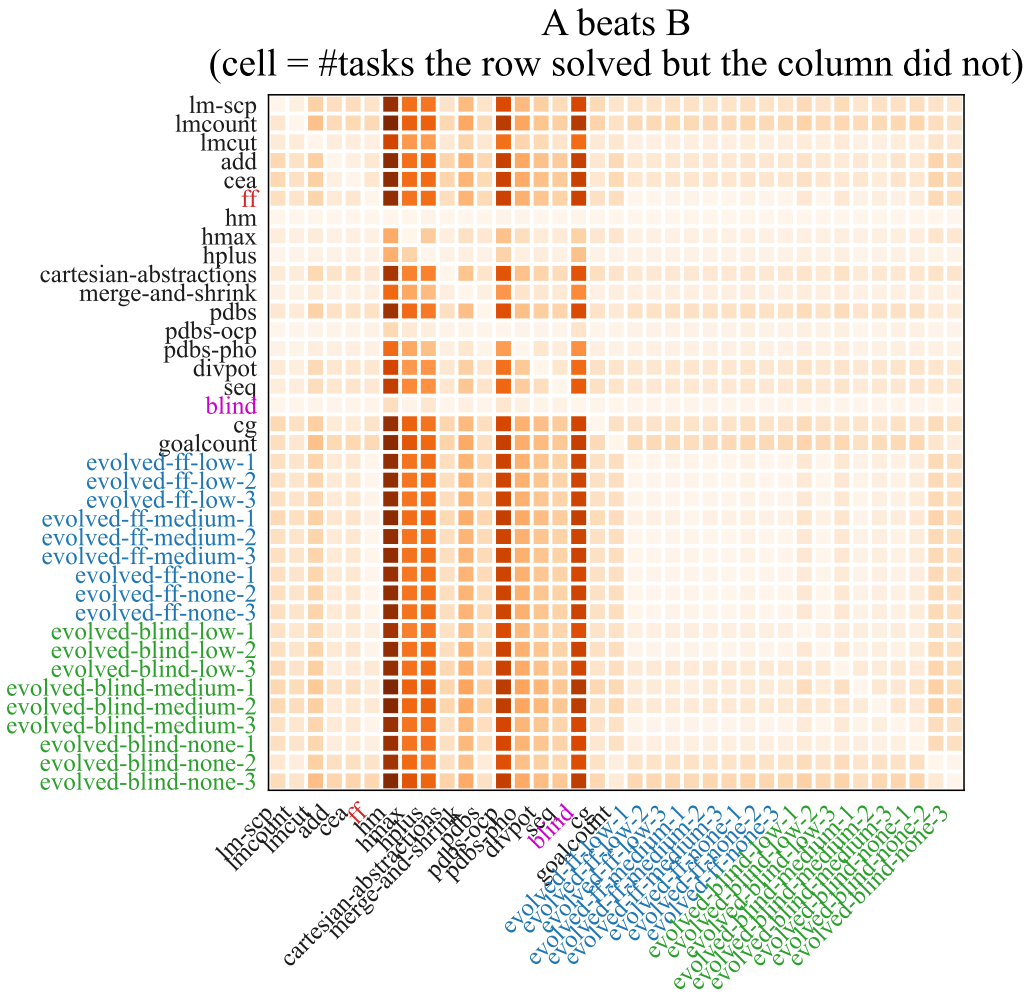


Figure 9: For each ordered pair of heuristics, the share of tasks in the pooled IPC Learning Track and held-out Autoscale test sets that the row solves and the column does not.

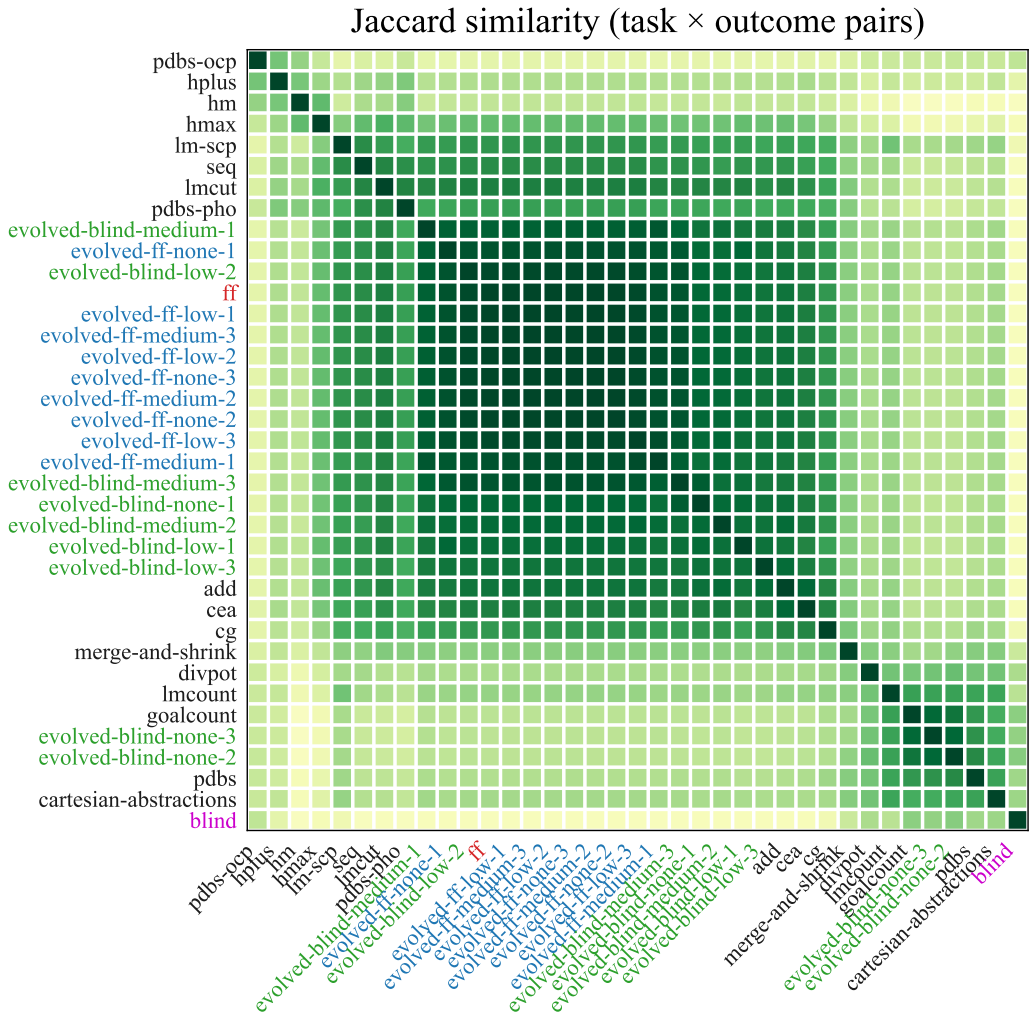


Figure 10: Jaccard similarity between the per-heuristic sets of solved tasks on the pooled IPC Learning Track and held-out Autoscale test sets. High values mark heuristics which are similar in behavior. We recover two primary clusters, the time-bound (left, around FF) and the memory-bound (right), along with smaller clusters.

D.3 Speedups and Statistical Tests

We rank all 37 heuristics by coverage on the pooled test sets (2640 tasks) and select the top half, rounding up to 19 heuristics. The cutoff is 1419 solved tasks, with no tie at the boundary. All 19 solve the same 1040 tasks (267 IPC and 773 Autoscale). To exclude trivial instances while preserving a common comparison set, we retain tasks on which at least one of these 19 heuristics takes at least 1 s of search time. This leaves 431 tasks (87 IPC and 344 Autoscale). For heuristics outside the top half, we use the subset of these 431 tasks that they solve and report its size n explicitly in Table 8. Rows with $n < 431$ therefore use different task sets and should not be ranked directly against rows using the full set.

Speedup is $\exp\big(\frac{1}{n}\sum_i \log(t_{\texttt{FF},i}/t_{h,i})\big)$, with search times floored at 0.01 s. Values above 1 favor the compared heuristic. We apply a one-sided paired Wilcoxon signed-rank test to these log ratios, with the alternative of a positive location shift. We omit zero differences and use SciPy's automatic method selection, rounding log ratios to 12 decimal places for the test to preserve numerical ties. The test concerns the signed ranks of log ratios rather than their mean, so its result need not agree with the direction of the geometric-mean speedup for skewed distributions. Table 8 reports both raw p-values and Holm-adjusted values across the 36 comparisons against `FF`. These are task-level tests conditional on the selected benchmark instances, not estimates of uncertainty across domains or evolutionary runs. The three top evolved heuristics have Holm-adjusted values $7.9{\cdot}10^{-40}$ (`evolved-blind-medium-1`), 0.194 (`evolved-blind-none-3`) and 1.000 (`evolved-ff-medium-3`). The table and a CSV containing both filtered and unfiltered results are generated by `code/gbfs_common_speed.py`.

Table 8: GBFS speedups over FF, pooling IPC and held-out Autoscale. Rows are ordered by total coverage out of 2640 tasks. The first 19 rows define the common task set, separated from the remaining heuristics by a rule. n counts solved tasks in its fixed 431-task nontrivial subset. Speedups are geometric means of paired search-time ratios, with values above 1 favoring the listed heuristic. p is the one-sided Wilcoxon value and p_{Holm} corrects across all 36 comparisons.

Heuristic	Coverage	n	Speed-up	p	p_{Holm}
evolved-blind-medium-1	1517	431	1.27	$2.3 \cdot 10^{-41}$	$7.9 \cdot 10^{-40}$
lmcount	1511	431	2.04	$3.9 \cdot 10^{-6}$	$1.1 \cdot 10^{-4}$
evolved-blind-medium-2	1487	431	1.21	$5.0 \cdot 10^{-5}$	0.001
evolved-blind-none-3	1485	431	1.39	0.008	0.194
evolved-ff-medium-1	1468	431	1.07	$1.7 \cdot 10^{-9}$	$5.6 \cdot 10^{-8}$
goalcount	1465	431	1.31	0.030	0.627
add	1459	431	0.87	0.152	1.000
cea	1453	431	0.41	1.000	1.000
evolved-blind-low-2	1441	431	1.14	$2.4 \cdot 10^{-35}$	$8.1 \cdot 10^{-34}$
evolved-ff-low-1	1439	431	0.99	0.999	1.000
ff	1439	431	1.00	—	—
evolved-ff-medium-3	1438	431	1.01	0.420	1.000
evolved-ff-none-3	1438	431	1.06	$1.3 \cdot 10^{-6}$	$3.9 \cdot 10^{-5}$
evolved-ff-low-2	1436	431	1.04	$2.8 \cdot 10^{-4}$	0.008
evolved-ff-medium-2	1436	431	1.04	0.003	0.071
evolved-ff-none-2	1436	431	0.95	1.000	1.000
evolved-ff-low-3	1433	431	0.95	0.252	1.000
evolved-blind-low-3	1425	431	0.87	0.039	0.783
cg	1419	431	1.25	0.001	0.028
lm-scp	1417	405	0.49	1.000	1.000
evolved-ff-none-1	1412	431	1.05	$8.0 \cdot 10^{-7}$	$2.5 \cdot 10^{-5}$
evolved-blind-medium-3	1409	431	1.27	$3.3 \cdot 10^{-61}$	$1.2 \cdot 10^{-59}$
evolved-blind-none-1	1403	431	1.26	$1.4 \cdot 10^{-11}$	$4.5 \cdot 10^{-10}$
pdbs	1390	403	0.71	0.641	1.000
evolved-blind-none-2	1374	416	1.23	0.027	0.601
evolved-blind-low-1	1370	419	0.81	0.118	1.000
cartesian-abstractions	1352	391	0.78	0.957	1.000
seq	1278	335	0.04	1.000	1.000
lmcut	1227	365	0.08	1.000	1.000
divpot	1220	306	1.61	0.001	0.029
merge-and-shrink	1051	318	0.83	0.350	1.000
pdbs-pho	1029	281	0.02	1.000	1.000
hmax	752	154	0.07	1.000	1.000
hplus	710	103	$5.3 \cdot 10^{-4}$	1.000	1.000
pdbs-ocp	497	49	$1.7 \cdot 10^{-3}$	1.000	1.000
blind	474	103	0.02	1.000	1.000
hm	270	9	$1.3 \cdot 10^{-4}$	1.000	1.000

E LAMA EVALUATION

Setup. We use Scorpion's LAMA 2011 configuration (`seq-sat-lama-2011`), replacing its `ff` heuristic with the listed evolved heuristic and, in the paired configurations, also replacing its landmark heuristic, while retaining the anytime search schedule. As for `ff` in default LAMA, the replaced heuristic sees unit action costs in the initial greedy iteration and costs increased by one in the subsequent weighted A* iterations. All LAMA runs use one CPU core per run and a Scorpion build without LP support, on a different cluster than the GBFS runs. Each run has a 5-minute overall time limit, including translation, and an 8 GB memory limit. We evaluate on the same 2640 tasks as the GBFS evaluation, excluding the training domains `sokoban` and `transport` from the IPC runs.

Preferred Operators. Besides heuristic values, LAMA uses `ff` to compute *preferred operators*, the actions in the relaxed plan that are applicable in the current state, which the evolved programs do not provide. For the FF-like heuristics `evolved-blind-medium-1` and `evolved-ff-medium-3`, we manually add FF's helpful-actions rule to the relaxed-plan extraction without changing the heuristic values. We add no preferred-operator rule to `evolved-blind-none-3`. The landmark heuristic supplies its own preferred operators whenever retained, including when `evolved-blind-none-3` replaces FF. The upper block of Table 2 uses these two adapted FF-like heuristics, the unmodified `evolved-blind-none-3` and default LAMA. In its lower block, the landmark heuristic is replaced by the unmodified `evolved-blind-none-3`, so preferred operators come only from the FF-like heuristic. Compared to default LAMA, the best such configuration, `evolved-blind-medium-1` with `evolved-blind-none-3`, gains most in `childsnack` (+31) and `blocksworld` (+28) and loses most in `scanalyzer` (-8), `parking` (-7) and `depots` (-6). Evolving preferred operators together with heuristic values is left to future work.

Metrics and Statistical Tests. Coverage counts tasks with at least one plan, and quality sums the IPC cost ratios using the cheapest final plan among the seven configurations in Table 2 as the reference for each task. Time to first plan includes translation, and speedup is the geometric mean of the ratio of default LAMA's time to the variant's on tasks both solve, with times floored at 0.01 s. For `evolved-blind-medium-1`, the comparison includes 1933 tasks solved by both configurations. We count a task as faster or slower only when the time difference is at least 0.1 s and at least 10% of the slower time. This gives 349 faster tasks, 70 slower tasks and 1514 ties, with $p = 6.1 \cdot 10^{-46}$ from a one-sided exact sign test favoring `evolved-blind-medium-1`, excluding ties. For the domain-level comparison, we pool the IPC and Autoscale instances of each domain before taking the median of the paired log time ratios. Among the 34 distinct domains, 12 favor `evolved-blind-medium-1`, none favor default LAMA and 22 are ties, using a symmetric tolerance of $\log(1.05)$ around zero. The corresponding one-sided exact sign test gives $p = 2.4 \cdot 10^{-4}$. These raw p-values describe task-level and domain-level consistency of the first-plan speed improvement.

Combined Heuristics. With `evolved-blind-medium-1` and `evolved-blind-none-3` together, coverage is 451/720 on IPC and 1535/1920 on Autoscale. Against default LAMA, the pair uniquely solves 106 tasks and loses 56; against `evolved-blind-medium-1` with landmarks, it gains 107 and loses 61; against `FF` with `evolved-blind-none-3`, it gains 10 and loses 6. Two-sided exact McNemar tests give $p = 1.1 \cdot 10^{-4}$, $4.8 \cdot 10^{-4}$ and 0.45, respectively ($3.2 \cdot 10^{-4}$, $9.6 \cdot 10^{-4}$ and 0.45 after Holm correction across these three exploratory coverage comparisons). Resampling the 34 distinct domains with replacement, pooling each domain across suites, gives percentile 95% intervals of $[-24, 146]$ tasks and $[-28.4, 188.2]$ quality points for the gains over default LAMA (100 000 resamples, seed 20260926). Thus, the aggregate gains do not establish an advantage across domains. The $1.11\times$ speedup uses 1880 jointly solved tasks (two-sided Wilcoxon test on nonzero log time ratios, raw $p = 3.9 \cdot 10^{-24}$). These comparisons use single runs per task in separate execution batches; timing variation and selection across the broader set of configurations are not covered by the correction. Replacing landmarks also removes their preferred operators, so these results do not isolate the effects of changing heuristic values and preferred operators.

Ablations. Table 9 gives per-suite coverage with and without the preferred operators of the FF-like component. Table 2 measures each speedup on the tasks that both compared configurations solve, so the task set differs between heuristics. Table 10 instead restricts each heuristic to the tasks that all four configurations solve: the heuristic with and without preferred operators and `ff` with and without preferred operators (three configurations for `evolved-blind-none-3`). This hardly changes

the results. The largest change is for `evolved-blind-medium-1` with preferred operators, whose speedup drops from 1.05 to 1.04 and remains significant.

Table 9: Coverage of LAMA with `ff` or an evolved heuristic, on the IPC Learning Track test set (720 tasks) and the held-out Autoscale test set (1920 tasks). "Pref. ops" adds FF's helpful-actions rule to the FF-like heuristic. Both columns for `evolved-blind-none-3` repeat the same unmodified configuration, which receives preferred operators only from the landmark heuristic. The `ff` row with preferred operators is default LAMA. Bold marks the best value per column.

	No pref. ops			Pref. ops		
	IPC	Autoscale	Total	IPC	Autoscale	Total
`ff`	365	1333	1698	**422**	1514	1936
`evolved-blind-medium-1`	**368**	1335	**1703**	**422**	**1518**	**1940**
`evolved-ff-medium-3`	365	1326	1691	421	1513	1934
`evolved-blind-none-3`	339	**1352**	1691	339	1352	1691

Table 10: Speedup of LAMA over `ff` as in Table 2, but on the tasks that all four configurations solve (heuristic and `ff`, each with and without heuristic preferred operators). For `evolved-blind-none-3`, both columns use the same unmodified heuristic and differ only in the `ff` reference, giving three configurations in the common-task intersection. "Tasks" is the size of this set. p is from a two-sided Wilcoxon signed-rank test on the per-task log time ratios. Bold marks the best value per column.

		No pref.		Pref.	
	Tasks	Speed-up	p	Speed-up	p
`evolved-blind-medium-1`	1672	1.053×	$7 \cdot 10^{-44}$	**1.039**×	$9 \cdot 10^{-22}$
`evolved-ff-medium-3`	1666	0.974×	$7 \cdot 10^{-6}$	1.005×	0.84
`evolved-blind-none-3`	1540	**1.092**×	$1 \cdot 10^{-30}$	0.612×	$4 \cdot 10^{-43}$

F SOURCE CODE

We list the complete C++ source of the two seeds and the three top evolved heuristics.

F.1 EVOLUTION SEEDS

The two seed heuristics used to initialize evolution in Section 4 are the `blind` seed and the `FF` seed.

F.1.1 BLIND SEED

```
// EVOLVE-BLOCK-START
#include "../heuristic.h"
#include "../plugins/plugin.h"
#include "../task_utils/task_properties.h"
#include "../utils/logging.h"

using namespace std;

namespace my_heuristic {

/*
 * Blind heuristic - evolution seed.
 *
 * Returns 1 for every non-goal state, regardless of how close the state is
 * to the goal.
 *
 */
class MyHeuristic : public Heuristic {
protected:
    virtual int compute_heuristic(const State &ancestor_state) override {
        State state = convert_ancestor_state(ancestor_state);

        // Always check for the goal first - return 0 immediately if reached.
        if (task_properties::is_goal_state(task_proxy, state))
            return 0;

        // Every non-goal state gets h = 1: we know at least one action is needed,
        // but we make no attempt to estimate how many.
        return 1;
    }

public:
    MyHeuristic(const shared_ptr<AbstractTask> &transform, bool cache_estimates,
                const string &description, utils::Verbosity verbosity)
        : Heuristic(transform, cache_estimates, description, verbosity) {}
};
// EVOLVE-BLOCK-END

class MyHeuristicFeature
    : public plugins::TypedFeature<Evaluator, MyHeuristic> {
public:
    MyHeuristicFeature() : TypedFeature("my_h") {
        document_title("An evolved heuristic");
        add_heuristic_options_to_feature(*this, "my_h");
    }
    virtual shared_ptr<MyHeuristic> create_component(
        const plugins::Options &opts) const override {
        return plugins::make_shared_from_arg_tuples<MyHeuristic>(
            get_heuristic_arguments_from_options(opts));
    }
};

static plugins::FeaturePlugin<MyHeuristicFeature> _plugin;
} // namespace my_heuristic
```

F.1.2 FF SEED

```
// EVOLVE-BLOCK-START
#include "relaxation_heuristic.h"
#include "../algorithms/priority_queues.h"
#include "../plugins/plugin.h"
#include "../task_utils/task_properties.h"
#include "../utils/logging.h"
#include <vector>

using namespace std;
using namespace relaxation_heuristic;

namespace my_heuristic {

/*
 * FF heuristic
 *
 * Returns the cost of the relaxed plan, i.e. the cheapest operator sequence
 * achieving the goals in the delete-relaxed task. This is computed in two
 * phases:
 *
 * 1. Forward Dijkstra on the delete-relaxed graph, computes for every fact the
 *    minimum-cost operator sequence that first achieves it (stored in `reached_by`).
 *    Implemented by `compute_add_and_ff()` defined below.
 *
 * 2. Relaxed plan extraction, backtracks from each unsatisfied goal through
 *    `reached_by` pointers, recursively collecting the operators needed. The set of
 *    unique operators collected is the relaxed plan; its total cost is h_FF.
 *
 *
 */

class MyHeuristic : public RelaxationHeuristic {
    static const int MAX_COST_VALUE = 100000000;

    priority_queues::AdaptiveQueue<PropID> queue;

    // One entry per task operator; true means the operator is in the relaxed plan.
    // Stored as a member so we avoid reallocating on every call.
    vector<bool> relaxed_plan;

    void enqueue_if_necessary(PropID prop_id, int cost, OpID op_id) {
        Proposition *prop = get_proposition(prop_id);
        if (prop->cost == -1 || prop->cost > cost) {
            prop->cost = cost;
            prop->reached_by = op_id;
            queue.push(cost, prop_id);
        }
    }

    // Clamp cost additions to avoid integer overflow on very long plans.
    void increase_cost(int &cost, int amount) {
        cost += amount;
        if (cost > MAX_COST_VALUE) cost = MAX_COST_VALUE;
    }

    // Reset all proposition costs and operator counters, seed the queue
    // with operators that have no preconditions.
    void setup_exploration_queue() {
        queue.clear();
        for (Proposition &prop : propositions) {
            prop.cost = -1;
            prop.marked = false;
        }
```

```
        for (UnaryOperator &op : unary_operators) {
            op.unsatisfied_preconditions = op.num_preconditions;
            op.cost = op.base_cost;
            if (op.unsatisfied_preconditions == 0)
                enqueue_if_necessary(op.effect, op.base_cost, get_op_id(op));
        }
    }

    // Mark all facts true in the current state as achieved at cost 0.
    void setup_exploration_queue_state(const State &state) {
        for (FactProxy fact : state)
            enqueue_if_necessary(get_prop_id(fact), 0, NO_OP);
    }

    // Dijkstra over the relaxed graph: propagate costs through operators
    // until all goals are reached or the queue is empty.
    void relaxed_exploration() {
        while (!queue.empty()) {
            auto [distance, prop_id] = queue.pop();
            Proposition *prop = get_proposition(prop_id);
            if (prop->cost < distance) continue;    // stale queue entry
            for (OpID op_id : precondition_of_pool.get_slice(
                     prop->precondition_of, prop->num_precondition_occurences)) {
                UnaryOperator *op = get_operator(op_id);
                increase_cost(op->cost, prop->cost);
                if (--op->unsatisfied_preconditions == 0)
                    enqueue_if_necessary(op->effect, op->cost, op_id);
            }
        }
    }

    // Run the full relaxed-graph Dijkstra and return h_add (sum of goal costs).
    // Returns DEAD_END if any goal is unreachable.
    // Also sets reached_by on every achieved proposition, enabling plan extraction.
    int compute_add_and_ff(const State &state) {
        setup_exploration_queue();
        setup_exploration_queue_state(state);
        relaxed_exploration();

        int total_cost = 0;
        for (PropID goal_id : goal_propositions) {
            const Proposition *goal = get_proposition(goal_id);
            int goal_cost = goal->cost;
            if (goal_cost == -1)
                return DEAD_END;
            increase_cost(total_cost, goal_cost);
        }
        return total_cost;
    }

    // Recursively marks the operators needed to achieve proposition goal_id
    // in the relaxed plan. Follows reached_by pointers set by compute_add_and_ff.
    void mark_relaxed_plan(PropID goal_id) {
        Proposition *prop = get_proposition(goal_id);
        if (prop->marked) return;           // already visited - avoid double-counting
        prop->marked = true;

        OpID op_id = prop->reached_by;
        if (op_id == NO_OP) return;         // fact was true in the initial state

        // Recursively collect operators needed for each precondition of this operator.
        for (PropID prec : get_preconditions(op_id))
            mark_relaxed_plan(prec);

        // operator_no == -1 for axioms, which don't count toward plan cost.
```

```
        int operator_no = get_operator(op_id)->operator_no;
        if (operator_no != -1)
            relaxed_plan[operator_no] = true;
    }

protected:
    virtual int compute_heuristic(const State &ancestor_state) override {
        State state = convert_ancestor_state(ancestor_state);

        // Phase 1: Dijkstra on the relaxed graph.
        // Returns h_add (sum of per-goal costs), or DEAD_END if any goal is unreachable.
        int h_add = compute_add_and_ff(state);
        if (h_add == DEAD_END)
            return DEAD_END;

        // Phase 2: extract the relaxed plan by backtracking from each goal.
        for (PropID goal_id : goal_propositions)
            mark_relaxed_plan(goal_id);

        // Sum costs of operators in the relaxed plan, then reset the bitvector.
        int h_ff = 0;
        for (size_t op_no = 0; op_no < relaxed_plan.size(); ++op_no) {
            if (relaxed_plan[op_no]) {
                relaxed_plan[op_no] = false;    // reset for the next compute_heuristic call
                h_ff += task_proxy.get_operators()[op_no].get_cost();
            }
        }
        return h_ff;
    }

public:
    MyHeuristic(tasks::AxiomHandlingType axioms,
                const shared_ptr<AbstractTask> &transform, bool cache_estimates,
                const string &description, utils::Verbosity verbosity)
        : RelaxationHeuristic(axioms, transform, cache_estimates, description, verbosity),
          relaxed_plan(task_proxy.get_operators().size(), false) {}
};
// EVOLVE-BLOCK-END

class MyHeuristicFeature
    : public plugins::TypedFeature<Evaluator, MyHeuristic> {
public:
    MyHeuristicFeature() : TypedFeature("my_h") {
        document_title("An evolved heuristic");
        relaxation_heuristic::add_relaxation_heuristic_options_to_feature(*this, "my_h");
    }
    virtual shared_ptr<MyHeuristic> create_component(
        const plugins::Options &opts) const override {
        return plugins::make_shared_from_arg_tuples<MyHeuristic>(
            relaxation_heuristic::get_relaxation_heuristic_arguments_from_options(opts));
    }
};

static plugins::FeaturePlugin<MyHeuristicFeature> _plugin;
} // namespace my_heuristic
```

F.2 Top Evolved Heuristics

These are the three top training scorers discussed in Section 4.2. Each file is the verbatim `best_program.cc` produced by the corresponding evolution run, including the OpenEvolve `EVOLVE-BLOCK` markers and the unchanged `MyHeuristicFeature` plugin registration. Pseudocode summaries of their state-evaluation logic appear in Section 4.2.

F.2.1 `evolved-blind-medium-1`

```
// EVOLVE-BLOCK-START
#include "relaxation_heuristic.h"
#include "../plugins/plugin.h"
#include "../task_utils/task_properties.h"
#include "../utils/logging.h"
#include "../algorithms/priority_queues.h"
#include <unordered_map>

using namespace std;
using namespace relaxation_heuristic;

namespace my_heuristic {

class MyHeuristic : public RelaxationHeuristic {
    static constexpr int INF = numeric_limits<int>::max() / 2;

    priority_queues::AdaptiveQueue<PropID> queue;
    vector<PropID> plan_stack;
    vector<int> prop_visited;
    vector<int> op_in_plan;
    int cur_gen;

    // Precomputed dead-end facts: (var, value) where value is a DTG sink
    // for a goal variable and value != goal value for that variable
    vector<pair<int, int>> dead_end_facts;

protected:
    int compute_heuristic(const State &ancestor_state) override {
        State state = convert_ancestor_state(ancestor_state);
        if (task_properties::is_goal_state(task_proxy, state))
            return 0;

        // Quick dead-end check using precomputed DTG sink analysis
        for (size_t i = 0; i < dead_end_facts.size(); ++i) {
            if (state[dead_end_facts[i].first].get_value() == dead_end_facts[i].second)
                return DEAD_END;
        }

        int gen = ++cur_gen;
        if (cur_gen < 0) {
            cur_gen = 1;
            gen = 1;
            fill(prop_visited.begin(), prop_visited.end(), 0);
            fill(op_in_plan.begin(), op_in_plan.end(), 0);
        }

        queue.clear();

        for (size_t i = 0; i < propositions.size(); ++i) {
            propositions[i].cost = INF;
            propositions[i].reached_by = NO_OP;
        }
        for (size_t i = 0; i < unary_operators.size(); ++i) {
            unary_operators[i].unsatisfied_preconditions =
                unary_operators[i].num_preconditions;
```

```
        unary_operators[i].cost = unary_operators[i].base_cost;
    }

    for (FactProxy fact : state) {
        PropID prop_id = get_prop_id(fact);
        propositions[prop_id].cost = 0;
        queue.push(0, prop_id);
    }

    int goals_reached = 0;
    int num_goals = goal_propositions.size();
    while (!queue.empty()) {
        auto [cost, prop_id] = queue.pop();
        if (cost > propositions[prop_id].cost)
            continue;
        if (propositions[prop_id].is_goal && ++goals_reached == num_goals)
            break;

        const Proposition &prop = propositions[prop_id];
        for (OpID op_id : precondition_of_pool.get_slice(
                 prop.precondition_of, prop.num_precondition_occurences)) {
            UnaryOperator &op = unary_operators[op_id];
            op.cost += cost;
            if (--op.unsatisfied_preconditions == 0) {
                if (op.cost < propositions[op.effect].cost) {
                    propositions[op.effect].cost = op.cost;
                    propositions[op.effect].reached_by = op_id;
                    queue.push(op.cost, op.effect);
                }
            }
        }
    }

    for (PropID goal_id : goal_propositions) {
        if (propositions[goal_id].cost == INF)
            return DEAD_END;
    }

    // Extract relaxed plan by backtracking from goals
    plan_stack.clear();
    for (PropID goal_id : goal_propositions) {
        plan_stack.push_back(goal_id);
    }

    int relaxed_plan_size = 0;
    while (!plan_stack.empty()) {
        PropID prop_id = plan_stack.back();
        plan_stack.pop_back();

        if (prop_visited[prop_id] == gen)
            continue;
        prop_visited[prop_id] = gen;

        const Proposition &prop = propositions[prop_id];
        if (prop.reached_by == NO_OP)
            continue;

        OpID op_id = prop.reached_by;
        const UnaryOperator &op = unary_operators[op_id];
        int op_no = op.operator_no;
        bool trace_preconds = true;
        if (op_no >= 0 && op_no < static_cast<int>(op_in_plan.size())) {
            if (op_in_plan[op_no] != gen) {
                op_in_plan[op_no] = gen;
                ++relaxed_plan_size;
```

```
                } else {
                    trace_preconds = false;
                }
            }

            if (trace_preconds) {
                for (PropID prec_id : get_preconditions(op_id)) {
                    if (prop_visited[prec_id] != gen) {
                        plan_stack.push_back(prec_id);
                    }
                }
            }
        }

        return max(relaxed_plan_size, 1);
    }

public:
    MyHeuristic(tasks::AxiomHandlingType axioms,
                const shared_ptr<AbstractTask> &transform, bool cache_estimates,
                const string &description, utils::Verbosity verbosity)
        : RelaxationHeuristic(axioms, transform, cache_estimates, description, verbosity),
          cur_gen(0) {
        prop_visited.resize(propositions.size(), 0);
        op_in_plan.resize(task_proxy.get_operators().size(), 0);
        plan_stack.reserve(propositions.size());

        // Precompute dead-end facts via DTG sink analysis
        // A value is a DTG sink if no operator/axiom can transition the variable away from it
        VariablesProxy vars = task_proxy.get_variables();
        int num_vars = static_cast<int>(vars.size());

        vector<vector<bool>> has_outgoing(num_vars);
        for (int v = 0; v < num_vars; ++v) {
            has_outgoing[v].resize(vars[v].get_domain_size(), false);
        }

        // Analyze operators for outgoing transitions
        for (OperatorProxy op : task_proxy.get_operators()) {
            unordered_map<int, int> eff_map;
            unordered_map<int, int> prec_map;

            for (EffectProxy eff : op.get_effects()) {
                int var = eff.get_fact().get_variable().get_id();
                int val = eff.get_fact().get_value();
                eff_map[var] = val;
            }
            for (FactProxy prec : op.get_preconditions()) {
                int var = prec.get_variable().get_id();
                int val = prec.get_value();
                prec_map[var] = val;
            }

            for (const auto &entry : eff_map) {
                int var = entry.first;
                int to_val = entry.second;
                auto it = prec_map.find(var);
                if (it != prec_map.end()) {
                    int from_val = it->second;
                    if (from_val != to_val) {
                        has_outgoing[var][from_val] = true;
                    }
                } else {
                    // Variable not in precondition: can transition from any value
                    int dom_size = vars[var].get_domain_size();
```

```
                    for (int v = 0; v < dom_size; ++v) {
                        if (v != to_val) {
                            has_outgoing[var][v] = true;
                        }
                    }
                }
            }
        }

        // Also analyze axioms for outgoing transitions
        for (OperatorProxy axiom : task_proxy.get_axioms()) {
            unordered_map<int, int> eff_map;
            unordered_map<int, int> prec_map;

            for (EffectProxy eff : axiom.get_effects()) {
                int var = eff.get_fact().get_variable().get_id();
                int val = eff.get_fact().get_value();
                eff_map[var] = val;
            }
            for (FactProxy prec : axiom.get_preconditions()) {
                int var = prec.get_variable().get_id();
                int val = prec.get_value();
                prec_map[var] = val;
            }

            for (const auto &entry : eff_map) {
                int var = entry.first;
                int to_val = entry.second;
                auto it = prec_map.find(var);
                if (it != prec_map.end()) {
                    int from_val = it->second;
                    if (from_val != to_val) {
                        has_outgoing[var][from_val] = true;
                    }
                } else {
                    int dom_size = vars[var].get_domain_size();
                    for (int v = 0; v < dom_size; ++v) {
                        if (v != to_val) {
                            has_outgoing[var][v] = true;
                        }
                    }
                }
            }
        }

        // A fact (var, val) is a dead end if: var is a goal variable,
        // val != goal value for var, and val has no outgoing transitions
        GoalsProxy goals = task_proxy.get_goals();
        for (size_t g = 0; g < goals.size(); ++g) {
            int gvar = goals[g].get_variable().get_id();
            int gval = goals[g].get_value();
            int dom_size = vars[gvar].get_domain_size();
            for (int val = 0; val < dom_size; ++val) {
                if (val != gval && !has_outgoing[gvar][val]) {
                    dead_end_facts.push_back(make_pair(gvar, val));
                }
            }
        }
    }
};
// EVOLVE-BLOCK-END

class MyHeuristicFeature
    : public plugins::TypedFeature<Evaluator, MyHeuristic> {
public:
```

```
    MyHeuristicFeature() : TypedFeature("my_h") {
        document_title("An evolved heuristic");
        add_relaxation_heuristic_options_to_feature(*this, "my_h");
    }
    virtual shared_ptr<MyHeuristic> create_component(
        const plugins::Options &opts) const override {
        return plugins::make_shared_from_arg_tuples<MyHeuristic>(
            get_relaxation_heuristic_arguments_from_options(opts));
    }
};

static plugins::FeaturePlugin<MyHeuristicFeature> _plugin;
} // namespace my_heuristic
```

F.2.2 EVOLVED-FF-MEDIUM-3

```
// EVOLVE-BLOCK-START
#include "relaxation_heuristic.h"
#include "../algorithms/priority_queues.h"
#include "../plugins/plugin.h"
#include "../task_utils/task_properties.h"
#include "../task_utils/causal_graph.h"
#include "../utils/logging.h"
#include <vector>
#include <algorithm>

using namespace std;
using namespace relaxation_heuristic;

namespace my_heuristic {

class MyHeuristic : public RelaxationHeuristic {
    static const int MAX_COST_VALUE = 100000000;

    priority_queues::AdaptiveQueue<PropID> queue;

    vector<char> relaxed_plan;
    vector<int> relaxed_plan_ops;
    vector<int> op_costs;
    int num_goals;

    // No extra members needed for FF heuristic

    void enqueue_if_necessary(PropID prop_id, int cost, OpID op_id) {
        Proposition *prop = get_proposition(prop_id);
        if (prop->cost == -1 || prop->cost > cost) {
            prop->cost = cost;
            prop->reached_by = op_id;
            queue.push(cost, prop_id);
        }
    }

    void increase_cost(int &cost, int amount) {
        cost += amount;
        if (cost > MAX_COST_VALUE) cost = MAX_COST_VALUE;
    }

    void setup_exploration_queue() {
        queue.clear();
        for (Proposition &prop : propositions) {
            prop.cost = -1;
            prop.marked = false;
        }
        for (UnaryOperator &op : unary_operators) {
```

```
            op.unsatisfied_preconditions = op.num_preconditions;
            op.cost = op.base_cost;
            if (op.unsatisfied_preconditions == 0)
                enqueue_if_necessary(op.effect, op.base_cost, get_op_id(op));
        }
    }

    void setup_exploration_queue_state(const State &state) {
        for (FactProxy fact : state)
            enqueue_if_necessary(get_prop_id(fact), 0, NO_OP);
    }

    void relaxed_exploration() {
        int goals_reached = 0;
        while (!queue.empty()) {
            auto [distance, prop_id] = queue.pop();
            Proposition *prop = get_proposition(prop_id);
            if (prop->cost < distance) continue;
            if (prop->is_goal && ++goals_reached == num_goals)
                return;
            for (OpID op_id : precondition_of_pool.get_slice(
                     prop->precondition_of, prop->num_precondition_occurences)) {
                UnaryOperator *op = get_operator(op_id);
                increase_cost(op->cost, prop->cost);
                if (--op->unsatisfied_preconditions == 0)
                    enqueue_if_necessary(op->effect, op->cost, op_id);
            }
        }
    }

    int compute_add_and_ff(const State &state) {
        setup_exploration_queue();
        setup_exploration_queue_state(state);
        relaxed_exploration();

        int total_cost = 0;
        for (PropID goal_id : goal_propositions) {
            const Proposition *goal = get_proposition(goal_id);
            int goal_cost = goal->cost;
            if (goal_cost == -1)
                return DEAD_END;
            increase_cost(total_cost, goal_cost);
        }
        return total_cost;
    }

    void mark_relaxed_plan(PropID goal_id) {
        Proposition *prop = get_proposition(goal_id);
        if (prop->marked) return;
        prop->marked = true;

        OpID op_id = prop->reached_by;
        if (op_id == NO_OP) return;

        for (PropID prec : get_preconditions(op_id))
            mark_relaxed_plan(prec);

        int operator_no = get_operator(op_id)->operator_no;
        if (operator_no != -1 && !relaxed_plan[operator_no]) {
            relaxed_plan[operator_no] = 1;
            relaxed_plan_ops.push_back(operator_no);
        }
    }

protected:
```

```
    virtual int compute_heuristic(const State &ancestor_state) override {
        State state = convert_ancestor_state(ancestor_state);
        if (task_properties::is_goal_state(task_proxy, state))
            return 0;

        int h_add = compute_add_and_ff(state);
        if (h_add == DEAD_END)
            return DEAD_END;

        for (PropID goal_id : goal_propositions)
            mark_relaxed_plan(goal_id);

        int h_ff = 0;
        for (int op_no : relaxed_plan_ops) {
            h_ff += op_costs[op_no];
            relaxed_plan[op_no] = 0;
        }
        relaxed_plan_ops.clear();

        return h_ff;
    }

public:
    MyHeuristic(tasks::AxiomHandlingType axioms,
                const shared_ptr<AbstractTask> &transform, bool cache_estimates,
                const string &description, utils::Verbosity verbosity)
        : RelaxationHeuristic(axioms, transform, cache_estimates, description, verbosity),
          relaxed_plan(task_proxy.get_operators().size(), 0),
          num_goals(task_proxy.get_goals().size()) {
        op_costs.reserve(task_proxy.get_operators().size());
        for (OperatorProxy op : task_proxy.get_operators())
            op_costs.push_back(op.get_cost());
        relaxed_plan_ops.reserve(task_proxy.get_operators().size());
    }
};

class MyHeuristicFeature : public plugins::TypedFeature<Evaluator, MyHeuristic> {
public:
    MyHeuristicFeature() : TypedFeature("my_h") {
        add_relaxation_heuristic_options_to_feature(*this, "my_h");
    }

    shared_ptr<MyHeuristic> create_component(const plugins::Options &opts) const override {
        return make_shared<MyHeuristic>(
            opts.get<tasks::AxiomHandlingType>("axioms"),
            opts.get<shared_ptr<AbstractTask>>("transform"),
            opts.get<bool>("cache_estimates"),
            opts.get<string>("description"),
            opts.get<utils::Verbosity>("verbosity")
        );
    }
};

static plugins::FeaturePlugin<MyHeuristicFeature> _plugin;

} // namespace my_heuristic
// EVOLVE-BLOCK-END
```

F.2.3 EVOLVED-BLIND-NONE-3

```
// EVOLVE-BLOCK-START
#include "../heuristic.h"
#include "../plugins/plugin.h"
#include "../task_utils/task_properties.h"
```

```
#include "../task_utils/causal_graph.h"
#include "domain_transition_graph.h"
#include "../algorithms/priority_queues.h"

#include <vector>
#include <algorithm>
#include <climits>

using namespace std;
using namespace domain_transition_graph;

namespace my_heuristic {

/*
 * DTG-distance goal-sum heuristic with iterative cross-variable refinement
 * and max-component for shared dependencies.
 *
 * For each goal variable, precomputes the minimum operator cost to reach
 * the goal value from any value using backward Dijkstra on the DTG.
 * Cross-variable dependencies are modeled by including precondition costs
 * in DTG edge weights, refined through iterative fixed-point propagation.
 *
 * Uses h_add-style sum but deduplicates via a max-component: when multiple
 * goals share a common precondition variable, we count the max single-goal
 * cost plus a fractional bonus for additional goals needing it.
 *
 * Achieves ~500K evals/s by doing only table lookups at evaluation time.
 */
class MyHeuristic : public Heuristic {
    struct GoalInfo {
        int var;
        int value;
        int dist_table_idx;  // stable index into dist_tables
        int weight;          // causal graph importance + DTG complexity
        int dep_bonus;       // coordination difficulty
        int landmark_level;  // estimated ordering level (0 = must be first)
    };
    vector<GoalInfo> goal_infos;
    vector<vector<int>> dist_tables;

protected:
    virtual int compute_heuristic(const State &ancestor_state) override {
        State state = convert_ancestor_state(ancestor_state);

        if (task_properties::is_goal_state(task_proxy, state))
            return 0;

        int h = 0;
        int h_max = 0;
        int unsat_count = 0;
        int weighted_unsat = 0;
        int max_landmark_level = 0;

        for (const GoalInfo &gi : goal_infos) {
            int curr_val = state[gi.var].get_value();
            if (__builtin_expect(curr_val != gi.value, 1)) {
                const vector<int> &dt = dist_tables[gi.dist_table_idx];
                int d = dt[curr_val];
                if (d == numeric_limits<int>::max()) {
                    return DEAD_END;
                }
                // Progressive weighting: earlier landmarks get multiplicative bonus
                int level_factor = 1 + gi.landmark_level;
                int weighted_d = d * level_factor;
```

```
                h += weighted_d + (weighted_d * gi.weight) / 4 + gi.dep_bonus;
                h_max = max(h_max, d);
                ++unsat_count;
                weighted_unsat += level_factor;
                max_landmark_level = max(max_landmark_level, gi.landmark_level);
            }
        }

        if (unsat_count == 0) return 0;

        // Critical path focus: hardest single goal
        h += h_max;

        // Landmark urgency: goals at earlier levels are more urgent
        h += max_landmark_level * 2;

        // Prefer states with fewer weighted unsatisfied goals
        h += weighted_unsat * 2;

        return max(1, h);
    }

public:
    MyHeuristic(const shared_ptr<AbstractTask> &transform, bool cache_estimates,
                const string &description, utils::Verbosity verbosity)
        : Heuristic(transform, cache_estimates, description, verbosity) {

        const causal_graph::CausalGraph &cg = task_proxy.get_causal_graph();
        VariablesProxy vars = task_proxy.get_variables();
        int num_vars = vars.size();

        // Build DTGs for all variables
        auto pruning = [](int, int) { return false; };
        DTGFactory factory(task_proxy, false, pruning);
        vector<unique_ptr<DomainTransitionGraph>> dtgs = factory.build_dtgs();

        // Build reverse adjacency for all variables
        vector<vector<vector<pair<int, int>>>> all_reverse_adj(num_vars);
        for (int var = 0; var < num_vars; ++var) {
            int domain_size = vars[var].get_domain_size();
            DomainTransitionGraph *dtg = dtgs[var].get();

            vector<vector<pair<int, int>>> reverse_adj(domain_size);
            for (int v = 0; v < domain_size; ++v) {
                ValueNode &node = dtg->nodes[v];
                for (ValueTransition &trans : node.transitions) {
                    int target_val = trans.target->value;
                    int min_cost = numeric_limits<int>::max();
                    for (ValueTransitionLabel &label : trans.labels) {
                        int op_cost = label.is_axiom ? 0 :
                            task_proxy.get_operators()[label.op_id].get_cost();
                        min_cost = min(min_cost, op_cost);
                    }
                    if (min_cost < numeric_limits<int>::max()) {
                        reverse_adj[target_val].push_back({v, min_cost});
                    }
                }
            }
            all_reverse_adj[var] = move(reverse_adj);
        }

        // Identify goal variables
        vector<int> goal_value(num_vars, -1);
        GoalsProxy goals = task_proxy.get_goals();
        for (FactProxy goal : goals) {
```

```
        goal_value[goal.get_variable().get_id()] = goal.get_value();
    }

    // Compute local DTG distances for goal variables
    vector<vector<int>> local_dist(num_vars);
    for (int var = 0; var < num_vars; ++var) {
        int domain_size = vars[var].get_domain_size();
        if (goal_value[var] >= 0) {
            vector<int> dist(domain_size, numeric_limits<int>::max());
            dist[goal_value[var]] = 0;

            priority_queues::AdaptiveQueue<int> queue;
            queue.push(0, goal_value[var]);

            while (!queue.empty()) {
                auto [cost, val] = queue.pop();
                if (cost > dist[val]) continue;
                for (auto &[src, edge_cost] : all_reverse_adj[var][val]) {
                    int new_cost = cost + edge_cost;
                    if (new_cost < dist[src]) {
                        dist[src] = new_cost;
                        queue.push(new_cost, src);
                    }
                }
            }
            local_dist[var] = move(dist);
        } else {
            local_dist[var].assign(domain_size, 0);
        }
    }

    // Non-goal variable: estimate minimum cost to change value
    vector<int> avg_change_cost(num_vars, 1);
    for (int var = 0; var < num_vars; ++var) {
        if (goal_value[var] < 0) {
            int min_change = numeric_limits<int>::max();
            for (int v = 0; v < vars[var].get_domain_size(); ++v) {
                for (auto &[src, edge_cost] : all_reverse_adj[var][v]) {
                    if (src != v) {
                        min_change = min(min_change, edge_cost);
                    }
                }
            }
            avg_change_cost[var] = (min_change < numeric_limits<int>::max()) ? min_change : 0;
        }
    }

    // Iterative refinement with cross-variable precondition costs
    const int MAX_ITERATIONS = 3;
    for (int iteration = 0; iteration < MAX_ITERATIONS; ++iteration) {
        bool changed = false;

        for (int var = 0; var < num_vars; ++var) {
            if (goal_value[var] < 0) continue;

            int domain_size = vars[var].get_domain_size();
            DomainTransitionGraph *dtg = dtgs[var].get();

            // Build enhanced reverse adjacency with cross-variable costs
            vector<vector<pair<int, int>>> enhanced_reverse_adj(domain_size);
            for (int v = 0; v < domain_size; ++v) {
                ValueNode &node = dtg->nodes[v];
                for (ValueTransition &trans : node.transitions) {
                    int target_val = trans.target->value;
                    int best_cost = numeric_limits<int>::max();
```

```
                    for (ValueTransitionLabel &label : trans.labels) {
                        int op_cost = label.is_axiom ? 0 :
                            task_proxy.get_operators()[label.op_id].get_cost();
                        int prec_cost = 0;
                        bool impossible = false;
                        for (const LocalAssignment &prec : label.precond) {
                            int prec_var = dtg->local_to_global_child[prec.local_var];
                            if (prec_var == var) continue;
                            if (goal_value[prec_var] >= 0) {
                                if (prec.value != goal_value[prec_var]) {
                                    int d = local_dist[prec_var][prec.value];
                                    if (d == numeric_limits<int>::max()) {
                                        impossible = true;
                                        break;
                                    }
                                    prec_cost += d;
                                }
                            } else {
                                prec_cost += avg_change_cost[prec_var];
                            }
                        }
                        if (!impossible) {
                            best_cost = min(best_cost, op_cost + prec_cost);
                        }
                    }
                    if (best_cost < numeric_limits<int>::max()) {
                        enhanced_reverse_adj[target_val].push_back({v, best_cost});
                    }
                }
            }

            // Recompute backward Dijkstra with enhanced costs
            vector<int> new_dist(domain_size, numeric_limits<int>::max());
            new_dist[goal_value[var]] = 0;

            priority_queues::AdaptiveQueue<int> queue;
            queue.push(0, goal_value[var]);

            while (!queue.empty()) {
                auto [cost, val] = queue.pop();
                if (cost > new_dist[val]) continue;
                for (auto &[src, edge_cost] : enhanced_reverse_adj[val]) {
                    int new_cost = cost + edge_cost;
                    if (new_cost < new_dist[src]) {
                        new_dist[src] = new_cost;
                        queue.push(new_cost, src);
                    }
                }
            }

            for (int v = 0; v < domain_size; ++v) {
                if (new_dist[v] != local_dist[var][v]) {
                    changed = true;
                    break;
                }
            }
            local_dist[var] = move(new_dist);

            if (!changed) break;
        }
        if (!changed) break;
    }

    // Compute landmark levels using topological approximation on causal graph
    // Goals that are predecessors of other goals in CG must be achieved first
```

```
        vector<int> goal_vars;
        for (FactProxy goal : goals) {
            goal_vars.push_back(goal.get_variable().get_id());
        }

        // Compute approximate landmark levels via reverse propagation in CG
        vector<int> landmark_level(num_vars, 0);
        {
            vector<vector<int>> cg_succ(num_vars);
            for (int v = 0; v < num_vars; ++v) {
                cg_succ[v] = cg.get_successors(v);
            }

            bool changed = true;
            for (int round = 0; changed && round < num_vars; ++round) {
                changed = false;
                for (int i = 0; i < (int)goal_vars.size(); ++i) {
                    int gv = goal_vars[i];
                    for (int succ : cg_succ[gv]) {
                        if (goal_value[succ] >= 0) {
                            int new_level = max(landmark_level[succ] + 1, landmark_level[gv]);
                            if (new_level > landmark_level[gv]) {
                                landmark_level[gv] = new_level;
                                changed = true;
                            }
                        }
                    }
                }
            }
        }

        // Normalize levels so max is not too large
        int max_level = 0;
        for (int gv : goal_vars) {
            max_level = max(max_level, landmark_level[gv]);
        }
        if (max_level > 3) {
            for (int gv : goal_vars) {
                landmark_level[gv] = (landmark_level[gv] * 3) / max_level;
            }
        }

        // Build goal info with stable indexing
        goal_infos.reserve(goals.size());
        dist_tables.reserve(goals.size());

        for (size_t i = 0; i < goal_vars.size(); ++i) {
            int var = goal_vars[i];
            int value = goals[i].get_value();

            const vector<int> &preds = cg.get_predecessors(var);
            int num_preds = (int)preds.size();

            int dep_bonus = max(1, num_preds);

            // weight: combine CG importance with DTG complexity
            int dtg_complexity = 0;
            DomainTransitionGraph *dtg = dtgs[var].get();
            for (int v = 0; v < vars[var].get_domain_size(); ++v) {
                dtg_complexity += dtg->nodes[v].transitions.size();
            }
            int weight = num_preds + min(dtg_complexity / 4, 5);

            // Stable storage
            int table_idx = dist_tables.size();
```

```
            dist_tables.push_back(move(local_dist[var]));

            goal_infos.push_back({
                var, value, table_idx, weight, dep_bonus,
                landmark_level[var]
            });
        }

        // Sort by landmark_level descending for cache-friendly access
        stable_sort(goal_infos.begin(), goal_infos.end(),
            [](const GoalInfo &a, const GoalInfo &b) {
                return a.landmark_level > b.landmark_level;
            });
    }
};
// EVOLVE-BLOCK-END

class MyHeuristicFeature
    : public plugins::TypedFeature<Evaluator, MyHeuristic> {
public:
    MyHeuristicFeature() : TypedFeature("my_h") {
        document_title("An evolved heuristic");
        add_heuristic_options_to_feature(*this, "my_h");
    }
    virtual shared_ptr<MyHeuristic> create_component(
        const plugins::Options &opts) const override {
        return plugins::make_shared_from_arg_tuples<MyHeuristic>(
            get_heuristic_arguments_from_options(opts));
    }
};

static plugins::FeaturePlugin<MyHeuristicFeature> _plugin;
} // namespace my_heuristic
```