Autotelic LLM-based exploration for goal-conditioned RL

Guillaume Pourcel
Inria (Flowers)
University of Bordeaux, France
guillaume.pourcel@inria.fr

Grgur Kovač
Inria (Flowers)
University of Bordeaux, France

Thomas Carta
Inria (Flowers)
University of Bordeaux, France

Pierre-Yves Oudeyer
Inria (Flowers)
University of Bordeaux, France

Abstract

Designing autotelic agents capable of autonomously generating and pursuing their own goals represents a promising endeavor for open-ended learning and skill acquisition in reinforcement learning. This challenge is especially difficult in open worlds that require inventing new previously unobserved goals. In this work, we propose an architecture where a single generalist autotelic agent is trained on an automatic curriculum of goals. We leverage large language models (LLMs) to generate goals as code for reward functions based on learnability and difficulty estimates. The goal-conditioned RL agent is trained on those goals sampled based on learning progress. We compare our method to an adaptation of OMNI-EPIC to goal-conditioned RL. Our preliminary experiments imply that our method generates a higher proportion of learnable goals, suggesting better adaptation to the goal-conditioned learner.

1 Introduction

Reinforcement learning (Sutton and Barto [2018]) has successfully allowed agents to master complex tasks (Mnih [2013]). However, current agents are still unable to autonomously explore and develop skills in open-ended environments. It has been proposed that achieving such autonomy requires agents to explore by continuously setting and pursuing their own goals (Oudeyer and Kaplan [2007]), e.g. autotelic curiosity-driven agents (Colas et al. [2022]). Various intrinsic rewards have been proposed to drive goal-based exploration such as diversity (Eysenbach et al. [2018]), intermediate-difficulty (Florensa et al. [2018]), and, most relevant for this work, learning progress (Kaplan and Oudeyer [2003], Schmidhuber [1991]). Several works explored autotelic agents that generate novel goals, i.e. goals previously unobserved in the environment. For a goal-conditioned RL agent, MUGL (Laversanne-Finot et al. [2018]) samples goals from a disentangled VAE latent space, and IMAGINE (Colas et al. [2020]) leverages the compositionality of language to generate new linguistic goals. Recently, LLMs have been applied to generate novel goals to explore textworlds (Colas et al. [2023]) and minecraft (Wang et al. [2023]) for an LLM-based learner. A parallel line of work focused on generating curricula in the form of environments (Portelas et al. [2020a]) with the aim to generate environments in the suitable order for an RL learner. (Portelas et al. [2020b]) compare different methods for adapting the curriculum to the agents' morphologies, e.g. generating water environments for swimmers and ground environments for walkers. Recently, LLMs have been deployed to generate environments, LLM-POET (Aki et al. [2024]) uses LLMs to generate environments for bipedal

Accepted at the Intrinsically Motivated Open-ended Learning workshop at NeurIPS 2024.

walker agents. OMNI-EPIC (Faldor et al. [2024]) uses an LLM in tandem with a vision language model to generate both reward functions and environments as Python code.

The motivation of this work is an autotelic agent leveraging an LLM to generate goals for a goal-conditioned RL learner. The most similar related work is OMNI-EPIC (Faldor et al. [2024]), with a key conceptual distinction. In OMNI-EPIC, for each new environment, the RL agent corresponding to the closest environment is cloned and fine-tuned, i.e. they create a lookup table of specialized agents as opposed to one generalist goal-conditioned agent. Our preliminary results imply that simply adapting the OMNI-EPIC goal generation to a setup with a single goal-conditioned learner results in many unlearnable goals. To overcome this, we propose a novel method for goal generation and selection, which leverages difficulty and learnability estimates to generate goals better adapted to the current goal-conditioned RL agent.

In this work, we present an autotelic agent (see Figure 1), to autonomously explore a 2D minecraft-like environment Craftax (Matthews et al. [2024]). It has two modules: a goal generator, and a goal-conditioned learner consisting of goal selection and a goal-conditioned RL-agent, which communicate through a goal archive. Inside the goal-conditioned learner, a goal-conditioned RL agent is trained on goals sampled from the archive based on learning progress (LP). Then, the difficulty (D) and learnability (L) of those goals are estimated. Inside the goal generator, goals are sampled based on learnability (L). Those goals, are used as in-context examples for the LLM-based goal generation alongside a numerical estimate of their difficulty and learnability estimates. We compare the learnability of goals generated by our method to that generated by a OMNI-EPIC-like baseline (an adaptation of OMNI-EPIC to a goal-conditioned agent). Our preliminary experiments imply that our method is able to leverage the learnability estimates to generate a higher percentage of learnable goals.

The main contributions of this work are: 1) a first study of LLM-goal generation for a goal-conditioned (autotelic) RL agent 2) a method leveraging learnability and difficulty to improve goal generation 3) a comparison with a naive baseline (an adaptation of OMNI-EPIC to a goal-conditioned RL setting)

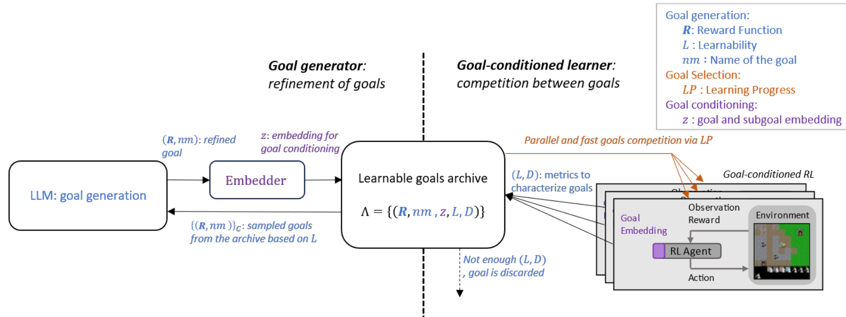

Figure 1: The autotelic agent architecture consists of two main modules: the goal generator and the goal-conditioned learner, along with a goal archive. In the goal generator, goals are sampled based on their learnability. These goals, their learnability (L) and difficulty (D) estimates, are given to LLM to generate new goals: reward functions (R) and names (nm). Names are embedded for goal conditioning of the RL agent. In the goal-conditioned learner, goals for training the RL agent are sampled based on Learning Progress (LP), and their learnability (L) and difficulty (D) are estimated.

2 Method: Open-ended generation of goals in code

In our setting, a goal g is a tuple $g = (nm, \mathbf{R})$, where nm is the name of the goal and $\mathbf{R}$ is the associated reward function. We designed an autotelic agent that generates new goals based on the difficulty of previous goals it has successfully completed. The agent operates through two distinct modules (Fig.1) to facilitate the open-ended generation and learning of goals. In the first module, an LLM adaptively generates and evaluates goals based on the agent's current skill level, using a combination of learnability and difficulty measures. All tasks are stored in an archive. In the second module, a multi-goal DeepRL agent learns the tasks from the archive.

The process is initialized by adding a set of hand-crafted goals to the archive. The manually created goals are very simple and allow the LLM to learn how to generate syntactically correct reward functions. The multi-goal agent is then trained on these goals for a few updates, involving rollouts and weight updates. Then, based on the success of the agent in learning the different tasks, the archive is updated: tasks that are too difficult or too easy to learn are removed from the archive. This iterative process allows the model to continuously generate and adapt goals that align with the agent's evolving capabilities, fostering an open-ended learning environment.The pseudo-algorithm for the process described above is provided in Appendix A.1.

2.1 The goal conditioned learner

While any RL algorithm could be used, we train the goal-conditioned agent using Proximal Policy Optimization (PPO). Appendix B.1 details the hyperparameters used for this training. The agent is a causal transformer (Dai et al. [2019]), conditioned on both textual and visual information. The textual information is the embedding of the nm, $z \in \mathbb{R}^{n_e}$, with n_e being the dimensionality of the embedding space. The visual information consists of the last n_{obs} images returned by the environment.

To determine which goals should be kept in the archive, they are evaluated based on two metrics: difficulty D which corresponds to the learner's success rate, and learnability L, which is the difference between the maximum and minimum success rates during the learning stage. After each training session, all the goals in the archive are ranked, and the n_{best} are kept. To class the goals we use a fitness function $f(g) = L_g \times D_g$, where L_g is the learnability of g and D_g its difficulty. During training, the learner's progress is also measured for each goal (see Appendix A.3).

2.2 The goal generator

We model the goal generator using an LLM, which is prompted with in-context examples of goals and their associated learning progress (*LP*) and difficulty (*D*) . These examples inform the LLM about the capacities of the goal-conditioned agent. The procedure for selecting the in-context examples is described in Appendix A.2 and a full prompt is given in Appendix D. The goal generator proposes new goals or modifies existing ones to maximize the estimated *LP* and *D* of the tasks (the tasks, while difficult, must be learnable). The LP metric helps select learnable tasks from a larger pool of non-learnable ones, while the difficulty metric filters out tasks that are too simple.

After the generation, a procedure (see Appendix A.5) removes goals g whose reward functions $\mathbf{R}$ are not syntactically correct (that do not compile). $\mathbf{R}$ can use privileged information such as the current state of the game engine, the agent's current action, and a reward state where memory can be stored. This allows the agent to target challenging goals such as time-extended goals (e.g., "move up three times") and goals involving optimization under selected constraints (e.g., "build a shelter while maintaining your health above 5"). Examples of such reward code are provided in Appendix C.

3 Experiments

We conduct our preliminary experiments on a 2D Minecraft-like environment, Crafter (Hafner [2021]) with a fast JAX implementation (Matthews et al. [2024]). The environment contains various materials (e.g. wood, stone) and mobs (creatures such as zombie, skeleton, cow, etc.). The observation is a 8x8 semantic grid, and the action space contains actions for movement, attacking, and crafting. Crafting enables the building of additional objects (e.g. table pickaxe). We explore the following two questions: 1) Does our method adapt to the learner, i.e. generate learnable goals? 2) What kind of goals are discovered, and are their implementations semantically plausible?

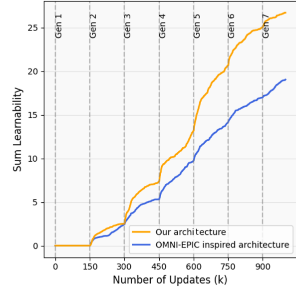

Figure 2: Cumulative learnability of all goals generated by our method compare to the OMNI-EPIC-like baseline.

3.1 Does our method adapt to the learner better, i.e. generates more learnable goals, than the OMNI-EPIC-like baseline?

In this experiment, we compare our method to a baseline inspired by OMNI-EPIC. The baseline differs from our method in the goal-generator by filtering goals for in-context examples based on

difficulty (see section 2.2 and appendix A.2), and no additional metrics are included in the prompt. We compare methods based on the generated goals' learnability. As learnability is low for goals that are trivially easy, impossible, or too hard, it serves as a good estimate of the adaptability of the goal generator to the current capabilities of the learner.

Figure 2 compares the cumulative learnability over all the goals in the archive at a given time by our method to that generated by the OMNI-EPIC-like baseline. We observe that our method generated more learnable goals, implying better adaptation to the learner. However, since these results are based on a single seed, further experiments are needed for stronger conclusions.

3.2 What kind of goals are discovered, are their implementations semantically plausible?

In this section, we present examples of reward functions (names and implementations) discovered during our preliminary experiments. We aim to build intuition of how the exploration evolves, and the limitations of our approach. We observe that simple tasks that align well with the original game code are generated easily with correct semantics, such as one that creates a sequence of subgoals culminating in using a wooden sword (Fig.3d). Furthermore, we observe implications that our architecture can generate increasingly complex goals. The agent first generates a simple shelter reward function based solely on checking whether there are non-grass blocks in its perimeter (Fig.3a). Then, another reward function is created checking for the presence of stones in the perimeter (Fig.3b), and finally, a further reward function checks for a "mob trap", which should correspond to a shelter around another character - a mob (Fig.3c). However, we note that the LLM fails to adequately match the semantics of those more abstract goals: the shelters created are not enclosed, allowing a mob to always attack the agent, and for a "mob trap" it merely checks if there is a creature close to the agent and >5 non-grass objects. Similarly, we observe other goals that represent interesting directions for exploration, even though some of them might be implemented in semantically implausible ways. Some of those interesting goals include checking for resources inside the shelter, damage from mobs over multiple timesteps, surviving in the proximity of lava, moving in certain patterns and building mazes.

In future work, we hope that the LLM would adjust those reward functions to make them more semantically plausible or more interesting. Given that building a valid shelter, or trapping a character would be harder, this would increase their learnability (as the starting performance would be low), thereby making them more likely to be selected.

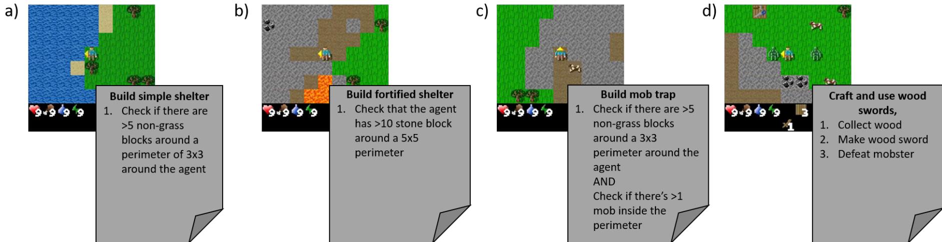

Figure 3: Examples of generated reward functions, pseudocode, and visualizations of the observations at the task completion state. Complexity increases from a)-c), but implementations semantically limited (shelters are not enclosed). In d), a more standard (less abstract) minecraft-like goal is implemented well, despite being multi-step.

4 Conclusion and future work

We present an autotelic agent that leverages an LLM to generate novel goals (names and reward function as code) tailored to a goal-conditioned RL agent. Our approach uses learnability estimates to filter in-context examples, which are given alongside their difficulty and learnability scores. We evaluate the adaptation of the goal generation to the RL learner with the cumulative learnability of generated goals. Our preliminary experiments, in which we compare with a baseline inspired by OMNI-EPIC, suggest that our method generates more learnable goals and adapts better to the learner. We observe a gradual increase in the goal complexity, but also note that a common issue is that implementations often do not semantically correspond to the goal name. In future refinements of our method, we hope that it could lead to more interesting behavior through better usage of the LLM's background common-sense and the iterative selection of more learnable goals.

References

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. Bradford Books, Cambridge, Massachusetts London, England, 2nd edition edition, November 2018. ISBN 978-0-262-03924-6.

Volodymyr Mnih. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? a typology of computational approaches. *Frontiers in neurorobotics*, 1:108, 2007.

Cédric Colas, Tristan Karch, Olivier Sigaud, and Pierre-Yves Oudeyer. Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: a short survey. *Journal of Artificial Intelligence Research*, 74:1159–1199, 2022.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *International conference on machine learning*, pages 1515–1528. PMLR, 2018.

Frédéric Kaplan and Pierre-Yves Oudeyer. Maximizing learning progress: an internal reward system for development. In *Embodied Artificial Intelligence: International Seminar, Dagstuhl Castle, Germany, July 7-11, 2003. Revised Papers*, pages 259–270, 2003.

J. Schmidhuber. Curious model-building control systems. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, pages 1458–1463 vol.2, 1991. doi: 10.1109/IJCNN.1991.170605.

Adrien Laversanne-Finot, Alexandre Pere, and Pierre-Yves Oudeyer. Curiosity driven exploration of learned disentangled goal spaces. In *Conference on Robot Learning*, pages 487–504. PMLR, 2018.

Cédric Colas, Tristan Karch, Nicolas Lair, Jean-Michel Dussoux, Clément Moulin-Frier, Peter Dominey, and Pierre-Yves Oudeyer. Language as a cognitive tool to imagine goals in curiosity driven exploration. *Advances in Neural Information Processing Systems*, 33:3761–3774, 2020.

Cédric Colas, Laetitia Teodorescu, Pierre-Yves Oudeyer, Xingdi Yuan, and Marc-Alexandre Côté. Augmenting autotelic agents with large language models. In Sarath Chandar, Razvan Pascanu, Hanie Sedghi, and Doina Precup, editors, *Proceedings of The 2nd Conference on Lifelong Learning Agents*, volume 232 of *Proceedings of Machine Learning Research*, pages 205–226. PMLR, 22–25 Aug 2023. URL `https://proceedings.mlr.press/v232/colas23a.html`.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic curriculum learning for deep rl: A short survey. *arXiv preprint arXiv:2003.04664*, 2020a.

Rémy Portelas, Cédric Colas, Katja Hofmann, and Pierre-Yves Oudeyer. Teacher algorithms for curriculum learning of deep rl in continuously parameterized environments. In *Conference on Robot Learning*, pages 835–853. PMLR, 2020b.

Fuma Aki, Riku Ikeda, Takumi Saito, Ciaran Regan, and Mizuki Oka. Llm-poet: Evolving complex environments using large language models. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 243–246, 2024.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. OMNI-EPIC: Open-endedness via Models of human Notions of Interestingness with Environments Programmed in Code, May 2024. URL `http://arxiv.org/abs/2405.15568`. arXiv:2405.15568 [cs].

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A Lightning-Fast Benchmark for Open-Ended Reinforcement Learning, February 2024. URL http://arxiv.org/abs/2402.16801. arXiv:2402.16801 [cs].

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context, June 2019. URL http://arxiv.org/abs/1901.02860. arXiv:1901.02860 [cs, stat].

Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. Omni: Open-endedness via models of human notions of interestingness. *arXiv preprint arXiv:2306.01711*, 2023.

Gautier Hamon. transformerXL_PPO_JAX, July 2024. URL https://inria.hal.science/hal-04659863.

A Pseudo-algorithm and equation

A.1 Pseudo-algorithm

Algorithm 1 Autotelic open-ended goal generation

Input: Goal archive $\Lambda = \{(nm, z, \mathbf{R}, L_{ini}, D_{ini})\}$ with hand-made goals
Initialize: Initialize policy $\pi(\phi|(\cdot))$
for $generation = 1, 2, \ldots, G$ **do**
  for $update : k = 1, 2, \ldots, K$ **do**
    Sample a batch S from the goal selection distribution, $\{(z, \mathbf{R})\}_S \sim \mathcal{D}_\Lambda$
    Collect p steps of policy rollout $\{\tau\}_S$ per sampled goals $\{(z, \mathbf{R})\}_S$
    Evaluate the goal-specific success rate $\{SR_k\}$
    Update the goal-specific Learning Progress $\{LP_k\}$
    Update the goal-specific Learnability and Difficulty $\{(L_k, D_k)\}_S$
    Update the goal-condition policy $\pi(\phi|(\cdot))$ with $\{\mathbf{R}(\tau)\}_S$ and $\{z\}_S$
  end for
  Update the Archive Λ with the goal learnability and difficulty $\{L_K, D_K\}$
  Update the archive by keeping the N-fittest goals $\Lambda \leftarrow \{\lambda \in \Lambda | rank(\lambda, L_\lambda \times D_\lambda) \leq N\}$
  for $i = 1, 2, \ldots, I$ **do**
    Sample a goals from the archive based on the filtering distribution $((nm, \mathbf{R}))_C \sim \mathcal{F}_\Lambda$
    Generate a new goal from the sampled goals $(nm, \mathbf{R}) \sim LLM((nm, \mathbf{R})_C, G_{prompt})$
    Create Embedding $z_g = \big(L_e(nm_g), L_e(nm_{s_g}(1)), L_e(nm_{s_g}(2)), \ldots)\big)$
    Test validity of the goal generated (syntax+compilation)
    if valid goal **then**
      Add the goal to the archive $\Lambda \leftarrow \Lambda \cup (nm, z, \mathbf{R}, L_{ini}, D_{ini})$
    end if
  end for
end for

The goal selection distribution $\mathcal{D}_\Lambda$ is based on the learning progress (LP). Both the distribution and the calculation of LP are conducted in the same manner as described in the OMNI Zhang et al. [2023] architecture with the same hyperparameters.

A.2 $\mathcal{F}_\Lambda$ the filter function

The Filter distribution $\mathcal{F}_\Lambda$ is defined by the following procedure:

1. Define the sets of learnable and not-learnable rewards:
   $R_L = \{r \in \Lambda \mid L(r) > 0.1\}, \quad R_N = \{r \in \Lambda \mid L(r) \leq 0.1\}.$

2. Sample $g \sim \text{Uniform}(R_L)$.

3. Select the two closest learnable rewards to g in embedding space (excluding g) :
   $r_{L1},\ r_{L2} = \arg\min_{r \in R_L \setminus \{g\}} \text{distance}(\text{embedding}(r),\ \text{embedding}(g)).$

4. Select the two closest not-learnable rewards to g in embedding space:
   $r_{N1},\ r_{N2} = \arg\min_{r \in R_N} \text{distance}(\text{embedding}(r),\ \text{embedding}(g)).$

5. Sample $g'_1,\ g'_2 \sim \text{Uniform}(R_L)$.This two goals are sampled for creative combinations with dissimilar goals.

Then, we obtain the sample $(g,\ r_{L1},\ r_{L2},\ r_{N1},\ r_{N2},\ g'_1,\ g'_2) \sim \mathcal{F}_\Lambda$.

In the OMNI-EPIC-like baseline the Filtering distribution is similar except that the two sets R_L and R_N are replaced by:

$$R_D\{r \in \Lambda \mid D(r) > 0.1\}, \quad R_N = \{r \in \Lambda \mid LP(r) \leq 0.1\} \tag{1}$$

A.3 D, L and LP

The different metrics are all based on the success rate defined by computing:

$$SR(k_{\text{tot}}) = \frac{\sum \text{(times the goal was achieved in } \{\tau(k_{\text{tot}})\}_S)}{\sum \text{(times the goal was not achieved in } \{\tau(k_{\text{tot}})\}_S)} \tag{2}$$

which is a then exponentially smoothed with a constant of 0.1 leading to $SR_{smooth}(k_{tot})$. Also k_{tot} refers to the index of update since the begining of the simulation, not only the index of the update in the current generation like in the pseudo-code.

The difficulty is defined by:

$$D(k_{tot}) = SR_{smooth}(k_{tot}) \tag{3}$$

The learnability is defined by:

$$L(k_{tot}) = \max_{l_{tot} \in [0, k_{tot}]} (SR_{smooth}(l_{tot}) - \min_{l_{tot} \in [0, k_{tot}]} (SR_{smooth}(l_{tot}) \tag{4}$$

The learning progress is defined in the same manner as described in the OMNI architecture Zhang et al. [2023].

A.4 Goal and subgoals embedding

When generating a goal, the language model (LLM) is tasked with creating a name, denoted as nm, which includes both the name of the goal (nm_g) and the names of its subgoals ($(nm_{s_g}(1), nm_{s_g}(2), \ldots)$). These names are then embedded into a vector of size 512 using the OpenAI text-embedding-3-small embedder.

This process results in the following embedding:

$$z_g(i) = \left(L_e(nm_g), L_e(name_{s_g}(i))\right)$$

Here, i corresponds to the index of the current subgoal which is determined by the reward state, which changes a counter to indicate the current subgoal. The term nm_g represents the name of the goal, and nm_{s_g} represents the name of the subgoal.

A.5 Filtering valid goals

Before adding a goal generated by the LLM to the archive we are testing its validity by:

- Checking if it's syntactically correct by executing it's code
- Checking if the name of the reward state, the goal and subgoal names and the reward function is consistent.
- Checking if the reward state and the reward function is jit compatible by trying to compile and execute on a random input.

B Hyperparameters

B.1 PPO AGENT

We re-used the hyperparameters of the JAX implementation of PPO with the transformerXL architecture Hamon [2024]

Hyperparameter	Value
Learning Rate	2e-4
Batch Size	8
Number of Epochs	4
Clip Range	0.2
Discount Factor (Gamma)	0.999
GAE Lambda	0.8
Entropy Coefficient	2e-3
Value Function Coefficient	0.5
Max Gradient Norm	1.
Number of Layers	2
Number of Heads	8
Size hidden layers	256
QKV features	256
Window memory	128
Window gradient	64
Size embedding	256
max time step per rollout	100

Table 1: PPO transformerXL Hyperparameters

B.2 Autotelic architecture hyperparameters

Hyperparameter	Value
Update per generation K	150
Samples goals for training $\#S$	1024
Number of steps per rollout p	128
Number of initial goal seed	33
Max size of the archive N	200

Table 2: Autotelic architecture hyperparameters

Overall the autotelic agent is tested on $128 \times 1024 \times 7 \times 150 = 16800000$ steps

C Example of reward functions generated by the LLM

We present here the code of the reward function displayed in Fig.3.

The structure is the following:

- the reward state is defined with a class

- the reward function is a function
- the names are stored in a list

```
[<name goal>, <name sugboal 1>, <name subgoal 2>, ...]
```

Goal Craft and use wood swords:

```
@struct.dataclass
class RewardState_craft_and_use_wood_sword:
    counter_prompt: int = 0

def craft_and_use_wood_sword(state, action, rng, reward_state):
    """
    Check the sequenced goal that includes the sequence of
        subgoals: collect_wood, make_wood_sword, and
        defeat_monster.
    """

    # first subtask: collect wood
    subtask_1_achieved = state.inventory.wood >= 2
    counter_updated = jax.lax.select(jnp.logical_and(
        subtask_1_achieved, reward_state.counter_prompt == 0), 1,
        reward_state.counter_prompt)

    # second subtask: make wood sword
    subtask_2_achieved = state.inventory.wood_sword == 1
    counter_updated = jax.lax.select(jnp.logical_and(
        subtask_2_achieved, reward_state.counter_prompt == 1), 2,
        counter_updated)

    # third subtask: defeat monster
    monster_defeated = jnp.any(state.zombies.health <= 0)
    counter_updated = jax.lax.select(jnp.logical_and(
        monster_defeated, reward_state.counter_prompt == 2), 3,
        counter_updated)

    task_achieved = counter_updated == 3

    reward_state = reward_state.replace(counter_prompt=
        counter_updated)

    return task_achieved, reward_state, reward_state.
        counter_prompt

prompt_sequence_craft_and_use_wood_sword = ['collect wood', 'make
    wood sword', 'defeat monster']
```

Goal Build shelter:

```
@struct.dataclass
class RewardState_build_shelter:
    has_placed_enough_blocks: bool = False
    counter_prompt: int = 0

def build_shelter(state, action, rng, reward_state):
    """
    Check whether the agent builds a shelter using available
        resources (wood, stone) within a defined area.
    """
    def helper_in_bounds(state, position):
```

```
        in_bounds_x = jnp.logical_and(0 <= position[0], position
            [0] < state.map.shape[0])
        in_bounds_y = jnp.logical_and(0 <= position[1], position
            [1] < state.map.shape[1])
        return jnp.logical_and(in_bounds_x, in_bounds_y)

    # Define the shelter area (3x3 area around the player)
    shelter_area = jnp.array([
        [-1, -1], [-1, 0], [-1, 1],
        [0, -1], [0, 0], [0, 1],
        [1, -1], [1, 0], [1, 1]
    ], dtype=jnp.int32)

    # Check if the agent has placed blocks in the shelter area
    def check_placed_blocks(unused, loc_add):
        pos = state.player_position + loc_add
        is_in_bounds = helper_in_bounds(state, pos)
        is_placed = jnp.logical_and(is_in_bounds, state.map[pos
            [0], pos[1]] != BlockType.GRASS.value)
        return None, is_placed

    _, is_placed_blocks = jax.lax.scan(check_placed_blocks, None,
        shelter_area)

    # Check if the agent has placed at least 5 blocks in the
        shelter area
    has_placed_enough_blocks = is_placed_blocks.sum() >= 5

    # Update the reward state
    reward_state = reward_state.replace(has_placed_enough_blocks=
        has_placed_enough_blocks, counter_prompt=jax.lax.select(
        has_placed_enough_blocks, 1, 0))


    return has_placed_enough_blocks, reward_state, reward_state.
        counter_prompt

prompt_sequence_build_shelter = ['gather resources', 'place blocks
    ', 'form shelter']
```

Goal build fortified shelter

```
    @struct.dataclass
class RewardState_build_fortified_shelter:
    has_placed_enough_blocks: bool = False
    counter_prompt: int = 0

def build_fortified_shelter(state, action, rng, reward_state):
    """
    Check whether the agent builds a fortified shelter using stone
         and iron blocks, ensuring it is well-protected.
    """
    # Define the shelter area (5x5 around the player)
    shelter_area = jnp.array([
        [-2, -2], [-2, -1], [-2, 0], [-2, 1], [-2, 2],
        [-1, -2], [-1, -1], [-1, 0], [-1, 1], [-1, 2],
        [0, -2], [0, -1], [0, 0], [0, 1], [0, 2],
        [1, -2], [1, -1], [1, 0], [1, 1], [1, 2],
        [2, -2], [2, -1], [2, 0], [2, 1], [2, 2]
    ], dtype=jnp.int32)
```

```
    # Check if the agent has placed blocks in the shelter area
    def check_placed_blocks(unused, loc_add):
        pos = state.player_position + loc_add
        is_in_bounds = jnp.logical_and(0 <= pos[0], pos[0] < state
            .map.shape[0])
        is_in_bounds = jnp.logical_and(is_in_bounds, jnp.
            logical_and(0 <= pos[1], pos[1] < state.map.shape[1]))
        is_placed = jnp.logical_and(is_in_bounds, jnp.logical_or(
            state.map[pos[0], pos[1]] == BlockType.STONE.value,
            state.map[pos[0], pos[1]] == BlockType.IRON.value))
        return None, is_placed

    _, is_placed_blocks = jax.lax.scan(check_placed_blocks, None,
        shelter_area)

    # Check if the agent has placed at least 10 blocks in the
        shelter area
    has_placed_enough_blocks = is_placed_blocks.sum() >= 10

    # Update the reward state
    reward_state = reward_state.replace(has_placed_enough_blocks=
        has_placed_enough_blocks, counter_prompt=jax.lax.select(
        has_placed_enough_blocks, 1, 0))


    return has_placed_enough_blocks, reward_state, reward_state.
        counter_prompt

prompt_sequence_build_fortified_shelter = ['gather stone', 'gather
     iron', 'place stone blocks', 'place iron blocks', 'form a 5x5
     shelter']
```

Goal Build mob trap:

```
@struct.dataclass
class RewardState_build_mob_trap:
    task_achieved: bool = False
    counter_prompt: int = 0

def build_mob_trap(state, action, rng, reward_state):
    """
    Check whether the agent builds a trap to catch mobs by
        strategically placing blocks and using resources.
    """

    # Define the trap area (3x3 around the player)
    trap_area = jnp.array([
        [-1, -1], [-1, 0], [-1, 1],
        [0, -1], [0, 0], [0, 1],
        [1, -1], [1, 0], [1, 1]
    ], dtype=jnp.int32)

    # Check if the agent has placed blocks in the trap area
    def check_placed_blocks(unused, loc_add):
        pos = state.player_position + loc_add
        is_in_bounds = jnp.logical_and(0 <= pos[0], pos[0] < state
            .map.shape[0])
        is_in_bounds = jnp.logical_and(is_in_bounds, jnp.
            logical_and(0 <= pos[1], pos[1] < state.map.shape[1]))
```

```
        is_placed = jnp.logical_and(is_in_bounds, state.map[pos
            [0], pos[1]] != BlockType.GRASS.value)
        return None, is_placed

    _, is_placed_blocks = jax.lax.scan(check_placed_blocks, None,
        trap_area)

    # Check if the agent has placed at least 5 blocks in the trap
        area
    has_placed_enough_blocks = is_placed_blocks.sum() >= 5

    # Check if any mobs are trapped in the area
    def check_trapped_mobs(unused, loc_add):
        pos = state.player_position + loc_add
        is_in_bounds = jnp.logical_and(0 <= pos[0], pos[0] < state
            .map.shape[0])
        is_in_bounds = jnp.logical_and(is_in_bounds, jnp.
            logical_and(0 <= pos[1], pos[1] < state.map.shape[1]))
        is_mob = jnp.logical_and(is_in_bounds, state.mob_map[pos
            [0], pos[1]])
        return None, is_mob

    _, is_trapped_mobs = jax.lax.scan(check_trapped_mobs, None,
        trap_area)

    # Check if there is at least one mob trapped
    has_trapped_mobs = is_trapped_mobs.sum() > 0

    # Task achievement is based on having placed enough blocks and
         trapping mobs
    task_achieved = jnp.logical_and(has_placed_enough_blocks,
        has_trapped_mobs)

    # Update the reward state
    reward_state = reward_state.replace(task_achieved=
        task_achieved, counter_prompt=jax.lax.select(task_achieved
        , 1, 0))

    # Reward is given when the task is achieved
    reward = jax.lax.select(task_achieved, 1., 0.)

    return task_achieved, reward_state, reward_state.
        counter_prompt

prompt_sequence_build_mob_trap = ['gather resources', 'place
    blocks strategically', 'trap mobs']
```

D Prompt for the LLM

```
You are a player in an open-world game trying to write reward
    functions to train an Deep-Reinforcement learning AI agent.You
     want the AI agent to achieve as many different interesting
    and complex tasks as possible. For this you want to design a
    curriculum of tasks tailored to the AI agent capabilities. The
     AI agent is acting in the crafter environement. Crafter is an
     open-world game designed to evaluate the general abilities of
     intelligent agents within a single environment. Developed for
     reinforcement learning research, Crafter provides a
```

```
    procedurally generated 2D world where agents must perform
    tasks such as foraging for food, finding water, building tools
     and constructions, and defending against monsters. Crafter is
     in 2D.


Instructions:
- The next task should be learnable:
    - Not too difficult given the difficulty of the previous tasks
       . Don't create tasks that are not learnable.
    - Realistic for the agent to achieve in the current
        environment

- The next task should be interesting:
    - Not too similar to the current tasks. Do not copy the
        existing tasks.
    - Not too easy for the agent to achieve. The agent should
        learn something new.
    - Useful according to humans, making it worth learning.
    - Creative or surprising.
    - Optionally, the task can be fun and engaging to watch.

- The task should be well-defined:
    - The code should reflect the task description (in the
        docstring). If it's not the case, create a new reward
        function (with a different name). Tasks that are are not
        learnable usually are not well defined and can be
        corrected by changing how the code reflects the task
        description.
    - If you want to re-use a helper function, you have to re-
        write it in the reward function.
    - Pay attention to the state representation (EnvState and
        Inventory) and the game logic. The reward function should
        be compatible with the game logic and the EnvState.

- Jax and jit compatible:
    - don't use python conditionals and boolean operations.
        - example:
            - python (not jax and jit compatible): has_resources =
                state.inventory.stone >= 3 and state.inventory.
               wood >= 2
            - jax and jit compatible: has_resources = jnp.
               logical_and(state.inventory.stone >= 3, state.
               inventory.wood >= 2)
    - jnp.logical_or() and jnp.logical_and() takes only 2
        positional arguments, so you can't use it with more than 2
         arguments. Alternatively you can use & and | but add
        parentheses around the condition.
    - if you use array in the reward state, you should use the
        following syntax using field:
class RewardState_move_far_away:
     start_position: jnp.ndarray = field(default_factory=lambda:
        jnp.array([0, 0], dtype=jnp.int32))
     counter_prompt: int = 0


The world-state is updated based on a state representation ("state
    ") that is updated after each action of the agent by the
    following game logic. The state has the following structure:
```

```
# Inventory class to track player's collected resources and
  crafted tools
class Inventory:
    wood: int = 0
    stone: int = 0
    coal: int = 0
    iron: int = 0
    diamond: int = 0
    sapling: int = 0
    wood_pickaxe: int = 0
    stone_pickaxe: int = 0
    iron_pickaxe: int = 0
    wood_sword: int = 0
    stone_sword: int = 0
    iron_sword: int = 0

# Mobs class to represent creatures (zombies, cows, skeletons) in
  the game
class Mobs:
    position: jnp.ndarray
    health: int
    mask: bool
    attack_cooldown: int

# EnvState class to represent the entire game state
class EnvState:
    map: jnp.ndarray= map # 64x64 grid representing the world
    mob_map: jnp.ndarra= jnp.zeros((64, 64), dtype=bool)

    player_position: jnp.ndarray = player_position
    player_direction: int = Action.UP.value

    # Player's vital stats (health, food, water, energy)
    player_health: int = 9
    player_food: int = 9
    player_drink: int = 9
    player_energy: int = 9
    is_sleeping: bool = False

    # Rates of change for player's vital stats
    player_recover: float = 0.0
    player_hunger: float = 0.0
    player_thirst: float = 0.0
    player_fatigue: float = 0.0

    inventory: Inventory
    zombies: Mobs
    cows: Mobs
    skeletons: Mobs
    arrows: Mobs
    arrow_directions: jnp.ndarray

    # Tracking planted crops
    growing_plants_positions: jnp.ndarray
    growing_plants_age: jnp.ndarray
    growing_plants_mask: jnp.ndarray
```

```
# Enum for different block types in the game world
class BlockType(Enum):
    INVALID = 0
    OUT_OF_BOUNDS = 1
    GRASS = 2
    WATER = 3
    STONE = 4
    TREE = 5
    WOOD = 6
    PATH = 7
    COAL = 8
    IRON = 9
    DIAMOND = 10
    CRAFTING_TABLE = 11
    FURNACE = 12
    SAND = 13
    LAVA = 14
    PLANT = 15
    RIPE_PLANT = 16

# Enum for possible actions the player can take
class Action(Enum):
    NOOP = 0
    LEFT = 1
    RIGHT = 2
    UP = 3
    DOWN = 4
    DO = 5
    SLEEP = 6
    PLACE_STONE = 7
    PLACE_TABLE = 8
    PLACE_FURNACE = 9
    PLACE_PLANT = 10
    MAKE_WOOD_PICKAXE = 11
    MAKE_STONE_PICKAXE = 12
    MAKE_IRON_PICKAXE = 13
    MAKE_WOOD_SWORD = 14
    MAKE_STONE_SWORD = 15
    MAKE_IRON_SWORD = 16

# Blocks that the player cannot walk through
SOLID_BLOCKS = jnp.array(
    [
        BlockType.WATER.value,
        BlockType.STONE.value,
        BlockType.TREE.value,
        BlockType.COAL.value,
        BlockType.IRON.value,
        BlockType.DIAMOND.value,
        BlockType.CRAFTING_TABLE.value,
        BlockType.FURNACE.value,
        BlockType.PLANT.value,
        BlockType.RIPE_PLANT.value,
    ],
    dtype=jnp.int32,
)

# Directions for movement and interaction
DIRECTIONS = jnp.concatenate(
```

```
    (
        jnp.array([[0, 0], [0, -1], [0, 1], [-1, 0], [1, 0]],
            dtype=jnp.int32),
        jnp.zeros((11, 2), dtype=jnp.int32),
    ),
    axis=0,
)

# Adjacent blocks for interaction
CLOSE_BLOCKS = jnp.array(
    [
        [0, -1],
        [0, 1],
        [-1, 0],
        [1, 0],
        [-1, -1],
        [-1, 1],
        [1, -1],
        [1, 1],
    ],
    dtype=jnp.int32,
)


# Example usage
view = extract_agent_view(state).

The code you create needs to be compatible with jax and jit. To
    create a new task you should write a function with the same
    input and output as the previous examples. Also add a reward
    class like the previous example. The reward class is a way to
    store information across different step updates and generate
    reward based on information contained in a sequence of states
    of the world. Also add a list of brief description of the
    sequential subtasks (less than 10 words per subtasks and less
    than 10 subtasks)to guide the AI agent in the learning process
    . Create one new task by writing the code for its reward
    function with its associated reward state and prompt sequence.
     Take into account the agent achievements to make a novel
    interesting task that is not too easy and not too hard for the
     AI agent. Answer based on the syntax of the following format:


My proposed task is:
Reasoning for what the next reward should be:
<brief reasoning>

Next The code for the reward function and its reward state and
    prompt sequence is:
'''python
'''
<reward function>

<reward state>

<prompt_sequence = [<description of the task>, <description of
    subtask 1>, <description of subtask 2>, ...]>
'''
```

```
Please create a new reward function. Don't re-use the same name of
    function. You should take inspiration from the reward
    functions below to create a new reward function that is
    interesting, novel and achievable.For each reward functions a
    score between 0 and 100 is given to describe the difficuly and
    learnability based on the AI-agent experience in craftax.

Below are some rewards that are learnable:

The learnability of the following goal is: <score learnability>
The difficulty of the following goals is: <score difficulty>
<g, main goal sampled uniformly among the learnable ones>

The learnability of the following goal is: <score learnability>
The difficulty of the following goals is: <score difficulty>
<r_{L1}, closest learnable goal in embedding space >

The learnability of the following goal is: <score learnability>
The difficulty of the following goals is: <score difficulty>
<r_{L2}, second closest learnable goal in embedding space >

The learnability of the following goal is: <score learnability>
The difficulty of the following goals is: <score difficulty>
<g_1', creative goal sampled uniformly among the learnable ones>

The learnability of the following goal is: <score learnability>
The difficulty of the following goals is: <score difficulty>
<g_2', creative goal sampled uniformly among the learnable ones>


Below are some rewards that are not learnable:

The learnability of the following goal is: <score learnability>
The difficulty of the following goals is: <score difficulty>
<r_{N1}, closest non-learnable goal in embedding space >

The learnability of the following goal is: <score learnability>
The difficulty of the following goals is: <score difficulty>
<r_{N2}, second closest non-learnable goal in embedding space >

My proposed task is:
```