PROGRAMMATIC SEARCH POLICIES FOR PLANNING

Anonymous authors
Paper under double-blind review

ABSTRACT

We study the problem of transferring knowledge from solved embodied planning problems to related future problems. Embodied planning problems are typically solved by a high-level planner that explores an environment by proposing sub-goals, which are then realized by a low-level control system. We are interested in the setting where the space of sub-goals must be rich enough to encode useful sub-goals and where the low-level control system cannot realize all sub-goals; this setting makes the low-level control system the dominant cost during exploration. Thus, the key challenge is to propose useful sub-goals that allow the planner to efficiently find trajectories that maximize reward. We refer to a system that generates sub-goals during exploration as a *search policy*. We describe an algorithm that synthesizes programmatic search policies — programs that generate candidate sub-goals for a given state, which drive the planning process. The search policies are represented as Python programs, which allows them to generate sub-goals in an expressive language and delegate some computation to the program interpreter, and they are amenable to being efficiently updated to novel environments via Language Models. Thus, by synthesizing a programmatic search policy, our proposed algorithm stores the knowledge acquired during the exploration in a representation that allows easy adaptation to novel tasks. We empirically show, in simulated environments, that the policies synthesized by our algorithm can be used to transfer knowledge across tasks.

1 INTRODUCTION

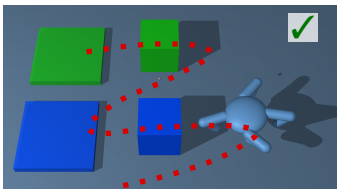
Solving an embodied task means finding a trajectory, i.e., a sequence of low-level actions from some initial state, that satisfies some given criteria. This is difficult when the horizon is long: the set of candidate trajectories grows rapidly with the number of actions, and only a small fraction of them solve the task. Consider as an example a robot that must deliver some objects, each to its destination, where delivering each object takes many actions. Standard techniques like reinforcement learning or trajectory optimization have difficulty scaling to settings like these, due to their long horizon and sparse or complex success criteria. Instead, the most effective approaches search for ways to decompose a task into sub-goals (Sacerdoti, 1975; Erol et al., 1995; Garrett et al., 2021). These involve a high-level planner leveraging a system that proposes sub-goals and a low-level control system searching for a trajectory for each sub-goal.

A growing body of work uses language models as the source of these sub-goals, since they carry general knowledge about how tasks are structured and require no hand-built domain model to supply it. Such work shows that a model can generate rich task decompositions, as a goal specification a planner consumes or as code composing primitive skills, without fine-tuning on a given domain (Chen et al., 2024b; Pan et al., 2023; Huang et al., 2023).

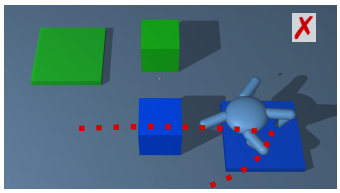
However, even with state-of-the-art language models, this approach to solving tasks remains expensive or impossible in many settings. One of the core challenges is that evaluation of sub-goals is often expensive and difficult to amortize. Here, we focus on attaining more reward for a given evaluation budget by transferring knowledge from a previously solved task, improving the quality of proposed goals so that fewer sub-goals must be generated. Standard existing techniques to do this involve either finetuning the decomposition model (Finn et al., 2017) or storing and retrieving experiences into the model's prompt (Majumder et al., 2024; Chen et al., 2024a).

The key idea behind our system is to learn a programmatic search policy, taking as a starting point an LM sub-goal proposer. We accumulate knowledge programmatically because a program admits

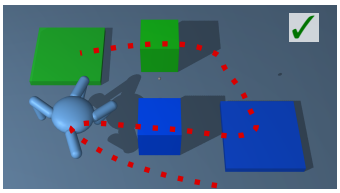
Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping 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.



(a) the source task



(b) a new task, same search policy



(c) the new task, repaired

```python
def push_side(state):
    # a fixed guess, not
    # read from the task
    return (1.0, 0.0)
```

(d) before

```python
def push_side(state):
    ix, iy = staged_item(state).item_spec.position
    sx, sy = station_of(state)          # where it must end up
    dx, dy = sx - ix, sy - iy
    d = (dx * dx + dy * dy) ** 0.5
    return (-dx / d, -dy / d)           # stand on the far side
```

(e) after

Figure 1: A component of a programmatic search policy, the update a run adopted for it, and the behaviors they generate, respectively. Note that this is only one helper, with the full search policy being a Python program that takes as input a state and generates a list of goals. Top row: the dotted line is the robot's route and the paths it drives the items along, and the mark indicates whether both items reach the platform of the corresponding colour. Bottom row: the helper in the programmatic search policy that was updated. Note that the old helper had 'hardcoded' the approach to the box from a fixed direction, and the updated helper instead computes the approach geometrically from the position of the target platform.

updates that target specific behaviors (Le Goues et al., 2019) and need little data (Tang et al., 2024), and because it is a natural representation for rich sub-goal spaces. Figure 1 is one such update: on a task where a platform switching sides makes the policy's original attempted sub-goal of pushing the box unsuccessful, one six-line programmatic helper repairs the failure mode without changing the correct components of the behavior. Our programmatic search policy learning algorithm alternates between search and synthesis phases. During the search phase, the environment is explored by generating sub-goals, with each selected sub-goal's execution evaluated by the low-level control system; the sub-goals are proposed by the LLM planner but also by the learned search policy. During the synthesis phase, the learned search policy is updated with the data collected so far to maximize planning performance. The learned search policy is transferred to a new task, so the knowledge acquired while solving a task is preserved in an executable and easy-to-update representation.

While there has been research into the mechanisms by which programmatic policies can be synthesized (Verma et al., 2018; 2019; Bastani et al., 2018), interaction of those policies with an imperfect low-level control is underexplored. For instance, approaches that compose reliable primitive skills into task programs, such as Liang et al. (2023), presuppose a reliable set of low level control primitives. Approaches that write a fresh objective for each task, such as Yu et al. (2023); Ma et al. (2023), do not accumulate knowledge across tasks. Recently, systems that accumulate past experience and store it as task-keyed text or code templates have shown promise, such as Zha et al. (2024); Christie et al. (2026); but in these systems, goals are regenerated at each use, so a fact learned once is not reused directly. None of these works focus on developing representations and learning mechanisms that simultaneously account for the limitations of the low-level control system and transferring programmatic knowledge across tasks.

We make the following contributions.[1] **(i) a programmatic search policy learning algorithm**, which takes an environment and learns and updates a programmatic search policy during exploration of the environment, with the programmatic policy being easily adaptable to a new task, and; and **(ii) an empirical study** across pairs of source and target tasks in several domains, measuring how much more reward a policy learned on the source attains on the target task for a given budget, against planning from scratch; this study includes a novel domain, BOXDELIVERY, of contact-rich embodied tasks in MuJoCo, designed to measure how well a learned search policy generalizes

[1]Source code is available in `https://anonymous.4open.science/r/programmatic-search-policy-6E18` and will be made public upon acceptance.

2 OVERVIEW

As a running example, consider the following 3D rigid body simulated domain. A quadruped robot is sharing a flat arena with boxes and platforms, and each box must be moved by the robot to a target platform of matching color (see fig. 1). The robot moves boxes by touching and pushing them. The contact dynamics are discontinuous and pushing boxes requires locomotion and interacting with objects, so the low-level control problem is challenging. We refer to this domain as BOXDELIVERY.

Given an environment, the problem is to find, as efficiently as possible, a sequence of actions that maximize the accumulated reward. We refer to the system that determines how the environment is explored as the *search policy*. We synthesize programmatic search policies during planning, and measure whether they attain more reward on unseen tasks for a given budget.

We assume there is a low-level control system that attempts to achieve proposed sub-goals, but we allow it to be imperfect. In our running example, a sub-goal could be "the robot made contact with box 1 and eventually the box reached platform 2". These sub-goals are represented in a language that the low-level control system can act on. Whether a sub-goal can be executed by the low-level control system depends on many factors, and in some cases the sub-goals will not be executable until the system tries another goal first (e.g., "approach box 1 from the left"). The number of sub-goals dominates the cost of the search. Learning to propose good sub-goals (i.e., a search policy) is therefore a mechanism to efficiently find a solution.

Our system learns such a search policy. It takes as input the environment and optionally a programmatic search policy learned in previous tasks. The system's execution proceeds as follows: on the first task, there is no previous learned search policy, so the system uses the LLM to propose sub-goals and explore the MDP. The exploration records which sub-goals raise the reward and which the low-level control system can achieve. Our system uses those records to synthesize and update the programmatic search policy, so that it generates the sub-goals that were useful for the search.

3 SETTING AND PROBLEM

Task A task $w \in \mathcal{M}$ is a Markov decision process (MDP) with continuous state space S, action space A, deterministic dynamics $f_w : S \times A \to S$, initial state $s_{0,w} \in S$, and reward $R_w : S \to [0, 1]$, all of them given and fixed by the domain. The reward measures how much of the task a state has accomplished, and is bounded, which we take to be the unit interval without loss of generality. The reward of an action is therefore the increase it produces, $R_w(s_{t+1}) - R_w(s_t)$, discounted by a factor $\gamma \in (0, 1)$ per action, so the same progress made sooner is worth more and a trajectory ending where it began has reward zero. A trajectory of w from a state s is a finite sequence $\tau = \big((s_i, a_i)\big)_{0 \le i < k}$ with $s_0 = s$ and $s_{i+1} = f_w(s_i, a_i)$, the set of trajectories is $\mathcal{T}$, and we write $\mathrm{first}(\tau) = s_0$, $\mathrm{last}(\tau) = f_w(s_{k-1}, a_{k-1})$ and $|\tau| = k$.

Adaptation across related tasks In our setting, tasks share S and A and are indexed by a latent context $z \in \mathcal{Z}$, which fixes the dynamics and the reward. This corresponds to a contextual MDP (Hallak et al., 2015), and the set of tasks is defined by $\mathcal{M} = \{w_z : z \in \mathcal{Z}\}$. Because we are interested in studying adaptation efficiency, we consider pairs (v, w) of tasks: a learner is given the source task v first, and must solve it, after which it will move to the target task w.

Sub-goals We assume a task can be decomposed into sub-goals: some finite sequence of them has trajectories that, run by the low-level control system one after another from $s_{0,w}$, reach a state of maximal reward. For convenience, our system represents sub-goals in the language of Signal Temporal Logic, which is a very flexible language to describe predicates that involve temporal constraints. Formally, sub-goals are predicates over trajectories. We use $\mathcal{G}$ to refer to the set of such formulas built from the temporal operators and a user-given vocabulary of domain-specific predicates. Note that $\mathcal{G}$ is a rich, combinatorial space: formulas may contain real arguments, and contain the temporal operators $\Diamond_{[0,H]}$, $\Box_{[0,H]}$, $\wedge$, $\vee$ and $\neg$, for a time horizon H. We write $\rho(g, \tau)$ for the robustness of τ with respect to g, which is positive exactly when τ satisfies g (this is the standard relaxation of truth values into real numbers; please refer to e.g., Takano et al. (2021) for details).

	candidate edge at s	from	what G records
1	$\Diamond_{[0,10]}$ `RobotDistanceToBox(red, 0.35)`	σ	reached, cost 1
2	$\Diamond_{[0,10]}$ `RobotAtLocation(1.789, 1.296)`	σ	failed
3	$\Diamond_{[0,10]}$ `BoxInContactWithPlatform(red)`	σ_{LM}	reached, cost 9, delivers

Figure 2: Three sub-goals evaluated at one state of a BOXDELIVERY transfer, and what G stores for each. The first goal in the table reads, 'Eventually, in the next 10 timesteps, the robot is within 0.35 units of the red box'. Goals are formatted for display.

Low-level control system We consider a low-level control system $L : S \times \mathcal{G} \to \mathcal{T} \cup \{\mathsf{fail}\}$, which searches for a trajectory from a given state satisfying a given sub-goal and returns fail when its search finds none. L is sound but not complete: if $L(s, g) = \tau \neq \mathsf{fail}$ then $\mathrm{first}(\tau) = s$ and $\rho(g, \tau) > 0$, while $L(s, g) = \mathsf{fail}$ does not imply that no trajectory from s satisfies g. Note that this is a more general assumption than assuming access to an API of robust control primitives. We assume L is given and fixed (though see Sacks & Boots, 2023 for how the distribution it samples from might be learned). Running L on a pair (s, g) *evaluates* that pair, and evaluations are the unit in which we measure cost because they dominate the search cost.

Search policy A search policy is a function $\sigma : S \to \mathcal{G}^*$, where $\mathcal{G}^*$ is the set of sequences of goals it may propose at a state. The set of search policies is Σ, and we write $\mathcal{P}$ for those that are programs, which is what a learner transfers from one task to the next. A system *explores* an MDP by using a search policy to generate sub-goals and move between its states.

Planning graph Exploring an MDP induces a graph, $G = \langle N, E \rangle$, recording the exploration, where $N \subseteq S$ is the set of nodes and $E \subseteq S \times \mathcal{G} \times S$ the set of edges. When the low-level controller fails to meet a subgoal in a state, we say that the edge failed. We further assume the graph is annotated with the rewards along each transition and which transitions have been unsuccessful.

We consider planners that learn as they explore an MDP. A *learner* is a pair of functions SEARCH : $\mathfrak{G} \times \mathcal{P} \times \mathbb{N} \to \mathfrak{G}$, and UPDATE : $\mathfrak{G} \times \mathcal{P} \times 2^{\mathfrak{G}} \to \mathcal{P}$, the first exploring an MDP for some budget using a search policy and returning the planning graph they generated, and the second updating a search policy using the data in a planning graph. Running a learner alternates them until its budget is spent.

Problem statement We are interested in the following measurement: *how effectively can a learner leverage knowledge acquired in a previous task (the 'source') to maximize reward in a novel task (the 'target')?*

Let $w, v \in \mathcal{M}$ be two tasks. For a fixed budget, write $R^\star(w; v, B)$ for what a learner attains on the target task w, the greatest $R_w(\mathrm{last}(\tau))$ over the trajectories it recorded, having been given the source task v first, and $R^\star(w; \varnothing, B)$ for what the same learner attains having been given nothing, with the learning having spent an equal budget on both searches. The problem we tackle in this work is designing a learner that maximizes $R^\star(w; v, B) - R^\star(w; \varnothing, B)$, over pairs of source and target tasks drawn from the family.

4 METHOD

We propose a learner that synthesizes and adapts a search policy. The core algorithmic insight is twofold: first, the use of a program as the representation of the search policy allows for targeted updates with a simple synthesis objective. Second, leveraging an off-the-shelf LM as an auxiliary sub-goal proposer allows exploration when the programmatic search policy has not been synthesized. This section describes the two components of our proposed learner.

A programmatic search policy has two advantages over representations such as natural language or neural networks. First, it is easy to edit in a very targeted way and, often, to generalize over a set of behaviors. Second, it is interpretable, which makes edits easier to design. Together they let a language model revise the policy from what the exploration recorded.

Algorithm 1 SEARCHANDSYNTHESIZE: the learner (SEARCH, UPDATE) of §3, run on one task

Input: a task w and its low-level control system L; an evaluation budget B and a round size b
a programmatic search policy σ, and the graphs $\mathcal{H}$ of the tasks it was transferred through
a language model, supplying a search policy σ_{LM} and a repair $\text{REPAIR}_{\mathsf{LM}}$
Output: the graph G the search built, the revised policy σ, and $\mathcal{H} \cup \{G\}$ for the next task

procedure SEARCH(G, σ, b) ▷ §4.1
  for b times **do**
    $s \leftarrow$ the node G ranks first: live, most reward recorded, most recent
    $g \leftarrow$ the first candidate edge at s that G has no record of
    $\tau \leftarrow L(s, g)$ ▷ the only evaluation
    $G \leftarrow G$ with τ recorded; if the state s' it reaches is new, add its candidate edges
    $\sigma(s')_1,\ \sigma_{\mathsf{LM}}(s')_1,\ \sigma(s')_2,\ \sigma_{\mathsf{LM}}(s')_2,\ \ldots$, in that order ▷ one draw of σ_{LM}
  end for
  return G
end procedure

procedure UPDATE($G, \sigma, \mathcal{H}$) ▷ §4.2
  $C \leftarrow \{\sigma\} \cup \text{REPAIR}_{\mathsf{LM}}(\sigma, G)$
  return the candidate of C that replays best over $\mathcal{H} \cup \{G\}$, keeping σ if none beats it
end procedure

$G \leftarrow$ the graph whose only node is $s_{0,w}$, with its candidate edges added
while fewer than B evaluations have been made **do**
  $G \leftarrow \text{SEARCH}(G, \sigma, b)$
  $\sigma \leftarrow \text{UPDATE}(G, \sigma, \mathcal{H})$
end while
return G, σ, $\mathcal{H} \cup \{G\}$

Below we describe the two components of the learner (see Algorithm 1): a SEARCH which is a round of exploration using the learned search policy and the off-the-shelf LM sub-goal proposer, and an UPDATE that revises the learned search policy using the constructed planning graph so far.

4.1 SEARCH

The learner leverages two search policies to explore the MDP and construct the planning graph. The first is an off-the-shelf language model that proposes sub-goals, used to bootstrap the search, σ_{LM}, and it is not updated. The *programmatic search policy*, $\sigma \in \mathcal{P} \subset \Sigma$, is learned and updated as the planning graph is constructed. It is empty on the first task and then transferred to the next.

For simplicity, in our experiments we use best-first search over the nodes of the planning graph: nodes are ranked by the reward so far, descending, and ties are broken newest first. We also admit a heuristic for discarding nodes.

The learner chooses a sub-goal as follows: each node is associated with queue of sub-goals, if the queue is not empty, then the next scheduled sub-goal is chosen. Otherwise, the node's queue is initialized to a list containing the sub-goals generated by the two search policies interleaved. A node's candidate edges are thus initialized to $\hat{\sigma}(s) = \sigma(s)_1,\ \sigma_{\mathsf{LM}}(s)_1,\ \sigma(s)_2,\ \sigma_{\mathsf{LM}}(s)_2,\ \ldots$, in order, an entry occurring twice kept at its earlier position. Adopting a revision replaces σ's candidates at every node and leaves σ_{LM}'s as drawn, so a better program changes what is tried at states already visited without a further query. Where a search has evaluated every candidate at every node and budget remains, σ_{LM} is drawn again at every live node and the new entries are appended, at most four times in a run.

Once a node and sub-goal have been chosen, the low-level control system is called on the sub-goal and the state in the node, recording the outcome either as a new edge and new node or as a low-level control failure.

4.2 UPDATE

The learner updates the programmatic search policy using the planning graph explored so far to maximize search performance. To this end, the learner uses $\text{REPAIR}_{\mathsf{LM}} : \mathcal{P} \times \mathfrak{G} \to 2^{\mathcal{P}}$, an off-the-shelf language-model which returns a finite set of candidate programs, each a revision of the programmatic search policy it was given. The update we describe does not rely on further calls to the low-level control system, so the update step can evaluate many proposals in parallel offline.

Data given to the model A revision is a program repair task, so the model is shown states at which the current policy did worst rather than a score summarising them, as many as a prompt can carry. At each it is shown the entries the policy offered there, what G records for each of them, and the entry G records as best. It is also given the source of the current policy, the task's description, the vocabulary sub-goals are built from, and the entries already queued for evaluation. It is not the prompt σ_{LM} is drawn from, which asks for sub-goals and must not see the program.

Scoring a candidate A candidate is scored by $\text{REPLAY} : \mathcal{P} \times \mathfrak{G} \times \mathbb{N} \to \mathbb{R}$, which replays it on a recorded graph, costing no evaluation, over a route of its own: $s_0 = s_{0,w}$, and at s_k the sub-goal g_k is the first entry of $\sigma'(s_k)$ that G records as succeeding there. An edge records a sub-goal as succeeding when it was recorded for that sub-goal, and otherwise when it is the cheapest edge from s whose end state satisfies it; a failure matches only the pair it was recorded for, because L is not complete and so bears on no other sub-goal. The candidate's own list is replayed and not $\hat{\sigma}$, because a replay following the interleaved order credits a candidate for the model's entries. The replay charges one evaluation for every entry it passes, whether it succeeded or failed, and K is the last step it reaches within the round's budget b, so candidates are compared at equal cost; it stops at an entry G has no record of, or when no entry is left. The score is the reward that route banks, $\text{REPLAY}(\sigma', G, b) = \sum_{k<K} R(s_k, g_k)$.

Adoption and transfer Every candidate is replayed on this task's graph and on every graph retained from a task worked on earlier, and the candidate attaining the greatest total is adopted; a tie keeps the current policy. A revision that proposes well here and contradicts what an earlier task recorded attains less on that earlier graph. The learner transfers the adopted policy and the graphs, $\mathcal{H} \cup \{G\}$, and a graph is evidence for scoring only: its nodes are states of the task it was built in, so no search on the next task begins from any of them.

5 EXPERIMENTS

Our experiments answer two questions. **(RQ1)** Does transferring a programmatic search policy learned on one task raise the reward a related task attains for a given budget? **(RQ2)** Does that knowledge have to be a program, or would giving the language model what an earlier task's graph records serve as well?

Every measurement is a pair of runs. A learner is run first on a *source* task until it solves or its budget is spent, then on a *target* task from the same family, transferring into it one of the three things below and nothing else. Each run is measured on the task it plans. We run every ordered pair of distinct tasks from a small set drawn from the family, so each task serves as both source and target. A source that adopts no policy has nothing to transfer, so it contributes no pair. The run on a source task transfers nothing, so it is also the *no transfer* control for that task.

We instantiate the setting of §3 into three domains, described as follows. First, we propose and describe BOXDELIVERY, a novel MuJoCo benchmark in which a quadruped robot delivers boxes to platforms. Each box must be taken to a corresponding platform that accepts it by pushing it across the floor one at a time. Each task in this experiment consists of delivering two pairs of boxes and platforms, with each delivery awarding a positive reward of 1; tasks differ in the positions of the boxes and platforms within a flat surface. The target task sends its two boxes to platforms on opposite sides. This domain uses CMA-ES (Hansen & Ostermeier, 2001) as the low-level control system, scoring a candidate by the robustness of the sub-goal it was given. Note that every delivery is contact-rich manipulation of a body the robot cannot grasp, which means that despite its apparent simplicity, the low-level control problem is very challenging. In particular, this domain requires the search policy to learn to approach objects and to incrementally push them towards the target platform. For instance, pushing a box towards a platform requires the robot to first position itself properly relative to the box

and the target platform, and then to set itself sub-goals to push the box incrementally; attempting the motion in a single sub-goal is too challenging for the low-level controller. Section A describes the implementation of this domain in detail.

SOKOBAN is the standard 7×7 gridworld in which an agent pushes three boxes onto three targets. It credits a box on its target with a whole unit and gives partial credit only to the nearest box that is not. We use the domain's default environment, whose generator places every box on a target and pulls it backwards, so every level is solvable by construction. A task varies which wall the targets sit on, where along that wall they sit, and where the boxes and the agent start. Its low-level control system is off-the-shelf A* over grid states under a fixed budget of expanded nodes, so exhausting that budget reports that no plan was found and not that none exists.

MINIGRID is the MULTIROOM gridworld, in which an agent must pass through a chain of six rooms, opening the closed door between each pair. Its reward is the fraction of the task's milestones the state meets, one for each of the five doors standing open and one for reaching the goal, so it has a level for each door rather than one for the whole task. We use the domain's default environment, in which a task varies the rooms' shapes and the doors' positions. Its low-level control system searches with A* under a budget of 64 expanded nodes. A task qualifies for the experiment when every room is reachable within one such search and no room can be skipped; we use the first three qualifying layouts in seed order.

Baselines We run learners on the same tasks, under the same budget, with the same low-level control system and the same language model, differing only in what transfers from the source task into the target task. *Ours* executes Algorithm 1 and transfers the programmatic search policy the source task produced. *ACE-style* is given effectively the same domain knowledge and loop as our method, but does not transfer a program. Instead, it prompts the model, at every state, with an itemized playbook initialized from what the source task's graph records and revised after every round, which is our reimplementation of Zhang et al. (2026), the mechanism of systems that transfer past experience into the prompt as text (Zha et al., 2024; Majumder et al., 2024; Chen et al., 2024a); see section B for details and departures from the published method. *No transfer* does not transfer anything, but instead executes Algorithm 1 from an empty learned search policy. Everything else is reset between tasks in all three.

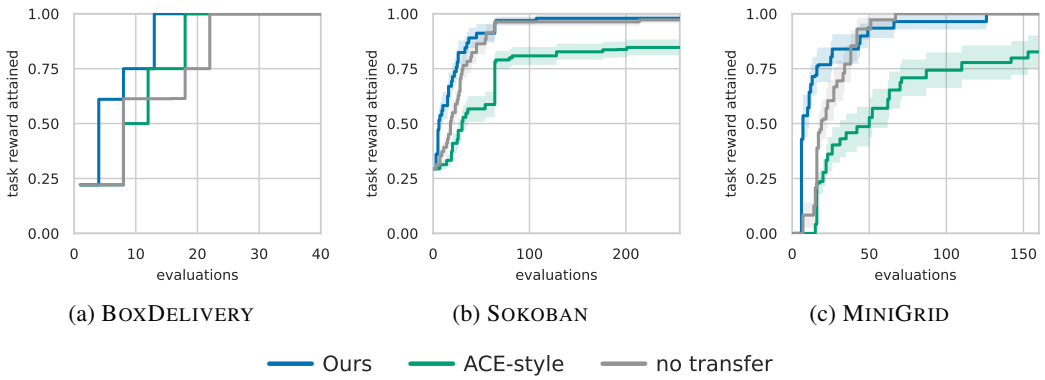
Our proposed learner starts with a skeleton Python file, which is updated as described by Algorithm 1. Each LLM involved is given domain-specific prompts that explain the tasks at a high-level. The sub-goal proposers are given a description of the Signal Temporal Logic language and the domain-specific operators as an API that they use to write sub-goals in.

5.1 RESULTS

A policy learned on one task makes adaptation easier Transferring a programmatic search policy can raise the reward a learner attains for a given budget (see fig. 3). In MINIGRID the transferred curve leads over the first forty evaluations, which is where the difference in attained reward is made, and the curves converge after. In BOXDELIVERY it leads throughout. In SOKOBAN it leads on 19 of 32 pairs, which the sign test does not separate from chance. Table 1 reads the same difference off the curves one seed at a time.

The programmatic representation increases efficiency of adaptation Transferring a program beats transferring the same experience as text: over 14 MINIGRID pairs it attains more on 13 and less on 1, a median difference of $+0.344$ ($p = 0.000916$). The playbook does not beat transferring nothing at all: over 24 MINIGRID pairs it attains more on 6 and less on 18, a median difference of -0.196 ($p = 0.0113$), and over 36 SOKOBAN pairs more on 11 and less on 25, a median of -0.108.

Transferring weights instead of a program We also train PPO on each domain, from scratch and from a policy trained on another task of the same family. We find that adapting a neural policy naively is not effective. From scratch PPO is competitive in two gridworlds, although in BOXDELIVERY no policy holds a completed task from one checkpoint to the next. We give PPO a budget comparable to or larger than the search's: the median search run of either condition reaches the task's maximum on 2.9–5.1M steps in BOXDELIVERY and 35–71k in MINIGRID, against PPO's 4.96M and 3.28M



(a) BOXDELIVERY (b) SOKOBAN (c) MINIGRID

Figure 3: Performance of search algorithms in adapting a policy, measured as the reward of the best recorded route each algorithm has found, as a fraction of the task's maximum, against the evaluations spent so far. Each curve is the mean over runs and the band is one standard error of that mean, drawn only where a curve has at least three runs. A curve reaches the top only where every run reached the task's maximum. BOXDELIVERY's curves are two runs each; section A gives every panel's runs. 'No transfer' is the performance of the same learner given nothing to transfer.

Table 1: For each pair of runs on one seed, the mean reward the transfer attained over the reported window, minus the same for transferring nothing, as a fraction of the task's maximum. Every pair is counted, whether or not either run reached the maximum. A policy is transferred to more than one target, so the pairs are fewer policies than they appear: 28 MINIGRID pairs come from 14 and 32 SOKOBAN pairs from 16.

		the transfer attained			
domain	pairs	more	less	the same	median difference
BOXDELIVERY	2	2	0	0	+0.129
SOKOBAN	32	19	12	1	+0.038
MINIGRID	28	21	7	0	+0.066

(table 5). But importantly, in the two gridworlds a policy initialized from an earlier task is much worse than one trained from scratch, collapsing or plateauing (section C).

6 RELATED WORK

Fitting a program to a search. Our loop is expert iteration: a search produces behaviour better than the current policy, and a policy is fit to what it found (Anthony et al., 2017; Silver et al., 2018; Christiano et al., 2018), usually a network fit to Monte-Carlo tree search or to its trace (Grill et al., 2020; Danihelka et al., 2021; Hamrick et al., 2020; Guez et al., 2018; Lehnert et al., 2024; Gandhi et al., 2024), and in language-model systems a fixed prompt that keeps nothing between tasks (Yao et al., 2023; Hao et al., 2023; Zhou et al., 2024), though a learned guide has precedent there (Guan et al., 2025). Where a program is fit instead, it is projected from a neural oracle by mirror descent in a mixed space (Verma et al., 2019; 2018; Bastani et al., 2018), optimized in a learned embedding, searched as partial programs, or fit without an oracle (Trivedi et al., 2021; Liu et al., 2023; Shah et al., 2020; Qiu & Zhu, 2022), sometimes with a language model writing candidates that a search repairs (Liu et al., 2024; Hu et al., 2025; Moraes et al., 2025; Moraes & Lelis, 2024). We take the same step and projection, and differ in the policy object (policy vs. search policy) and optimization method, and we do not search program space directly, which Carvalho et al. (2023) also prefer. We fit a program rather than a network.

Fitting a policy to a planner or a trajectory optimizer. Guided policy search fits a policy to samples from a trajectory optimizer instead of to reward (Levine & Koltun, 2013; Zhang et al., 2016), as does planner amortization (Lowrey et al., 2018; Byravan et al., 2021). That work assumes an

optimizer that solves what it is given, where ours cannot imitate its output because for most pairs there is none; it learns which pairs the optimizer can solve. The nearest statement of that problem estimates per-skill competence from outcomes and plans over the estimate (Kumar et al., 2024; Rabiee et al., 2022). We pin between runs a warm start of the complementary kind that improves the optimizer by learning the distribution it samples from (Sacks & Boots, 2023; Sacks et al., 2023) (section A). Since our expert cannot be re-queried cheaply, data aggregation (Ross et al., 2011) is unavailable, which is why a candidate is scored only on what the graph records of it.

Language models that write programs and accumulate skills. Recent work composes primitive skills into task programs (Liang et al., 2023; Huang et al., 2023), grows skills across tasks (Wang et al., 2023; Tziafas & Kasaei, 2024; Lu et al., 2026; Sygkounas et al., 2026; Jena et al., 2026), or compresses programs into reusable abstractions (Ellis et al., 2021; Bowers et al., 2023; Cao et al., 2023; Grand et al., 2023; Stengel-Eskin et al., 2024; Woods & Seifi, 2026). A verifier rather than the model decides what is kept, by agreement across uses, a checker, a separate model, planning competence, or an improved replayed score (Wang et al., 2024; Yang et al., 2026; Zhao et al., 2024c; Liang et al., 2025; Zhang et al., 2024; Agrawal et al., 2025). Ours differs in what it evaluates: a whole candidate replayed against transitions already recorded, so comparing candidates costs no evaluation, where executing them pays for each. Measured gains in this area have been attributed to self-correction rather than to reuse (Berlot-Attwell et al., 2026), which is why comparison to the learner given nothing to transfer is valuable (§5). Producing goal specifications from language is a separate literature (Kress-Gazit et al., 2008; Pan et al., 2023; Chen et al., 2024b; Yu et al., 2023; Ma et al., 2023; Curtis et al., 2024; Kumar et al., 2026; Guo et al., 2025).

Transferring knowledge from one task to the next. Most systems transfer text: insights, rules, manuals or corrections supplied whole (Zhao et al., 2024a; Majumder et al., 2024; Chen et al., 2024a; Zha et al., 2024), retrieved by what matches the state at hand (Fu et al., 2024; Kagaya et al., 2024), or revised from their own outcomes (Zhang et al., 2026; Ouyang et al., 2025). In all of them the model is in the loop at every use, so what was established is re-derived each time. Fewer transfer a program: workflows as prompt memory, a skillbook of human corrections, skills induced at inference, and typed libraries grown over a lifelong sequence (Wang et al., 2025b; Christie et al., 2026; Huang et al., 2026; Wang et al., 2025a; Valkov et al., 2018; Zhao et al., 2024b). Closest to our setting, Mendez-Mendez et al. (2023) transfers neural proposers into an expensive planner over such a stream, and Wong et al. (2023) learns action abstractions from language for a planner to use.

Hierarchical planning and TAMP. Task and motion planning, hierarchical planning are the foundational approaches we build on, although in this work we focus on a specific part of the problem. These approaches search symbolic structure, and account for complex constraints present in embodied planning (Garrett et al., 2021; Erol et al., 1995; Kaelbling & Lozano-Perez, 2010; Kaelbling & Lozano-Pérez, 2011; Garrett et al., 2018; Toussaint, 2015; Lozano-Pérez & Kaelbling, 2014; Garrett et al., 2020); our focus is knowledge transfer rather than the planning method.

7 Conclusion

We presented a method that learns a *programmatic search policy*: a program that maps a state to an ordered list of sub-goals to evaluate there, repaired by a language model from the decisions on which it did worst, and adopted only when replaying it on the transitions already recorded reaches more of the task. Because that test replays what the search has already recorded, choosing between candidates costs no evaluation, and because a search policy is a program, revising one helper changes every sub-goal computed from it.

Limitations The low-level control system, the abstraction of the state it reads, and the vocabulary sub-goals are built from are given rather than learned, and the control system must interface with Signal Temporal Logic, which asks a non-trivial amount of domain expertise and effort of the user. The search assumes a state recorded earlier can be restored, which is a convenience rather than a fundamental assumption. Our experiments cover three simulated domains, and pairs of tasks rather than long sequences of them. Due to computational constraints, we were restricted in the number of seeds our experiments were performed in.

AI DISCLOSURE

A language model is a component of the system under study: `qwen3-32b` proposes sub-goals and revises programs, and section A gives its configuration. Separately, the authors used generative AI tools to aid or polish writing, retrieval and discovery, research execution. We have not used generative AI tools for generating synthetic datasets or proving mathematical claims. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Section A gives each component's configuration and the per-domain choices, and section D names the model, the seeds, the commits the reported runs ran from, and what cannot be reproduced and why. Every figure and table is computed from the recorded runs by a command in the source code linked anonymously in the introduction.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. October 2025. URL https://openreview.net/forum?id=RQm2KQTM5r.

Thomas Anthony, Zheng Tian, and David Barber. Thinking Fast and Slow with Deep Learning and Tree Search. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/hash/d8e1344e27a5b08cdfd5d027d9b8d6de-Abstract.html.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/hash/e6d8545daa42d5ced125a4bf747b3688-Abstract.html.

Ian Berlot-Attwell, Tobias Sesterhenn, Frank Rudzicz, and Xujie Si. Is This LLM Library Learning? Evaluation Must Account For Compute and Behaviour. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3534–3568, Rabat, Morocco, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.163. URL https://aclanthology.org/2026.eacl-long.163.

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-Down Synthesis for Library Learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, January 2023. ISSN 2475-1421. doi: 10.1145/3571234. URL https://dl.acm.org/doi/10.1145/3571234.

Arunkumar Byravan, Leonard Hasenclever, Piotr Trochim, Mehdi Mirza, Alessandro Davide Ialongo, Yuval Tassa, Jost Tobias Springenberg, Abbas Abdolmaleki, Nicolas Heess, Josh Merel, and Martin Riedmiller. Evaluating Model-Based Planning and Planner Amortization for Continuous Control. October 2021. URL https://openreview.net/forum?id=SS8F6tFX3-.

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proceedings of the ACM on Programming Languages*, 7(POPL):396–424, January 2023. ISSN 2475-1421. doi: 10.1145/3571207. URL https://dl.acm.org/doi/10.1145/3571207.

Tales Henrique Carvalho, Kenneth Tjhia, and Levi Lelis. Reclaiming the Source of Programmatic Policies: Programmatic versus Latent Spaces. October 2023. URL https://openreview.net/forum?id=NGVljI6HkR.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. AutoManual: Constructing Instruction Manuals by LLM Agents via Interactive Environmental Learning, November 2024a. URL http://arxiv.org/abs/2405.16247. arXiv:2405.16247 [cs.AI].

Yongchao Chen, Jacob Arkin, Charles Dawson, Yang Zhang, Nicholas Roy, and Chuchu Fan. AutoTAMP: Autoregressive Task and Motion Planning with LLMs as Translators and Checkers. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6695–6702, May 2024b. doi: 10.1109/ICRA57147.2024.10611163. URL https://ieeexplore.ieee.org/document/10611163.

Paul Christiano, Buck Shlegeris, and Dario Amodei. Supervising strong learners by amplifying weak experts, October 2018. URL http://arxiv.org/abs/1810.08575. arXiv:1810.08575 [cs.LG].

Benjamin A. Christie, Yinlong Dai, Mohammad Bararjanianbahnamiri, Simon Stepputtis, and Dylan P. Losey. From Local Corrections to Generalized Skills: Improving Neuro-Symbolic Policies with MEMO, June 2026. URL http://arxiv.org/abs/2603.04560. arXiv:2603.04560 [cs.RO].

Aidan Curtis, Nishanth Kumar, Jing Cao, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Trust the PRoC3S: Solving Long-Horizon Robotics Problems with LLMs and Constraint Satisfaction, September 2024. URL http://arxiv.org/abs/2406.05572. arXiv:2406.05572 [cs].

Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with Gumbel. October 2021. URL https://openreview.net/forum?id=bERaNdoegnO.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 835–850, New York, NY, USA, June 2021. Association for Computing Machinery. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

Kutluhan Erol, James A. Hendler, and Dana S. Nau. Semantics for hierarchical task-network planning. 1995. URL https://drum.lib.umd.edu/bitstreams/60efd9c6-c9ef-4551-a953-07e6adc4d5d2/download.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 1126–1135. PMLR, July 2017. URL https://proceedings.mlr.press/v70/finn17a.html.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. AutoGuide: Automated Generation and Selection of Context-Aware Guidelines for Large Language Model Agents. In *Advances in Neural Information Processing Systems*, volume 37, pp. 119919–119948. Curran Associates, Inc., 2024. doi: 10.52202/079017-3811. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/d8efbb5dd415974eb095c3f06bff1f48-Abstract-Conference.html.

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of Search (SoS): Learning to Search in Language, April 2024. URL http://arxiv.org/abs/2404.03683. arXiv:2404.03683 [cs.LG].

Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. FFRob: Leveraging symbolic planning for efficient task and motion planning. *The International Journal of Robotics Research*, 37(1):104–136, January 2018. ISSN 0278-3649. doi: 10.1177/0278364917739114. URL https://doi.org/10.1177/0278364917739114.

Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. PDDLStream: Integrating Symbolic Planners and Blackbox Samplers via Optimistic Adaptive Planning. *Proceedings of the*

International Conference on Automated Planning and Scheduling, 30:440–448, June 2020. ISSN 2334-0843. doi: 10.1609/icaps.v30i1.6739. URL https://ojs.aaai.org/index.php/ICAPS/article/view/6739.

Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated Task and Motion Planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(Volume 4, 2021):265–293, May 2021. ISSN 2573-5144. doi: 10.1146/annurev-control-091420-084139. URL https://www.annualreviews.org/content/journals/10.1146/annurev-control-091420-084139.

Gabriel Grand, Lionel Wong, Matthew Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. LILO: Learning Interpretable Libraries by Compressing and Documenting Code. October 2023. URL https://openreview.net/forum?id=TqYbAWKMIe.

Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Remi Munos. Monte-Carlo Tree Search as Regularized Policy Optimization. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 3769–3778. PMLR, November 2020. URL https://proceedings.mlr.press/v119/grill20a.html.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rStar-Math: Small LLMs Can Master Math Reasoning with Self-Evolved Deep Thinking. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 20640–20661. PMLR, October 2025. URL https://proceedings.mlr.press/v267/guan25f.html.

Arthur Guez, Theophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Remi Munos, and David Silver. Learning to search with MCTSnets. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 1822–1831. PMLR, July 2018. URL https://proceedings.mlr.press/v80/guez18a.html.

Weihang Guo, Zachary Kingston, and Lydia E. Kavraki. CaStL: Constraints as Specifications Through Llm Translation for Long-Horizon Task and Motion Planning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11957–11964, May 2025. doi: 10.1109/ICRA55743.2025.11127555. URL https://ieeexplore.ieee.org/abstract/document/11127555.

Assaf Hallak, Dotan Di Castro, and Shie Mannor. Contextual Markov Decision Processes, February 2015. URL http://arxiv.org/abs/1502.02259. arXiv:1502.02259 [stat.ML].

Jessica B. Hamrick, Abram L. Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Holger Buesing, Petar Veličković, and Theophane Weber. On the role of planning in model-based deep reinforcement learning. October 2020. URL https://openreview.net/forum?id=IrM64DGB21.

Nikolaus Hansen and Andreas Ostermeier. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*, 9(2):159–195, June 2001. ISSN 1063-6560. doi: 10.1162/106365601750190398. URL https://ieeexplore.ieee.org/abstract/document/6790628.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 8154–8173, 2023. URL https://aclanthology.org/2023.emnlp-main.507/.

Qinglong Hu, Tong Xialiang, Mingxuan Yuan, Fei Liu, Zhichao Lu, and Qingfu Zhang. Multimodal LLM-assisted Evolutionary Search for Programmatic Control Policies. October 2025. URL https://openreview.net/forum?id=OHFNJoNtjW.

Siyuan Huang, Zhengkai Jiang, Hao Dong, Yu Qiao, Peng Gao, and Hongsheng Li. Instruct2Act: Mapping Multi-modality Instructions to Robotic Actions with Large Language Model, May 2023. URL http://arxiv.org/abs/2305.11176. arXiv:2305.11176 [cs].

Zixi Huang, Xiheng Wang, Andrew Wang, William Jurayj, Bernal Jiménez Gutiérrez, Daniel Khashabi, and Nicholas Andrews. Better, Faster, Stronger: Programmatic Skill Learning Best Reduces Agent Cost, August 2026. URL http://arxiv.org/abs/2608.11338. arXiv:2608.11338 [cs.CL].

Gaytri Jena, Kapil Wanaskar, Vinija Jain, Aman Chadha, Vasu Sharma, and Amitava Das. Weights or Skills? A Survey of Robot-Learning Techniques: from Action-Predicting Weights to Robots that Write their Own Skills, August 2026. URL http://arxiv.org/abs/2608.01851. arXiv:2608.01851 [cs.RO].

L. Kaelbling and Tomas Lozano-Perez. Hierarchical Planning in the Now. July 2010. URL https://www.semanticscholar.org/paper/Hierarchical-Planning-in-the-Now-Kaelbling-Lozano-Perez/46cd76f721fe0a7635c5ff61492012efc52039b4.

Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical task and motion planning in the now. In *2011 IEEE International Conference on Robotics and Automation*, pp. 1470–1477, May 2011. doi: 10.1109/ICRA.2011.5980391. URL https://ieeexplore.ieee.org/document/5980391.

Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. RAP: Retrieval-Augmented Planning with Contextual Memory for Multimodal LLM Agents, February 2024. URL http://arxiv.org/abs/2402.03610. arXiv:2402.03610 [cs.LG].

Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating Structured English to Robot Controllers. *Advanced Robotics*, 22(12):1343–1359, January 2008. ISSN 0169-1864. doi: 10.1163/156855308X344864. URL https://doi.org/10.1163/156855308X344864. _eprint: https://doi.org/10.1163/156855308X344864.

Nishanth Kumar, Tom Silver, Willie McClinton, Linfeng Zhao, Stephen Proulx, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Jennifer Barry. Practice Makes Perfect: Planning to Learn Skill Parameter Policies, May 2024. URL http://arxiv.org/abs/2402.15025. arXiv:2402.15025 [cs.RO].

Nishanth Kumar, William Shen, Fabio Ramos, Dieter Fox, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Caelan Reed Garrett. Open-World Task and Motion Planning via Vision-Language Model Generated Constraints. *IEEE Robotics and Automation Letters*, 11(3):3366–3373, March 2026. ISSN 2377-3766. doi: 10.1109/LRA.2026.3656799. URL https://ieeexplore.ieee.org/document/11361071.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, November 2019. ISSN 0001-0782, 1557-7317. doi: 10.1145/3318162. URL https://dl.acm.org/doi/10.1145/3318162.

Lucas Lehnert, Sainbayar Sukhbaatar, DiJia Su, Qinqing Zheng, Paul Mcvay, Michael Rabbat, and Yuandong Tian. Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping, April 2024. URL http://arxiv.org/abs/2402.14083. arXiv:2402.14083 [cs.AI].

Sergey Levine and Vladlen Koltun. Guided Policy Search. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 1–9. PMLR, May 2013. URL https://proceedings.mlr.press/v28/levine13.html.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as Policies: Language Model Programs for Embodied Control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500, May 2023. doi: 10.1109/ICRA48891.2023.10160591. URL https://ieeexplore.ieee.org/document/10160591.

Yichao Liang, Nishanth Kumar, Hao Tang, Adrian Weller, Joshua B. Tenenbaum, Tom Silver, João F. Henriques, and Kevin Ellis. Visualpredicator: Learning abstract world models with neuro-symbolic predicates for robot planning. In *International Conference on Learning Representations*, volume

2025, pp. 60416–60444, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/975db59bfa6eba2175a410f6afcecd99-Abstract-Conference.html.

Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical Programmatic Reinforcement Learning via Learning to Compose Programs. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 21672–21697. PMLR, July 2023. URL https://proceedings.mlr.press/v202/liu23p.html.

Max Liu, Chan-Hung Yu, Wei-Hsu Lee, Cheng-Wei Hung, Yen-Chun Chen, and Shao-Hua Sun. Synthesizing Programmatic Reinforcement Learning Policies with Large Language Model Guided Search. October 2024. URL https://openreview.net/forum?id=8DBTq09LgN.

Kendall Lowrey, Aravind Rajeswaran, Sham Kakade, Emanuel Todorov, and Igor Mordatch. Plan Online, Learn Offline: Efficient Learning and Exploration via Model-Based Control. September 2018. URL https://openreview.net/forum?id=Byey7n05FQ.

Tomás Lozano-Pérez and Leslie Pack Kaelbling. A constraint-based method for solving sequential manipulation planning problems. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3684–3691, September 2014. doi: 10.1109/IROS.2014.6943079. URL https://ieeexplore.ieee.org/document/6943079.

Runyu Lu, Yubo Wu, Ethan Kou, Letian Fu, Wenli Xiao, Ajay Mandlekar, Yinzhen Xu, Guanya Shi, Ken Goldberg, Ang Chen, Mosharaf Chowdhury, Yuke Zhu, Linxi "Jim" Fan, and Guanzhi Wang. ASPIRE: Agentic /Skills Discovery for Robotics, June 2026. URL http://arxiv.org/abs/2607.00272. arXiv:2607.00272 [cs.RO].

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-Level Reward Design via Coding Large Language Models. October 2023. URL https://openreview.net/forum?id=IEduRUO55F.

Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. CLIN: A Continually Learning Language Agent for Rapid Task Adaptation and Generalization. August 2024. URL https://openreview.net/forum?id=xS6zx1aBI9.

Jorge Mendez-Mendez, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Embodied Lifelong Learning for Task and Motion Planning, November 2023. URL http://arxiv.org/abs/2307.06870. arXiv:2307.06870 [cs.RO].

Rubens O. Moraes and Levi H. S. Lelis. Searching for Programmatic Policies in Semantic Spaces. volume 7, pp. 5990–5998, August 2024. doi: 10.24963/ijcai.2024/662. URL https://www.ijcai.org/proceedings/2024/662.

Rubens O. Moraes, Quazi Asif Sadmine, Hendrik Baier, and Levi H. S. Lelis. InnateCoder: Learning Programmatic Options with Foundation Models, May 2025. URL http://arxiv.org/abs/2505.12508. arXiv:2505.12508 [cs.LG].

Siru Ouyang, Jun Yan, I.-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. October 2025. URL https://openreview.net/forum?id=jL7fwchScm.

Jiayi Pan, Glen Chou, and Dmitry Berenson. Data-Efficient Learning of Natural Language to Linear Temporal Logic Translators for Robot Task Specification. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11554–11561, May 2023. doi: 10.1109/ICRA48891.2023.10161125. URL https://ieeexplore.ieee.org/document/10161125.

Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *The Tenth International Conference on Learning Representations*, 2022. URL https://openreview.net/pdf?id=6Tk2noBdvxt.

Sadegh Rabiee, Connor Basich, Kyle Hollins Wray, Shlomo Zilberstein, and Joydeep Biswas. Competence-Aware Path Planning via Introspective Perception, January 2022. URL http://arxiv.org/abs/2109.13974. arXiv:2109.13974 [cs.RO].

Stephane Ross, Geoffrey Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 627–635. JMLR Workshop and Conference Proceedings, June 2011. URL https://proceedings.mlr.press/v15/ross11a.html.

Earl D. Sacerdoti. A Structure for Plans and Behavior. August 1975. URL https://apps.dtic.mil/sti/html/tr/ADA458657/.

Jacob Sacks and Byron Boots. Learning Sampling Distributions for Model Predictive Control. In *Proceedings of The 6th Conference on Robot Learning*, pp. 1733–1742. PMLR, March 2023. URL https://proceedings.mlr.press/v205/sacks23a.html.

Jacob Sacks, Rwik Rana, Kevin Huang, Alex Spitzer, Guanya Shi, and Byron Boots. Deep Model Predictive Optimization, October 2023. URL http://arxiv.org/abs/2310.04590. arXiv:2310.04590 [cs.RO].

Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning Differentiable Programs with Admissible Neural Heuristics. In *Advances in Neural Information Processing Systems*, volume 33, pp. 4940–4952. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/hash/342285bb2a8cadef22f667eeb6a63732-Abstract.html.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, December 2018. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.aar6404. URL https://www.science.org/doi/10.1126/science.aar6404.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. ReGAL: Refactoring Programs to Discover Generalizable Abstractions, June 2024. URL http://arxiv.org/abs/2401.16467. arXiv:2401.16467 [cs.SE].

Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs. *Artificial Intelligence*, 112(1-2):181–211, August 1999. doi: 10.1016/S0004-3702(99)00052-1. URL https://dl.acm.org/doi/abs/10.1016/S0004-3702%2899%2900052-1.

Alkis Sygkounas, Victor Aregbede, Amy Loutfi, and Andreas Persson. MEMENTO: Memory-Guided Memetic Code-as-Policy Evolution, July 2026. URL http://arxiv.org/abs/2607.22832. arXiv:2607.22832 [cs.LG].

Rin Takano, Hiroyuki Oyama, and Masaki Yamakita. Continuous Optimization-Based Task and Motion Planning with Signal Temporal Logic Specifications for Sequential Manipulation. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8409–8415, May 2021. doi: 10.1109/ICRA48506.2021.9561209.

Hao Tang, Darren Key, and Kevin Ellis. WorldCoder, a Model-Based LLM Agent: Building World Models by Writing Code and Interacting with the Environment. In *Advances in Neural Information Processing Systems*, volume 37, pp. 70148–70212. Curran Associates, Inc., 2024. doi: 10.52202/079017-2243. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/820c61a0cd419163ccbd2c33b268816e-Abstract-Conference.html.

Marc Toussaint. Logic-Geometric Programming: An Optimization-Based Approach to Combined Task and Motion Planning. July 2015. URL https://www.semanticscholar.org/paper/Logic-Geometric-Programming%3A-An-Optimization-Based-Toussaint/9939d9b4549c04cbfbdc636ce1c88d6647c8e108.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to Synthesize Programs as Interpretable and Generalizable Policies. In *Advances in Neural Information Processing Systems*, volume 34, pp. 25146–25163. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/hash/d37124c4c79f357cb02c655671a432fa-Abstract.html.

Georgios Tziafas and Hamidreza Kasaei. Lifelong Robot Library Learning: Bootstrapping Composable and Generalizable Skills for Embodied Control with Language Models, July 2024. URL http://arxiv.org/abs/2406.18746. arXiv:2406.18746 [cs.RO].

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. *Advances in neural information processing systems*, 31, 2018. URL https://proceedings.neurips.cc/paper_files/paper/8086-houdini-lifelong-learning-as-program-synthesis.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 5045–5054. PMLR, July 2018. URL https://proceedings.mlr.press/v80/verma18a.html.

Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-Projected Programmatic Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/hash/5a44a53b7d26bb1e54c05222f186dcfb-Abstract.html.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models, October 2023. URL http://arxiv.org/abs/2305.16291. arXiv:2305.16291 [cs.AI].

Zhiruo Wang, Graham Neubig, and Daniel Fried. TroVE: Inducing Verifiable and Efficient Toolboxes for Solving Programmatic Tasks. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 51177–51191. PMLR, July 2024. URL https://proceedings.mlr.press/v235/wang24az.html.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing Programmatic Skills for Agentic Tasks, August 2025a. URL http://arxiv.org/abs/2504.06821. arXiv:2504.06821 [cs.CL].

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 63897–63911. PMLR, October 2025b. URL https://proceedings.mlr.press/v267/wang25bx.html.

Lionel Wong, Jiayuan Mao, Pratyusha Sharma, Zachary S. Siegel, Jiahai Feng, Noa Korneev, Joshua B. Tenenbaum, and Jacob Andreas. Learning adaptive planning representations with natural language guidance, December 2023. URL http://arxiv.org/abs/2312.08566. arXiv:2312.08566 [cs.AI].

John Woods and Hasti Seifi. SkillComposer: Learning Reusable Skills for Natural-Language Robot Programming, August 2026. URL http://arxiv.org/abs/2608.14944. arXiv:2608.14944 [cs.RO].

Yunhao Yang, Neel P. Bhatt, Kevin Wang, Samuel Tetteh, Zhangyang Wang, and Ufuk Topcu. VASO: Formally Verifiable Self-Evolving Skills for Physical AI Agents, June 2026. URL http://arxiv.org/abs/2606.05395. arXiv:2606.05395 [cs.RO].

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822. Curran Associates, Inc., 2023. doi: 10.52202/075280-0517. URL https://papers.nips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montserrat Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to Rewards for Robotic Skill Synthesis. In *Proceedings of The 7th Conference on Robot Learning*, pp. 374–404. PMLR, December 2023. URL https://proceedings.mlr.press/v229/yu23a.html.

Lihan Zha, Yuchen Cui, Li-Heng Lin, Minae Kwon, Montserrat Gonzalez Arenas, Andy Zeng, Fei Xia, and Dorsa Sadigh. Distilling and Retrieving Generalizable Knowledge for Robot Manipulation via Language Corrections. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 15172–15179, May 2024. doi: 10.1109/ICRA57147.2024.10610455. URL https://ieeexplore.ieee.org/document/10610455.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Y. Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *International Conference on Learning Representations*, 2026:86069–86100, April 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/8a94ff6f922d995d7d3f4ebf4143e442-Abstract-Conference.html.

Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search, February 2016. URL http://arxiv.org/abs/1509.06791. arXiv:1509.06791 [cs.LG].

Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. Agent-Pro: Learning to Evolve via Policy-Level Reflection and Optimization. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5348–5375, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.292. URL https://aclanthology.org/2024.acl-long.292/.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642, March 2024a. ISSN 2374-3468. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Bonan Zhao, Christopher G. Lucas, and Neil R. Bramley. A model of conceptual bootstrapping in human cognition. *Nature Human Behaviour*, 8(1):125–136, January 2024b. ISSN 2397-3374. doi: 10.1038/s41562-023-01719-1. URL https://www.nature.com/articles/s41562-023-01719-1.

Xufeng Zhao, Cornelius Weber, and Stefan Wermter. Agentic Skill Discovery, August 2024c. URL http://arxiv.org/abs/2405.15019. arXiv:2405.15019 [cs.RO].

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 62138–62160. PMLR, July 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

A Implementation details

Language model. Every query to the model uses `qwen3-32b` through an OpenAI-compatible endpoint served with vLLM, with reasoning disabled and responses cached. BoxDelivery decodes at temperature 0.8 with up to 4,096 new tokens, MiniGrid at 0.8 with up to 1,024, and Sokoban at 0.9 with up to 2,048, each domain using one setting for both language models; several draws are taken per round and each is seeded by its index. The model is asked for four draws at each state in BoxDelivery and MiniGrid, and one in Sokoban. The repair model is called only between rounds, never while the search is evaluating.

Low-level control system. One call climbs a ladder of three searches, cheapest first, and the ladder stops at the first that satisfies the sub-goal it was given. The first searches with CMA-ES (Hansen & Ostermeier, 2001) over eight parameters of a gait program shared by every leg: a frequency, an amplitude, gain and offset for the hips and the same three for the knees, and a phase lag between them. Its population is 32 over 30 generations from an initial step of 0.5, and it spends that budget in full rather than stopping at the first success, returning whichever satisfying plan arrives earliest. The second, reached only when the first satisfies nothing, searches the same way over 32 parameters, four for each of the robot's eight actuators. The third, reached only when both fail, searches the full 160-dimensional control sequence with population 20, initial step 0.3, at most 48 iterations, a stagnation window of 10 and a function-history tolerance of 10^{-4}, and stops at the first satisfying candidate. Only the third is warm-started. A sub-goal must hold within 10 sub-steps of 0.1 s inside a solver horizon of 20, the satisfaction slack is 0.15, and robustness uses a smooth min/max relaxation with smoothing 10^{-3}. There is no wall-clock cap on a solve: its cost is bounded by the rollout budgets above. A rollout is one simulated trajectory inside one of these searches, and is not an evaluation: an evaluation in the sense of §3 is one call to the whole ladder, which spends many rollouts. A BOXDELIVERY run is given 32 worker processes, which it spends as one solve at a time, matching the CMA-ES population of the first two searches. The gridworlds need none: an A* solve is a single search rather than a pool of rollouts, so each runs one solve at a time in its own process.

Warm start. Because navigation dominates the cost of a solve, the third search is warm-started with a goal-conditioned policy that proposes its first 20-step rollout. That policy is an MLP with two hidden layers of 512 units, pretrained by behavioural cloning on the navigation trajectory store and then fine-tuned with PPO, the deployed checkpoint being the one at $5{,}048{,}384$ environment steps. The pretraining run's own record of its settings was later overwritten, so we give them as the tracked configuration specified them at the time rather than as a record of that run: 400 epochs at learning rate 10^{-3}, batch 256, weight decay 10^{-4}, dropout 0.1, and 15% held out for validation. It seeds the search's first trajectory and never alters the satisfaction gate. The network itself is never trained during a run, and the whole warm start is reset to one pinned snapshot at the start of every run, the navigation store it retrieves from holding $8{,}969$ solves at that point. That store does grow within a run, with the run's own solves, so the warm start is pinned between runs rather than frozen inside one.

How fig. 3 and table 1 are read. A run's attained reward after e evaluations is the reward of the best route it had recorded by then, over the task's maximum, and its mean over the reported window is one number per run, defined whether or not the run ever reached the maximum. A pair's difference is the transferred run's mean minus its control's, so no pair is excluded and a run that never solves counts at what it did attain. The sign test is exact and one-sided, over the pairs whose difference is not zero. MINIGRID's row and curves are one condition run end to end at width 1 under a budget of 320 evaluations, over 28 transferred runs and 24 controls. BOXDELIVERY reports transfers into its target task, the designed direction, over a budget of 80 evaluations, over 2 transferred runs and 2 controls. SOKOBAN's row and curves are 32 transferred runs and 36 controls. From seed 230 on, its program transfers stop after four rounds of 64 evaluations, which the reported window of 256 never reads past, and its ACE-style transfers run all eight. Each panel's ACE-style curve is over 24 runs in MINIGRID, 36 in SOKOBAN and 2 in BOXDELIVERY. SOKOBAN's budget is 512; fig. 3 and every statistic read from it cover the first half of each of these budgets, so a curve stops at 160, 256 or 40.

What each domain instantiates. The components §4 fixes are the same in every domain; what follows is chosen per domain.

domain	node order	tie	evaluations per choice	rounds	redraw
MINIGRID	recorded reward	most recent	1	20×16	on, at most 4
SOKOBAN	recorded reward	most recent	1	8×64	off
BOXDELIVERY	recorded reward	most recent	1	10×8	off

Table 2: Per-domain choices. Every domain re-chooses after each landing, so a round's evaluations are not fixed in advance; BOXDELIVERY runs one solver pool of 32 workers and the gridworlds one solve at a time. A source stops when the policy alone reaches the task's maximum, or when its rounds are spent.

What the node order reads in each domain. During node selection we use the simple heuristic of checking for cornered and wall-locked boxes to discard dead nodes. A node is live unless its state is lost, the test §4.1 ranks on, and it holds only in SOKOBAN, where a box off target is lost when it is cornered, a wall to its north or south and a wall to its east or west, or wall-locked, a wall to its north or south with no target in its row or one to its east or west with none in its column. Other boxes do not count as walls, so the test can miss a lost state. It calls no live state lost on these levels, which we checked over every reachable state, but the wall-lock rule assumes a box beside a wall stays in its line, which a level with interior walls could break. In MINIGRID no state is lost, which a check establishes by enumerating every reachable state, and BOXDELIVERY has no test, so every node there is live and the key is inert in both. Banked progress is the reward of §5 at the node's state, on a scale of the task's item count rather than of its maximum, which orders nodes identically because the factor is positive. An entry counts as taken once the graph records it at that node, as an edge or a refutation.

Search as a decision problem. Choosing which sub-goals to evaluate is itself a Markov decision process: its state is G, its actions are the pairs (s, g) with s a node of G, and its transition is the outcome of $L(s, g)$. An evaluation is a temporally extended action in the sense of the options framework (Sutton et al., 1999): it returns a trajectory, and its reward is the discounted sum of the increments along that trajectory. Writing $R(s, g)$ for that sum, and with $c(s, g)$ the edge cost already read off the graph, its recorded edges give

$$Q(s, g) = R(s, g) + \gamma^{\,c(s,g)} V\big(\delta(s, g)\big), \qquad V(s) = \max_{g} Q(s, g),$$

with $V(s) = 0$ where G records nothing at s, and $d(s)$ the cost of the cheapest recorded route to s. The learner may make B evaluations in all, and is a policy that uses them to make $V(s_{0,w})$ as large as it can.

The node orders compared. As part of preliminary experiments during the design process, different ordering methods of the nodes were run on one seed pair of SOKOBAN control tasks, six tasks each, under one budget, and read at the same round. *Banked progress* is the order of §4.1. *Local improvement* ranks a pair by $\gamma^{\,d(s)} \cdot \hat{p}(s, g) \cdot \max(0, \hat{Q}(s, g) - V(s))$, with $\hat{p}$ and $\hat{Q}$ fit to the graph, so it scores a pair by what it would add over the best value already recorded at its node. *Banked plus increment* adds to a node's banked progress a learned estimate of what the pair would contribute, and is the decomposition of expected terminal reward into the part the graph records and the part that must be predicted.

Table 3: Node orders on six SOKOBAN control tasks. The last column is the share of evaluations spent at states where no box had yet been delivered, which is why the orders differ: an order that scores a pair by its improvement over the best value already recorded penalises the nodes furthest along, since those have the highest recorded value.

order	reached the maximum	median evaluations	spent at no progress
banked progress (§4.1)	6 of 6	97.5	0.085
banked plus increment	6 of 6	151.5	0.12
expected terminal reward	6 of 6	139	0.12
increase over the best route	6 of 6	135	0.155
the same, discounted	6 of 6	135	0.195
local improvement	0 of 6	—	0.955

Only *banked progress* and *banked plus increment* are compared on the median of the same six tasks; the three middle rows are read at the first round and are reported for their ordering rather than as a matched cost. *Local improvement* reached two of three boxes on three tasks and delivered all three on none.

The tie-breaks compared. Ties in banked progress are frequent in SOKOBAN, because most recorded moves advance the agent and not a box, so the tie-break decides much of the search there. The reported runs break them newest first; the alternative prefers the node whose recorded route is cheapest, which reaches the same states in a different order.

Preferring the cheapest recorded route spends more of the budget before the first box, and in exchange solves one more source.

Table 4: Two tie-breaks on 12 SOKOBAN sources each, at 8 rounds of 64 with one evaluation per choice, seeds 150–153 against 170–173. *No progress* is the share of evaluations spent at states holding no delivered box, pooled over the 12 sources up to each first full solve, an unsolved source counting its whole run. The median ranks an unsolved source above every cost; over the solved sources alone the two medians are 42.5 and 54.

tie-break	reached the maximum	median evaluations	spent at no progress
most recent (reported)	10 of 12	43	0.045
cheapest recorded route	11 of 12	54	0.122

Refutation and already-satisfied sub-goals. A pair is refuted after one failed evaluation in SOKOBAN and MINIGRID, whose low-level control systems are deterministic, and after three in BOXDELIVERY, whose search is randomized. MINIGRID screens a sub-goal that already holds with its goal test and BOXDELIVERY screens a robot-pose sub-goal within tolerance, both passing it over at no cost, while SOKOBAN screens nothing: such an entry takes a slot, the solver returns a plan of no moves, and the graph records a failed evaluation.

Replaying a candidate on a recorded graph. The replay of §4.2 needs the reward of every edge it passes, and a graph records the reward at each node and the cost of each edge but not the increments inside a trajectory. An edge is therefore credited $\gamma^{c-1}\big(R(s') - R(s)\big)$, the whole of its increment at its last action. A route that returns to where it began still scores zero, which is what §3 requires of it. Where a candidate is replayed on a graph retained from an earlier task, the model's entries at a state are the ones that task recorded rather than a fresh query, because our model servers do not honour a seed and a fresh call would not reproduce them; the replay stops there only when it needs an answer the earlier task never recorded.

Which realising edge the replay follows. Where several recorded edges from a state realise one sub-goal, the replay takes the edge of least recorded cost, then the greatest Q through it, then the first in a fixed order on end states. Every key is a property of an edge rather than of the order the evaluations were recorded in, so the same graph scores a candidate the same however it was built, and a check asserts that reversing the order the edges were recorded in leaves every candidate's score unchanged.

What is checked rather than asserted. Some of the properties this paper states about its own mechanism are enforced by checks that run with the code, each of which fails on the implementation that preceded it and passes on the one the reported runs use. Three cover the properties interleaving rests on: that the prompt producing the model's list holds no part of the programmatic search policy nor anything computed from it, that the two lists strictly alternate with a repeated entry kept at its earlier position, and that the learned ordering ranks nodes and never the entries within one. Three more cover the adoption line of algorithm 1: that a candidate is chosen by the replay summed over the retained graphs and the current one, that a tie keeps the incumbent, and that the deficit takes no part in the choice.

Goal language. Goals are formulas over the simulator's generic predicate vocabulary, which is the same whatever task is posed and therefore names kinds of object a BOXDELIVERY task does not contain, such as doors and dispensers, alongside the robot, items and platforms it does. A goal naming an absent object is well formed and cannot be satisfied, and no run proposed one. The predicates cover robot and item positions and orientations, robot-to-item and item-to-platform distances, and contacts, composed with `Eventually`, `Always`, `And`, `Or` and `Not`, each given quantitative (robustness) semantics so the search has a smooth objective to climb.

B THE ACE-STYLE BASELINE

ACE-style baseline: what it transfers and where it is read. The ACE-style condition transfers a *playbook*, a list of short natural-language lessons about which sub-goals tend to be reached and under

what conditions, and shows it to the language model in the prompt of σ_{LM}at every state. Each *item* of the playbook is one lesson of at most 300 characters, stored with a unique identifier, the update that added it, and two counts: *helpful*, the number of times the Reflector (defined below) judged a recorded outcome to bear the lesson out, and *harmful*, the number of times it judged one to contradict it. An item is printed in ACE's bullet format, `[pb-00007] helpful=2 harmful=0 ::` followed by the lesson. The playbook only grows: items are appended and are never removed or reworded, and no count is ever lowered. What the prompt shows is bounded instead: the whole playbook is rendered, not retrieved per state, and its item lines are capped at 4,000 characters, the bound of the prompt slot it occupies. The cap binds, and where it does the condition is shown part of its own playbook: over the reported ACE-style runs it truncated the block in 856 of 1,456 SOKOBAN prompts and 186 of 924 MINIGRID prompts, and in none of BOXDELIVERY's. When the items do not fit, they are ranked by *net votes* (helpful minus harmful), then newest first, and shown in that order until the next one does not fit, followed by a line stating how many were left out; net votes come first because the counts exist to keep lessons the records bore out, and recency second so that a new, not yet voted lesson can be shown and earn votes. The block, a fixed header followed by the items, fills the one optional slot of each domain's σ_{LM}prompt, after the task, state, goal-vocabulary and domain sections and before the reply format; it precedes the instruction on ordering the list in SOKOBAN and follows it in BOXDELIVERY, and MINIGRID has no such instruction. An empty playbook renders nothing, so the prompt is then the no-transfer control's byte for byte, and the playbook is the only difference in what σ_{LM}reads.

ACE-style baseline: how the playbook is updated. The playbook is updated from recorded outcomes alone, once from the source task before the run and then after every round on the target task, where a *round* is one iteration of algorithm 1. The evidence is a list of *records*, each one evaluation a search graph holds: a one-line description of the state, the sub-goal, and whether the low-level control system reached it, with the plan length and the change in reward when it did. Records include every evaluated sub-goal, whether the source's program or σ_{LM}proposed it, so the condition reads the same outcomes that the program condition is synthesised from. Initialisation reads every record of the source run's final graph; the update after round i reads the records that round added to the target run's own graph; no update follows the last round, since no prompt would read it. An update splits its records into consecutive batches of at most 32, and each batch passes through three steps, starting from the playbook the previous batch left. First, the *Reflector*, one language-model call, reads the batch and the playbook exactly as σ_{LM}sees it, and returns a short diagnosis, at most three *insights* (candidate lessons), and *votes* that tag shown items helpful or harmful. Second, the *Curator*, a second call, reads the insights and the same playbook and returns at most three ADD operations, each one new lesson; it is skipped when the Reflector found no insight. Third, a deterministic *merge*, with no model in it, raises each voted count by one and appends each new lesson unless it is a near-duplicate of an item already present, meaning equal after lower-casing and collapsing punctuation, or sharing at least 90% of its words (Jaccard similarity of the word sets at least 0.9). A malformed reply of either call is refused with a logged reason: a refused Reflector reply leaves its batch unused, and a refused Curator reply adds no lesson while the Reflector's votes still count. Because $\sigma_{\mathsf{LM}}$is asked once per state and its answer is kept, an update changes the lists of states first asked after it, as an online ACE update reaches the next sample and not the last.

ACE-style baseline: departures from the published method. We reimplement the mechanism of Zhang et al. (2026), not the authors' code, and depart from it in seven ways. (i) σ_{LM}, which plays ACE's Generator, does not mark which items it used, because its reply is the goal list the search reads, in a format held identical across conditions; the Reflector votes against the recorded outcomes instead, and sees the sub-goals that were not reached as well as those that were. (ii) The Reflector makes one call per batch rather than refining one reflection over up to five rounds, so that every lesson is what a single call wrote. (iii) The Curator's only operation is ADD, and the playbook is one flat list without sections: an edit or a removal would be a rewrite, which ACE's incremental update exists to avoid, and ranking by votes takes the place of sections. (iv) De-duplication compares word sets rather than semantic embeddings, because the merge must be deterministic and free of any model and we run no embedding model; the threshold of 0.9 is conservative because a false merge loses a lesson for good, while a missed duplicate only spends prompt space. (v) An update reads batches of 32 records rather than one sample per delta, and one reply may add at most three lessons: a batch of one would cost a Reflector call per record, several hundred for one SOKOBAN source, and the per-reply bound sets how fast the playbook grows against the cap; both values are the same in every domain and are stated

on every command line. (vi) Offline adaptation is one pass over the source's records rather than up to five epochs. Every run initialises its own playbook, with no cache shared across runs, so that runs stay independent. (vii) Lessons drawn from the source task must state relations between the objects rather than cells or coordinates, because the target's layout differs from the source's, and every lesson must pass the same guard as σ_{LM}'s prompt, which refuses the program's identifiers and code, so that no program reaches the condition as text. As in ACE's label-free setting, no ground-truth answer is used: the only feedback is whether the low-level control system reached each evaluated sub-goal.

ACE-style baseline: what it shares with ours. An ACE-style transfer from a source task X to a target task T is the no-transfer control on T with two changes, so that it differs from ours only in how what is transferred is represented. The control and both transfers run the same runner on the same target task from the same seed, under the same budget of evaluations, the same low-level control system and the same σ_{LM}prompt apart from the slot above, and both transfers stop at their first full delivery. The Reflector and the Curator call the same language model, on the same server and at the same sampling temperature, as σ_{LM}, as ACE uses one model for all three roles, so no stronger model writes the lessons. The first change is the playbook, initialised from the same finished source run on X whose final program ours transfers. The second is that no program is synthesised: the proposer is never built, $\textsc{repair}_{\mathsf{LM}}$never runs, the policy stays the skeleton, and $\hat{\sigma}$at every state is σ_{LM}'s list alone, so the playbook is the only thing the condition adapts. Ours differs from the same control in the program it starts from and, in BOXDELIVERY, also in the source graph it retains for scoring candidates. Both conditions keep learning on the target, ours by repairing its program and ACE-style by curating its playbook, so the comparison isolates the representation of what is transferred and learned: a program or text. Model calls are not equalised between conditions; every ACE-style run records its σ_{LM}, Reflector and Curator calls by kind. The domains differ only in what the playbook's prompts print: BOXDELIVERY describes a state by the robot's position and the staged item's name and position and writes a sub-goal's time bound as the planning horizon, while the gridworlds write it as `END`.

C A REINFORCEMENT-LEARNING BASELINE

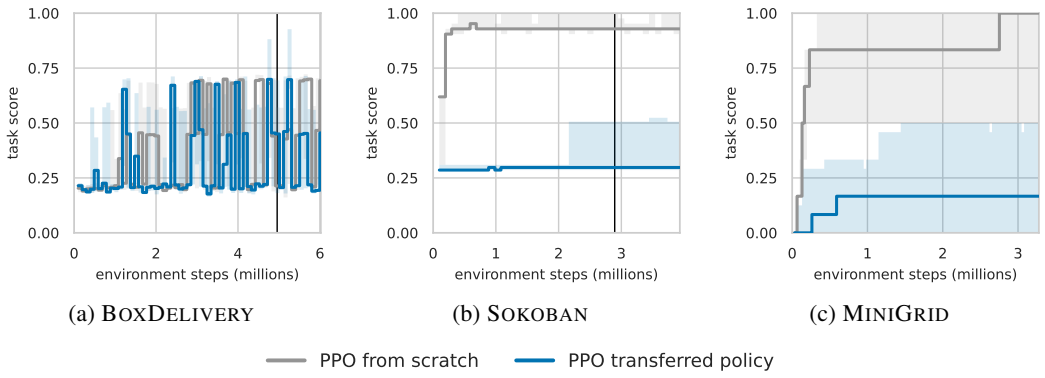
We train PPO on each domain, from scratch and from a policy trained on another task of the same family, as a point of comparison for what a learned controller achieves without a search.

Why it is not on fig. 3. The two are not measured in the same unit, and putting them on one axis would imply they were. A search spends evaluations and reports the best route it has recorded, which is a property of the graph it built. PPO spends environment steps and reports the score of its own policy at a checkpoint, which is a property of that policy. We therefore compare at compute parity with the corresponding search, and report the two quantities separately rather than in adjacent columns. A search run's own steps are bounded rather than counted, BOXDELIVERY's because the low-level control system re-simulates a number of candidates that nothing records and MINIGRID's from the most steps one evaluation can cost, while SOKOBAN's reports expanded nodes and so has no step count at all. Both are inflated by contention, because runs share the model servers and queue behind one another. A search run is given 32 worker processes in BOXDELIVERY and none in the gridworlds, which solve serially (§A), while PPO trains with 64, 32 and 16 environments, so equal wall-clock is equal wall-clock on two systems that are not equally parallel, and the factor differs in each domain and in each direction.

BOXDELIVERY. This policy is trained on `core_5x3_2task`, the same family as fig. 3, so the comparison is like for like. Every leg delivers one item early, first doing so by 1.28M steps, and delivers at least one at 11 to 32 of its 62 checkpoints, so PPO reaches half of this task readily. What it does not do is hold the whole task: both items are delivered at 14 of 744 checkpoints, in 6 of the 12 legs, and no leg delivers both at two consecutive checkpoints. Nine of the twelve legs end on the zero-item level. So the comparison here is one of environment steps rather than of what a trained policy retains. Over six transfer pairs the transferred leg reaches the full delivery first on four, the control on one, and neither on one; on the other hand a transferred leg delivers at least one item at fewer checkpoints than a source leg, 122 of 372 against 150.

Table 5: PPO at wall-clock parity with the search of fig. 3. Scores are the greatest fraction of the task any checkpoint up to that point attains, on the same reward as the search, which is the quantity a search's best recorded route is comparable to and the one a practitioner keeps by selecting a checkpoint; fig. 4 instead draws each checkpoint's own score, which can fall. A BOXDELIVERY score of 1.000 rests on a single checkpoint, since no leg delivers both items at two consecutive ones. MINIGRID has no parity row, because parity was not measured for it; its scores are given at the last checkpoint instead.

domain	measured at	statistic	from scratch	transferred
BOXDELIVERY	parity, 14,393 s (4.96M steps)	range over runs	0.701–1.000	0.717–1.000
	end of training (6.02M steps)	range over runs	0.706–1.000	0.717–1.000
SOKOBAN	parity, 785 s (2.89M steps)	median over runs	0.952	0.298
MINIGRID	the last checkpoint (3.28M steps)	median over runs	1.000	0.167



(a) BOXDELIVERY (b) SOKOBAN (c) MINIGRID

Figure 4: The score PPO's own policy attains at each checkpoint, against the environment steps it has trained for. Each curve is the median over runs and each band the interquartile range; this is the policy's score, not a running maximum, so it may fall. BOXDELIVERY draws on 6 from-scratch runs and 6 transferred ones over three seeds, on the two tasks of fig. 3, and SOKOBAN and MINIGRID on 9 and 18 each. The vertical rule is wall-clock parity with the search of fig. 3, at 4.96M steps for BOXDELIVERY and 2.89M for SOKOBAN, being 14,393 and 785 seconds. Parity is the step count an uncontended run reaches in that time; legs that trained beside a search reached fewer steps in it, so the rule is generous to PPO. MINIGRID has no rule. The axes end at 6.02M, 3.93M and 3.28M steps, and the first checkpoint is at 98,304 in the first two and 32,768 in MINIGRID.

SOKOBAN. Here the comparison is exact: the same rooms, move rule and reward as the search. From scratch PPO is strong, solving one task on all three seeds and reaching two of three boxes on the others. Transferred, it is much worse, and 13 of 18 transfers never deliver a box at all. The measured cause is that the source policy's entropy collapses over training, from 1.38 to 0.0006, so the policy that arrives at the new room has stopped exploring.

MINIGRID. The worlds are the search's own, so the comparison is exact, and two controls fix the scale: the search's own plan replays to the maximum on all three worlds, while a policy that does nothing scores 0 and a uniform random one scores 0.008 to 0.067, a mean over 20 episodes of which only the ones that open the first door score at all, and never passes the first door. From scratch PPO reaches the maximum on 5 of 9 runs and a median of 1.0 by the end of training. Transferred, no run of 18 reaches it, the best reaching five of the six rooms, at a median of 0.17. Paired against a from-scratch run on the same seed and target, the transferred policy scores lower on 14 of 18, higher on 3 and ties on the last, all three on the one task where training from scratch stalled.

What the contrast shows. A trained policy transfers worse than it trains from scratch in both domains where the comparison is exact, where a programmatic search policy does not. Transferring weights costs SOKOBAN most of its from-scratch performance, and costs MINIGRID the task entirely, because what transfers includes the collapse of exploration that made the source task succeed.

Transferring a program moves a rule for computing sub-goals and leaves the search that spends evaluations untouched, so what is transferred does not carry the collapsed exploration with it.

D REPRODUCTION

Section A gives each component's configuration and table 2 the per-domain choices. This section records what a reader needs beyond those, and what cannot be reproduced.

Model and serving. Every language-model call goes to one of four `vLLM` 0.21.0 servers of the same model, Qwen3-32B at its BF16 checkpoint, quantized to FP8 by the server at load, with a maximum length of 16,384 tokens and Qwen3's reasoning block disabled. Each request is a single user message with the domain's temperature and reply budget from section A and a per-draw seed. The servers started before the first reported run and did not restart during any of them. Runs share the servers and queue behind one another, which is why wall-clock comparisons in section C are approximate.

Seeds. A seed is reported once every leg it plans has finished and written its records, and one seed-selection rule in the code is read by both the figures and the tables, so a panel and its row always cover the same seeds. MINIGRID reports seeds 160–167, SOKOBAN 226–237, and BOXDELIVERY the settle-own seeds of its relaunch. Each gridworld seed contributes three sources, one per task, and each adopted source is transferred to each of the two other tasks.

Sources of non-determinism in the experiment. *The language model's replies.* Every draw passes a seed, but the server's continuous batching makes a seeded sample depend on the other requests in its batch, so the same prompt at the same seed need not return the same reply: on one prompt 5 of 24 repeats came back byte-identical and on another none did. Rerunning any command therefore takes a different trajectory and gives different numbers.

BOXDELIVERY*'s warm start.* Every BOXDELIVERY run begins from one pinned snapshot of the retrieval store and the navigation policy, which is what makes the low-level control system identical across conditions (§5). The snapshot itself is kept, but the behavioural-cloning checkpoint it was built from was overwritten, so the snapshot cannot be rebuilt from its source.

SOKOBAN*'s world selection.* The world set is kept and the generator is available, but the script that selected these worlds from the generator's output is not, so the set can be used and not re-derived.

Reproducible analysis. The runs have some non-determinism, but the analysis is fully reproducible. Every run records the sub-goals the model proposed, the graph the search built, and the programs the repair model wrote and the run adopted, and every figure and table in this paper is computed from those records alone by a command in the released source code. So the numbers can be recomputed exactly from the released records, and a fresh run of the same configuration will not reproduce them. Every leg runs from a frozen checkout, so the reported runs name the code that produced them: commit `65ec31bb` for MINIGRID and SOKOBAN, and `d85c8a6f` for BOXDELIVERY.