

CODE-SHARP: Continuous Open-ended Discovery and Evolution of Skills as Hierarchical Reward Programs

Richard Bornemann¹ Pierluigi Vito Amadori² Antoine Cully¹

Abstract

Developing agents capable of open-endedly discovering and learning novel skills is a grand challenge in Artificial Intelligence. While reinforcement learning offers a powerful framework for training agents to master complex skills, it typically relies on hand-designed reward functions. This is infeasible for open-ended skill discovery, where the set of meaningful skills is not known *a priori*. While recent methods have shown promising results towards automating reward function design, they remain limited to refining rewards for pre-defined tasks. To address this limitation, we introduce **Continuous Open-ended Discovery and Evolution of Skills as Hierarchical Reward Programs** (CODE-SHARP), a novel framework leveraging Foundation Models (FM) to open-endedly expand and refine a hierarchical skill archive, structured as a directed graph of executable reward functions in code. We show that a goal-conditioned agent trained exclusively on the rewards generated by the discovered SHARP skills learns to solve increasingly long-horizon goals in the Craftax environment. When composed by a high-level FM-based planner, the discovered skills enable a single goal-conditioned agent to solve complex, long-horizon tasks, outperforming both pretrained agents and task-specific expert policies by over 134% on average. We will open-source our code and provide additional videos [here](#).

1. Introduction

A core quality of general intelligence is the ability to open-endedly expand and refine its set of mastered skills autonomously (Hughes et al., 2024). Consequently, creating such an agent has been a long-standing grand challenge

of research in artificial intelligence (Turing, 2007; Norvig, 1995). In theory, Reinforcement Learning (RL) provides the framework to train an agent capable of open-endedly learning novel skills and improving them (Silver et al., 2021). However, manually designing reward functions for open-ended skill acquisition is infeasible, as it requires the expert to have extensive knowledge over the skill space or to anticipate potential pitfalls, such as reward hacking (Amodei et al., 2016; Ibrahim et al., 2024). Automating the discovery of these reward functions could serve as a catalyst for training truly generalist agents capable of rapidly adapting to unseen environments and acquiring new skills in an open-ended fashion. Recent advances in the capabilities of Foundation Models (FMs) have led to impressive progress in the automated discovery of high-performing reward functions (Ma et al.; Kwon & Michael, 2023; Xie et al., 2024). While these methods effectively refine predefined skills, they lack the ability to autonomously discover novel ones. General intelligence requires an agent to not only master existing tasks but to autonomously discover and learn entirely new skills, continuously expanding its capabilities in an open-ended fashion.(Hughes et al., 2024).

In this work, we introduce **Continuous Open-ended Discovery and Evolution of Skills as Hierarchical Reward Programs** (CODE-SHARP), a novel framework that bridges the gap between open-ended skill discovery and autonomous reward function design. CODE-SHARP leverages Foundation Models (FMs) to open-endedly grow and refine an interconnected archive of Skills defined as Hierarchical Reward Programs (SHARPs) (Figure 2): executable Python programs that return a reward and an active goal based on the environment state. Each novel SHARP skill is composed of previously added SHARP skills from the archive, resulting in an emergent skill hierarchy of increasing depth over time. To generate novel SHARP skills and refine the existing skill archive, CODE-SHARP consists of two open-ended iterative processes: 1) FM-driven open-ended discovery of SHARP skills based on the current skill archive and 2) FM-driven open-ended refinement of the skill archive by mutating already implemented SHARP skills. Simultaneously, CODE-SHARP trains a single instructable, goal-conditioned agent exclusively on the rewards and goals from the growing skill archive, to continuously expand the agent’s

¹Department of Computing, Imperial College London ²Sony Interactive Entertainment. Correspondence to: Richard Bornemann <r.bornemann25@imperial.ac.uk>.

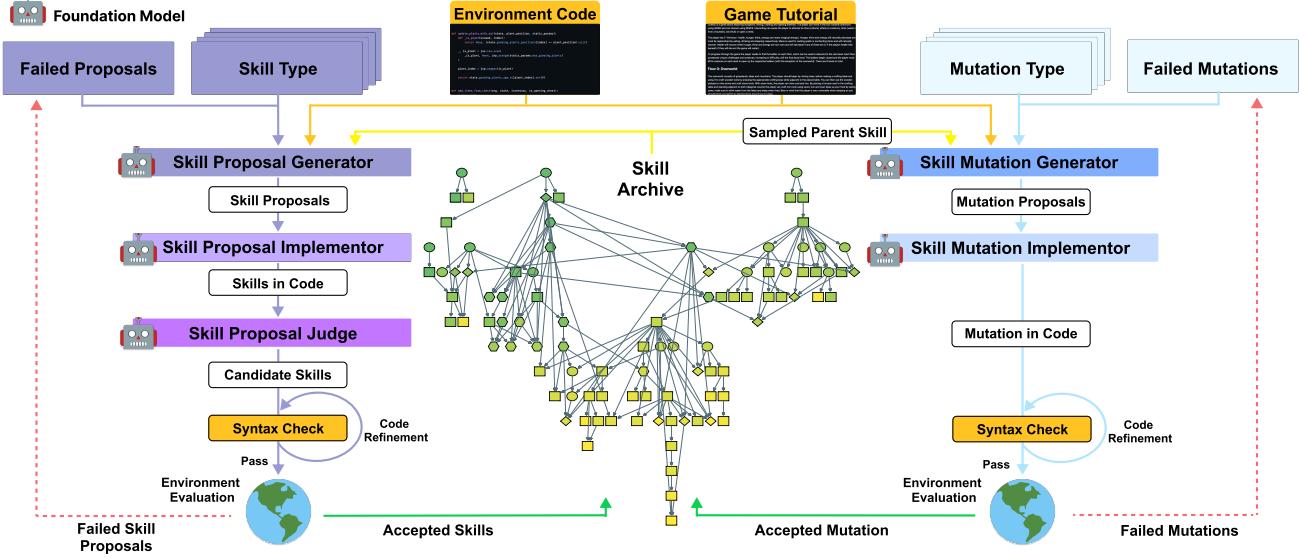


Figure 1. CODE-SHARP consists of two FM-driven iterative processes to discover novel SHARP skills and refine SHARP skills already present in the skill archive. CODE-SHARP utilises a pipeline of FM-based **skill proposal generator**, **implementor**, and **judge** to first generate and filter novel SHARP skills before environment evaluation. Skill refinement is based on the FM-based **skill mutation generator** and **implementor**. Skill mutation proposals are then directly evaluated in the environment.

set of learned capabilities and solve increasingly complex goals.

We evaluate CODE-SHARP in the Craftax environment (Matthews et al.). Our results show that CODE-SHARP discovers, on average, 90 diverse SHARP skills, spanning a broad spectrum of the Craftax skill space. A goal-conditioned agent trained exclusively on the rewards generated by the discovered SHARP skills learns to solve complex, long-horizon goals not achieved by any of our baselines. When composed into high-level policies-in-code by an FM-based policy planner, we find that the discovered SHARP skills can effectively guide the trained agent to complete complex sequences of long-horizon goals, outperforming both pretrained agents and task-specific expert policies by over 134% on average.

2. Method

CODE-SHARP curates a continuously growing, interconnected archive implemented as a directed graph of skills. Each node in the skill archive represents a Skill as Hierarchical Reward Program (SHARP) (Figure 2), generated and implemented in Python code by a set of FMs. CODE-SHARP consists of two FM driven iterative loops as shown in Figure 1. The proposal of novel SHARP skills is performed over three sequential steps: generation, implementation, and selection. First, the skill proposal generator proposes a set of potentially interesting skills. These skills are then implemented in Python code by the skill proposal implementor. Finally, the skill proposal judge selects two SHARP skill

candidates among the generated skills to be evaluated in the environment. Skills deemed as learnable by the current agent are added to the skill archive, while rejected skills are added to the list of failed proposals. To refine proposals, CODE-SHARP randomly samples a SHARP skill from the skill archive and prompts the mutation proposal generator to propose a set of potentially useful mutations, which the mutation proposal implementor translates into SHARP skills in Python code. The implemented mutation proposals are then evaluated using the goal-conditioned policy, and the best-performing mutation replaces the current elite version of the sampled skill if it achieves a higher success rate. Throughout the skill discovery process, CODE-SHARP trains a goal-conditioned agent on rewards generated by the discovered SHARP skills to expand its capabilities in an open-ended manner. The full algorithm of CODE-SHARP is summarised in Section A.

2.1. Problem Formalisation

We model the environment as a Goal-Conditioned Markov Decision Process defined by the tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{G}, P, \gamma \rangle$. Here, \mathcal{S} and \mathcal{A} denote the state and action spaces, $P : \mathcal{S} \times \mathcal{A} \rightarrow \Delta(\mathcal{S})$ represents the transition dynamics, and $\gamma \in [0, 1]$ is the discount factor. The set \mathcal{G} represents the space of all semantic goals. We formalise the problem of open-ended skill discovery as the iterative construction of a dynamic skill archive, structured as a directed acyclic graph $\Lambda_t = (\mathcal{V}_t, \mathcal{E}_t)$, where each node in $\sigma \in \mathcal{V}_t$ represents a skill corresponding to a semantic goal in \mathcal{G} .

Drawing on the Options framework (Sutton et al., 1999),

we define each skill as a tuple $\sigma = \langle \phi_\sigma, \psi_\sigma, \rho_\sigma \rangle$, where $\phi_\sigma : \mathcal{S} \times \mathcal{S} \rightarrow \{0, 1\}$ is a success function acting as the termination condition, and ρ_σ estimates the agent's success rate of completing σ . We define the set of initiation functions $\psi_\sigma = ((c_i, u_i))_{i=1}^m$ as an ordered list mapping environment condition functions c to prerequisite skills u , with each u corresponding to a chosen skill $\sigma \in \mathcal{V}_t$. Directed edges $(u, v) \in \mathcal{E}_t$ are implicitly defined by these dependencies, indicating that satisfying prerequisite skill u_i is required when condition c_i of skill v is not met. The skill discovery process corresponds to a mapping $f : \Lambda_t \rightarrow \Lambda_{t+1}$ that identifies and adds novel, learnable skills to the graph.

The agent acts according to a universal policy $\pi : \mathcal{S} \times \mathcal{V}_t \rightarrow \Delta(\mathcal{A})$, which maps a state and a specific skill from the archive to a distribution over actions. The reward function R is defined relative to the active skill σ : $R(s, a, s' | \sigma) = \alpha(\sigma) \cdot \mathbb{I}[\phi_\sigma(s') = 1]$, where $\alpha(\sigma)$ is an adaptive scaling factor. Consequently, the agent's objective is to maximise the expected cumulative return over the expanding curriculum of discovered skills:

$$J(\pi) = \mathbb{E}_{\substack{\sigma \sim \mathcal{P}(\mathcal{V}_t) \\ \tau \sim \pi(\cdot | \sigma)}} \left[\sum_{k=1}^{\infty} \gamma^k R(s_k, a_k, s_{k+1} | \sigma) \right] \quad (1)$$

where $\mathcal{P}(\mathcal{V}_t)$ is a sampling distribution over the archive.

2.2. Skills as Hierarchical Reward Programs

Each node in the skill archive Λ is represented by an FM-generated Python program, which we term a Skills as Hierarchical Reward Program (SHARP). Each SHARP skill consists of a success condition function ϕ_σ that represents a semantic goal in code (i.e. `agent.inventory.iron_pickaxe >= 1`) which returns a reward upon completion. In addition, the FM defines a set of environment condition functions c_i which verify that the current environment state satisfies the requirements needed (i.e. `agent.inventory.iron >= 1`) to attempt to complete ϕ_σ in s_t . The FM assigns an existing SHARP skill as prerequisite skill u_i (i.e. `MineIron`) that is called if c_i is not satisfied in s_t . SHARP skills receive the current state s_t and previous state s_{t-1} as input, and output their own skill ID or a prerequisite skills ID together with the reward achieved by the agent in s_t .

We define a transition operator $\mathcal{T} : \mathcal{V}_t \times \mathcal{S} \rightarrow \mathcal{V}_t$ that maps a target SHARP skill σ_{target} to the SHARP skill σ_{pre} assigned as prerequisite skill u for the first unmet environment condition c :

$$\mathcal{T}(\sigma_{\text{target}}, s) = \begin{cases} u_i & \text{if } \exists i : c_i(s) = 0 \wedge (\forall j < i, c_j(s) = 1) \\ \sigma_{\text{target}} & \text{otherwise} \end{cases} \quad (2)$$

Given a high-level target skill σ_{target} , CODE-SHARP iteratively applies \mathcal{T} to traverse the hierarchy of dependencies

defined by the environment condition functions, terminating at a fixed point σ_{terminal} where $\mathcal{T}(\sigma_{\text{terminal}}, s_t) = \sigma_{\text{terminal}}$. This terminal skill is then treated as the active skill used to condition the policy $\pi(a | s_t, \sigma_{\text{terminal}})$ and the reward function $R(s_k, a_k, s_{k+1} | \sigma_{\text{terminal}})$ to return any rewards achieved by the agent in s_t while following skill σ_{terminal} . This hierarchical traversal repeats at every environment step, enabling the rapid adaptation of the active skill to stochastic changes in the environment state.

2.3. Open-Ended Archive Expansion

CODE-SHARP adopts an FM-based propose-implement-judge approach to generating novel SHARP skills (Faldor et al.). All prompts used for skill discovery can be found in Section H.

CraftStonePickaxe
Environment Condition Functions
Wood < 1? → MineWoodID
Stone < 1? → MineStoneID
Not near Table? → CraftCraftingTableID
Success Condition Function
Has Stone Pickaxe?: reward = 1.0 done = True
else: reward = 0.0 done = False
return CraftStonePickaxeID, reward, done

Figure 2. Pseudo-Code version of the SHARP skill defining a skill to craft a stone pickaxe.

Skill Proposal Generator The proposal generator produces a set of n skill candidates formatted as pseudo-code. Each proposal specifies a high-level description, a binary success condition ϕ , and a dictionary mapping environment conditions to prerequisite SHARP skills in the existing archive. The skill proposal generator receives as context the pseudo-code of the current skill archive, failed proposals from the previous iterations, the environment source code, and auxiliary information such as tutorials. To ensure diversity, we condition the generation on n skill categories sampled uniformly with replacement from a predefined set of heuristics.

Skill Proposal Implementor The skill proposal implementor translates the pseudo-code candidates generated by the skill proposal generator into executable SHARP skills. As guidance, the skill proposal implementor receives a SHARP skill class template, along with the same context provided to the proposal generator and the existing SHARP skills archive.

Skill Proposal Judge Recent work has shown FMs to be effective proxies for human understanding of interestingness and novelty (Zhang et al.; Faldor et al.). CODE-SHARP therefore leverages an FM-based judge to evaluate the implemented skill proposals against three criteria: 1) *Correctness*: The code must compile and validly reference dependencies in \mathcal{V} . 2) *Feasibility*: The skill must be learnable given the agent's current capabilities. 3) *Novelty*: The skill must tar-

get a distinct region of the skill space compared to existing skills. The Judge selects two candidate skills that maximise the three criteria according to its judgement.

Learnability Evaluation If any of the selected SHARP skill candidates fail to compile, the code and error trace are fed back to the implementor for iterative refinement. This process is repeated up to three times. Once the chosen skills have been successfully implemented, we train a copy of the goal-conditioned agent on the candidate SHARP skills. If the agent shows learning progress on a candidate, it is added to the skill archive. Otherwise, it is added to the archive of failed proposals.

2.4. Open-Ended Archive Refinement

CODE-SHARP leverages FMs to continuously refine SHARP skills that are already present in the skill archive. Each iteration, skills are sampled for refinement with a probability $P(k) \propto (1 - \rho_k)$, where ρ_k is the skill's current success rate. All prompts used for the archive refinement can be found in Section I.

Mutation Proposal Generation The mutation proposal generator is tasked with proposing m possible mutations of the sampled skill, with the aim of increasing the rate of successful skill completion. As context, the mutation generator receives the sampled SHARP skills code, the skill archive, the environment code, and any auxiliary information. We additionally sample m mutation heuristics from a pre-defined set to encourage diverse mutation proposals. The mutation proposal generator generates a mutated skill as pseudo-code, including a description of the mutation, the skill's success condition, the set of environment conditions, and the respective prerequisite skills.

Mutation Proposal Implementor The mutation proposal implementor translates the generated mutation proposals into SHARP skills in Python. It receives the generic SHARP skill class template, the sampled skills' original SHARP skill in code, as well as the environment code, auxiliary information, and the code of all SHARP skills present in the skill archive.

Zero-Shot Filtering. Mutations of the SHARP skills are focused on the environment condition functions and the associated prerequisite skills. As policy π is only conditioned on the active SHARP skill in each environment step, evaluating proposed mutations does not require any retraining of the agent. We evaluate each mutation σ'_k in the environment in isolation based on the current agent. CODE-SHARP updates the node in the archive: $\sigma_k \leftarrow \sigma'_k$ if a mutation demonstrates a higher skill success rate than the currently best performing version of the sampled SHARP skill σ_k .

2.5. Agent Training

We train the agent in a continual, open-ended fashion exclusively on the rewards provided by the SHARP skills generated by CODE-SHARP. At the beginning of each episode, we uniformly sample a target SHARP skill for the agent to complete. If the target skill is completed successfully or 300 environment steps have elapsed, we sample a new target SHARP skill. At each step, we traverse the target SHARP skills dependency graph to find the active SHARP skill, as described in Section 2.2. We use a text embedding model to encode the active SHARP skill's name and concatenate it with the environment state s_t provided to the agent. The episode is terminated once the agent has successfully completed all skills in the archive or 4096 environment steps elapse.

Prerequisite-Aware Importance Sampling To exploit hard-to-reach environment states, we re-weight the sampling probability of a SHARP skill σ_i by the cumulative success rates of the SHARP skills assigned to its satisfied environment conditions in s_t . We define a *satisfied conditions matrix* $\mathbf{N} \in \{0, 1\}^{|\mathcal{V}| \times |\mathcal{V}|}$. An entry $\mathbf{N}_{jk} = 1$ if the environment condition c_i of the SHARP skill σ_j is currently satisfied in state s_t and is assigned the prerequisite SHARP skill σ_k as u_i . We calculate a base sampling weight \mathbf{B}_i for skill σ_j inversely scaled by the cumulative success rates ρ of all SHARP skills assigned as prerequisites in \mathcal{D}_i .

$$\mathbf{B}_j = \frac{1}{\prod_{k=1}^{|\mathcal{V}|} (\rho_k + \epsilon)^{\mathbf{N}_{jk}}} \quad (3)$$

Finally, we apply a state-filter \mathbf{I}_j to remove SHARP skills whose success condition is already satisfied in s_t and use a Top-K filter \mathcal{F}_K to narrow down the distribution. The final sampling probability is $P_j \propto \mathcal{F}_K(\mathbf{B}_j \cdot \mathbf{I}_j)$.

Adaptive Reward Scaling Due to the hierarchical nature of SHARP skills, simply up-sampling low-performing skills to incentivise learning is insufficient, as it results in equal up-sampling of all prerequisite skills. To counter this and provide a more direct learning feedback, we implement adaptive reward scaling (Kwon et al., 2025). We scale the reward r_i provided by SHARP skill σ_i inversely to its success rate:

$$r_i = \min\left(\frac{1}{\rho_i}, 10.0\right). \quad (4)$$

3. Experiments

We evaluate CODE-SHARP in Craftax (Matthews et al.), a procedurally generated environment combining mechanics from Minecraft and NetHack (Küttler et al., 2020), resulting in a rich and essentially open-ended task space. We conduct three independent runs consisting of 100 skill proposal iterations and 85 refinement iterations, training the agent

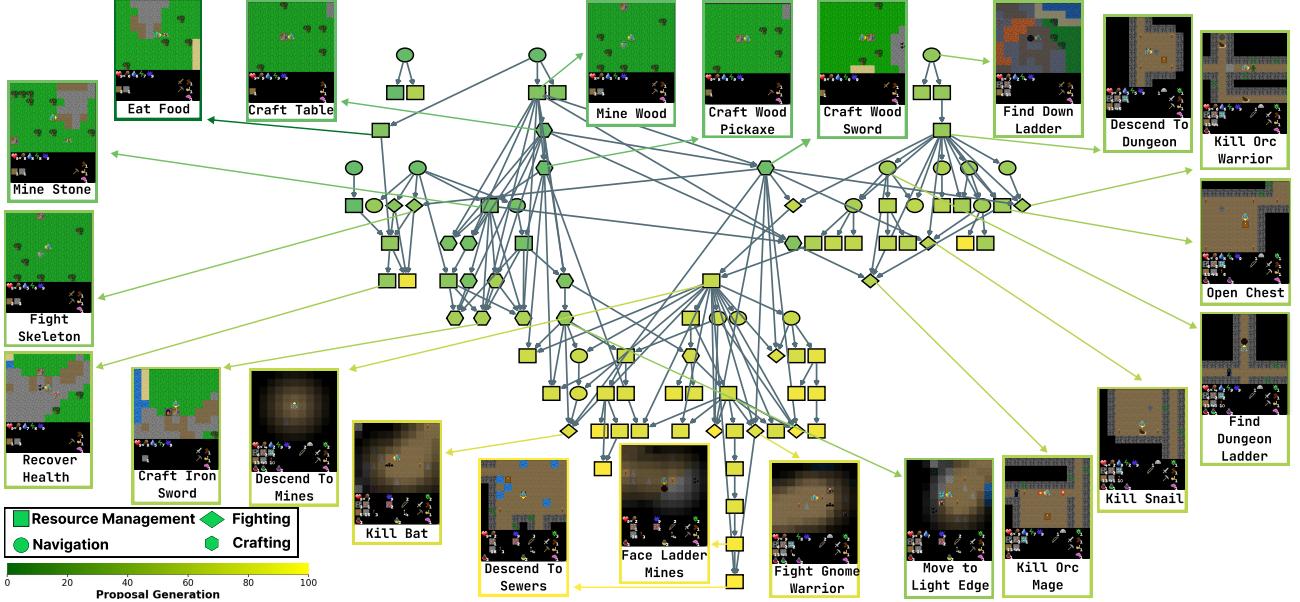


Figure 3. Interconnected archive of discovered SHARP skills. CODE-SHARP continuously builds on existing SHARP skills in the archive to define novel, meaningful skills in line with the natural curriculum of Craftax. Initial skill discovery focuses on the *Overworld* before progressing to the *Dungeon* then the *Mines* and finally the *Sewers*.

for a total of $2e9$ environment steps. The agent architecture is a JAX (Bradbury et al., 2018) implementation of TransformerXL (Dai et al., 2019; Hamon, 2024) trained via PPO (Schulman et al., 2017). For all experiments involving FMs we utilise the open-source Qwen3-235B-A22B-Thinking-2507 (Yang et al., 2025) model. To bootstrap the proposal process, the archive is initialised with three elementary skills: `FindTree`, `FindLake`, and `FindCow`. CODE-SHARP is provided with the environment source code, with any reference to rewards or achievements removed, and a game tutorial provided by the developers of Craftax. In this work, our primary objective is to evaluate FMs’ capacity to autonomously define semantically meaningful, executable skills and structure them into an open-ended archive. To isolate the validity of the generated SHARP skills from orthogonal survival constraints, we enforce a minimum health threshold to prevent premature episode termination due to extrinsic factors, such as hunger or energy depletion.

3.1. Skill Discovery Analysis

CODE-SHARP autonomously discovered an average of 90 SHARP skills per run, generating a skill curriculum that mirrors the natural progression through the different levels of Craftax. We provide a visual overview of the archive with a subset of the skills discovered by CODE-SHARP in Figure 3. CODE-SHARP initially populates the archive with foundational skills in the *Overworld*, such as `MineWood` and `CraftWoodPickaxe`, before building on the discovered `DescendToDungeon` skill to generate skills for the *Dungeon*. After generating the skills

needed to unlock the down ladder on the *Dungeon* level, CODE-SHARP focuses on generating skills for the *Mines* level. Finally, at iteration 100, CODE-SHARP generated the `DescendToSewers` skill. This represents an extremely long-horizon goal, requiring the agent to traverse the *Overworld*, *Dungeon*, and *Mines* by eliminating eight hostile mobs on each level before locating the unlocked ladder down to the next level. Among the baselines evaluated in Section 3.2, only the agent trained on the hand-designed Craftax rewards reaches the *Sewers*, while all others only manage to sparingly descend to the *Dungeon* level.

We find that CODE-SHARP effectively exploits specific game mechanics to define novel skills. For instance, the *Mines* level is initially in full darkness. CODE-SHARP discovered the `MoveToEdgeOfLightLevel2` skill, rewarding the agent for moving to the edge of a lit area. `PlaceTorchAtEdgeLevel2` directly builds on this, rewarding the agent for first moving to the light edge, then placing a torch to extend the lit area’s boundary. By repeatedly calling `PlaceTorchAtEdgeLevel2`, the agent is incentivised to illuminate the entire level, implicitly leading to broad exploration. CODE-SHARP exploits this mechanic to define multiple skills requiring exploration of the *Mines* level.

Overall, CODE-SHARP demonstrates a remarkable ability to effectively reason over the environment code, autonomously growing the skill archive following a sensible learning curriculum of meaningful SHARP skills. The goal-conditioned agent trained exclusively on rewards generated

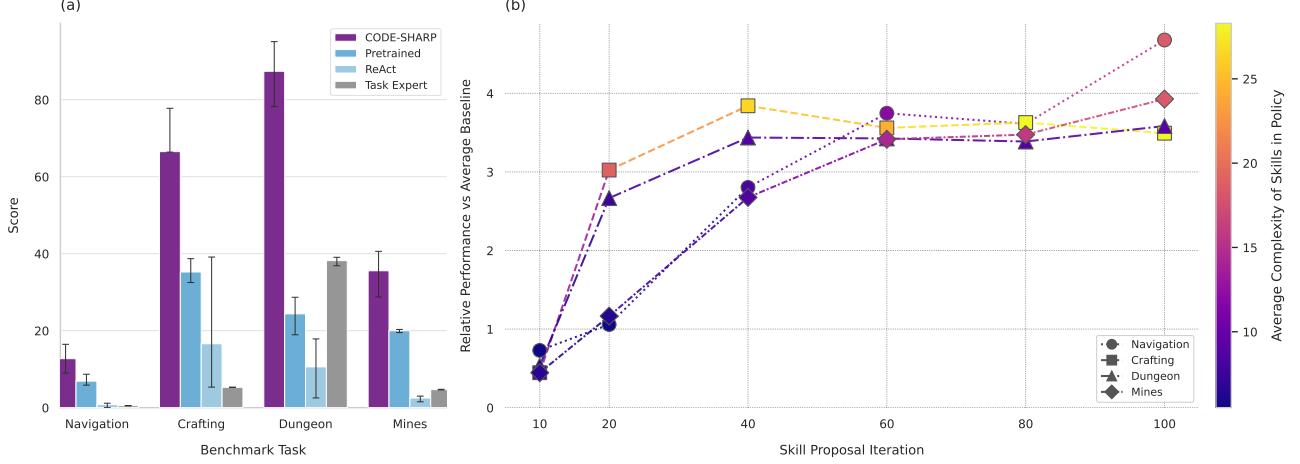


Figure 4. ((a)) Average score achieved on each benchmark task. CODE-SHARP outperforms the zero-shot ReAct LLM agent, the agent pretrained on environment rewards, and the task experts. (b) Evolution of agent capabilities over the course of open-ended skill discovery. The policy planner utilises increasingly complex SHARP skills to define policies-in-code throughout training, resulting in large performance gains relative to the average baseline.

by CODE-SHARP learned to solve complex goals that none of the pretrained and expert baselines managed to achieve, despite large-scale pretraining and access to environment and task-specific rewards. As the skills discussed here represent only a fraction of the skill archive generated by CODE-SHARP, we provide the complete set of discovered skills in pseudo-code from one run in Section K.

3.2. Skill Alignment Evaluation

We rigorously evaluate whether the discovered SHARP skills faithfully encode the semantic intent of their natural-language descriptions and produce reward signals that effectively train the goal-conditioned agent to achieve the stated goal. To this end, we assess the zero-shot composable of the discovered SHARP skills into policies-in-code generated by an FM-based policy planner, following a methodology similar to MaestroMotif (Klissarov et al.). The policy planner is provided with the benchmark task description, the environment source code, and the archive of discovered SHARP skills. The FM-based policy planner constructs the policies-in-code by defining sequential environment conditions and mapping them to existing SHARP skills, thereby decomposing the benchmark scenarios into a series of executable sub-goals for the agent. We provide examples of the policies generated by the policy-planner in Section G.

We introduce four distinct benchmark scenarios designed to probe diverse regions of the Craftax skill space: *Navigation* focuses on locating an enchantment table in the *Sewers* level, requiring advanced exploration and fighting skills to advance through the preceding levels. *Crafting* focuses on crafting a set of advanced iron tools which can be fully completed in the Overworld. *Dungeon* focuses on a scenario

where the agent must first gather useful supplies before advancing to the *Dungeon* to complete a range of tasks there. *Mines* focuses on the *Mines* level, where the agent has to complete crucial survival skills before locating a diamond. For a detailed description of the four benchmark tasks, see Section B. Each benchmark task is designed as a sequence of 5 to 11 milestones which the agent must complete. We compute the geometric mean of the milestone success rates as the final score.

To contextualise the performance of the generated policies-in-code, we compare CODE-SHARP against three distinct baselines. First, we evaluate task-specific expert policies trained exclusively on each benchmark, using rewards based on milestone completion. Second, we compare against an agent trained on the original hand-designed Craftax rewards and applied zero-shot to the benchmark tasks. For these baselines, we use the same TransformerXL architecture trained via PPO and set the same training hyperparameters. Finally, we utilise Qwen3 as a language-instructable zero-shot baseline via the ReAct framework (Yao et al., 2022; Paglieri et al., 2025).

Figure 4 (a) demonstrates that the policy planner successfully composes the autonomously discovered SHARP skills into high-level policies-in-code. The generated policies effectively instruct the goal-conditioned agent to complete complex sequences of milestones, outperforming all baseline scores by over 134% on average across all benchmarks. When analysing the generated policies-in-code, we find that the policy planner routinely utilises skills from the archive that are not required for the benchmarks but may aid the agent in achieving better performance, such as crafting additional tools or stockpiling resources. Our results show that

the agent pretrained on the original Craftax is the strongest baseline. However, its performance is capped by its inability to be instructed to reach specific goals. The task-specific experts perform competitively when milestones are easy to reach and follow each other within short-horizons, such as on the *Dungeon* task, but perform poorly on *Navigation* and *Mines* tasks where milestones require long-horizon skills to complete. Consistent with prior work (Klissarov et al.), we observe that the zero-shot ReAct agent struggles with low-level control, resulting in the lowest overall performance across the benchmarks.

Our results empirically validate that the SHARP skills implemented by CODE-SHARP generate rewards that are semantically aligned with their stated goals. Overall, we find that the goal-conditioned agent trained solely on rewards generated by the SHARP skills discovered by CODE-SHARP is the only method that achieves the advanced milestones of locating the enchantment table and diamonds. Additionally, our goal-conditioned agent reliably completes all milestones in the *Dungeon* and *Crafting* benchmarks, achieving overall success rates of 68.0% and 34.9%, respectively, compared to just 2.5% and 3.1% for the best-performing baseline. For detailed performance across all methods and benchmark milestones, see Section C.

3.3. Archive Evolution Analysis

To analyse how the skill archive’s performance evolves with the addition of new SHARP skills, we prompt the policy planner to generate policies over the current archive at fixed iteration intervals. Additionally, we compute the average complexity of the SHARP skills used by the policy planner for each benchmark task and iteration. We define the complexity of a SHARP skills as one plus the sum of complexities of all its assigned prerequisite SHARP skills: $C(\sigma_k) = 1 + \sum_{u_{k,i} \in \psi_{\sigma_k}} C(\sigma_{u_{k,i}})$. The result of this analysis can be seen in Figure 4 (b). Our findings show that performance consistently improves relative to the baselines as the archive expands. Moreover, the distinct performance trajectories reveal an emergent curriculum: Performance on benchmark tasks focused on earlier levels (*Crafting* and *Dungeon*) increases more quickly than tasks predominantly focused on later levels (*Navigation* and *Mines*). We additionally observe a positive correlation between the performance at a given iteration and the average complexity of the SHARP skills used in the policies generated by the high-level planner, with jumps in performance coinciding with large increases in average skill complexity.

These findings demonstrate that CODE-SHARP effectively leverages the hierarchical structure of SHARP skills to open-endedly compose meaningful and novel skills from previously discovered ones. Consequently, the goal-conditioned agent, trained on this emergent, open-ended

curriculum, learns to solve increasingly long-horizon goals. It is important to note that the discovery process was halted at 100 iterations solely due to computational resource constraints, not due to convergence. As evidenced by the non-saturating performance trend in Figure 4 (b), we hypothesise that CODE-SHARP would continue to discover novel skills and further improve performance if permitted to run for additional iterations.

3.4. Skill Refinement Analysis

Beyond continuously expanding the frontier of discovered skills, CODE-SHARP actively refines the existing archive, generating on average 20 mutated elite versions of SHARP skills. As SHARP skills can be sampled multiple times for refinement, the number of actually successful mutations is higher. To quantify the impact of these mutations, we compare the final success rates of the elite variants against their original base implementations using the same goal-conditioned policy. Our findings show a substantial performance gain of 68.80%, increasing the absolute success rate from 24.30% for the base versions to 41.02% for the elite versions.

When inspecting elite versions of mutated SHARP skills, we find that successful mutations primarily target the re-ordering of environment conditions and the substitution of prerequisite skills with better-suited alternatives. A compelling example is the *KillOrcWarrior* skill. The base implementation required the agent to first descend to the *Dungeon* and subsequently craft a stone sword, which requires resources only found in the *Overworld*. The elite mutation streamlines the condition order and first instructs the agent to craft the sword before descending, drastically improving the success rate.

These findings demonstrate that CODE-SHARP can autonomously identify potential misspecifications and inefficiencies in existing SHARP skills and propose effective refinements.

4. Related Work

4.1. Foundation Models for Open-Ended Skill Discovery

The ability to acquire novel skills and reflect on one’s own mastered abilities is central to general intelligence (Hughes et al., 2024; Jiang et al., 2023; Wang et al., 2020; Team et al., 2023). With the recent rise in FMs’ reasoning abilities and semantic world knowledge, multiple works have proposed utilising FMs to drive open-ended autonomous skill discovery and learning. One prominent class of methods deploys FMs directly as open-ended agents, acting through low-level APIs or direct text commands (Yao et al., 2022; Wang et al., 2023a; 2024; 2023b; Bolton et al., 2025). While these approaches are very general, they typically suffer from

high computational costs and latency due to the need for frequent FM inference at each decision step.

To minimise computational costs, alternative frameworks utilise FMs to guide the learning of RL agents rather than having them directly act in the environment. In these approaches, FMs drive open-ended skill discovery by curating curricula over skills (Zhang et al.; Lu et al., 2025), and novel training environments (Faldor et al.; Liang et al., 2024), evolving agent populations (Lehman et al., 2022), or proposing diverse goals for the agent to pursue (Pourcel et al., 2024; Colas et al., 2023).

4.2. Foundation Models for Autonomous Reward Function Design

Manual reward design is prone to specification errors and reward hacking (Amodei et al., 2016; Ibrahim et al., 2024). With the advancing reasoning capabilities and semantic knowledge of FMs, there has been a growing interest in automating reward function design using FMs. Early works utilised FMs directly as reward models (Klissarov et al., 2023; Klissarov et al.), while recent approaches leverage FMs to generate executable reward code (Yu et al., 2023; Kwon & Michael, 2023). These methods have advanced from zero-shot generation to iterative refinement (Song et al., 2023; Xie et al., 2024) and evolutionary strategies that mutate reward populations to optimize agent performance (Ma et al.; Li et al., 2025). Furthermore, FMs have been used to decompose long-horizon tasks into sub-goals with auxiliary rewards (Castanyer et al., 2025; Pourcel et al., 2024). While these approaches demonstrate the power of FMs for automated reward function design, they remain restricted to optimizing rewards for pre-specified tasks. CODE-SHARP extends this by autonomously discovering novel skills and their corresponding rewards without external direction.

4.3. Foundation Models for Hierarchical Learning

Decomposing long-horizon tasks into temporally extended abstractions is a fundamental challenge in AI, historically addressed through Hierarchical Task Networks (HTNs) (Ghallab et al., 2004; Erol et al., 1994) and the Options framework in reinforcement learning (Sutton et al., 1999; Bacon et al., 2017; Klissarov & Precup, 2021).

Recent neuro-symbolic approaches leverage the reasoning capabilities of FMs to automate task decomposition. These methods translate high-level instructions into sequences of sub-goals, guiding embodied agents either through executable code policies (Ahn et al., 2022; Liang et al., 2022; Prakash et al., 2023; Wang et al., 2023a) or via dense auxiliary rewards and completion signals (Venuto et al., 2024; Castanyer et al., 2025; Pourcel et al., 2024; Nottingham et al., 2023). Bridging these paradigms, MaestroMotif (Klissarov et al.) employs a two-stage approach, utilising

FMs to provide reward feedback to train options (Klissarov et al., 2023) and synthesising high-level policies-in-code over the trained options to complete various challenges in the Nethack Learning Environment (Küttler et al., 2020).

5. Conclusion

In this work, we introduce CODE-SHARP, a general framework that leverages FMs to train a goal-conditioned agent to open-endedly learn to solve novel, increasingly long-horizon goals in complex, stochastic environments. CODE-SHARP achieves this fully autonomously, eliminating the need for hand-designed reward functions. We define Skills as Hierarchical Reward Programs (SHARPs), executable Python programs that return rewards and active goals based on the environment state. By continuously expanding and refining an interconnected skill archive structured as a graph of SHARP skills, CODE-SHARP effectively bridges the gap between open-ended skill discovery and automated optimisation of reward functions.

Thorough experiments in the Craftax environment show that CODE-SHARP generates and refines diverse skill archives, containing on average 90 SHARP skills. We find that CODE-SHARP efficiently leverages the hierarchical structure of SHARP skills to compose novel skills, following a sensible curriculum. Our results demonstrate that the SHARP skills generated by CODE-SHARP faithfully encode their stated semantic goals into executable code. Consequently, an FM-based high-level policy planner can effectively compose the discovered skills into policies-in-code, guiding the goal-conditioned agent to solve a range of complex benchmark tasks. The guided agent significantly outperforms both pretrained agents and expert task-specific baselines by over 134% across all benchmarks. Furthermore, we find that CODE-SHARP continuously generates increasingly complex and novel SHARP skills throughout the discovery process, which the policy-planner effectively integrates into increasingly better performing policies-in-code.

A major limitation of CODE-SHARP is the requirement for the environment to be specified in code to generate the SHARP skills, limiting its applicability to real-world scenarios such as robotics. A promising direction for future research is to extend CODE-SHARP to non code based environments via learned reward models or rewards based on natural-language feedback. Despite this limitation, CODE-SHARP represents a significant step towards autonomous and open-ended training of generalist agents that can solve increasingly complex and diverse goals. CODE-SHARP achieves this without the need for expert-defined rewards, paving the way for the emergence of more general artificial intelligence.

Impact Statement

This paper presents work aimed at advancing research on Open-Endedness in Artificial Intelligence. We acknowledge that as open-ended systems scale, their ability to autonomously acquire capabilities poses potential risks regarding controllability and safety. However, as our research is confined to simulated environments and represents only an initial step towards truly open-ended systems in Artificial Intelligence, we do not feel the need to highlight specific societal impacts here.

References

- Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Fu, C., Gopalakrishnan, K., Hausman, K., et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.
- Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. Concrete problems in ai safety, 2016. URL <https://arxiv.org/abs/1606.06565>.
- Bacon, P.-L., Harb, J., and Precup, D. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.
- Bolton, A., Lerchner, A., Cordell, A., Moufarek, A., Bolt, A., Lampinen, A., Mitenkova, A., Hallingstad, A. O., Vujatovic, B., Li, B., et al. Sima 2: A generalist embodied agent for virtual worlds. *arXiv preprint arXiv:2512.04797*, 2025.
- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL <http://github.com/jax-ml/jax>.
- Castanyer, R. C., Mohamed, F., Castro, P. S., Neary, C., and Berseth, G. Arm-fm: Automated reward machines via foundation models for compositional reinforcement learning. *arXiv preprint arXiv:2510.14176*, 2025.
- Colas, C., Teodorescu, L., Oudeyer, P.-Y., Yuan, X., and Côté, M.-A. Augmenting autotelic agents with large language models, 2023. URL <https://arxiv.org/abs/2305.12487>.
- Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context, 2019. URL <https://arxiv.org/abs/1901.02860>.
- Erol, K., Hendler, J., and Nau, D. S. Htn planning: complexity and expressivity. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, pp. 1123–1128, 1994.
- Faldor, M., Zhang, J., Cully, A., and Clune, J. Omni-epic: Open-endedness via models of human notions of interestingness with environments programmed in code. In *The Thirteenth International Conference on Learning Representations*.
- Ghallab, M., Nau, D., and Traverso, P. *Automated Planning: theory and practice*. Elsevier, 2004.
- Hamon, G. transformerXL_PPO_JAX, July 2024. URL <https://inria.hal.science/hal-04659863>.
- Hughes, E., Dennis, M., Parker-Holder, J., Behbahani, F., Mavalankar, A., Shi, Y., Schaul, T., and Rocktaschel, T. Open-endedness is essential for artificial superhuman intelligence, 2024. URL <https://arxiv.org/abs/2406.04268>.
- Ibrahim, S., Mostafa, M., Jnadi, A., Salloum, H., and Osinenko, P. Comprehensive overview of reward engineering and shaping in advancing reinforcement learning applications. *IEEE Access*, 2024.
- Jiang, M., Rocktäschel, T., and Grefenstette, E. General intelligence requires rethinking exploration. *Royal Society Open Science*, 10(6):230539, 2023.
- Klissarov, M. and Precup, D. Flexible option learning. *Advances in Neural Information Processing Systems*, 34: 4632–4646, 2021.
- Klissarov, M., Henaff, M., Raileanu, R., Sodhani, S., Vincent, P., Zhang, A., Bacon, P.-L., Precup, D., Machado, M. C., and D’Oro, P. Maestromotif: Skill design from artificial intelligence feedback. In *The Thirteenth International Conference on Learning Representations*.
- Klissarov, M., D’Oro, P., Sodhani, S., Raileanu, R., Bacon, P.-L., Vincent, P., Zhang, A., and Henaff, M. Motif: Intrinsic motivation from artificial intelligence feedback. *arXiv preprint arXiv:2310.00166*, 2023.
- Kwon, M. and Michael, S. Reward design with language models. In *International Conference on Learning Representations (ICLR)*, 2023.
- Kwon, M., ElSayed-Aly, I., and Feng, L. Adaptive reward design for reinforcement learning, 2025. URL <https://arxiv.org/abs/2412.10917>.
- Küttler, H., Nardelli, N., Miller, A. H., Raileanu, R., Selvatici, M., Grefenstette, E., and Rocktäschel, T. The nethack learning environment, 2020. URL <https://arxiv.org/abs/2006.13760>.

- Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C., and Stanley, K. O. Evolution through large models, 2022. URL <https://arxiv.org/abs/2206.08896>.
- Li, P., Tang, H., Qiao, J., ZHENG, Y., and HAO, J. Lares: Evolutionary reinforcement learning with LLM-based adaptive reward search. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL <https://openreview.net/forum?id=jRjvcqtA>.
- Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., and Zeng, A. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.
- Liang, W., Wang, S., Wang, H.-J., Bastani, O., Jayaraman, D., and Ma, Y. J. Eurekaverse: Environment curriculum generation via large language models, 2024. URL <https://arxiv.org/abs/2411.01775>.
- Lu, C., Hu, S., and Clune, J. Intelligent go-explore: Standing on the shoulders of giant foundation models, 2025. URL <https://arxiv.org/abs/2405.15143>.
- Ma, Y. J., Liang, W., Wang, G., Huang, D.-A., Bastani, O., Jayaraman, D., Zhu, Y., Fan, L., and Anandkumar, A. Eureka: Human-level reward design via coding large language models. In *The Twelfth International Conference on Learning Representations*.
- Matthews, M., Beukman, M., Ellis, B., Samvelyan, M., Jackson, M. T., Coward, S., and Foerster, J. N. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *Forty-first International Conference on Machine Learning*.
- Norvig, P. A modern approach. 1995.
- Nottingham, K., Ammanabrolu, P., Suhr, A., Choi, Y., Hajishirzi, H., Singh, S., and Fox, R. Do embodied agents dream of pixelated sheep: Embodied decision making using language guided world modelling, 2023. URL <https://arxiv.org/abs/2301.12050>.
- Pagliari, D., Cupiał, B., Coward, S., Piterbarg, U., Wolczyk, M., Khan, A., Pignatelli, E., Łukasz Kuciński, Pinto, L., Fergus, R., Foerster, J. N., Parker-Holder, J., and Rocktäschel, T. Balrog: Benchmarking agentic llm and vlm reasoning on games, 2025. URL <https://arxiv.org/abs/2411.13543>.
- Pourcel, G., Carta, T., Kovač, G., and Oudeyer, P.-Y. Autotelic LLM-based exploration for goal-conditioned RL. In *Intrinsically-Motivated and Open-Ended Learning Workshop @NeurIPS2024*, 2024. URL <https://openreview.net/forum?id=n2OINaKF1e>.
- Prakash, B., Oates, T., and Mohsenin, T. Llm augmented hierarchical agents. *arXiv preprint arXiv:2311.05596*, 2023.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017. URL <https://arxiv.org/abs/1707.06347>.
- Silver, D., Singh, S., Precup, D., and Sutton, R. S. Reward is enough. *Artificial intelligence*, 299:103535, 2021.
- Song, J., Zhou, Z., Liu, J., Fang, C., Shu, Z., and Ma, L. Self-refined large language model as automated reward function designer for deep reinforcement learning in robotics. *CoRR*, 2023.
- Sutton, R. S., Precup, D., and Singh, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.
- Team, A. A., Bauer, J., Baumli, K., Baveja, S., Behbahani, F., Bhoopchand, A., Bradley-Schmiege, N., Chang, M., Clay, N., Collister, A., Dasagi, V., Gonzalez, L., Gregor, K., Hughes, E., Kashem, S., Loks-Thompson, M., Openshaw, H., Parker-Holder, J., Pathak, S., Perez-Nieves, N., Rakicevic, N., Rocktäschel, T., Schroecker, Y., Sygnowski, J., Tuyls, K., York, S., Zacherl, A., and Zhang, L. Human-timescale adaptation in an open-ended task space, 2023. URL <https://arxiv.org/abs/2301.07608>.
- Turing, A. M. Computing machinery and intelligence. In *Parsing the Turing test: Philosophical and methodological issues in the quest for the thinking computer*, pp. 23–65. Springer, 2007.
- Venuto, D., Islam, S. N., Klissarov, M., Precup, D., Yang, S., and Anand, A. Code as reward: Empowering reinforcement learning with vlms. *arXiv preprint arXiv:2402.04764*, 2024.
- Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023a.
- Wang, R., Lehman, J., Rawal, A., Zhi, J., Li, Y., Clune, J., and Stanley, K. O. Enhanced poet: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions, 2020. URL <https://arxiv.org/abs/2003.08536>.
- Wang, Z., Cai, S., Chen, G., Liu, A., Ma, X., and Liang, Y. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023b.

Wang, Z., Cai, S., Liu, A., Jin, Y., Hou, J., Zhang, B., Lin, H., He, Z., Zheng, Z., Yang, Y., et al. Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

Xie, T., Zhao, S., Wu, C. H., Liu, Y., Luo, Q., Zhong, V., Yang, Y., and Yu, T. Text2reward: Reward shaping with language models for reinforcement learning. In *ICLR*, 2024.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Zhou, J., Lin, J., Dang, K., Bao, K., Yang, K., Yu, L., Deng, L., Li, M., Xue, M., Li, M., Zhang, P., Wang, P., Zhu, Q., Men, R., Gao, R., Liu, S., Luo, S., Li, T., Tang, T., Yin, W., Ren, X., Wang, X., Zhang, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Zhang, Y., Wan, Y., Liu, Y., Wang, Z., Cui, Z., Zhang, Z., Zhou, Z., and Qiu, Z. Qwen3 technical report, 2025. URL <https://arxiv.org/abs/2505.09388>.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Yu, W., Gileadi, N., Fu, C., Kirmani, S., Lee, K.-H., Arenas, M. G., Chiang, H.-T. L., Erez, T., Hasenclever, L., Humplik, J., et al. Language to rewards for robotic skill synthesis. In *Conference on Robot Learning*, pp. 374–404. PMLR, 2023.

Zhang, J., Lehman, J., Stanley, K., and Clune, J. Omni: Open-endedness via models of human notions of interestingness. In *The Twelfth International Conference on Learning Representations*.

A. Algorithm

Algorithm 1 CODE-SHARP: Continuous Open-ended Discovery & Evolution

```

1: Input: Max iterations  $E$ , Initial Archive  $\Lambda_0 = (\mathcal{V}_0, \mathcal{E}_0)$ , Initial Policy  $\pi_\theta$ 
2: Initialize: Failed Proposals  $\mathcal{H} \leftarrow \emptyset$ , Success Rates  $\rho \leftarrow \text{Init}(\Lambda_0)$ 
3: for  $t = 1$  to  $E$  do
4:   {Phase 1: Open-Ended Discovery (Sec. 3.3)}
5:    $\mathcal{P}_{\text{raw}} \leftarrow \text{ProposalGenerator}(\Lambda_{t-1}, \mathcal{H}, \text{Context})$ 
6:    $\mathcal{P}_{\text{impl}} \leftarrow \text{ProposalImplementor}(\mathcal{P}_{\text{raw}})$ 
7:    $\mathcal{S}_{\text{cand}} \leftarrow \text{ProposalJudge}(\mathcal{P}_{\text{impl}})$ 
8:   for  $\sigma_{\text{new}} \in \mathcal{S}_{\text{cand}}$  do
9:      $\pi_{\text{copy}} \leftarrow \text{Copy}(\pi_\theta)$ 
10:     $\rho_{\text{new}} \leftarrow \text{EvaluateLearnability}(\pi_{\text{copy}}, \sigma_{\text{new}})$ 
11:    if  $\rho_{\text{new}} > \tau_{\text{learn}}$  then
12:       $\Lambda_t \leftarrow \Lambda_{t-1} \cup \{\sigma_{\text{new}}\}$ 
13:      Update  $\rho(\sigma_{\text{new}})$ 
14:    else
15:       $\mathcal{H} \leftarrow \mathcal{H} \cup \{\sigma_{\text{new}}\}$ 
16:    end if
17:   end for
18:   {Phase 2: Archive Refinement (Sec. 3.4)}
19:   Sample  $\sigma_k \sim P(k) \propto (1 - \rho_k)$ 
20:    $\mathcal{M}_{\text{impl}} \leftarrow \text{MutationPipeline}(\sigma_k, \Lambda_t)$ 
21:    $\rho_{\text{mut}} \leftarrow \text{ZeroShotEval}(\pi_\theta, \mathcal{M}_{\text{impl}})$ 
22:   if  $\rho_{\text{mut}} > \rho_k$  then
23:      $\Lambda_t[\sigma_k] \leftarrow \mathcal{M}_{\text{impl}}$ 
24:     Update  $\rho(\sigma_k)$  with  $\rho_{\text{mut}}$ 
25:   end if
26:   {Phase 3: Agent Training (Sec. 3.5)}
27:    $\pi_\theta, \rho \leftarrow \text{TRAINEPOCH}(\pi_\theta, \Lambda_t, \rho)$ 
28: end for

```

Algorithm 2 Training Epoch

```

1: function TRAINEPOCH( $\pi, \Lambda, \rho$ )
2:   Construct Satisfied Conditions Matrix  $\mathbf{N} \in \{0, 1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ 
3:   Calculate Base Weights  $\mathbf{B}$  using Eq. 3 (Prerequisite-Aware)
4:   Sample target  $\sigma_{\text{target}} \sim \text{TopK}(\mathbf{B} \cdot \mathbf{I})$ 
5:   while Episode not done do
6:     Observe state  $s_t$ 
7:     {Graph Traversal (Eq. 2)}
8:      $\sigma_{\text{active}} \leftarrow \mathcal{T}(\sigma_{\text{target}}, s_t)$ 
9:      $a_t \sim \pi(\cdot | s_t, \sigma_{\text{active}})$ 
10:    Step environment:  $s_{t+1}, \text{done} \leftarrow \text{Env}(a_t)$ 
11:    {Adaptive Reward (Eq. 4)}
12:     $r_t \leftarrow \alpha(\sigma_{\text{active}}) \cdot \mathbb{I}[\phi_{\sigma_{\text{active}}}(s_{t+1})]$ 
13:    Store transition in buffer
14:   end while
15:   Update  $\pi$  via PPO
16:   return  $\pi$ , Updated  $\rho$ 
17: end function

```

B. Description of Benchmark Scenarios

The Craftax environment presents a vast and ostensibly open-ended skill space that remains challenging even for expert human players. Consequently, our benchmark suite is designed to probe the diverse spectrum of capabilities required within the environment, specifically targeting the first four levels, which to our knowledge are the only levels that have been reached by any algorithm trained on Craftax. Drawing on the evaluation methodology of MaestroMotif (Klissarov et al.), we defined four distinct tasks targeting essential high-level competencies: *Navigation*, *Crafting*, and two composite scenarios, *Dungeon* and *Mines*, which require the integration of multiple distinct competencies.

Each benchmark is implemented as a sequence of milestones that must be achieved in order. These tasks are intentionally designed as ambitious, long-horizon objectives intended to quantify incremental progress toward complex goals, rather than as binary pass/fail assessments. Detailed specifications for each benchmark task are provided below. In the following we present the benchmark tasks in detail.

Navigation The navigation benchmark is comprised of 6 milestones which follow the natural progression through the levels of Craftax. First the agent must locate the down ladder on the *Overworld* level to descend to the *Orcish Dungeons*. Once on *Orcish Dungeons*, the agent must find and eliminate 8 hostile orc mobs before being able to unlock the next level, before exploring the level to locate the ladder down to the *Gnomish Mines*. Here the agent must eliminate 8 hostile gnome mobs to unlock the *Sewers* level. The *Gnomish Mines* is initially fully dark prohibiting any form of exploration unless the agent places down torches to light up a limited perimeter. Resources to craft torches can only be found on the preceding levels, requiring the agent to effectively plan ahead. After the agent has located the down ladder the agent should descend to the *Sewers* level. On the *Sewers* level paths are often blocked by water, requiring the agent to place down blocks to displace it and clear a path. The benchmark terminates once the agent has located an enchantment table, an important in game item which can enchant weapons and armour with certain effects.

Crafting The crafting benchmark focuses on 5 milestones related to crafting advanced iron tools. While diamond tools represent the most advanced tool class in Craftax they require diamonds which are not guaranteed to spawn in the overworld. To be able to evaluate an agents crafting ability without it being constrained by its ability to descend to subsequent levels we therefore focus on the iron tools and armour. The first milestone requires the agent to craft a *WoodPickaxe* which requires a total of 3 blocks of wood. For the next milestone the agent needs to craft a *StonePickaxe* which requires the previously crafted pickaxe to mine stone and another block of wood. Next the agent should craft an *IronPickaxe*, which requires a furnace adjacent to a crafting table, a block of coal and the *StonePickaxe* to mine a block of iron. Next, the agent must craft an *IronSword* which again requires the same resources as the *IronPickaxe*. Finally, the benchmark terminates when the agent crafts a set of *IronArmour*, requiring a furnace adjacent to a crafting table as well as 3 blocks of coal and 3 blocks of iron. To solve this benchmark the agent must repeatedly explore the environment to obtain the required resources and complete multi-step crafting processes.

Dungeon The dungeon benchmark focuses on 11 milestones across the *Overworld* and *Orcish Dungeon* levels of Craftax, simulating the process of the agent first having to gather helpful resources and tools before descending to subsequent levels. The agent must first craft a *WoodPickaxe* before crafting a *StonePickaxe* and a *StoneSword*. After these essential tools, the agent should continue by crafting *Torches* and *Arrows* before heading to the grassland and collecting a *Sapling*. Next, the agent must descend to the *Dungeon* level, where it should locate and open loot chests until a *Potion* is dropped. Next, the agent should eliminate at least 2 hostile orcish mobs before locating the upwards ladder and ascending back to the *Overworld*. Finally, the benchmark terminates when the agent has replenished its hunger via eating fruit or killing a cow. To solve this benchmark, the agent must complete a large number of sub-goals for each milestone, making it extremely challenging to fully complete.

Mines The mines benchmark focuses on the *Gnomish Mines* level, an area characterised by a higher spawn rate for rare resources such as diamonds. This benchmark evaluates the agent's ability to descend through the environment and locate a diamond via 10 sequential milestones. Similar to the *Dungeon* benchmark, the agent must first craft a *WoodPickaxe*, *StonePickaxe*, and *Torches* in the *Overworld* before locating the ladder to the *Orcish Dungeon*. After descending, the agent is required to eliminate 8 hostile orc mobs to unlock the passage to the *Gnomish Mines*. Upon reaching the level, the agent must demonstrate survival proficiency by placing a torch and hunting a bat for food. Next the agent must navigate the *Gnomish Mines* level, ideally illuminating its way via the crafted torches, to locate a water source and replenish its thirst. This benchmark is extremely challenging to solve, as it requires strong exploration skills to first descend to the *Gnomish Mines* and understanding of the game mechanics to deal with the darkness before attempting to locate a diamond block comprising approximately 0.5% of blocks on the *Gnomish Mines* level.

C. Detailed Performance Breakdown on Benchmark Tasks

We present the detailed results scores for the policies-in-code over the skills discovered by CODE-SHARP and all baselines in Table 1. For each method, we present the average, minimum, and maximum achieved score. We find that the high-level FM-based policy planner demonstrates a strong ability to compose the discovered SHARP skills to guide the goal-conditioned agent trained by CODE-SHARP to complete complex long-horizon tasks, consistently outperforming all baselines across benchmark tasks.

Table 1. Benchmark Score Results

Benchmark	Zero-Shot		Environment Rewards	Task-Specific training
	CODE-SHARP	ReAct Agent	PPO Pretrained Agent	PPO Task Experts
Navigation	12.72 (8.15, 17.47)	0.773 (0.0, 1.16)	6.89 (5.84, 8.69)	0.50 (0.48, 0.54)
Crafting	66.52 (1.68, 93.29)	16.60 (5.33, 39.13)	35.24 (32.50, 38.71)	5.29 (5.28, 5.31)
Dungeon	87.40 (71.45, 96.20)	10.59 (2.52, 17.86)	24.37 (18.93, 28.68)	38.19 (36.87, 39.06)
Mines	35.56 (28.63, 47.69)	1.52 (2.99, 4.68)	19.98 (19.50, 20.33)	4.68 (4.60, 4.77)

Table 2. Navigation Benchmark Milestone Success Rates

Milestone	CODE-SHARP	Task Specific Expert	Pretrained	ReAct agent
descend_to_dungeon	86.2% (70.3-96.5)	10.3% (9.4-12.1)	84.6% (80.5-89.8)	66.7% (0.0-100.0)
kill_8_dungeon_monsters	78.9% (57.8-89.8)	0.0%	30.6% (25.4-34.8)	0.0%
descend_to_gnomish_mines	59.8% (38.3-75.8)	0.0%	21.0% (17.6-23.4)	0.0%
kill_8_gnomish_mines_monsters	12.6% (0.0-45.7)	0.0%	1.8% (0.8-3.1)	0.0%
descend_to_sewers	0.9% (0.0-3.1)	0.0%	0.7% (0.0-2.0)	0.0%
find_enchantment_table	0.1% (0.0-0.8)	0.0%	0.0%	0.0%

Table 3. Crafting Benchmark Milestone Success Rates

Milestone	CODE-SHARP	Task Specific Expert	Pretrained	ReAct agent
craft_wood_pickaxe	92.6% (46.1-99.5)	99.0% (98.8-99.2)	98.7% (98.0-99.2)	100.0% (0.0-100.0)
craft_stone_pickaxe	87.5% (1.9-99.5)	97.7% (96.9-98.4)	91.5% (88.7-93.0)	100.0% (0.0-100.0)
craft_iron_pickaxe	75.8% (0.0-98.7)	0.0%	45.1% (41.0-50.0)	33.3% (0.0-100.0)
craft_iron_sword	69.7% (0.0-97.1)	0.0%	35.4% (33.6-38.3)	33.3% (0.0-100.0)
craft_iron_armour	34.9% (0.0-74.5)	0.0%	3.1% (2.0-4.3)	0.0%

Table 4. Dungeon Benchmark Milestone Success Rates

Milestone	CODE-SHARP	Task Specific Expert	Pretrained	ReAct Agent
craft_wood_pickaxe	98.3% (93.6-99.5)	99.2% (98.8-99.6)	98.2% (96.9-98.8)	100.0%
craft_stone_pickaxe	98.0% (92.5-99.5)	99.0% (98.4-99.6)	91.8% (91.0-92.2)	100.0%
craft_stone_sword	97.8% (91.7-99.5)	99.0% (98.4-99.6)	85.5% (84.8-85.9)	100.0%
craft_torches	97.0% (90.2-99.5)	98.8% (98.0-99.6)	79.9% (78.9-81.6)	66.7% (0.0, 100.0)
craft_arrows	96.8% (90.2-99.5)	98.8% (98.0-99.6)	73.8% (71.1-77.3)	66.7% (0.0, 100.0)
collect_sapling	96.8% (90.2-99.5)	98.8% (98.0-99.6)	41.5% (32.4-46.5)	66.7% (0.0, 100.0)
descend_to_dungeon	88.9% (73.3-96.9)	38.3% (36.7-40.2)	11.1% (5.1-16.0)	33.3% (0.0, 100.0)
find_potion	83.0% (60.5-94.3)	24.3% (18.4-34.4)	7.7% (3.9-10.9)	0.0%
kill_dungeon_mob	82.2% (58.7-94.3)	23.6% (17.6-34.0)	7.4% (3.9-10.5)	0.0%
ascend_to_overworld	68.9% (39.4-88.8)	10.7% (7.8-13.3)	4.2% (2.0-6.2)	0.0%
replenish_hunger	68.0% (39.0-88.6)	0.4% (0.0-1.2)	2.5% (2.0-3.1)	0.0%

Table 5. Mines Benchmark Milestone Success Rates

Milestone	CODE-SHARP	Task Specific Expert	Pretrained	ReAct Agent
craft_wood_pickaxe	98.5% (97.3-99.2)	98.8% (98.0-99.6)	98.7% (98.0-99.2)	100.0%
craft_stone_pickaxe	98.3% (96.5-99.2)	98.2% (97.3-98.8)	92.6% (91.8-93.0)	100.0%
craft_torches	97.6% (95.3-99.2)	97.5% (96.5-98.4)	90.0% (88.7-91.8)	66.7% (0.0-100.0)
descend_to_dungeon	89.1% (78.1-96.1)	34.9% (30.1-39.8)	68.6% (66.0-72.7)	0.0%
kill_8_dungeon_monsters	69.6% (48.8-90.6)	0.0%	28.4% (27.0-29.7)	0.0%
descend_to_gnomish_mines	47.3% (27.3-70.7)	0.0%	19.5% (17.6-21.5)	0.0%
place_torch	40.1% (18.0-65.2)	0.0%	19.0% (17.2-21.5)	0.0%
kill_bat	31.7% (11.7-59.4)	0.0%	17.8% (16.0-19.9)	0.0%
drink_water	5.9% (1.6-14.8)	0.0%	0.3% (0.0-0.4)	0.0%
find_diamond	1.3% (0.4-2.7)	0.0%	0.0%	0.0%

Figure 5 shows the evolution of the absolute score achieved by the goal-conditioned agent guided by the policies-in-code as the skill archive evolves. We observe large increases in performance for the *Dungeon* and *Crafting* benchmarks which are focused on the first two levels of Craftax. Performance on the *Navigation* and *Mines* benchmarks, which are focused on the later levels of Craftax, continue to increase steadily over the course of skill discovery. These results show that CODE-SHARP continuously explores further into the Craftax skill space to discover meaningful novel skills.

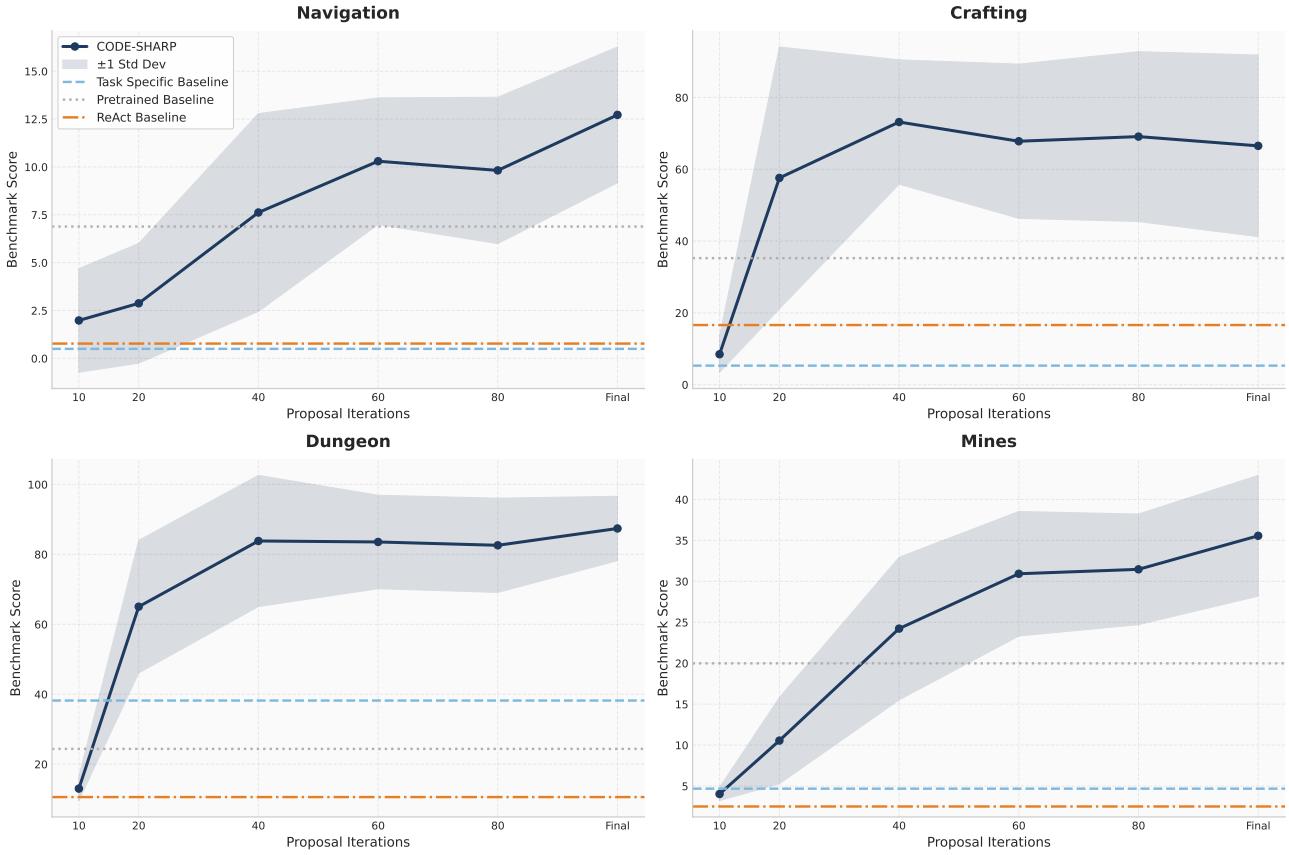


Figure 5. Absolute Benchmark Score vs Proposal Iterations

D. Training Hyperparameters

Table 6. CODE-SHARPHyperaprameters

Hyperparameter	Value
Proposal Iterations	100
Training Episodes per Iteration	10
Mutation Start Iteration	15
Proposal Evaluation Episodes	5
Skill Proposals per Iteration	10
Mutation Proposals per Iteration	5
Sampled Skills for Mutation per Iteration	1

Table 7. Hyperparameters for PPO Training

Hyperparameter	Value
Start Learning Rate	2×10^{-5}
Learning Rate Decay	Linear
End Learning Rate	2×10^{-6}
Batch Size	1024
Optimizer	AdamW
Discount Factor (γ)	0.99
Entropy Coefficient	0.02
Clip Range (ϵ)	0.2
GAE Parameter (λ)	0.99
Total Environment Steps	2×10^9

Table 8. Transformer Architecture Hyperparameters

Hyperparameter	Value
Architecture	Transformer-XL
Number of Layers	2
Number of Attention Heads	8
Embedding Dimension (d_{model})	256
QKV Dimension (d_{qkv})	256
MLP Hidden Size	256
Memory Window Length (N_{mem})	128
Gradient Window Length (N_{grad})	128
Gating	True (Bias 2.0)

E. Ablations on Individual Benchmark Tasks

We perform a detailed ablation study to isolate the contributions of three core components described in Section 2.5: open-ended training (OE), adaptive reward scaling (SR), and opportunistic sampling (OS). For all conditions, the agent is trained on the full generated skill archive, following the chronological order of skill discovery. We evaluate the following configurations:

No OE+SR+OS To evaluate the necessity of open-ended training, we train the agent in a strictly episodic setting. At the start of each episode, a target SHARP skill is sampled uniformly at random. The episode terminates after the target SHARP skills success condition returns True or when the time limit of 300 steps is reached.

No SR+OS We enable open-ended training but disable adaptive reward scaling and opportunistic sampling. In this setting, the next target SHARP skill is sampled uniformly from the skill archive discovered up to the current iteration of training.

No OS We retain open-ended training and adaptive reward scaling but disable opportunistic sampling. New target SHARP skills are sampled uniformly from the skill archive discovered up to the current iteration of training, with the returned rewards being scaled based on their current overall success rate.

CODE-SHARP The complete framework utilising open-ended training, adaptive reward scaling, and opportunistic sampling.

Table 9. Ablation Impact on Average Score over Benchmarks

Condition	Average Score	Absolute Decrease
CODE-SHARP	50.55	—
No OS	31.93	-18.62
No SR+OS	21.20	-29.35
No OE+SR+OS	13.50	-37.05

As detailed in Table 9, we observe a monotonic degradation in performance as components are removed. The full CODE-SHARP framework achieves an average score of 50.09, whereas removing opportunistic sampling (No OS) results in a significant absolute decrease of 21.84, dropping the score to 28.25. Reverting to a purely episodic setting (No OE+SR+OS) results in the lowest performance with an average score across all baselines of 15.34.

Figure 6 further illustrates the impact of these components across individual benchmark tasks. We observe that opportunistic sampling is critical for mastering complex, long-horizon tasks. While all ablations contribute to the agent’s success, the data suggest that opportunistic sampling, by dynamically shifting the training distribution toward the frontier of the agent’s capabilities, provides the largest singular contribution to performance on challenging benchmarks.

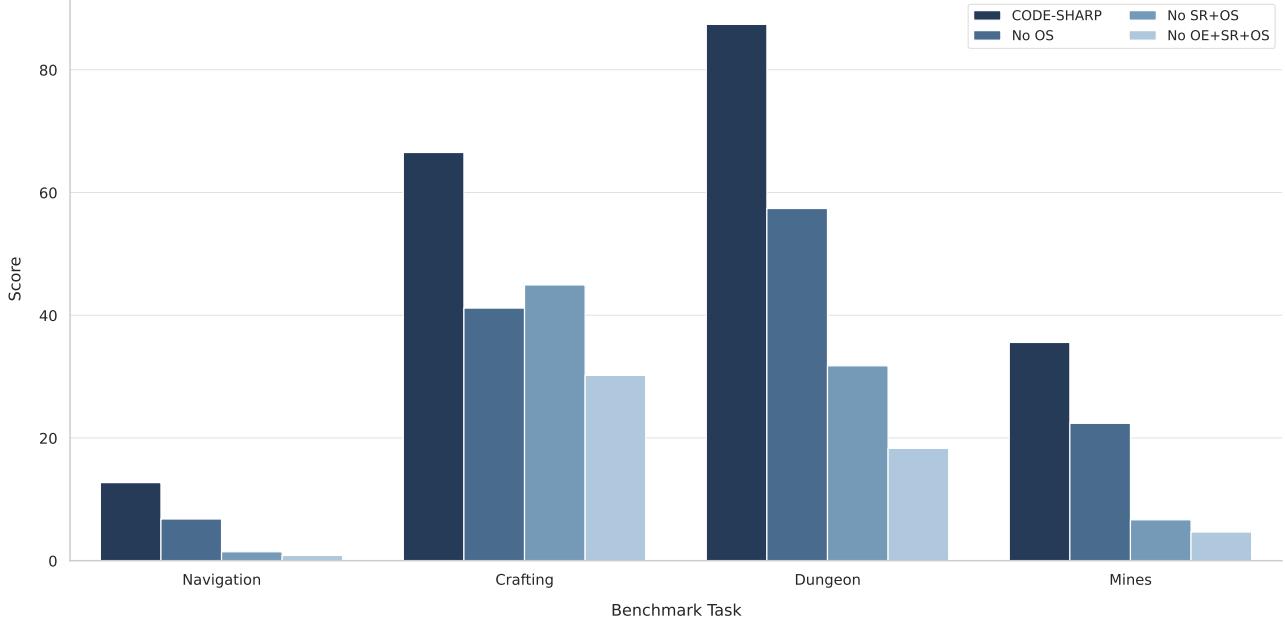


Figure 6. Detailed Ablation Results per Baseline

F. Archive Complexity Evolution

We analyse the evolution of the average complexity of the SHARP skills present in the skill archive. For each iteration, we compute each SHARP skill’s complexity according to the complexity equation presented in Section 3.3. We observe that the average skill complexity in the skill archive steadily increases with subsequent iterations, as seen in Figure 7. This

indicates that SHARP skills proposed by CODE-SHARPlater on in the skill discovery process effectively build on top of the previously implemented SHARP skills already present in the skill archive.

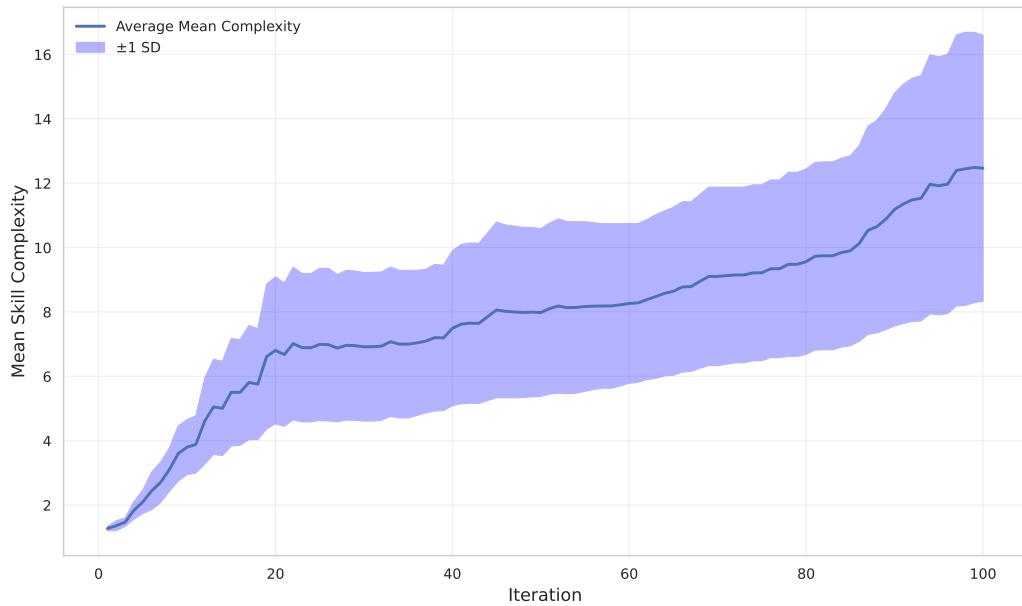


Figure 7. Evolution of average SHARP skill complexity present in the skill archive

G. Examples of Policies-in-Code

We provide an example for the high-level policies-in-code generated by the FM-based policy planner for each benchmark task.

Policy-in-Code for the Crafting Benchmark

```

class BenchmarkSolver:
    def __init__(self):
        self.parent_skill_index = 500
        self.assigned_reward = 1.0

        self.cond_fns = (
            lambda states: self._has_crafting_table(*states),
            lambda states: self._has_wooden_pickaxe(*states),
            lambda states: self._has_furnace(*states),
            lambda states: self._has_stone_pickaxe(*states),
            lambda states: self._has_iron_pickaxe(*states),
            lambda states: self._has_iron_sword(*states),
            lambda states: self._has_iron_armour(*states),
        )

        self.prereq_fns = (
            lambda states: self._do_place_crafting_table(*states),
            lambda states: self._do_craft_wood_pickaxe(*states),
            lambda states: self._do_place_furnace(*states),
            lambda states: self._do_craft_stone_pickaxe(*states),
            lambda states: self._do_craft_iron_pickaxe(*states),
            lambda states: self._do_craft_iron_sword(*states),
            lambda states: self._do_craft_iron_armour(*states),
        )

    def _has_crafting_table(self, prev, cur):
        # Requires external definition of exists_local_object and BlockType
        return exists_local_object(cur, BlockType.CRAFTING_TABLE.value, radius=2,
                                   check_items=False)

    def _has_wooden_pickaxe(self, prev, cur):
        return cur.inventory.pickaxe >= 1

    def _has_furnace(self, prev, cur):
        return exists_local_object(cur, BlockType.FURNACE.value, radius=2, check_items=False)

    def _has_stone_pickaxe(self, prev, cur):
        return cur.inventory.pickaxe >= 2

    def _has_iron_pickaxe(self, prev, cur):
        return cur.inventory.pickaxe >= 3

    def _has_iron_sword(self, prev, cur):
        return cur.inventory.sword >= 3

    def _has_iron_armour(self, prev, cur):
        return jnp.any(cur.inventory.armour == 1)

    def _do_place_crafting_table(self, prev, cur):
        return 20, 0.0, False

    def _do_craft_wood_pickaxe(self, prev, cur):
        return 27, 0.0, False

    def _do_place_furnace(self, prev, cur):

```

```

        return 39, 0.0, False

def _do_craft_stone_pickaxe(self, prev, cur):
    return 38, 0.0, False

def _do_craft_iron_pickaxe(self, prev, cur):
    return 75, 0.0, False

def _do_craft_iron_sword(self, prev, cur):
    return 100, 0.0, False

def _do_craft_iron_armour(self, prev, cur):
    return 93, 0.0, False

def _finalize(self, prev, cur):
    success = self._has_iron_armour(prev, cur)
    reward = jnp.where(success, self.assigned_reward, 0.0)
    return self.parent_skill_index, reward, success

```

Policy-in-Code for the Navigation Benchmark

```

class BenchmarkSolver:
    def __init__(self):
        self.parent_skill_index = 500
        self.assigned_reward = 1.0

        self.cond_fns = (
            lambda states: self._has_drunk_water(*states),
            lambda states: self._has_eaten_food(*states),
            lambda states: self._has_stone_sword(*states),
            lambda states: self._is_on_level1_or_beyond(*states),
            lambda states: self._has_killed_8_mobs_level1(*states),
            lambda states: self._is_on_level2_or_beyond(*states),
            lambda states: self._has_killed_8_mobs_level2(*states),
            lambda states: self._is_on_level3_or_beyond(*states),
            lambda states: self._is_near_enchantment_table(*states),
        )

        self.prereq_fns = (
            lambda states: self._do_drink_water(*states),
            lambda states: self._do_eat_food(*states),
            lambda states: self._do_craft_stone_sword(*states),
            lambda states: self._do_descend_ladder(*states),
            lambda states: self._do_kill_eight_mobs_level1(*states),
            lambda states: self._do_descend_ladder_level1(*states),
            lambda states: self._do_kill_eight_mobs_level2(*states),
            lambda states: self._do_descend_ladder_level2(*states),
            lambda states: self._do_enter_new_room_from_corridor(*states),
        )

    def _has_drunk_water(self, prev: Any, cur: Any) -> jnp.bool_:
        target = 7 + 2 * cur.player_dexterity
        on_level0 = jnp.equal(cur.player_level, 0)
        has_full_water = jnp.greater_equal(cur.player_drink, target)
        return jnp.logical_or(jnp.logical_not(on_level0), has_full_water)

    def _has_eaten_food(self, prev: Any, cur: Any) -> jnp.bool_:
        target = 7 + 2 * cur.player_dexterity
        on_level0 = jnp.equal(cur.player_level, 0)
        has_full_food = jnp.greater_equal(cur.player_food, target)
        return jnp.logical_or(jnp.logical_not(on_level0), has_full_food)

```

```

def _has_stone_sword(self, prev: Any, cur: Any) -> jnp.bool_:
    return jnp.greater_equal(cur.inventory.sword, 2)

def _is_on_level1_or_beyond(self, prev: Any, cur: Any) -> jnp.bool_:
    return jnp.greater_equal(cur.player_level, 1)

def _has_killed_8_mobs_level1(self, prev: Any, cur: Any) -> jnp.bool_:
    return jnp.greater_equal(cur.monsters_killed[1], 8)

def _is_on_level2_or_beyond(self, prev: Any, cur: Any) -> jnp.bool_:
    return jnp.greater_equal(cur.player_level, 2)

def _has_killed_8_mobs_level2(self, prev: Any, cur: Any) -> jnp.bool_:
    return jnp.greater_equal(cur.monsters_killed[2], 8)

def _is_on_level3_or_beyond(self, prev: Any, cur: Any) -> jnp.bool_:
    return jnp.greater_equal(cur.player_level, 3)

def _is_near_enchantment_table(self, prev: Any, cur: Any) -> jnp.bool_:
    # Requires external definition of exists_local_object and BlockType
    on_level3 = jnp.equal(cur.player_level, 3)
    near_table = exists_local_object(
        cur,
        BlockType.ENCHANTMENT_TABLE_ICE.value,
        radius=2,
        check_items=False
    )
    return jnp.logical_and(on_level3, near_table)

def _do_drink_water(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 4, 0.0, False # DrinkWater

def _do_eat_food(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 61, 0.0, False # EatFood

def _do_craft_stone_sword(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 41, 0.0, False # CraftStoneSword

def _do_descend_ladder(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 103, 0.0, False # DescendLadder

def _do_kill_eight_mobs_level1(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 183, 0.0, False # KillEightMobsLevel1

def _do_descend_ladder_level1(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 196, 0.0, False # DescendLadderLevel1

def _do_kill_eight_mobs_level2(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 251, 0.0, False # KillEightMobsLevel2

def _do_descend_ladder_level2(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 314, 0.0, False # DescendLadderLevel2

def _do_enter_new_room_from_corridor(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    return 172, 0.0, False # EnterNewRoomFromCorridorLevel1

def _finalize(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:

```

```

success = self._is_near_enchantment_table(prev, cur)
reward = jnp.where(success, self.assigned_reward, 0.0)
return self.parent_skill_index, reward, success

```

Policy-in-Code for the Dungeon Benchmark

```

class BenchmarkSolver:
    def __init__(self):
        self.parent_skill_index = 500
        self.assigned_reward = 1.0

        self.cond_fns = (
            lambda states: self._has_drunk_water(*states),
            lambda states: self._has_wood_pickaxe(*states),
            lambda states: self._has_stone_pickaxe(*states),
            lambda states: self._has_stone_sword(*states),
            lambda states: self._has_torches(*states),
            lambda states: self._has_arrows(*states),
            lambda states: self._has_sapling(*states),
            lambda states: self._has_potion(*states),
            lambda states: self._has_killed_two_mobs(*states),
            lambda states: self._has_ascended(*states),
            lambda states: self._has_eaten_food(*states),
        )

        self.prereq_fns = (
            lambda states: self._do_drink_water(*states),
            lambda states: self._do_craft_wood_pickaxe(*states),
            lambda states: self._do_craft_stone_pickaxe(*states),
            lambda states: self._do_craft_stone_sword(*states),
            lambda states: self._do_craft_torches(*states),
            lambda states: self._do_craft_arrows(*states),
            lambda states: self._do_collect_sapling(*states),
            lambda states: self._do_find_potion(*states),
            lambda states: self._do_kill_two_mobs(*states),
            lambda states: self._do_ascend(*states),
            lambda states: self._do_eat_food(*states),
        )

    def _has_drunk_water(self, prev, cur):
        # Requires external definition of has_recovered_intrinsic and IntrinsicType
        return has_recovered_intrinsic(prev, cur, IntrinsicType.WATER)

    def _has_wood_pickaxe(self, prev, cur):
        return cur.inventory.pickaxe >= 1

    def _has_stone_pickaxe(self, prev, cur):
        return cur.inventory.pickaxe >= 2

    def _has_stone_sword(self, prev, cur):
        return cur.inventory.sword >= 2

    def _has_torches(self, prev, cur):
        return cur.inventory.torches >= 4

    def _has_arrows(self, prev, cur):
        return cur.inventory.arrows >= 2

    def _has_sapling(self, prev, cur):
        return cur.inventory.sapling > 0

```

```

def _has_potion(self, prev, cur):
    return jnp.any(cur.inventory.potions > 0)

def _has_killed_two_mobs(self, prev, cur):
    return cur.monsters_killed[1] >= 2

def _has_ascended(self, prev, cur):
    return cur.player_level == 0

def _has_eaten_food(self, prev, cur):
    # Requires external definition of has_eaten_food
    return has_eaten_food((prev, cur))

def _do_drink_water(self, prev, cur):
    return 11, 0.0, False # DrinkWater skill index

def _do_craft_wood_pickaxe(self, prev, cur):
    return 27, 0.0, False # CraftWoodPickaxe

def _do_craft_stone_pickaxe(self, prev, cur):
    return 38, 0.0, False # CraftStonePickaxe

def _do_craft_stone_sword(self, prev, cur):
    return 72, 0.0, False # CraftStoneSword

def _do_craft_torches(self, prev, cur):
    return 54, 0.0, False # CraftTorches

def _do_craft_arrows(self, prev, cur):
    return 85, 0.0, False # CraftArrow

def _do_collect_sapling(self, prev, cur):
    return 123, 0.0, False # GatherSapling

def _do_find_potion(self, prev, cur):
    return 234, 0.0, False # LootPotionFromChest

def _do_kill_two_mobs(self, prev, cur):
    return 118, 0.0, False # KillOrcSoldier

def _do_ascend(self, prev, cur):
    return 175, 0.0, False # PositionOnUpwardLadderDungeon

def _do_eat_food(self, prev, cur):
    return 167, 0.0, False # EatSnail

def _finalize(self, prev, cur):
    success = self._has_eaten_food(prev, cur)
    reward = jnp.where(success, self.assigned_reward, 0.0)
    return self.parent_skill_index, reward, success

```

Policy-in-Code for the Mines Benchmark

```

class BenchmarkSolver:
    def __init__(self):
        self.parent_skill_index = 500
        self.assigned_reward = 1.0

        self.cond_fns = (
            lambda states: self._condition_craft_wood_sword(*states),
            lambda states: self._condition_craft_wood_pickaxe(*states),

```

```

        lambda states: self._condition_craft_stone_pickaxe(*states),
        lambda states: self._condition_craft_torches(*states),
        lambda states: self._condition_descend_to_dungeon(*states),
        lambda states: self._condition_kill_8_dungeon_monsters(*states),
        lambda states: self._condition_descend_to_gnomish_mines(*states),
        lambda states: self._condition_place_torch(*states),
        lambda states: self._condition_kill_bat(*states),
        lambda states: self._condition_drink_water(*states),
        lambda states: self._condition_find_diamond(*states),
    )

    self.prereq_fns = (
        lambda states: (20, 0.0, False),
        lambda states: (11, 0.0, False),
        lambda states: (32, 0.0, False),
        lambda states: (93, 0.0, False),
        lambda states: (41, 0.0, False),
        lambda states: (133, 0.0, False),
        lambda states: (146, 0.0, False),
        lambda states: (165, 0.0, False),
        lambda states: (290, 0.0, False),
        lambda states: (211, 0.0, False),
        lambda states: (282, 0.0, False),
    )

    def _condition_craft_wood_sword(self, prev, cur):
        return cur.inventory.sword >= 1

    def _condition_craft_wood_pickaxe(self, prev, cur):
        return cur.inventory.pickaxe >= 1

    def _condition_craft_stone_pickaxe(self, prev, cur):
        return cur.inventory.pickaxe >= 2

    def _condition_craft_torches(self, prev, cur):
        return cur.inventory.torches >= 4

    def _condition_descend_to_dungeon(self, prev, cur):
        return cur.player_level == 1

    def _condition_kill_8_dungeon_monsters(self, prev, cur):
        return cur.monsters_killed[1] >= 8

    def _condition_descend_to_gnomish_mines(self, prev, cur):
        return cur.player_level == 2

    def _condition_place_torch(self, prev, cur):
        return prev.inventory.torches > cur.inventory.torches

    def _condition_kill_bat(self, prev, cur):
        # Requires external definition of was_mob_killed, MobType, and PassiveMobType
        return was_mob_killed(prev, cur, MobType.PASSIVE.value, PassiveMobType.BAT.value)

    def _condition_drink_water(self, prev, cur):
        # Requires external definition of has_recovered_intrinsic and IntrinsicType
        return jnp.logical_and(
            cur.player_level == 2,
            has_recovered_intrinsic(prev, cur, IntrinsicType.WATER)
        )

    def _condition_find_diamond(self, prev, cur):
        # Requires external definition of is_near_block and BlockType

```

```

        return jnp.logical_and(
            cur.player_level == 2,
            is_near_block(cur, BlockType.DIAMOND.value)
        )

    def _finalize(self, prev, cur):
        success = self._condition_find_diamond(prev, cur)
        reward = jnp.where(success, self.assigned_reward, 0.0)
        return self.parent_skill_index, reward, success

```

H. Skill Discovery Prompts

Skill Proposal Generation Prompt

You are a helpful teacher guiding an agent to learn a diverse and powerful set of hierarchical skills in the Craftax environment. Your task is to propose an interesting and novel skill for the agent to learn. The skills should be implemented as hierarchical reward programs. Each skill has Precondition Functions to check if necessary sub-goals (e.g., "possess iron," "near crafting table") are met. If a precondition fails, the skill calls a prerequisite skill, itself defined with its own preconditions, dedicated to achieving that sub-goal, creating a dependency chain. At each environment step, these preconditions are checked sequentially, allowing the system to traverse this chain to find the first skill whose prerequisites are currently satisfied. This sequential, step-by-step checking means previously met conditions can become false, re-triggering their respective sub-goal skills. Finally, a Success Condition verifies if the main objective has been completed, at which point the skill returns a reward to the agent. Using this formulation, your task is to iteratively expand the agents skill archive via these interconnected skills. As the agent is consecutively trained on your proposed skills, they should form a logical curriculum of stepping stones towards more advanced skills. You must first start with very simple skills and then build on top of the learned skills to form more complex skills.

You will be provided with:

- * The agent's current skill repertoire.
- * The category for the new skill you must generate (Navigation, Survival, Gathering Resources, Crafting or Fighting mobs).
- * The environment tutorial and code.
- * Previously proposed skills which failed the evaluation
- * Current skill performance success rate scores

Your proposed skill must fit the given category and help the agent explore the world, manage its survival, gather resources, and craft tools. Carefully review all provided materials to define your proposed skill.

Your Task

1. Propose a Single Skill: Your proposal must be for exactly one novel and diverse skill that fits within the provided category. Ensure the proposed skill is novel. Simple repetitions of existing skills, e.g. MineThreeWood when MineWood is present, are not acceptable. Novelty implies a functional difference, not just a parametric one. \\
2. Build Upon Existing Skills: The proposed skill must expand the agent's current repertoire by building on existing skills. Consider the logical order of skill acquisition to maximize the agent's potential success. The prerequisite skills used to define the proposed skill must be present in the archive. \\
3. Form a Curriculum: Ensure that you start out with simple skills. Continuously build on already learned skills to form a curriculum of increasingly complex skills. The curriculum should be as easy as possible for the agent to follow. \\

4. Skill Exploration: The ultimate goal is for the agent to possess as many diverse and meaningful skills as possible. For this it is crucial to explore as many levels as possible in the environment but also thoroughly explore each level for any meaningful skills proposals not yet present in the agent's skill repertoire. \\
 5. Define Success Condition: Clearly define a condition that indicates the skill has been successfully completed. For crafting tools the success condition must be given as absolute values, i.e. for wood pickaxe it must be `cur.inventory.pickaxe $ >= 1$`. For navigation skills, ensure the success condition includes the level the agent should be on to fulfill the skills' goal. \\
 6. State Starting Conditions: Carefully review the provided environment code to identify any necessary starting conditions. State these conditions using the provided template. If no specific conditions are needed, you must write "No implementation for a condition needed". \\
 7. Condition Order: The agent will complete each condition in the order that is specified in the skill definition. Carefully analyse the environment and the logic of the skill to decide the order in which you place the conditions.
 8. Link Prerequisites: If a starting condition is required, you must name a prerequisite skill from the agent's repertoire that can fulfill it. If a necessary prerequisite skill does not exist, you must disregard your initial idea and instead propose the missing prerequisite skill. \\
 9. Assign Reward: Assign a reward of one to the skill. \\
 10. Proposal History: Carefully go over previously failed skill proposals, analyse why they might not have achieved a high enough performance to be accepted and use them as potential inspiration for new proposals. Never directly repropose one of the failed skills. You can propose a skill with the same objective but it must use a different structure so as to avoid it being rejected for the same reason as the failed proposal. \\
-

ENVIRONMENT DESCRIPTION

`$environment_description`

SAMPLED CATEGORY

`$category`

SKILL REPERTOIRE:

Current Agent Skill Repertoire: `$skill_repertoire`

Success rates: `$success_rates`

FAILED PROPOSALS:

Use the previously failed proposals as inspiration for new proposals. Do not directly reimplement them as they were rejected before. Instead think about why they might have not been successful and propose a fixed version of the skill.

`\$failed_proposals`

OUTPUT FORMAT:

To make it easier to parse your skill proposal, follow this exact format for describing the novel skill. End your text with the skill proposal. Do not generate anything after that:

```
</proposal_start>

"Skill Name": "SkillName1", #Do not include spaces or _ in your skill names
"Skill Type": "The skill type that the proposed skill belongs to",
"Skill Description": "Describe the skill...",
"Assigned Reward": "Integer value for assigned reward",
"Success Condition": "The condition...",
"Starting Conditions": {
    "Condition 1": ["description of function to check if condition is satisfied", "SkillName from repertoire or proposals to satisfy condition 1"],
    "Condition 2": ["description of function to check if condition is satisfied", "SkillName from repertoire or proposals to satisfy condition 2"],
    "Condition 3": ....}
</proposal_end>

<think>
```

Skill Types

Navigation: Navigation skills aim at giving the agent the ability to explore the environment and find specific stationary objects, descending or ascending to different floors, raw materials, resources or mobs present in the environment. As the environment has multiple levels, ensure in the success condition that the agent is on the intended level.

Resource Gathering: Resource gathering skills should give the agent the ability to mine raw materials after locating them to use them for crafting or survival.

Crafting: Crafting skills should aim at allowing the agent to craft increasingly powerful tools based previously gathered resources.

Fighting: Fighting skills should give the agent the ability to attack and defeat individual mobs that the agent can encounter in the environment on the different floors.

Skill Proposal Implementor Prompt

You are a JAX coding assistant. Your goal is to translate a "skill proposal" for the Craftax game environment into a valid Python class, implementing it as a new skill for an agent.

The skills should be implemented as hierarchical reward programs. Each skill has Precondition Functions to check if necessary sub-goals (e.g., "possess iron," "near crafting table") are met.

If a precondition fails, the skill calls a prerequisite skill, itself defined with its own preconditions, dedicated to achieving that sub-goal, creating a dependency chain. At each environment step, these preconditions are checked sequentially, allowing the system to traverse this chain to find the first skill whose prerequisites are currently satisfied. This sequential, step-by-step checking means previously met conditions can become false, re-triggering their respective sub-goal skills. Finally, a Success Condition verifies if the main objective has been completed, at which point the skill returns a reward to the agent. Using this formulation, your task is to iteratively expand the agents skill archive via these interconnected skills.

INPUT: SKILL PROPOSAL

=====

You will receive a proposal containing:

- * name: The skill's name.
- * description: A brief explanation of what the skill does.
- * index: The proposed integer ID for the skill.
- * success_condition: A condition that is True only when the skill is completed successfully.
- * start_conditions (Optional): A set of prerequisite environment states. Each is paired with a prerequisite skill that can achieve it if not specified otherwise.

=====

YOUR TASK: VALIDATION & IMPLEMENTATION

Follow these rules to implement the skill using the provided JAX class template.

1. Validate the Proposal

First, check the proposal for correctness. If it's fundamentally flawed and cannot be fixed, reject it and explain why.

* Success Condition:

- Ensure it's logical and unambiguous.
- Correct minor logical errors.
- For crafting tools the success condition must be given as absolute values, i.e. for wood pickaxe it must be cur.inventory.pickaxe ≥ 1 . For resources it should be relative to the previous state.

* Start Conditions:

- Verify that every prerequisite skill exists in the agent's repertoire.
- If a prerequisite skill is invalid, try to replace it with a valid one. If none exists, reject the proposal.

2. Implement the JAX Class

If the proposal is valid (or you've corrected it), implement the class.

* Indexing:

- Use the proposed index. If it's already taken, increment the index by one until it is unique.
- Refer to prerequisite skills by their correct index.

* Code Requirements:

- Your entire implementation MUST be JAX and JIT-compatible.
- ALWAYS use JAX logical operators (e.g., jnp.logical_and, jnp.logical_or) instead of Python's native 'and', 'or'.
- If the proposal specifies a condition with "No implementation for a condition needed" or you can not add the condition to the skill class. You are allowed to simply ignore it by either not adding it to the code or only adding it as a comment

ENVIRONMENT CODE

\$environment_description\$

SKILL FUNCTION TEMPLATE

```

-----
class Skill:
    """
        A pure logic container for the JAX SkillEnvWrapper.
        The Wrapper handles the traversal, recursion, and cycle detection.
        This class only defines the conditions, prerequisites, and success criteria.
    """

    def __init__(self):
        # Implement: Add the skill's unique index here (e.g., 20)
        self.parent_skill_index = 0
        # Implement: The assigned reward for the skill here
        self.assigned_reward = 1.0

        ##### Implement the tuples containing the condition and prerequisite functions
        #####
        # NOTE: The Wrapper requires these to be tuples of lambdas accepting a single
        'states' argument.
        # Always keep the dummy functions first to prevent empty tuple errors during
        JIT compilation.

        self.cond_fns = (
            # lambda states: self._condition_1(*states),
            # lambda states: self._condition_2(*states),
        )

        self.prereq_fns = (
            # lambda states: self._do_condition1(*states),
            # lambda states: self._do_condition2(*states),
        )

##### Implement the logic functions here #####
# def _condition_1(self, prev: Any, cur: Any) -> jnp.bool_:
#     # Return True if the condition is satisfied (e.g., item is in inventory)
#     return cur.inventory.wood >= 1

# def _do_condition1(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
#     # Return the index of the skill required to satisfy _condition_1
#     # Returns: (Prerequisite Skill Index, 0.0, False)
#     return 5, 0.0, False # Example: Index 5 is "Collect Wood"

##### Implement the success condition (Finalize) here #####
def _finalize(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    # Implement the ultimate success check for this skill
    # e.g., success = cur.inventory.crafting_table >= 1
    success = False

    reward = jnp.where(success, self.assigned_reward, 0.0)
    return self.parent_skill_index, reward, success

```

SKILL REPERTOIRE:

Current Agent Skill Repertoire:

```
$skill_reertoire_dicts$  
$skill_reertoire$
```

SKILL PROPOSAL

The following is the skill proposal with its name, description, success condition, and necessary starting conditions. First, verify that all properties are correctly specified according to the rules and environment mechanics, and that all prerequisite skills are present in the agent's skill repertoire. If not, correct these errors before creating the skill class in Python. If the proposal cannot be corrected, do not output a skill class.

\$skill_proposal_one\$

OUTPUT FORMAT:

To make it easier to parse your skill proposal, follow this exact format for describing the novel skill. End your text with the skill proposal. Do not generate anything after that:

```
</start_proposal>

"Skill Name": "SkillName1",
"Skill Index": "The skill's index corresponds to the index provided by the skill
               proposal",
"Skill Description": "Describe the skill...",
"Skill Function": "The implemented JAX skill function goes here"

</end_proposal>
```

<think>

Skill Proposal Judge Prompt

You are the official Skill Novelty Judge for the Craftax agent's skill development program.
Your primary mandate is to strategically guide the agent's learning path by selecting the two most valuable new skills from a batch of proposals.
Each skill has Precondition Functions to check if necessary sub-goals (e.g., "possess iron," "near crafting table") are met.
If a precondition fails, the skill calls a prerequisite SHARP, itself defined with its own preconditions, dedicated to achieving that sub-goal, creating a dependency chain. At each environment step, these preconditions are checked sequentially, allowing the system to traverse this chain to find the first skill whose prerequisites are currently satisfied. This sequential, step-by-step checking means previously met conditions can become false, re-triggering their respective sub-goal skills. Finally, a Success Condition verifies if the main objective has been completed, at which point the skill returns a reward to the agent. Your decision must balance the creation of a coherent, step-by-step learning curriculum with the goal of developing a diverse and high-performing set of abilities to best guide the iterative expansion of the agents skill archive. For your decision you must strike a balance between choosing more advanced skills which maximally increase the agents capabilities or more novel skills, which might not directly build on top of agents current skill archive
but add a new skill tree and maximally increase the archives diversity.

The Selection Process

You must first scrutinize each proposal individually for quality, compliance, and logical soundness. Any proposal that fails this phase is immediately disqualified.

- * **Criterion 1: Novelty and Feasibility**
 - * **Judge Novelty:** Is the proposed skill a genuinely new capability, or is it a trivial variation or duplication of a skill already in the agent's archive, e.g. CollectThreeWood if CollectWood already exists? **Reject redundant proposals.**
 - * **Evaluate Feasibility:** Does the skill represent a meaningful and achievable behavior within the Craftax environment? **Reject nonsensical or conceptually impossible proposals.**
 - * **Failed Proposals:** Compare the novel skill proposals against the list of previously failed proposals. Reject proposals which strictly reimplement a previously failed proposal.
 - * **Skill Performance:** For each proposal analyse the success rates of prerequisite skills and judge if it is feasible for the novel proposal to be completed given the success rates of any prerequisite skills.
 - * **Skill Balance:** You should balance the number of skills of each category in the skill archive. If recent additions have mostly been from a specific category type, you should aim to select proposals from a different category next.
 - * **Skill Exploration:** You must encourage the agent to reach new levels, while balancing the exploration of previous levels for meaningful skills not present in the skill archive. Favor proposals that either (a) meaningfully unlock new levels to explore, or (b) fill obvious gaps in the current/previous levels (e.g., missing resource collection or mob fighting skills).
 - * **Independent Judging:** You must first and foremost adhere to all the rules described here. Even if there already have been skills added to the archive that violate any of the rules described here, you must not repeat the same mistake by adding another skill that violates the same rule.
- * **Criterion 2: Logical Scrutiny**
 - * **Success Condition:** The condition must be unambiguous and precisely define the skill's unique outcome. For tools, success must be measured by the absolute count (e.g., `current_state.inventory.pickaxe >= 1`). For resources, it must be measured relative to the previous state.
 - * **Start Conditions & Prerequisites:** All start conditions must be strictly necessary. The specified prerequisite skills must be appropriate, exist in the agent's repertoire, and correctly satisfy the conditions.
 - * **No Correction Authority:** You are **not** allowed to make corrections to the code. If a proposal requires any changes to be valid (e.g., syntax errors, incorrect logic, missing prerequisites), you must **reject it**. Only select proposals that are ready to be used as-is.
- * **Criterion 3: Technical Compliance**
 - * **JAX Compliance:** The implementation must be fully JIT-compatible, using only JAX operators.
 - * **Template Adherence:** The code must strictly follow the provided JAX skill class template.
 - * **Index Uniqueness:** If the proposed index is taken, assign the next available unique integer.

EXAMPLES:

The following are a few examples of skill proposal novelty.

KillMobWithIronSword if KillMob exists is **not novel**! Even if killing the mob with an iron sword is more efficient, there already exists a skill with the same general functionality.

GoDownLadderFromLake if GoDownLadder exists is **not novel**! The change in ability is

```

to minor to warrant a completely separate skill.

KillOrcWarrior if KillOrcMage exists is **novel**! The skill focuses on a related but
different mob, so it is novel.

MineThreeIron if MineIron exists is **not novel**! Simple repetitions of existing skills
are not interesting and novel.

DrinkWaterSafe if DrinkWater exists is **not novel**! Simple variations of existing
skills are not interesting and novel and are being handled by the mutation loop.

#### Final Verdict and Report

Your final output must be a clear and structured report containing the following:

1. **Selected Skill:** At most two skills can be selected. If no skill passes your
judgement you are allowed to reject them all.
2. **Filtering:** Start by comparing each skill proposal against the example skills.
Filter out all skills which are would not be classified as novel. Then, make your
final decision based on the set of novel skills.
3. **Justification:** Provide a concise explanation for your choice, explicitly
referencing how the selected skill excels in **Curriculum Coherence**, **Strategic
Value**, and/or **Skill Diversity** compared to the other candidates.
4. **Output:** Provide the class of the skill which you choose as the optimal next
skill to add to agents repertoire in the format provided to you.
5. **Criteria:** The most important criteria your selected skill must possess is
feasibility. Always pick a skill that presents a logical incremental improvement
over extremely difficult skills for which the agent does not possess all
prerequisite skills.

```

ENVIRONMENT CODE

```
$environment_description$
```

SKILL FUNCTION TEMPLATE

```

class Skill:
    """
        A pure logic container for the JAX SkillEnvWrapper.
        The Wrapper handles the traversal, recursion, and cycle detection.
        This class only defines the conditions, prerequisites, and success criteria.
    """

    def __init__(self):
        # Implement: Add the skill's unique index here (e.g., 20)
        self.parent_skill_index = 0
        # Implement: The assigned reward for the skill here
        self.assigned_reward = 1.0

        ##### Implement the tuples containing the condition and prerequisite functions
        #####
        # NOTE: The Wrapper requires these to be tuples of lambdas accepting a single
        'states' argument.
        # Always keep the dummy functions first to prevent empty tuple errors during
        JIT compilation.

        self.cond_fns = (
            # lambda states: self._condition_1(*states),
            # lambda states: self._condition_2(*states),
        )

```

```

        self.prereq_fns = (
            # lambda states: self._do_condition1(*states),
            # lambda states: self._do_condition2(*states),
        )

##### Implement the logic functions here #####
# def _condition_1(self, prev: Any, cur: Any) -> jnp.bool_:
#     # Return True if the condition is satisfied (e.g., item is in inventory)
#     return cur.inventory.wood >= 1

# def _do_condition1(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
#     # Return the index of the skill required to satisfy _condition_1
#     # Returns: (Prerequisite Skill Index, 0.0, False)
#     return 5, 0.0, False # Example: Index 5 is "Collect Wood"

##### Implement the success condition (Finalize) here #####
def _finalize(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    # Implement the ultimate success check for this skill
    # e.g., success = cur.inventory.crafting_table >= 1
    success = False

    reward = jnp.where(success, self.assigned_reward, 0.0)
    return self.parent_skill_index, reward, success

SKILL REPERTOIRE:
-----
Current Agent Skill Repertoire: $skill_reertoire$

Skill performances: $skill_performances$

FAILED PROPOSALS:
-----
$failed_proposals$

SKILL PROPOSAL
-----
The following is the skill proposal with its name, description, success condition, and necessary starting conditions.

$candidate_dict$
$candidate_code$


OUTPUT FORMAT:
-----
If the skill proposal passes your judgement, output your decision following this exact
template specified below. Do not output anything
after the </decision_end> token

</decision_start>
[
{
    "Skill Names": "The name of the first skill you choose",

```

```

        "Proposal Index": Must correspond to the index of the chosen skill proposal,
        "Decision Reason": "Explain your reasoning for choosing this specific skill"
    },
    {
        "Skill Names": "The name of the second skill you choose",
        "Proposal Index": Must correspond to the index of the chosen skill proposal,
        "Decision Reason": "Explain your reasoning for choosing this specific skill"
    }
]
</decision_end>

If the skill does not pass your judgement, simply output "Skill did not pass judgement
".

<think>

```

I. Skill Mutation Prompts

Skill Mutation Proposal Prompt

You are a helpful skill engineer with the job of designing skills as hierarchical reward programs (SHARP) to train an agent in the game of Craftax. Each SHARP consists of precondition functions which define the necessary environment conditions that need to be satisfied in order to be able to execute the parent skill. Each precondition can call an already existing skill from the agents archive of learned skills to satisfy the respective condition. Each SHARP further has a success condition which can check if the parent skill is completed or not. This also returns the reward to train the agent. During the initial implementation there can often be mistakes, oversights or inefficiencies in the defined SHARPs. Your task is to evolve and mutate an already existing SHARP to optimise it based on its current structure and the other skills present in the agents skill archive. For this you will receive a heuristic which you will use as guidance on how to mutate the sampled SHARP based on the agents skill archive and the description of Craftax. Your mutation will be evaluated against the parent skill in the game of Craftax. If it has a better performance in terms of the success rate of completing the skill, the mutation will be accepted as the new elite.

Mutation Rules:

1. Ensure your mutations follow the exact heuristic specification given to you and is sensible.
2. If you add a new new precondition function, ensure that a relevant skill to satisfy it is present in the agents skill archive.
3. Directly follow the output template given to you to define your mutation proposal.
4. All precondition functions should be clearly marked in the mutation preconditions.
5. Carefully analyse the previous failed mutation attempts, if available, to intelligently propose a next mutation. You should not directly reimplement one of the previous failed mutations.
6. Under no circumstances should you mutate the success condition of the parent skill.

ENVIRONMENT DESCRIPTION

\$environment_description

SKILL ARCHIVE

```

$skill_reertoire

PARENT SKILL

$sampled_parent_skill

$sampled_parent_skill_code

PREVIOUSLY FAILED PROPOSED MUTATIONS

$previous_failed_elites

MUTATION HEURISTIC

$heuristic_name

\$heuristic

OUTPUT

The mutation for this index is \$mutation_index. Be sure to use this exact value for
the mutation output dict.
To make it easier to parse your skill proposal, follow this exact format for
describing the novel skill. End your text with the skill proposal. Do not generate
anything after that:

</mutation_start>

"Skill Name": "ParentSkillName",
"Mutation Name": "Name of the mutated skill",
"Skill Index": "The skill's index corresponds to the skill idx of the parent skill",
"Mutation Index": "\$mutation_index",
"Skill Description": "Describe the skill...",
"Assigned Reward": "The reward you assign to the skill",
"Mutation Description": "Describe the mutation and how it should improve the skill
success rate...",
"Success Condition": "The condition...",
"Mutation Starting Conditions": {
    "Condition 1": ["pythonian environment condition", "SkillName from repertoire or
proposals to satisfy condition 1"],
    "Condition 2": ["pythonian environment condition", "SkillName from repertoire or
proposals to satisfy condition 2"]
}
</mutation_end>

<think>

```

Mutation Heuristics

```

{
    "Crossover": "Add new preconditions and prerequisite skills to the parent skill that
synergize with the sampled skill to enhance its success rate. Focus on synergies
that improve survival, ability to navigate the environment, or resource management
(e.g., optimized tool usage). You can also replace prerequisite skills for
existing preconditions with skills from the agent's repertoire that are more
likely to successfully fulfil the specified preconditions.",

    "Efficiency": "Streamline the skill's execution flow. Analyze the logical sequence
of preconditions, ensuring they are ordered for maximum effectiveness (e.g.,

```

```
gathering tools before attempting tasks).",  
    "Simplicity": "Optimize the skill's implementation to be as efficient and  
    streamlined as possible by identifying at least one redundant or distracting  
    precondition that can be removed. Redundant preconditions are those that repeat  
    other preconditions objectives. Distracting preconditions are those that force the  
    agent to complete unnecessary objectives, diverting attention away from the skill  
    's primary objective. Carefully analyse which conditions are absolutely necessary  
    to achieve the sampled skills stated goal and then remove all preconditions  
    deemed distracting or redundant",  
    "Hyperparameters": "Optimize the numerical parameters governing the skill, such as  
    precondition thresholds and reward values. Adjust these parameters to minimize the  
    need for re-visiting preconditions. Ensure the agent gathers sufficient resources  
    in the first instance to sustain the skill's execution."  
}
```

Skill Mutation Implementor Prompt

You are a JAX coding assistant. Your goal is to translate a "skill mutation proposal" for the Craftax game environment into a valid Python class, implementing it as a new skill for an agent.

The skills should be implemented as hierarchical reward programs. Each skill has Precondition Functions to check if necessary sub-goals (e.g., "possess iron," "near crafting table") are met.

If a precondition fails, the skill calls a prerequisite skill, itself defined with its own preconditions, dedicated to achieving that sub-goal, creating a dependency chain. At each environment step, these preconditions are checked sequentially, allowing the system to traverse this chain to find the first skill whose prerequisites are currently satisfied. This sequential, step-by-step checking means previously met conditions can become false, re-triggering their respective sub-goal skills. Finally, a Success Condition verifies if the main objective has been completed, at which point the skill returns a reward to the agent. Using this formulation, your task is to iteratively expand the agents skill archive via these interconnected skills. Your task is to translate a proposed mutation of an existing skill into a valid Python class.

INPUT: MUTATION PROPOSAL

You will receive a mutation proposal containing:

- * 'SkillName': The skill's name.
- * 'SkillIndex': The integer ID for the skill, which you must use.
- * 'SkillDescription': A brief explanation of what the skill does.
- * 'SuccessCondition': A condition that is 'True' only when the skill is completed successfully.
- * 'Mutation Starting Conditions': A set of prerequisite environment states for the mutated skill. Each is paired with a prerequisite skill that can achieve it.

YOUR TASK: VALIDATION & IMPLEMENTATION

Follow these rules to implement the mutated skill using the provided JAX class template.

1. Validate the Proposal

First, check the mutation proposal for correctness. If it's fundamentally flawed and

cannot be fixed, reject it and explain why.

- * **Success Condition:**
 - * Ensure it's logical and unambiguous.
 - * Correct minor logical errors.
 - * For crafting tools, the success condition must be given as absolute values (e.g., `cur.inventory.wood_pickaxe >= 1`). For resources, it should be relative to the previous state.
- * **Mutation Starting Conditions:**
 - * Verify that every prerequisite skill exists in the agent's repertoire.
 - * If a prerequisite skill is invalid, try to replace it with a valid one. If no valid replacement exists, reject the proposal.

2. Implement the JAX Class

If the proposal is valid (or you've corrected it), implement the class.

- * **Indexing:**
 - * Use the exact 'SkillIndex' provided in the proposal.
 - * Refer to prerequisite skills by their correct index from the skill repertoire.
- * **Code Requirements:**
 - * Your entire implementation **MUST** be JAX and JIT-compatible.
 - * **ALWAYS** use JAX logical operators (e.g., `jnp.logical_and`, `jnp.logical_or`) instead of Python's native 'and' or 'or'.
 - * If the proposal specifies a condition with "No implementation for a condition needed," you must implement the condition function but **must not** append it to the `self.cond_fns` list, and the respective prerequisite skill must not be added to the `self.prereq_fns` list.

ENVIRONMENT CODE

\$environment_description

SKILL FUNCTION TEMPLATE

```
class Skill:
    """
        A pure logic container for the JAX SkillEnvWrapper.
        The Wrapper handles the traversal, recursion, and cycle detection.
        This class only defines the conditions, prerequisites, and success criteria.
    """

    def __init__(self):
        # Implement: Add the skill's unique index here (e.g., 20)
        self.parent_skill_index = 0
        # Implement: The assigned reward for the skill here
        self.assigned_reward = 1.0

        ##### Implement the tuples containing the condition and prerequisite functions
        #####
        # NOTE: The Wrapper requires these to be tuples of lambdas accepting a single
        'states' argument.
        # Always keep the dummy functions first to prevent empty tuple errors during
        JIT compilation.

        self.cond_fns = (
            # lambda states: self._condition_1(*states),
            # lambda states: self._condition_2(*states),
```

```

        )

        self.prereq_fns = (
            # lambda states: self._do_condition1(*states),
            # lambda states: self._do_condition2(*states),
        )

##### Implement the logic functions here #####
# def _condition_1(self, prev: Any, cur: Any) -> jnp.bool_:
#     # Return True if the condition is satisfied (e.g., item is in inventory)
#     return cur.inventory.wood >= 1

# def _do_condition1(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
#     # Return the index of the skill required to satisfy _condition_1
#     # Returns: (Prerequisite Skill Index, 0.0, False)
#     return 5, 0.0, False # Example: Index 5 is "Collect Wood"

##### Implement the success condition (Finalize) here #####
def _finalize(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
    # Implement the ultimate success check for this skill. This must exacactly
    # match the success condition of the original parent skill, regardless of
    # any mutations that have been applied.
    # e.g., success = cur.inventory.crafting_table >= 1
    success = False

    reward = jnp.where(success, self.assigned_reward, 0.0)
    return self.parent_skill_index, reward, success

SKILL REPERTOIRE
Current Agent Skill Repertoire: $skill_reertoire

MUTATION PROPOSAL
The following is the mutation proposal. First, verify that all properties are
correctly specified according to
the rules and environment mechanics and that all prerequisite skills are present in
the agent's skill repertoire.
If not, correct these errors before creating the skill class in Python. If the
proposal cannot be corrected,
do not output a skill class.

$mutation_proposal

SAMPLED PARENT SKILL CODE
This is the code of the sampled parent skill. It can diverge from the code in the base
repertoire. Always use the version
given to you here as the base for your code.

$sampled_parent_skill

OUTPUT FORMAT
To make it easier to parse your skill definition, follow this exact format:

</start_proposal>

"Skill Name": "SkillName1",
"Skill Index": "The skill's index corresponds to the index provided by the skill
proposal",
"Skill Description": "Describe the skill...",
"Skill Function": "The implemented JAX skill function goes here"

</end_proposal>

```

<think>

J. Policy Planner Prompt

Policy Planner Prompt

You are an expert AI architect guiding an agent to solve complex tasks in the Craftax environment. Your goal is to design a high-level policy skill, 'BenchmarkSolver', that orchestrates existing skills from a repertoire to complete a specific benchmark.

The 'BenchmarkSolver' acts as a meta-policy. You must define a sequence of ** preconditions** (checks for specific states) and map each to a **prerequisite skill** (an action to achieve that state). The agent will execute these in order.

YOUR TASK: IMPLEMENT BENCHMARK SOLVER

Implement the 'BenchmarkSolver' skill using the JAX class template provided. This skill orchestrates the agent to achieve the provided **Benchmark Milestones**.

1. Skill Structure

- * **Class Name:** 'BenchmarkSolver'
- * **Index:** \$next_skill_index
- * **Template:** Use the provided JAX class structure.

2. Logic & Strategy Requirements

You must define the 'cond_fns' (preconditions) and 'prereq_fns' (actions) lists. Construct them using the following logic:

A. Strategic Planning

Before strictly following the provided milestones, you must analyze the task requirements to ensure agent survival and efficiency.

- * **Preparation:** You are encouraged to define a limited set of auxiliary preconditions for essential tools that are not explicitly listed in the milestones but can assist the agent in completing the milestones.
- * **Focus:** Focus on the only the most critical tools that are required to complete the milestones. Avoid adding too many preparatory steps as it may lead to the agent getting distracted from the main task of solving milestones.
- * **Efficiency:** The agent can get distracted if you add too many preparatory steps. Ensure that you limit the number of preparatory steps to a minimum and that they can be completed easily by the agent.
- * **Ordering:** These preparatory steps must be placed only at the **beginning** of your precondition sequence, ensuring the agent is equipped *before* attempting to solve the milestones.

B. Milestone Execution

- * After your preparatory steps, define preconditions that mirror the **Benchmark Milestones** in order.
- * Ensure the logical flow allows the agent to progress from one milestone to the next without getting stuck.

C. Skill Selection

- * For each precondition, assign the most effective skill from the **Skill Repertoire **.
- * **Best Fit:** Choose the skill with the highest success rate that achieves the goal.
- * **Approximation:** If no skill perfectly matches a milestone, use the closest

```
available alternative or a skill that covers multiple milestones (e.g., a "Mine Stone" skill might naturally cover a "Craft Stone Pickaxe" milestone).  
* **Constraint:** You must use *existing* skills only. Do not invent new skill names.
```

3. Code Requirements

```
* **JAX Compatibility:** The implementation must be pure JAX and JIT-compatible.  
* **Operators:** ALWAYS use 'jnp.logical_and', 'jnp.logical_or', etc., instead of Python native operators.  
* **Imports:** Do not add import statements; assume the environment is pre-loaded.
```

CRAFTAX ENVIRONMENT

```
$environment_description
```

BENCHMARK MILESTONES

```
The benchmark task consists of the following milestones that should be achieved in order:
```

```
$milestones
```

SKILL REPERTOIRE

```
Current Agent Skill Repertoire:
```

```
$skill_reertoire
```

```
Skill Success Rates:
```

```
$success_rates
```

SKILL FUNCTION TEMPLATE

```
class BenchmarkSolver:  
    # Top-level policy for solving the benchmark task  
  
    def __init__(self):  
        self.parent_skill_index = $next_skill_index  
        self.assigned_reward = 1.0  
  
        ##### Implement the tuples containing the condition and prerequisite functions  
        #####  
        # NOTE: The Wrapper requires these to be tuples of lambdas accepting a single  
        'states' argument.  
        # Always keep the dummy functions first to prevent empty tuple errors during  
        JIT compilation.  
  
        self.cond_fns = (  
            # lambda states: self._condition_1(*states),  
            # lambda states: self._condition_2(*states),  
        )
```

```

        self.prereq_fns = (
            # lambda states: self._do_condition1(*states),
            # lambda states: self._do_condition2(*states),
        )

        ##### Implement the logic functions here #####
        # def _condition_1(self, prev: Any, cur: Any) -> jnp.bool_:
        #     # Return True if the condition is satisfied (e.g., item is in inventory)
        #     return cur.inventory.wood >= 1

        # def _do_condition1(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
        #     # Return the index of the skill required to satisfy _condition_1
        #     # Returns: (Prerequisite Skill Index, 0.0, False)
        #     return 5, 0.0, False # Example: Index 5 is "Collect Wood"

        ##### Implement the success condition (Finalize) here #####
        def _finalize(self, prev: Any, cur: Any) -> Tuple[int, float, bool]:
            # Implement the ultimate success check for this skill
            # e.g., success = cur.inventory.crafting_table >= 1
            success = False

            reward = jnp.where(success, self.assigned_reward, 0.0)
            return self.parent_skill_index, reward, success
    
```

OUTPUT FORMAT:

To make it easier to parse your skill proposal, follow this exact format for describing the novel skill. End your text with the skill proposal. Do not generate anything after that:

```

</start_proposal>

"Skill Name": "BenchmarkSolver",
"Skill Description": "Describe the skill...",
"Skill Function": "The implemented JAX skill function goes here"

</end_proposal>
    
```

K. Discovered Skill Archive

In the following we present the skill archive discovered during one of the three runs, consisting of 93 discovered and 3 expert provided skills (`FindTree`, `FindLake`, `FindCow`). The skills where discovered over 100 iterations of skill proposal and agent training.

FindTree

Description: The agent explores the environment until it is in the proximity of a tree (within a radius of 2 blocks).

Skill Type: Navigation

Assigned Reward: 1

Success Condition: `is_near_block(cur, BlockType.TREE.value)`

Starting Conditions: None

FindLake

Description: The agent explores the environment to locate a lake, a vital water source to manage its thirst. This skill aims to teach the agent to identify and approach water bodies for survival.

Skill Type: Navigation

Assigned Reward: 1

Success Condition: `is_near_block(cur, BlockType.WATER.value)`

Starting Conditions: None

FindCow

Description: The agent explores the overworld until it is within 2 blocks of a cow, a passive mob that can be killed for food.

Skill Type: Navigation

Assigned Reward: 1

Success Condition: `is_near_passive_mob(cur, 0, 2)`

Starting Conditions: None

DrinkWater

Description: The agent locates a water source and drinks from it to replenish its thirst intrinsic to maximum capacity, ensuring survival through hydration management.

Skill Type: Survival **Assigned Reward:** 1

Success Condition: `cur.player_drink >= (7 + 2 * cur.player_dexterity)`

Starting Conditions:

- **C1:** `is_near_block(cur, BlockType.WATER.value)` (Requires: FindLake)

MineWood

Description: The agent mines a tree to obtain wood, a fundamental resource required for crafting tools and structures. This skill builds upon tree location by teaching the agent to interact with trees to gather raw materials.

Skill Type: Resource Gathering

Assigned Reward: 1

Success Condition: `cur.inventory.wood >= 1`

Starting Conditions:

- **C1:** `is_near_block(cur, BlockType.TREE.value)` (Requires: FindTree)

KillCow

Description: The agent engages and defeats a cow, a passive mob found in the overworld, to obtain food resources. This skill teaches the agent basic combat mechanics against passive mobs.

Skill Type: Fighting Mobs

Assigned Reward: 1

Success Condition: `was_mob_killed(prev, cur, MobType.PASSIVE.value, PassiveMobType.COW.value)`

Starting Conditions:

- **C1:** `is_near_passive_mob(cur, PassiveMobType.COW.value, 2)` (Requires: FindCow)

PlaceCraftingTable

Description: The agent places a crafting table using wood resources, creating a necessary structure for crafting tools and enabling progression to more advanced skills.

Assigned Reward: 1

Skill Type: Crafting

Success Condition: `is_near_block(cur, BlockType.CRAFTING_TABLE.value)`

Starting Conditions:

- **C1:** `cur.inventory.wood >= 2` (Requires: MineWood)

FindStone

Description: The agent explores the overworld to locate stone deposits within a 2-block radius, which are essential for crafting stone tools and advancing resource gathering capabilities.

Skill Type: Navigation

Assigned Reward: 1

Success Condition: `is_near_block(cur, BlockType.STONE.value)`

Starting Conditions: None

CraftWoodPickaxe

Description: The agent crafts a wooden pickaxe at a crafting table, enabling stone mining for progression. This skill builds upon wood gathering and table placement to teach core crafting mechanics.

Skill Type: Crafting

Assigned Reward: 1

Success Condition: `cur.inventory.pickaxe >= 1`

Starting Conditions:

- **C1:** `cur.inventory.wood >= 1` (Requires: MineWood)
- **C1:** `is_near_block(cur, BlockType.CRAFTING_TABLE.value)` (Requires: PlaceCraftingTable)

CraftWoodSword

Description: The agent crafts a wooden sword at a crafting table using gathered wood resources. This skill builds upon wood acquisition and crafting table placement to teach the agent foundational combat tool creation.

Skill Type: Crafting

Assigned Reward: 1

Success Condition: `cur.inventory.sword >= 1`

Starting Conditions:

- **C1:** `cur.inventory.wood >= 1` (Requires: MineWood)
- **C1:** `is_near_block(cur, BlockType.CRAFTING_TABLE.value)` (Requires: PlaceCraftingTable)

MineStone

Skill Type: Resource Gathering

Description: The agent locates stone deposits and mines them using a wooden pickaxe to obtain stone resources.

Assigned Reward: 1

Success Condition: `cur.inventory.stone >= 1`

Starting Conditions:

- **C1:** `cur.inventory.pickaxe >= 1` (Req: CraftWoodPickaxe)
- **C2:** `is_near_block(cur, BlockType.STONE.value)` (Req: FindStone)

CraftStonePickaxe

Skill Type: Crafting

Description: The agent crafts a stone pickaxe at a crafting table using gathered wood and stone resources.

Assigned Reward: 1

Success Condition: `cur.inventory.pickaxe >= 2`

Starting Conditions:

- **C1:** `cur.inventory.wood >= 1` (Req: MineWood)
- **C2:** `cur.inventory.stone >= 1` (Req: MineStone)
- **C3:** `is_near_block(cur, BlockType.CRAFTING_TABLE.value)` (Req: PlaceCraftingTable)

FindCoal

Skill Type: Navigation

Description: The agent explores the overworld (level 0) to locate coal deposits within a 2-block radius.

Assigned Reward: 1

Success Condition: `(cur.player_level == 0) & is_near_block(cur, BlockType.COAL.value)`

Starting Conditions:

- **C1:** `is_near_block(cur, BlockType.STONE.value)` (Req: FindStone)

MineCoal

Skill Type: Resource Gathering

Description: The agent locates coal deposits identified in stone-rich areas and mines them using a wooden pickaxe.

Assigned Reward: 1

Success Condition: `cur.inventory.coal >= 1`

Starting Conditions:

- **C1:** `cur.inventory.pickaxe >= 1` (Req: CraftWoodPickaxe)
- **C2:** `is_near_block(cur, BlockType.COAL.value)` (Req: FindCoal)

CraftStoneSword

Skill Type: Crafting

Description: The agent crafts a stone sword at a crafting table using gathered wood and stone resources.

Assigned Reward: 1

Success Condition: `cur.inventory.sword >= 2`

Starting Conditions:

- **C1:** `cur.inventory.wood >= 1` (Req: MineWood)
- **C2:** `cur.inventory.stone >= 1` (Req: MineStone)
- **C3:** `is_near_block(cur, BlockType.CRAFTING_TABLE.value)` (Req: PlaceCraftingTable)

PlaceFurnace

Skill Type: Crafting

Description: The agent places a furnace using gathered stone resources, creating essential infrastructure for smelting ores.

Assigned Reward: 1

Success Condition: `is_near_block(cur, BlockType.FURNACE.value)`

Starting Conditions:

- **C1:** `cur.inventory.stone >= 1` (Req: MineStone)

CraftTorch

Skill Type: Crafting

Description: The agent crafts torches at a crafting table using gathered wood and coal resources.

Assigned Reward: 1

Success Condition: `cur.inventory.torches >= 1`

Starting Conditions:

- **C1:** `cur.inventory.wood >= 1` (Req: MineWood)
- **C2:** `cur.inventory.coal >= 1` (Req: MineCoal)
- **C3:** `is_near_block(cur, BlockType.CRAFTING_TABLE.value)` (Req: PlaceCraftingTable)

MineSapling

Skill Type: Resource Gathering

Description: The agent mines grass blocks in the overworld (level 0) near cows to obtain saplings.

Assigned Reward: 1

Success Condition: `cur.inventory.sapling >= 1`

Starting Conditions:

- **C1:** `is_near_passive_mob(cur, PassiveMobType.COW.value, 2)` (Req: FindCow)

EatFood

Skill Type: Survival

Description: The agent locates and kills cows in the overworld (level 0) until its food level reaches maximum capacity.

Assigned Reward: 1

Success Condition: `(cur.player_level == 0) & (cur.player_food >= (7 + 2 * cur.player_dexterity))`

Starting Conditions:

- **C1:** `is_near_passive_mob(cur, PassiveMobType.COW.value, 2)` (Req: FindCow)

CraftArrow

Skill Type: Crafting

Description: The agent crafts arrows at a crafting table using gathered wood and stone resources.

Assigned Reward: 1

Success Condition: `cur.inventory.arrows >= 1`

Starting Conditions:

- **C1:** `cur.inventory.wood >= 1` (Req: MineWood)
- **C2:** `cur.inventory.stone >= 1` (Req: MineStone)
- **C3:** `is_near_block(cur, BlockType.CRAFTING_TABLE.value)` (Req: PlaceCraftingTable)

KillZombie

Skill Type: Fighting Mobs

Description: The agent locates and defeats a zombie, an aggressive melee mob that spawns in stone-rich areas.

Assigned Reward: 1

Success Condition: `(cur.player_level == 0) & was_mob_killed(prev, cur, MobType.MELEE.value, MeleeMobType.ZOMBIE.value)`

Starting Conditions:

- **C1:** `is_near_melee_mob(cur, MeleeMobType.ZOMBIE.value, 2)` (Req: FindStone)

MineIron

Skill Type: Resource Gathering

Description: The agent mines iron ore deposits using a stone pickaxe, obtaining iron resources.

Assigned Reward: 1

Success Condition: `cur.inventory.iron >= 1`

Starting Conditions:

- **C1:** `cur.inventory.pickaxe >= 2` (Req: CraftStonePickaxe)
- **C2:** `is_near_block(cur, BlockType.STONE.value)` (Req: FindStone)

KillSkeleton

Skill Type: Fighting Mobs

Description: The agent locates and defeats a skeleton, a ranged mob that spawns in stone-rich areas.

Assigned Reward: 1

Success Condition: `(cur.player_level == 0) & was_mob_killed(prev, cur, MobType.RANGED.value, RangedMobType.SKELETON.value)`

Starting Conditions:

- **C1:** `cur.inventory.sword >= 1` (Req: CraftWoodSword)
- **C2:** `(cur.player_level == 0) & is_near_ranged_mob(cur, RangedMobType.SKELETON.value, 2)` (Req: FindStone)

SleepToFullEnergy

Skill Type: Survivability

Description: The agent ensures its survival intrinsics are stabilized and eliminates threats before sleeping to recover energy.

Assigned Reward: 1

Success Condition: `(cur.player_level == 0) & (cur.player_energy >= (7 + 2 * cur.player_dexterity))`

Starting Conditions:

- **C1:** `(cur.player_level == 0) & (cur.player_food >= (7 + 2 * cur.player_dexterity))` (Req: EatFood)
- **C2:** `(cur.player_level == 0) & (cur.player_drink >= (7 + 2 * cur.player_dexterity))` (Req: DrinkWater)
- **C3:** `not is_near_melee_mob(cur, MeleeMobType.ZOMBIE.value, 3)` (Req: KillZombie)
- **C4:** `not is_near_ranged_mob(cur, RangedMobType.SKELETON.value, 3)` (Req: KillSkeleton)

FindLadderDown

Skill Type: Navigation

Description: The agent explores the overworld (level 0) to locate the downward ladder required to descend to the dungeon level.

Assigned Reward: 1

Success Condition: `(cur.player_level == 0) & exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True)`

Starting Conditions: None

PlaceTorch

Skill Type: Survivability

Description: The agent utilizes crafted torches to place them on valid adjacent blocks, illuminating the immediate surroundings.

Assigned Reward: 1

Success Condition: `exists_stationary_object(cur, ItemType.TORCH.value)`

Starting Conditions:

- **C1:** `cur.inventory.torches >= 1` (Req: CraftTorch)

RecoverHealth

Skill Type: Survivability

Description: The agent ensures its survival intrinsics are fully replenished and in a safe environment, allowing passive health regeneration.

Assigned Reward: 1

Success Condition: (cur.player_level == 0) & (cur.player_health >= (8 + cur.player_strength))

Starting Conditions:

- **C1:** (cur.player_level == 0) & (cur.player_energy >= (7 + 2 * cur.player_dexterity)) (Req: SleepToFullEnergy)

StepOntoLadderDown

Skill Type: Navigation

Description: The agent moves from being within a 2-block radius of the downward ladder to standing exactly on the ladder block.

Assigned Reward: 1

Success Condition: (cur.player_level == 0) & (cur.item_map[cur.player_level, cur.player_position[0], cur.player_position[1]] == ItemType.LADDER_DOWN.value)

Starting Conditions:

- **C1:** exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True) (Req: FindLadderDown)

FindIron

Skill Type: Resource Gathering

Description: The agent explores the overworld (level 0) to locate iron ore deposits within a 2-block radius.

Assigned Reward: 1

Success Condition: (cur.player_level == 0) & is_near_block(cur, BlockType.IRON.value)

Starting Conditions:

- **C1:** (cur.player_level == 0) & is_near_block(cur, BlockType.STONE.value) (Req: FindStone)

DescendLadder

Skill Type: Navigation

Description: The agent executes the descend action while positioned on the downward ladder in the overworld to transition to the dungeon.

Assigned Reward: 1

Success Condition: cur.player_level == 1

Starting Conditions:

- **C1:** (cur.player_level == 0) & (cur.item_map[cur.player_level, cur.player_position[0], cur.player_position[1]] == ItemType.LADDER_DOWN.value) (Req: StepOntoLadderDown)

FindChestLevel

Skill Type: Navigation

Description: The agent navigates the dungeon level (level 1) to locate a chest within a 2-block radius.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & is_near_block(cur, BlockType.CHEST.value)

Starting Conditions:

- **C1:** cur.player_level == 1 (Req: DescendLadder)

FindOrcSoldierLevel1

Skill Type: Navigation

Description: The agent explores the dungeon level (level 1) to locate an Orc Soldier, a melee mob that spawns in dungeon rooms, within a 2-block radius.

Assigned Reward: 1

Success Condition: (cur.player.level == 1) & is_near_melee_mob(cur, MeleeMobType.ORC_SOLDIER.value, 2)

Starting Conditions:

- C1: cur.player.level == 1 (Req: DescendLadder)

FaceLadderDown

Skill Type: Navigation

Description: The agent positions itself adjacent to the downward ladder on the overworld (level 0) and rotates to face the ladder block.

Assigned Reward: 1

Success Condition: (cur.player.level == 0) & (cur.item_map[0, cur.player.position[0] + DIRECTIONS[cur.player.direction][0], cur.player.position[1] + DIRECTIONS[cur.player.direction][1]] == ItemType.LADDER_DOWN.value)

Starting Conditions:

- C1: (cur.player.level == 0) & exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True) (Req: FindLadderDown)

KillOrcWarrior

Skill Type: Fighting Mobs

Description: The agent locates and defeats an Orc Warrior, a melee mob that spawns in dungeon rooms on level 1.

Assigned Reward: 1

Success Condition: (cur.player.level == 1) & was_mob_killed(prev, cur, MobType.MELEE.value, MeleeMobType.ORC_SOLDIER.value)

Starting Conditions:

- C1: cur.player.level == 1 (Req: DescendLadder)
- C2: cur.inventory.sword >= 1 (Req: CraftWoodSword)
- C3: (cur.player.level == 1) & is_near_melee_mob(cur, MeleeMobType.ORC_SOLDIER.value, 2) (Req: FindOrcSoldierLevel1)

OpenChestForBowLevel1

Skill Type: Resource Gathering

Description: The agent locates and opens the first chest on the dungeon level (level 1) to obtain a bow.

Assigned Reward: 1

Success Condition: (cur.player.level == 1) & (cur.inventory.bow >= 1)

Starting Conditions:

- C1: cur.player.level == 1 (Req: DescendLadder)
- C2: (cur.player.level == 1) & is_near_block(cur, BlockType.CHEST.value) (Req: FindChestLevel1)

FindFountainLevel1

Skill Type: Navigation

Description: The agent leverages its ability to locate dungeon chests to efficiently explore adjacent rooms and identify fountains.

Assigned Reward: 1

Success Condition: (cur.player.level == 1) & is_near_block(cur, BlockType.FOUNTAIN.value)

Starting Conditions:

- C1: cur.player.level == 1 (Req: DescendLadder)
- C2: is_near_block(cur, BlockType.CHEST.value) (Req: FindChestLevel1)

FindDungeonTorch

Skill Type: Navigation

Description: The agent explores the dungeon level (level 1) to locate pre-placed torches in room corners, serving as navigation landmarks.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & exists_stationary_object(cur, ItemType.TORCH.value)

Starting Conditions:

- **C1:** cur.player_level == 1 (Req: DescendLadder)

OpenChestForArrowsLevel1

Skill Type: Resource Gathering

Description: The agent locates and opens a chest on the dungeon level (level 1) to obtain arrows.

Assigned Reward: 1

Success Condition: cur.inventory.arrows >= 1

Starting Conditions:

- **C1:** cur.player_level == 1 (Req: DescendLadder)
- **C2:** is_near_block(cur, BlockType.CHEST.value) (Req: FindChestLevel1)

DrinkFromFountainLevel1

Skill Type: Survability

Description: The agent locates a fountain in the dungeon level (level 1) and drinks from it to replenish thirst.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & (cur.player_drink >= (7 + 2 * cur.player_dexterity))

Starting Conditions:

- **C1:** (cur.player_level == 1) & is_near_block(cur, BlockType.FOUNTAIN.value) (Req: FindFountainLevel1)

PlaceFurnaceAdjacent

Skill Type: Crafting

Description: The agent locates the existing crafting table and strategically places a furnace in an adjacent block (including diagonally).

Assigned Reward: 1

Success Condition: is_near_block(cur, BlockType.CRAFTING_TABLE.value) & is_near_block(cur, BlockType.FURNACE.value)

Starting Conditions:

- **C1:** is_near_block(cur, BlockType.CRAFTING_TABLE.value) (Req: PlaceCraftingTable)
- **C2:** cur.inventory.stone >= 1 (Req: MineStone)

CraftIronArmour

Skill Type: Crafting

Description: The agent crafts a piece of iron armour at a crafting table while adjacent to a furnace using gathered iron and coal resources.

Assigned Reward: 1

Success Condition: jnp.any(cur.inventory.armour >= 1)

Starting Conditions:

- **C1:** cur.inventory.iron >= 3 (Req: MineIron)
- **C2:** cur.inventory.coal >= 3 (Req: MineCoal)
- **C3:** is_near_block(cur, BlockType.CRAFTING_TABLE.value) (Req: PlaceCraftingTable)
- **C4:** is_near_block(cur, BlockType.FURNACE.value) (Req: PlaceFurnace)

CraftIronPickaxe

Skill Type: Crafting

Description: The agent crafts an iron pickaxe at a crafting table while ensuring the furnace is placed adjacently.

Assigned Reward: 1

Success Condition: `cur.inventory.pickaxe >= 3`

Starting Conditions:

- **C1-C4:** Wood, Stone, Iron, Coal ≥ 1
- **C5:** `is_near_block(cur, BlockType.CRAFTING_TABLE.value) & is_near_block(cur, BlockType.FURNACE.value)` (Req: PlaceFurnaceAdjacent)

FindLadderDownLevel1

Skill Type: Navigation

Description: The agent explores the dungeon level (level 1) to locate the downward ladder by leveraging dungeon torches as navigation landmarks.

Assigned Reward: 1

Success Condition: `(cur.player.level == 1) & exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True)`

Starting Conditions:

- **C1:** `cur.player.level == 1` (Req: DescendLadder)
- **C2:** `exists_stationary_object(cur, ItemType.TORCH.value)` (Req: FindDungeonTorch)

CraftIronSword

Skill Type: Crafting

Description: The agent crafts an iron sword at a crafting table while ensuring furnace adjacency.

Assigned Reward: 1

Success Condition: `cur.inventory.sword >= 3`

Starting Conditions:

- **C1-C4:** Wood, Stone, Iron, Coal ≥ 1
- **C5:** `is_near_block(cur, BlockType.CRAFTING_TABLE.value) & is_near_block(cur, BlockType.FURNACE.value)` (Req: PlaceFurnaceAdjacent)

StepOntoLadderDownLevel1

Skill Type: Navigation

Description: The agent navigates from proximity to the downward ladder on the dungeon level (level 1) to stand precisely on the ladder block.

Assigned Reward: 1

Success Condition: `(cur.player.level == 1) & (cur.item_map[cur.player.level, cur.player_position[0], cur.player_position[1]] == ItemType.LADDER_DOWN.value)`

Starting Conditions:

- **C1:** `(cur.player.level == 1) & exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True)` (Req: FindLadderDownLevel1)

FindUpLadderLevel1

Skill Type: Navigation

Description: The agent explores the starting room on the dungeon level (level 1) to locate the upward ladder.

Assigned Reward: 1

Success Condition: `(cur.player.level == 1) & exists_ladder_object(cur, ItemType.LADDER_UP.value)`

Starting Conditions:

- **C1:** `cur.player.level == 1` (Req: DescendLadder)

MineGrassForSapling

Skill Type: Resource Gathering

Description: The agent mines grass blocks in the overworld (level 0) to obtain saplings, a critical resource for establishing sustainable food sources.

Assigned Reward: 1

Success Condition: cur.inventory.sapling >= 1

Starting Conditions:

- C1: (cur.player_level == 0) & is_near_block(cur, BlockType.GRASS.value) (Req: FindTree)

KillOrcMage

Skill Type: Fighting mobs

Description: The agent locates and defeats an Orc Mage, a ranged mob that spawns in dungeon rooms on level 1.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & was_mob_killed(prev, cur, MobType.RANGED.value, RangedMobType.ORC_MAGE.value)

Starting Conditions:

- C1: cur.player_level == 1 (Req: DescendLadder)
- C2: cur.inventory.bow >= 1 (Req: OpenChestForBowLevel1)
- C3: cur.inventory.arrows >= 1 (Req: CraftArrow)
- C4: is_near_ranged_mob(cur, RangedMobType.ORC_MAGE.value, 2) (Req: FindDungeonTorch)

ExitRoomToCorridorLevel1

Skill Type: Navigation

Description: The agent learns to exit a dungeon room by moving away from corner torches into the connecting corridor.

Assigned Reward: 1

Success Condition: not exists_stationary_object(cur, ItemType.TORCH.value)

Starting Conditions:

- C1: exists_stationary_object(cur, ItemType.TORCH.value) (Req: FindDungeonTorch)

EnterAdjacentRoomLevel1

Skill Type: Navigation

Description: The agent navigates from the starting room (containing the upward ladder) on the dungeon level to an adjacent room using torch landmarks.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & exists_stationary_object(cur, ItemType.TORCH.value) & exists_ladder_object(cur, ItemType.LADDER_UP.value)

Starting Conditions:

- C1: (cur.player_level == 1) & exists_ladder_object(cur, ItemType.LADDER_UP.value) (Req: FindUpLadderLevel1)

EnterNewRoomFromCorridorLevel1

Skill Type: Navigation

Description: The agent navigates from the dungeon corridor (after exiting a room) on level 1 to enter an adjacent room by detecting torches.

Assigned Reward: 1

Success Condition: exists_stationary_object(cur, ItemType.TORCH.value)

Starting Conditions:

- C1: not exists_stationary_object(cur, ItemType.TORCH.value) (Req: ExitRoomToCorridorLevel1)

FindSnailLevel1

Skill Type: Navigation

Description: The agent explores the dungeon level (level 1) to locate a snail, a passive mob that spawns in dungeon rooms.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & is_near_passive_mob(cur, PassiveMobType.SNAIL.value, 2)

Starting Conditions:

- **C1:** cur.player_level == 1 (Req: DescendLadder)
- **C2:** exists_stationary_object(cur, ItemType.TORCH.value) (Req: FindDungeonTorch)

KillSnailLevel1

Skill Type: Fighting mobs

Description: The agent locates and defeats a snail, a passive mob that spawns in dungeon rooms on level 1, to obtain food resources.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & was_mob_killed(prev, cur, MobType.PASSIVE.value, PassiveMobType.SNAIL.value)

Starting Conditions:

- **C1:** (cur.player_level == 1) (Req: DescendLadder)
- **C2:** cur.inventory.sword >= 1 (Req: CraftWoodSword)
- **C3:** is_near_passive_mob(cur, PassiveMobType.SNAIL.value, 2) (Req: FindFountainLevel1)

EnterNewRoomExcludingUpLadderLevel1

Skill Type: Navigation

Description: The agent navigates from the corridor to enter an adjacent room that does not contain the upward ladder, confirming new room entry.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & exists_stationary_object(cur, ItemType.TORCH.value) & !exists_ladder_object(cur, ItemType.LADDER_UP.value)

Starting Conditions:

- **C1:** not exists_stationary_object(cur, ItemType.TORCH.value) (Req: ExitRoomToCorridorLevel1)

KillEightMobsLevel1

Skill Type: Fighting mobs

Description: The agent eliminates 8 hostile or passive mobs to satisfy the requirement for opening the downward ladder to level 2.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & (cur.monsters_killed[1] >= 8)

Starting Conditions:

- **C1:** cur.player_level == 1 (Req: DescendLadder)
- **C2:** cur.inventory.sword >= 1 (Req: CraftWoodSword)

FaceLadderDownLevel1

Skill Type: Navigation

Description: The agent positions itself adjacent to the downward ladder on the dungeon level and rotates to face the ladder block.

Assigned Reward: 1

Success Condition: (cur.player_level == 1) & (cur.item_map[cur.player_level, cur.player_position[0] + DIRECTIONS[cur.player_direction][0], cur.player_position[1] + DIRECTIONS[cur.player_direction][1]] == ItemType.LADDER_DOWN.value)

Starting Conditions:

- **C1:** (cur.player_level == 1) & exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True) (Req: FindLadderDownLevel1)

ApproachLadderDownLevel1

Skill Type: Navigation

Description: The agent navigates from proximity to the downward ladder to a position directly adjacent (1 block away).

Assigned Reward: 1

Success Condition: (cur.player.level == 1) & ((cur.item.map[1, cur.player.position[0]+1, cur.player.position[1]] == ItemType.LADDER_DOWN.value) | (cur.item.map[1, cur.player.position[0]-1, cur.player.position[1]] == ItemType.LADDER_DOWN.value) | ...)

Starting Conditions:

- **C1:** (cur.player.level == 1) & exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True) (Req: FindLadderDownLevel1)

DescendLadderLevel1

Skill Type: Navigation

Description: The agent executes the descend action while positioned on the downward ladder to transition to the Gnomish Mines (level 2).

Assigned Reward: 1

Success Condition: cur.player.level == 2

Starting Conditions:

- **C1:** (cur.player.level == 1) & (cur.monsters_killed[1] >= 8) (Req: KillEightMobsLevel1)
- **C2:** (cur.player.level == 1) & (cur.item.map[cur.player.level, cur.player.position[0], cur.player.position[1]] == ItemType.LADDER_DOWN.value) (Req: StepOntoLadderDownLevel1)

FindWaterLevel2

Skill Type: Navigation

Description: The agent navigates the Gnomish Mines level (level 2) to locate a water pool within a 2-block radius.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & is_near_block(cur, BlockType.WATER.value)

Starting Conditions:

- **C1:** (cur.player.level == 2) (Req: DescendLadderLevel1)

FindUpLadderLevel2

Skill Type: Navigation

Description: The agent explores the Gnomish Mines level (level 2) to locate the upward ladder (leading back to level 1).

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & exists_ladder_object(cur, ItemType.LADDER_UP.value)

Starting Conditions:

- **C1:** cur.player.level == 2 (Req: DescendLadderLevel1)

FindWaterWithTorchLevel2

Skill Type: Navigation

Description: The agent leverages placed torches to illuminate the dark Gnomish Mines and locates water pools within a 2-block radius.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & is_near_block(cur, BlockType.WATER.value)

Starting Conditions:

- **C1:** cur.player.level == 2 (Req: DescendLadderLevel1)
- **C2:** exists_stationary_object(cur, ItemType.TORCH.value) (Req: PlaceTorch)

MoveToEdgeOfLightLevel2

Skill Type: Navigation

Description: The agent moves from the spawn point on level 2 to a position where the light level drops below 0.5.

Assigned Reward: 1

Success Condition: `(cur.player.level == 2) & (cur.light_map[2, cur.player.position[0], cur.player.position[1]] < 0.5)`

Starting Conditions:

- **C1:** `cur.player.level == 2` (Req: DescendLadderLevel1)

FindStalagmiteLevel2

Skill Type: Navigation

Description: The agent explores the Gnomish Mines level (level 2) to locate stalagmites within a 2-block radius.

Assigned Reward: 1

Success Condition: `(cur.player.level == 2) & is_near_block(cur, BlockType.STALAGMITE.value)`

Starting Conditions:

- **C1:** `cur.player.level == 2` (Req: DescendLadderLevel1)

MineStalagmiteLevel2

Skill Type: Resource Gathering

Description: The agent mines stalagmites in the Gnomish Mines (level 2) to obtain stone resources.

Assigned Reward: 1

Success Condition: `cur.inventory.stone >= 1`

Starting Conditions:

- **C1:** `cur.player.level == 2` (Req: DescendLadderLevel1)
- **C2:** `exists_stationary_object(cur, ItemType.TORCH.value)` (Req: PlaceTorch)
- **C3:** `cur.inventory.pickaxe >= 1` (Req: CraftWoodPickaxe)

NavigateToWaterFromStalagmiteLevel2

Skill Type: Navigation

Description: The agent leverages stalagmites as environmental landmarks in the Gnomish Mines to navigate toward nearby water pools.

Assigned Reward: 1

Success Condition: `(cur.player.level == 2) & is_near_block(cur, BlockType.WATER.value)`

Starting Conditions:

- **C1:** `(cur.player.level == 2) & is_near_block(cur, BlockType.STALAGMITE.value)` (Req: FindStalagmiteLevel2)
- **C2:** `exists_stationary_object(cur, ItemType.TORCH.value)` (Req: PlaceTorch)

PlaceTorchAtEdgeLevel2

Skill Type: Navigation

Description: The agent strategically places a torch at the boundary of existing light (light level > 0.5) in the Gnomish Mines.

Assigned Reward: 1

Success Condition: `(cur.player.level == 2) & (cur.item_map[2, cur.player.position[0], cur.player.position[1]] == ItemType.TORCH.value)`

Starting Conditions:

- **C1:** `(cur.player.level == 2) & (cur.light_map[2, cur.player.position[0], cur.player.position[1]] < 0.5)` (Req: MoveToEdgeOfLightLevel2)
- **C2:** `cur.inventory.torches >= 1` (Req: CraftTorch)

MoveIntoLitAreaLevel2

Skill Type: Survivability

Description: The agent moves from the torch placement position at the edge of darkness into the newly illuminated area.

Assigned Reward: 1

Success Condition: `(cur.player_level == 2) & (cur.light_map[2, cur.player_position[0], cur.player_position[1]] >= 0.5)`

Starting Conditions:

- **C1:** `(cur.player_level == 2) & (cur.light_map[2, cur.player_position[0], cur.player_position[1]] < 0.5)` (Req: MoveToEdgeOfLightLevel2)
- **C2:** `(cur.player_level == 2) & exists_stationary_object(cur, ItemType.TORCH.value)` (Req: PlaceTorchAtEdgeLevel2)

MoveToNextEdgeOfLightLevel2

Skill Type: Navigation

Description: The agent moves from the center of a newly created lit area to the next edge of darkness, enabling progressive exploration.

Assigned Reward: 1

Success Condition: `(cur.player_level == 2) & (cur.light_map[2, cur.player_position[0], cur.player_position[1]] < 0.5)`

Starting Conditions:

- **C1:** `cur.player_level == 2` (Req: DescendLadderLevel1)
- **C2:** `(cur.player_level == 2) & (cur.light_map[2, cur.player_position[0], cur.player_position[1]] >= 0.5)` (Req: MoveIntoLitAreaLevel2)

MoveAwayFromUpLadderLevel2

Skill Type: Navigation

Description: The agent navigates away from the upward ladder spawn point in the Gnomish Mines to a position at least 5 blocks distant.

Assigned Reward: 1

Success Condition: `(cur.player_level == 2) & (jnp.abs(cur.player_position[0] - cur.up_ladders[2][0]) + jnp.abs(cur.player_position[1] - cur.up_ladders[2][1]) >= 5)`

Starting Conditions:

- **C1:** `(cur.player_level == 2) & exists_ladder_object(cur, ItemType.LADDER_UP.value)` (Req: FindUpLadderLevel2)

FindBatLevel2

Skill Type: Navigation

Description: The agent explores the Gnomish Mines level (level 2) by leveraging water pools (located via torches) to find bats, passive mobs that spawn near water sources.

Assigned Reward: 1

Success Condition: `(cur.player_level == 2) & is_near_passive_mob(cur, PassiveMobType.BAT.value, 2)`

Starting Conditions:

- **C1:** `cur.player_level == 2` (Req: DescendLadderLevel1)
- **C2:** `(cur.player_level == 2) & is_near_block(cur, BlockType.WATER.value)` (Req: FindWaterWithTorchLevel2)

MineCoalNearWaterLevel2

Skill Type: Resource Gathering

Description: The agent leverages water pools as reliable landmarks in the Gnomish Mines to mine coal deposits within stone-rich areas.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & (cur.inventory.coal >= 1)

Starting Conditions:

- **C1:** cur.inventory.pickaxe >= 1 (Req: CraftWoodPickaxe)
- **C2:** exists_stationary_object(cur, ItemType.TORCH.value) (Req: PlaceTorch)
- **C3:** (cur.player.level == 2) & is_near_block(cur, BlockType.WATER.value) (Req: NavigateToWaterFromStalagmiteLevel2)

KillGnomeWarriorNearUpLadderLevel2

Skill Type: Fighting mobs

Description: The agent remains within the naturally lit area around the upward ladder spawn point and eliminates a Gnome Warrior.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & was_mob_killed(prev, cur, MobType.MELEE.value, MeleeMobType.GNOME_WARRIOR.value)

Starting Conditions:

- **C1:** cur.player.level == 2 (Req: DescendLadderLevel1)
- **C2:** (cur.player.level == 2) & exists_ladder_object(cur, ItemType.LADDER_UP.value) (Req: FindUpLadderLevel2)
- **C3:** cur.inventory.sword >= 1 (Req: CraftWoodSword)

KillBatLevel2

Skill Type: Fighting mobs

Description: The agent locates and defeats a bat, a passive mob that spawns near water pools in the Gnomish Mines level (level 2), to obtain food resources.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & was_mob_killed(prev, cur, MobType.PASSIVE.value, PassiveMobType.BAT.value)

Starting Conditions:

- **C1:** cur.player.level == 2 (Req: DescendLadderLevel1)
- **C2:** cur.inventory.sword >= 1 (Req: CraftWoodSword)
- **C3:** exists_stationary_object(cur, ItemType.TORCH.value) (Req: PlaceTorch)
- **C4:** is_near_passive_mob(cur, PassiveMobType.BAT.value, 2) (Req: FindBatLevel2)

CheckLadderInLitAreaLevel2

Skill Type: Navigation

Description: After moving into a newly illuminated area in the Gnomish Mines, the agent systematically checks for the downward ladder.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True)

Starting Conditions:

- **C1:** (cur.player.level == 2) & (cur.light_map[2, cur.player.position[0], cur.player.position[1]] >= 0.5) (Req: MoveIntoLitAreaLevel2)

DrinkWaterLevel2

Skill Type: Survivability

Description: The agent locates a water pool in the Gnomish Mines level (level 2) and drinks from it to replenish thirst.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & (cur.player.drink >= (7 + 2 * cur.player.dexterity))

Starting Conditions:

- **C1:** (cur.player.level == 2) & is_near_block(cur, BlockType.WATER.value) (Req: NavigateToWaterFromStalagmiteLevel2)

ApproachWaterLevel2

Skill Type: Survivability

Description: The agent moves from proximity (within 2 blocks) to a water pool in the Gnomish Mines to an adjacent position (1 block away).

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & ((cur.map[2, cur.player.position[0]+1, cur.player.position[1]] == BlockType.WATER.value) | ...)

Starting Conditions:

- **C1:** (cur.player.level == 2) & is_near_block(cur, BlockType.WATER.value) (Req: NavigateToWaterFromStalagmiteLevel2)

CheckLadderAfterTorchLevel2

Skill Type: Navigation

Description: After placing a torch at the edge of darkness, the agent immediately checks the full 11x11 illuminated area for the downward ladder.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & exists_stationary_object(cur, ItemType.LADDER_DOWN.value)

Starting Conditions:

- **C1:** (cur.player.level == 2) & exists_stationary_object(cur, ItemType.TORCH.value) (Req: PlaceTorchAtEdgeLevel2)

ApproachLadderDownLevel2

Skill Type: Navigation

Description: The agent moves from proximity to the downward ladder on the Gnomish Mines level (level 2) to a position directly adjacent.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & ((cur.item.map[2, cur.player.position[0]+1, cur.player.position[1]] == ItemType.LADDER_DOWN.value) | ...)

Starting Conditions:

- **C1:** (cur.player.level == 2) & exists_local_object(cur, ItemType.LADDER_DOWN.value, radius=1, check_items=True) (Req: CheckLadderInLitAreaLevel2)

CheckLadderInFullLitAreaLevel2

Skill Type: Navigation

Description: After moving into a newly illuminated area, the agent performs a comprehensive 10-block radius scan to detect the downward ladder.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & exists_stationary_object(cur, ItemType.LADDER_DOWN.value)

Starting Conditions:

- **C1:** cur.player.level == 2 (Req: DescendLadderLevel1)
- **C2:** (cur.player.level == 2) & (cur.light_map[2, cur.player.position[0], cur.player.position[1]] >= 0.5) (Req: MoveIntoLitAreaLevel2)

KillEightMobsLevel2

Skill Type: Fighting mobs

Description: The agent explores the Gnomish Mines level (level 2) while maintaining position in lit areas to eliminate 8 hostile or passive mobs.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & (cur.monsters_killed[2] >= 8)

Starting Conditions:

- **C1:** cur.player.level == 2 (Req: DescendLadderLevel1)
- **C2:** cur.inventory.sword >= 1 (Req: CraftWoodSword)
- **C3:** (cur.player.level == 2) & (cur.light_map[2, cur.player.position[0], cur.player.position[1]] >= 0.5) (Req: MoveIntoLitAreaLevel2)

MineCoalLevel2

Skill Type: Resource Gathering

Description: The agent locates and mines coal deposits in the Gnomish Mines (level 2) using a wooden pickaxe to obtain coal resources.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & (cur.inventory.coal > prev.inventory.coal)

Starting Conditions:

- **C1:** cur.player.level == 2 (Req: DescendLadderLevel1)
- **C2:** cur.inventory.pickaxe >= 1 (Req: CraftWoodPickaxe)
- **C3:** (cur.player.level == 2) & is_near_block(cur, BlockType.COAL.value) (Req: FindStalagmiteLevel2)

FaceLadderDownLevel2

Skill Type: Navigation

Description: The agent positions itself adjacent to the downward ladder on the Gnomish Mines level (level 2) and rotates to face the ladder block.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & (cur.item_map[2, cur.player.position[0] + DIRECTIONS[cur.player.direction][0], cur.player.position[1] + DIRECTIONS[cur.player.direction][1]] == ItemType.LADDER_DOWN.value)

Starting Conditions:

- **C1:** (cur.player.level == 2) & ((cur.item_map[2, cur.player.position[0]+1, cur.player.position[1]] == ItemType.LADDER_DOWN.value) | ...) (Req: ApproachLadderDownLevel2)

FaceWaterLevel2

Skill Type: Survivability

Description: The agent, when adjacent to a water pool in the Gnomish Mines level (level 2), rotates to face the water block.

Assigned Reward: 1

Success Condition: (cur.player.level == 2) & (cur.map[2, cur.player.position[0] + DIRECTIONS[cur.player.direction][0], cur.player.position[1] + DIRECTIONS[cur.player.direction][1]] == BlockType.WATER.value)

Starting Conditions:

- **C1:** (cur.player.level == 2) & ((cur.map[2, cur.player.position[0]+1, cur.player.position[1]] == BlockType.WATER.value) | ...) (Req: ApproachWaterLevel2)

ApproachUpLadderLevel2

Skill Type: Navigation

Description: The agent moves from proximity to the upward ladder on the Gnomish Mines level (level 2) to a position directly adjacent.

Assigned Reward: 1

Success Condition: (cur.player_level == 2) & ((cur.item_map[2, cur.player_position[0]+1, cur.player_position[1]] == ItemType.LADDER_UP.value) | ...)

Starting Conditions:

- **C1:** (cur.player_level == 2) & exists_ladder_object(cur, ItemType.LADDER_UP.value) (Req: FindUpLadderLevel2)

MoveForwardAfterTorchLevel2

Skill Type: Navigation

Description: After placing a torch at the edge of darkness, the agent moves forward one block into the newly illuminated area.

Assigned Reward: 1

Success Condition: (cur.player_level == 2) & (cur.light_map[2, cur.player_position[0], cur.player_position[1]] >= 0.5)

Starting Conditions:

- **C1:** (cur.player_level == 2) & (cur.light_map[2, cur.player_position[0], cur.player_position[1]] < 0.5) (Req: PlaceTorchAtEdgeLevel2)

RecoverManaSafe

Skill Type: Survivability

Description: The agent ensures survival intrinsics are fully replenished via SleepToFullEnergy, then actively verifies and eliminates threats before recovering mana.

Assigned Reward: 1

Success Condition: (cur.player_level == 0) & (cur.player_mana >= (6 + cur.player_intelligence))

Starting Conditions:

- **C1:** (cur.player_level == 0) & (cur.player_energy >= (7 + 2 * cur.player_dexterity)) (Req: SleepToFullEnergy)
- **C2:** not is_near_melee_mob(cur, MeleeMobType.ZOMBIE.value, 3) (Req: KillZombie)
- **C3:** not is_near_ranged_mob(cur, RangedMobType.SKELETON.value, 3) (Req: KillSkeleton)

StepOntoUpLadderLevel2

Skill Type: Navigation

Description: The agent moves from an adjacent position to the upward ladder on the Gnomish Mines level (level 2) to stand exactly on the ladder block.

Assigned Reward: 1

Success Condition: (cur.player_level == 2) & (cur.item_map[2, cur.player_position[0], cur.player_position[1]] == ItemType.LADDER_UP.value)

Starting Conditions:

- **C1:** (cur.player_level == 2) & ((cur.item_map[2, cur.player_position[0]+1, cur.player_position[1]] == ItemType.LADDER_UP.value) | ...) (Req: ApproachUpLadderLevel2)

FaceUpLadderLevel2

Skill Type: Navigation

Description: The agent positions itself adjacent to the upward ladder on the Gnomish Mines level (level 2) and rotates to face the ladder block.

Assigned Reward: 1

Success Condition: `(cur.player_level == 2) & (cur.item_map[2, cur.player_position[0] + DIRECTIONS[cur.player_direction][0], cur.player_position[1] + DIRECTIONS[cur.player_direction][1]] == ItemType.LADDER_UP.value)`

Starting Conditions:

- **C1:** `(cur.player_level == 2) & ((cur.item_map[2, cur.player_position[0]+1, cur.player_position[1]] == ItemType.LADDER_UP.value) | ...)` (Req: ApproachUpLadderLevel2)

AscendLadderLevel2

Skill Type: Navigation

Description: The agent executes the ascend action while positioned on the upward ladder in the Gnomish Mines to transition back to the dungeon level.

Assigned Reward: 1

Success Condition: `cur.player_level == 1`

Starting Conditions:

- **C1:** `(cur.player_level == 2) & (cur.item_map[2, cur.player_position[0], cur.player_position[1]] == ItemType.LADDER_UP.value)` (Req: StepOntoUpLadderLevel2)

ApproachFountainLevel1

Skill Type: Navigation

Description: The agent moves from proximity (within 2 blocks) to a fountain in the dungeon level to an adjacent position (1 block away).

Assigned Reward: 1

Success Condition: `(cur.player_level == 1) & ((cur.map[1, cur.player_position[0]+1, cur.player_position[1]] == BlockType.FOUNTAIN.value) | ...)`

Starting Conditions:

- **C1:** `(cur.player_level == 1) & is_near_block(cur, BlockType.FOUNTAIN.value)` (Req: FindFountainLevel1)

StepOntoLadderDownLevel2

Skill Type: Navigation

Description: The agent moves forward from an adjacent position while facing the downward ladder on the Gnomish Mines level to stand exactly on the ladder block.

Assigned Reward: 1

Success Condition: `(cur.player_level == 2) & (cur.item_map[2, cur.player_position[0], cur.player_position[1]] == ItemType.LADDER_DOWN.value)`

Starting Conditions:

- **C1:** `(cur.player_level == 2) & (cur.item_map[2, cur.player_position[0] + DIRECTIONS[cur.player_direction][0], cur.player_position[1] + DIRECTIONS[cur.player_direction][1]] == ItemType.LADDER_DOWN.value)` (Req: FaceLadderDownLevel2)

DrinkWaterLevel2Improved

Skill Type: Survivability

Description: The agent, after precisely approaching and correctly facing a water pool in the Gnomish Mines, drinks from it to replenish thirst.

Assigned Reward: 1

Success Condition: (cur.player_level == 2) & (cur.player_drink >= (7 + 2 * cur.player_dexterity))

Starting Conditions:

- **C1:** (cur.player_level == 2) & ((cur.map[2, cur.player_position[0]+1, cur.player_position[1]] == BlockType.WATER.value) | ...) (Req: ApproachWaterLevel2)
- **C2:** (cur.player_level == 2) & (cur.map[2, cur.player_position[0] + DIRECTIONS[cur.player_direction][0], cur.player_position[1] + DIRECTIONS[cur.player_direction][1]] == BlockType.WATER.value) (Req: FaceWaterLevel2)

MoveAwayFromWaterLevel2

Skill Type: Navigation

Description: The agent moves from an adjacent position to a water pool to a location where it is no longer adjacent, enabling exploration of surrounding areas.

Assigned Reward: 1

Success Condition: (cur.player_level == 2) & not ((cur.map[2, cur.player_position[0]+1, cur.player_position[1]] == BlockType.WATER.value) | ...)

Starting Conditions:

- **C1:** (cur.player_level == 2) & ((cur.map[2, cur.player_position[0]+1, cur.player_position[1]] == BlockType.WATER.value) | ...) (Req: ApproachWaterLevel2)

KillGnomeWarriorInLitAreaLevel2

Skill Type: Fighting mobs

Description: The agent, while positioned in a fully illuminated area on the Gnomish Mines level, locates and eliminates a Gnome Warrior.

Assigned Reward: 1

Success Condition: (cur.player_level == 2) & was_mob_killed(prev, cur, MobType.MELEE.value, MeleeMobType.GNOME_WARRIOR.value)

Starting Conditions:

- **C1:** cur.player_level == 2 (Req: DescendLadderLevel1)
- **C2:** cur.inventory.sword >= 1 (Req: CraftWoodSword)
- **C3:** (cur.player_level == 2) & (cur.light_map[2, cur.player_position[0], cur.player_position[1]] >= 0.5) (Req: MoveIntoLitAreaLevel2)
- **C4:** is_near_melee_mob(cur, MeleeMobType.GNOME_WARRIOR.value, 2) (Req: FindStalagmiteLevel2)

KillGnomeArcherWithSwordLevel2

Skill Type: Fighting mobs

Description: The agent leverages its sword and light management to locate and defeat a Gnome Archer in melee range within illuminated areas.

Assigned Reward: 1

Success Condition: (cur.player_level == 2) & was_mob_killed(prev, cur, MobType.RANGED.value, RangedMobType.GNOME_ARCHER.value)

Starting Conditions:

- **C1:** cur.player_level == 2 (Req: DescendLadderLevel1)
- **C2:** cur.inventory.sword >= 1 (Req: CraftWoodSword)
- **C3:** (cur.player_level == 2) & (cur.light_map[2, cur.player_position[0], cur.player_position[1]] >= 0.5) (Req: MoveIntoLitAreaLevel2)
- **C4:** is_near_ranged_mob(cur, RangedMobType.GNOME_ARCHER.value, 2) (Req: FindStalagmiteLevel2)

DescendLadderLevel2

Skill Type: Navigation

Description: The agent executes the descend action while positioned on the downward ladder in the Gnomish Mines level to transition to the Sewers (level 3).

Assigned Reward: 1

Success Condition: `cur.player.level == 3`

Starting Conditions:

- **C1:** `cur.player.level == 2` (Req: DescendLadderLevel1)
- **C2:** `(cur.player.level == 2) & (cur.monsters_killed[2] >= 8)` (Req: KillEightMobsLevel2)
- **C3:** `(cur.player.level == 2) & (cur.item_map[2, cur.player_position[0], cur.player_position[1]] == ItemType.LADDER_DOWN.value)` (Req: StepOntoLadderDownLevel2)