000
001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053

ORSO: ACCELERATING REWARD DESIGN VIA ONLINE REWARD SELECTION AND POLICY OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Reward shaping is a critical component in reinforcement learning (RL), particularly for complex tasks where sparse rewards can hinder learning. While shaping rewards have been introduced to provide additional guidance, selecting effective shaping functions remains challenging and computationally expensive. This paper introduces Online Reward Selection and Policy Optimization (ORSO), a novel approach that frames shaping reward selection as an online model selection problem. ORSO employs principled exploration strategies to automatically identify promising shaping reward functions without human intervention, balancing exploration and exploitation with provable regret guarantees. We demonstrate ORSO's effectiveness across various continuous control tasks using the Isaac Gym simulator. Compared to traditional methods that fully evaluate each shaping reward function, ORSO significantly improves sample efficiency, reduces computational time, and consistently identifies high-quality reward functions that produce policies comparable to those generated by domain experts through hand-engineered rewards with up to $16\times$ less compute. Code is available at `anonymized for submission`

1 INTRODUCTION

Reward functions are crucial in reinforcement learning (RL; Sutton & Barto (2018)) as they guide the learning of successful policies. In many real-world scenarios, the ultimate objective involves maximizing long-term rewards that are not immediately available, making optimization challenging. To address this, practitioners often introduce shaping rewards (Margolis & Agrawal, 2022; Liu et al., 2024; Mahmood et al., 2018; Ng et al., 1999) – to provide additional guidance during training. Instead of directly maximizing the task rewards (R), it is therefore common for the RL algorithm to maximize an easier-to-optimize shaped reward function F in the hope of obtaining high performance as measured by task rewards, R. While shaping rewards contain helpful hints, maximizing them does not necessarily solve the task. For instance, an agent tasked with finding an exit (i.e., longer-term reward in the future) may be provided with shaping rewards to avoid obstacles. However, the task success ultimately depends on reaching the exit, not just avoiding obstacles. If poorly designed, the shaped rewards F can mislead the RL algorithm, causing the agent to focus on maximizing F while neglecting R (Chen et al., 2022; Agrawal, 2021), leading to training failure or suboptimal performance.

Designing effective shaping reward functions F that improve RL algorithm performance is challenging and time-consuming. It requires multiple iterations of training agents with different shaping rewards, evaluating their performance on the task reward R, and refining F accordingly. This process is inefficient due to the lengthy training runs and because the performance measured early in training may be misleading, making it challenging to quickly iterate over different shaping rewards.

To address this challenge, we propose treating the design of the shaping reward function as an exploration-exploitation problem and to solve it using provably efficient online decision-making algorithms similar to those in multi-armed bandits (Auer et al., 2002; Auer, 2002) and model selection (Agarwal et al., 2017; Pacchiano et al., 2020; Dann et al., 2024; Foster et al., 2019; Lee et al., 2021). Each shaping reward function acts as an arm or model, with the agent's task reward R when trained with shaping reward F serving as the model's utility. Our goal is to identify the best shaping reward function within a fixed time budget.

054
055
056
057
058
059
060
061
062
063
064
065
066
067
068
069
070
071
072
073
074
075
076
077
078
079
080
081
082
083
084
085
086
087
088
089
090
091
092
093
094
095
096
097
098
099
100
101
102
103
104
105
106
107

This approach presents unique challenges. Unlike standard multi-armed bandit settings with stationary reward distributions, the utility of a shaping reward function in our case is nonstationary. As the agent explores new parts of the state space during training, the reward distribution changes. Additionally, we must balance exploration and exploitation to efficiently allocate training time among different shaping rewards without committing too early to high-performing options or wasting time on low-performing ones.

We introduce *Online Reward Selection and Policy Optimization* (ORSO), an algorithm that efficiently selects the best shaping reward function from a set of candidate shaping reward functions to improve RL performance on the task reward. ORSO provides regret guarantees and adaptively allocates training time to each shaping reward based on a model selection algorithm at each step. Our empirical results across various continuous control tasks using the Isaac Gym simulator (Makoviychuk et al., 2021) demonstrate that ORSO identifies the best auxiliary reward function much faster ($2\times$ or more) than current methods. Moreover, ORSO consistently selects reward functions that are comparable to, and sometimes surpass, those designed by domain experts with up to $16\times$ less compute.

2 PRELIMINARIES

Reinforcement Learning (RL) In RL, the objective is to learn a policy for an agent (e.g., a robot) that maximizes the expected cumulative reward during the interaction with the environment. The interaction between the agent and the environment is formulated as a Markov decision process (MDP) (Puterman, 2014), $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \gamma, \rho_0)$, where the $\mathcal{S}$ and $\mathcal{A}$ denote state and action spaces, respectively, $P : \mathcal{S} \times \mathcal{A} \to \Delta_{\mathcal{S}}$[1] is the state transition dynamics, $r : \mathcal{S} \times \mathcal{A} \to \Delta_{\mathbb{R}}$ denotes the reward function, $\gamma \in [0, 1)$ is the discount factor, and $\rho_0 \in \Delta_{\mathcal{S}}$ is the initial state distribution. At each timestep $t \in \mathbb{N}$ of interaction, the agent selects an action $a_t \sim \pi(\cdot \mid s_t)$ based on its policy π, receives a (possibly) stochastic reward $r_t \sim r(s_t, a_t)$, and transitions to the next state $s_{t+1} \sim P(\cdot \mid s_t, a_t)$ according to the transition dynamics. Here, r is the task reward, also referred to as extrinsic reward (Chen et al., 2022). RL algorithms aim to find a policy $\pi^\star$ that maximizes the discounted cumulative reward, i.e.,

$$\pi^\star \in \arg\max_{\pi} \mathcal{J}(\pi) := \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, \begin{array}{l} s_0 \sim \rho_0,\ a_t \sim \pi(\cdot \mid s_t), \\ r_t \sim r(s_t, a_t),\ s_{t+1} \sim P(\cdot \mid s_t, a_t) \end{array}\right]. \tag{1}$$

3 METHOD: REWARD DESIGN AS SEQUENTIAL DECISION MAKING

As previously stated, the reward function r encodes the task objective but can be sparse, making it difficult to directly optimize using RL methods. We formalize the reward design problem as follows.

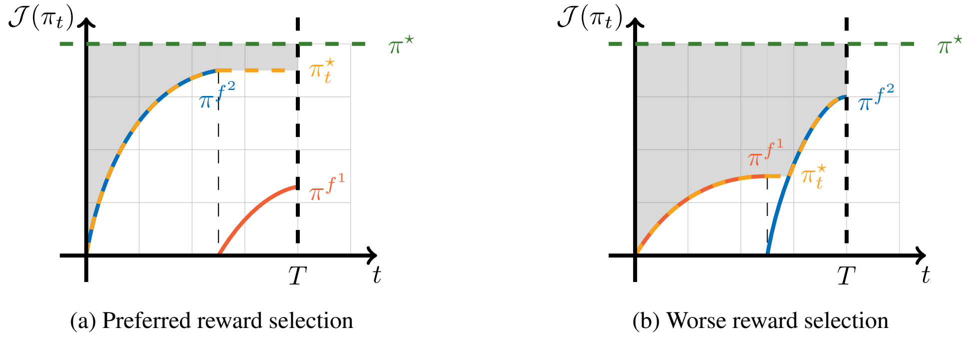
Definition 3.1 (Reward Design). *Let $\mathfrak{A}$ be a reinforcement learning algorithm that takes an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \gamma, \rho_0)$, a reward function f, and a number of interction steps with the environment N as input and returns a policy $\pi^f = \mathfrak{A}_f(\mathcal{M}, N)$ that approximately maximizes reward f in $\mathcal{M}$ after N interaction steps.*

Given $\mathcal{M}$ and $\mathfrak{A}$, the reward design problem aims to find a reward function $f : \mathcal{S} \times \mathcal{A} \to \Delta_{\mathbb{R}}$, with $f \in \mathcal{R}$, the space of reward functions, such that the policy $\pi^f = \mathfrak{A}_f(\mathcal{M}, N)$ achieves an expected return under the task reward r, such that $\mathcal{J}\left(\pi^f\right) \approx \max_{r' \in \mathcal{R}} \mathcal{J}(\pi^{r'}) = \mathcal{J}\left(\pi^\star\right)$.

While this could be achieved by running the algorithm $\mathfrak{A}$ on every possible reward function $r' \in \mathcal{R}$, this is computationally prohibitive. The reward space $\mathcal{R}$ can be extremely large, and attempting to optimize over all possible rewards is impractical, especially when the available interaction budget is constrained.

To make the problem tractable, we assume access to a finite set of candidate shaping reward functions $\mathcal{R}^K = \left\{f^1, \ldots, f^K\right\} \sim G(\mathcal{R})$, where G is a distribution over the set of reward functions, that contains at least one near-optimal reward function and a budget of iterations T. If the budget

[1] $\Delta_{\mathcal{S}}$ denotes the set of probability distributions over $\mathcal{S}$.



(a) Preferred reward selection

(b) Worse reward selection

Figure 1: Comparison of two reward selection strategies given a time budget T. The green dashed line represents the task reward of the optimal policy, $\pi^\star$. The red and blue curves show the cumulative maximum task rewards for policies trained withreward functions f^1 and f^2, respectively. The yellow curve, $\pi_t^\star$, tracks the maximum of the red and blue curves. The shaded gray area depicts cumulative regret in Equation (2) associated with each selection strategy. The preferred selection strategy, (a), spends most iterations on f^2, while the worse strategy, which initially focuses on f^1, leaves too little of the available budget T to fully exploit f^2, resulting in lower performance and higher regret.

does not allow training on each $f^i \in \mathcal{R}^K$, we need to allocate resources to gather useful information about the quality of each candidate, while simultaneously optimizing the most promising ones. This introduces a fundamental exploration-exploitation tradeoff. On one hand, we must explore various rewards to identify high performers; on the other, we need to exploit promising candidates to train performant policies.

Cumulative Regret $\implies$ Efficiency This tradeoff is captured by a regret-based objective, commonly used in sequential decision-making problems, which measures the suboptimality incurred by the current policy with respect to the optimal one. Therefore, we cast the reward selection problem as an online model selection problem. Let n_t^i denote the number of iterations reward function f^i has been used for training up to iteration t. Then the set of policies trained up to step t is $\Pi(t) = \{\pi_{n_j^i}^{f^{i_j}}\}_{j=1}^t$, where $\pi_{n_j^i}^{f^{i_j}} = \mathfrak{A}_{f^{i_j}}(\mathcal{M}, n_j^i)$, the policy trained with reward function f^{i_j} for n_j^i iterations. We define *model selection regret* as

$$\mathrm{MReg}(T) \coloneqq \sum_{t=1}^{T} \mathcal{J}(\pi^\star) - \mathcal{J}(\pi_t^\star), \tag{2}$$

where $\pi_t^\star \coloneqq \arg\max_{\pi \in \Pi(t)} \mathcal{J}(\pi)$. The choice of $\pi_t^\star$ in the definition of $\mathrm{MReg}(T)$ reflects a practical preference. Practitioners are generally more interested in the best-performing solution available at a given point, rather than the most recent update. For instance, in deploying a robotic running policy, one would select the fastest policy observed thus far – assuming the objective is to run as fast as possible.

The regret minimization framework is well-aligned with the goal of *efficient* reward design, as it emphasizes the speed at which effective policies are learned. In Figure 1, we compare two strategies: one that starts by training with the worse reward function, f^1, until convergence, and another that immediately focuses on the better reward function, f^2. The shaded area represents the regret incurred by each selection strategy (Equation (2)), which reflects the performance gap between the learned policy and the optimal one over time. The worse strategy spends too much of the available budget T on f^1, leaving insufficient iterations for training on f^2. As a result, the best policy trained with the suboptimal strategy reaches a lower performance and the selection strategy incurs higher regret. Conversely, starting with f^2 minimizes regret and maximizes the performance. A further discussion of the online model selection problem can be found in Appendix B.

162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215

3.1 ORSO: ONLINE REWARD SELECTION AND POLICY OPTIMIZATION

In this section, we introduce ORSO (*Online Reward Selection and Optimization*), a novel approach to *efficiently* and *effectively* design reward functions for reinforcement learning. Our method operates in two phases: (1) reward generation and (2) online reward selection and policy optimization.

Reward Generation In the first phase of ORSO, we generate a set of candidate reward functions $\mathcal{R}^K$ for the online selection phase. Given an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \rho_0)$ and a stochastic generator G, we sample a set of K reward function candidates, $\mathcal{R}^K = \{f^1, \ldots, f^K \mid \forall i \in [K],\ f^i : \mathcal{S} \times \mathcal{A} \to \Delta_{\mathbb{R}},\ f^i \sim G\}$, from G during the reward design phase. The generator G can be any distribution over the reward function space $\mathcal{R}$. For instance, if the set of possible reward functions is given by a linear combination of two reward components c_1, c_2, which are functions of the current state and action, such that $r(s, a) = w_1 c_1(s, a) + w_2 c_2(s, a)$, then the generator G can be represented by the means and variances of two normal distributions, one for each weight w_1, w_2.

Online Reward Selection and Policy Optimization Our algorithm for online reward selection and policy optimization is described in Algorithm 1. On a high level, the algorithm proceeds as follows. Given an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \rho_0)$, an RL algorithm $\mathfrak{A}$ and a reward generator G, we sample set of K reward functions $\mathcal{R}^K \sim G$ and initialize K distinct policies $\pi^1, \ldots, \pi^K$. At step t of the reward selection process, the algorithm selects a learner $i_t \in [K]$ according to a selection strategy. We then perform N iterations of training with algorithm $\mathfrak{A}$, updating the policy corresponding to reward function i_t to obtain π^{i_t}. Policy π^{i_t} is simultaneously evaluated under the task reward function r and the necessary variables for the model selection algorithm are then updated (e.g., reward estimates, reward function visitation counts, and confidence intervals). The algorithm returns the reward function $f_T^\star$ and the corresponding policy $\pi_T^\star$ that performs the best under the task reward function r.

Algorithm 1 ORSO: Online Reward Selection and Policy Optimization

Require: MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \rho_0)$, algorithm $\mathfrak{A}$, generator G
1: Sample K reward functions $\mathcal{R}^K = \left\{f^1, \ldots, f^K\right\} \sim G$
2: Initialize K policies $\left\{\pi^1, \ldots, \pi^K\right\}$
3: **for** $t = 1, 2, \ldots, T$ **do**
4: Select an model $i_t \in [K]$ according to a selection strategy
5: Update $\pi^{i_t} \leftarrow \mathfrak{A}_{f^{i_t}}(\mathcal{M}, N, \pi^{i_t})$
6: Evaluate $\mathcal{J}(\pi^{i_t}) \leftarrow \texttt{Eval}(\pi^{i_t})$
7: Update variables (e.g., reward estimates and confidence intervals)
8: **end for**
9: **return** $\pi_T^\star, f_T^\star = \arg\max_{i \in [K]} \mathcal{J}(\pi^i)$

Choice of Selection Algorithm While ORSO is a general algorithm that can employ any selection method to pick the reward function to train on, the performance depends on the choice of algorithm.

For instance, using a simple selection method like ε-greedy introduces an element of exploration by occasionally selecting a random reward function (with probability ε), but it risks overcommitting to a seemingly promising reward function early on. This can lead to suboptimal performance if the chosen reward function causes the task performance to plateau in the long run. However, greedier methods, such as ε-greedy, can achieve lower regret if they commit to the actual optimal reward function early in the process. These methods are particularly effective when early performance signals are strong indicators of long-term success.

However, if initial performance is not a reliable predictor of future outcomes, these greedy approaches may struggle, as they risk prematurely locking onto suboptimal rewards. In contrast, more exploratory algorithms like the exponential-weight algorithm for exploration and exploitation (Exp3) (Auer et al., 2002) maintain a broader search, potentially discovering better rewards in the long run, especially in environments where early signals are less informative. We empirically validate different choices of selection algorithms in Section 5.

4 THEORETICAL GUARANTEES

In this section, we provide regret guarantees for ORSO with the Doubling Data-Driven Regret Balancing (D^{3}RB) algorithm by Dann et al. (2024). A discussion of the intuition behind the D^{3}RB algorithm and the full pseudo-code for ORSO with D^{3}RB is provided in Appendix C. We note that the regret definition used in the online model selection literature is an upper bound for the model selection regret defined in Section 3. We provide a further discussion of this relationship in Appendix B.

We first introduce some useful definitions for our analysis.

Definition 4.1 (Definition 2.1 from Dann et al. (2024))**.** *The regret scale of learner i after being played t times is $\frac{\sum_{\ell=1}^{t} \mathrm{reg}(\pi^i_{(\ell)})}{\sqrt{t}}$ where $\mathrm{reg}(\pi^i_{(\ell)}) = \mathcal{J}(\pi^\star) - \mathcal{J}(\pi^i_{(\ell)})$ in the reward design problem.*

For a positive constant $d_{\min} > 0$, the regret coefficient of learner i after being played for t rounds is $d^i_{(t)} = \max\{d_{\min}, \sum_{\ell=1}^{t} \mathrm{reg}(\pi^i_{(\ell)})/\sqrt{t}\}$. That is, $d^i_{(t)} \geq d_{\min}$ is the smallest number such that the incurred regret is bounded as $\sum_{\ell=1}^{t} \mathrm{reg}(\pi^i_{(\ell)}) \leq d^i_{(t)}\sqrt{t}$.

Dann et al. (2024) use $\sqrt{t}$ as this is the most commonly targeted regret rate in stochastic settings.

The main idea underlying our regret guarantees is that the internal state of all suboptimal reward functions is only updated up to a point where the regret equals that of the best policy so far.

We assume there exists a learner that monotonically dominates every other learner.

Assumption 4.2. *There is a learner $i_\star$ such that at all time steps, its expected sum of rewards dominates any other learner, i.e., $u^{i_\star}_{(t)} \geq u^i_{(t)}$, for all $i \in [K], t \in \mathbb{N}$ and such that its average expected rewards are increasing, i.e., $\frac{u^{i_\star}_{(t)}}{t} \leq \frac{u^{i_\star}_{(t+1)}}{t+1}, \quad \forall t \in \mathbb{N}$. This is equivalent to saying that $d^{i_\star}_{(t)} \geq d^{i_\star}_{(t+1)}$, for all $t \in \mathbb{N}$.*

Assumption 4.2 guarantees that the cumulative expected reward of the optimal learner $i_\star$ is always at least as large as the cumulative expected reward of any other learner and that its average performance increases monotonically.

Following the notation of Dann et al. (2024), we refer to the event that the confidence intervals for the reward estimator are valid as $\mathcal{E}$.

Definition 4.3 (Definition 8.1 from Dann et al. (2024))**.** *We define the event $\mathcal{E}$ as the event in which for all rounds $t \in \mathbb{N}$ and learners $i \in [K]$ the following inequalities hold*

$$-c\sqrt{n^i_t \ln \frac{K \ln n^i_t}{\delta}} \leq \widehat{u}^i_t - u^i_t \leq c\sqrt{n^i_t \ln \frac{K \ln n^i_t}{\delta}} \tag{3}$$

for the algorithm parameter $\delta \in (0, 1)$ and a universal constant $c > 0$.

Then we can refine Lemma 9.3 from Dann et al. (2024) in the case where Assumption 4.2 holds.

Lemma 4.4. *Under event $\mathcal{E}$ and Assumption 4.2, with probability $1 - \delta$, the regret of all learners i is bounded in all rounds T as*

$$\sum_{t=1}^{n^i_T} \mathrm{reg}(\pi^i_{(t)}) \leq 6 d^{i_\star}_T \sqrt{n^{i_\star}_T + 1} + 5c\sqrt{(n^{i_\star}_T + 1) \ln \frac{K \ln T}{\delta}}, \tag{4}$$

where $d^{i_\star}_T = d^{i_\star}_{(n^{i_\star}_T)}$.

We provide the proof for Lemma 4.4 in Appendix D. Lemma 4.4 implies that when Assumption 4.2 holds, the regrets are perfectly balanced. This is in stark contrast with the regret guarantees of Dann et al. (2024) that prove the D^{3}RB algorithm's overall regret to scale as $\left(\bar{d}^{i_\star}_T\right)^2 \sqrt{T}$ where $\bar{d}^{i_\star}_t = \max_{\ell \leq t} d^{i_\star}_\ell$. Instead, our results above depend not on the monotonic regret coefficients $\bar{d}^{i_\star}_t$ but on the true regret coefficients $d^{i_\star}_t$. Even if learner $i_\star$ has a slow start (and therefore a large $\bar{d}^{i_\star}_T$), as long as monotonicity holds and the $i_\star$-th learner recovers in the later stages of learning, our results show that D^{3}RB will achieve a regret guarantee comparable with running learner $i_\star$ in isolation.

270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323

5 PRACTICAL IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we present a practical implementation[2] of ORSO and its experimental results on several continuous control tasks. We study the ability of ORSO to design effective reward functions with varying budget constraints. We also study how different sample sizes, K, of the set of reward functions $\mathcal{R}^K$ influence the performance of ORSO and compare different selection algorithms.

This section is structured as follows. First, we present the experimental setup, including the environments and baselines, and the practical consideration of the reward generator G and the algorithms used in the online reward selection phase. Then, we present the main results and ablate our design choices. Further experimental results can be found in Appendix H

5.1 EXPERIMENTAL SETUP

Environments and RL Algorithm We evaluate ORSO on a set of continuous control tasks using the Isaac Gym simulator (Makoviychuk et al., 2021). Specifically, we consider the following tasks: CARTPOLE and BALLBALANCE, which are relatively simple; two locomotion tasks, ANT and HUMNAOID, which have dense but unshaped task rewards – for instance, the agent is rewarded for running fast, but the reward function lacks terms to encourage upright posture or smooth movement; and two complex manipulation tasks, ALLEGROHAND and SHADOWHAND, which feature sparse task reward functions.

Our policies are trained using the proximal policy optimization (PPO) algorithm (Schulman et al., 2017), with our implementation built on CleanRL (Huang et al., 2022). We chose PPO because the Makoviychuk et al. (2021) provide hyperparameters, which we use, that enable it to perform well on these tasks when using the human-engineered reward functions.

5.1.1 BASELINES

In our experiments, we consider three baselines. We analyze the performance of policies trained using each reward function detailed below. We evaluate the reward function selection *efficiency* of ORSO compared to more naive selection strategies.

No Design *(Task Reward with No Shaping)* We train the agent with the task reward function r for each MDP. These reward functions can be sparse (for manipulation) or unshaped (for locomotion). We use the same reward definitions as prior work (Ma et al., 2024), which we report in Appendix E.

Human We consider the human-engineered reward functions for each task provided by (Makoviychuk et al., 2021). We note that these are constructed such that training PPO with the given hyperparameters yields a performant policy with respect to the task reward function. The function definitions are reported in Appendix E.

Naive Selection We employ EUREKA (Ma et al., 2024) as a baseline for the naive selection approach. EUREKA uses a large language model to generate Python code for the reward functions of several continuous control tasks. EUREKA uses an evolutionary scheme to evaluate and improve its reward functions. During each iteration, EUREKA samples a set of reward functions from an LLM, trains a policy on each reward function, and uses the best-performing reward function as a context for the LLM to perform the evolutionary step. However, this selection strategy can be seen as naive, as it uniformly explores each reward function for a fixed number of iterations, regardless of its actual performance on the task.

5.1.2 IMPLEMENTATION

Reward Generation Similarly to recent works on reward design, which demonstrate that LLMs can generate effective reward functions for training agents (Park et al., 2024; Ma et al., 2024; Xie et al., 2024), we follow this paradigm by using GPT-4 (Achiam et al., 2023) to avoid manually designing reward function components. The language model is prompted to generate reward function

[2]The code for ORSO is available at `anonymized for submission`

code in Python based on some minimal environment code describing the observation space and useful class variables. We employ prompts similar to those used by Ma et al. (2024). Since the exact prompts are not the primary focus of our work, we do not detail them here; instead, we refer readers to our codebase for further details on the prompt construction.

While the LLM produces seemingly good code, this does not guarantee that the sampled code is bug-free and runnable. In ORSO, we employ a simple rejection sampling technique to construct sets of only valid reward functions with high probability. We also note that the initial set of generated reward functions in ORSO might not contain an effective reward function.[3] To address this limitation, we introduce a mechanism for improving the reward function set through iterative resampling and in-context evolution of new sets $\mathcal{R}^K$. We provide more details on the rejection sampling mechanism and the iterative refinement process Appendix F.

Online Reward Selection Algorithms We evaluate multiple reward selection algorithms from the multi-armed bandit and online model selection literature: explore-then-commit (ETC), ε-greedy (EG), upper confidence bound (UCB) (Auer, 2002), exponential-weight algorithm for exploration and exploitation (EXP3) (Auer et al., 2002), and doubling data-driven regret balancing (D3RB) (Dann et al., 2024). We provide the pseudocode and the hyperparameters used for each selection algorithm in Appendix G. For every environment, we set the number of iterations N in Algorithm 1 used to train the policy before we select a different reward function to $N = \texttt{n_iters}/100$, where $\texttt{n_iters}$ is the number of iterations used to train the baselines, i.e., we perform at least 100 iterations of online reward selection before the iterative resampling.

5.2 RESULTS

In this section, we present the experimental results of ORSO. We evaluate ORSO's ability to efficiently select reward functions with varying budget constraints and reward function set size K. We consider budgets $B \in \{5, 10, 15\} \times \texttt{n_iters}$ and sample sizes $K \in \{4, 8, 16\}$.

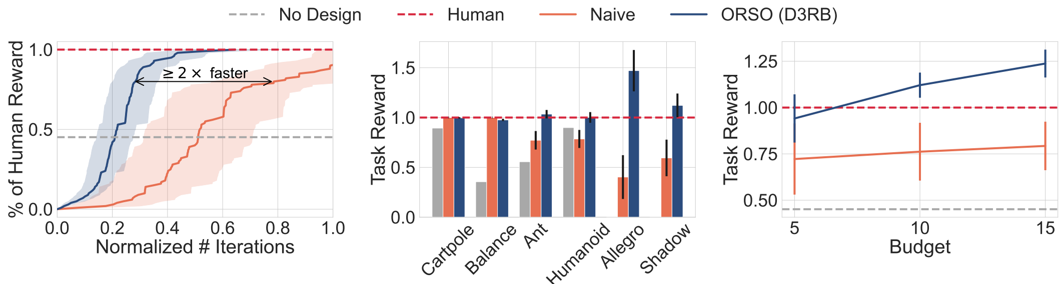


Figure 2: **Left**: ORSO achieves human-level performance in approximately half the time compared to the naive strategy. The curves represent the average percentage of human-designed reward functions across multiple tasks and iteration budget constraints. **Middle**: Normalized task rewards averaged over various iteration budgets and seeds. ORSO consistently matches or surpasses human-designed reward functions. The normalized task reward is averaged over different iteration budgets and seeds. **Right**: Normalized task rewards across different iteration budgets. ORSO effectively scales with increased iterations, with values averaged over multiple tasks and seeds. Shaded areas and vertical bars in the plots indicate 95% confidence intervals.

ORSO is Twice as Fast as the Naive Selection Strategy In Figure 2 (left), we plot the number of iterations required to reach different percentages of the performance achieved by policies trained with human-engineered reward functions. The y-axis represents the percentage of human performance, while the x-axis shows progress in the selection algorithm, normalized so that a value of 1.0 corresponds to $B \times \texttt{n_iters}$ for each task. Results are aggregated across 6 tasks, 3 different budgets, and 3 reward function sets, with 3 seeds per configuration, totaling 162 runs.

[3]An effective reward function is one that leads to high performance with respect to the task reward r when used for training.

We observe that ORSO with D^{3}RB achieves human-level performance more than twice as fast as the naive selection strategy. The naive selection strategy on average does not manage to select an effective reward function within the limited budget. Detailed per-task and per-budget results are reported in Appendix H.

ORSO Surpasses Human-Designed Reward Functions Not only does ORSO reach human-level performance quickly, but it also has the potential to surpass it. Figure 2 (middle) illustrates the average performance of ORSO compared to human-designed reward functions, the task reward function, and the naive selection strategy across different tasks. We observe that ORSO consistently matches or exceeds human-designed rewards, particularly in more complex environments. Again, the results are averaged over multiple seeds and configurations. The full breakdown is reported in Appendix H.

ORSO Scales with Budget Figure 2 (right) demonstrates how ORSO's performance scales with increasing budgets. While both ORSO and naive selection benefit from larger budgets, ORSO is consistently superior and surpasses human-designed rewards when $B \geq 10$.

ORSO Can Reach Human Performance with Fewer GPUs One advantage of the naive selection strategy is that it can be easily parallelized on many GPUs. In Figure 3 we show the estimated time required to achieve the same performance level as policies trained with human-designed reward functions, based on the number of GPUs. Notably, ORSO performs at a comparable speed to the naive selection strategy even when the latter leverages up to 16 GPUs in parallel, achieving similar performance within the same timeframe. It should be noted that the plotted time is an approximation based on the time needed to complete one iteration of PPO for each task. We report the results for all computational budgets in the Appendix H. We also note that one could in principle run ORSO on multiple GPUs in parallel and combie the final results, which would likely lead to further improvements in the efficiency of reward selection.

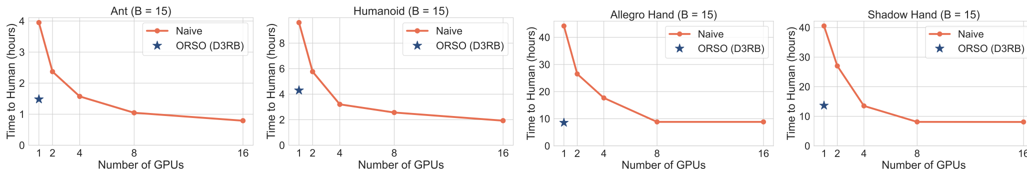


Figure 3: Median time to human-level performance as a function of number of parallel GPUs. Policies trained with ORSO can achieve the same performance as policies trained with the human-engineered reward functions with up to $16\times$ fewer GPUs.

5.3 Ablation Study

Choice of Selection Algorithm In Figure 4, we compare different selection algorithms for ORSO. We find that D^{3}RB performs best on average, consistently outperforming other algorithms, followed closely by Exp3. These algorithms allow ORSO to balance exploration and exploitation effectively, leading to superior performance compared to more greedy approaches like UCB, ETC, and EG. Interestingly, even simpler strategies like EG and ETC substantially outperform the naive strategy, which highlights the importance of properly balancing exploration and exploitation for efficient reward selection. By framing reward design as an exploration-exploitation problem, we demonstrate that even basic strategies offer considerable gains over static, inefficient methods.

Regret of Different Selection Algorithms To further quantify ORSO's performance, we analyze its regret with respect to human-engineered reward functions.[4] This formulation is motivated by two key considerations. First, we lack access to the true optimal policy $\pi^\star$. Second, the PPO hyperparameters used in our experiments were specifically tuned for the human-engineered reward function, making the policy trained with it a reasonable proxy for the optimal policy. Regret provides a useful

[4]The normalized cumulative regret with respect to the human-engineered reward functions is defined as $\frac{1}{T}\sum_{t=1}^{T}\frac{\texttt{Human}-\mathcal{J}(\pi_t^\star)}{\texttt{Human}}$.

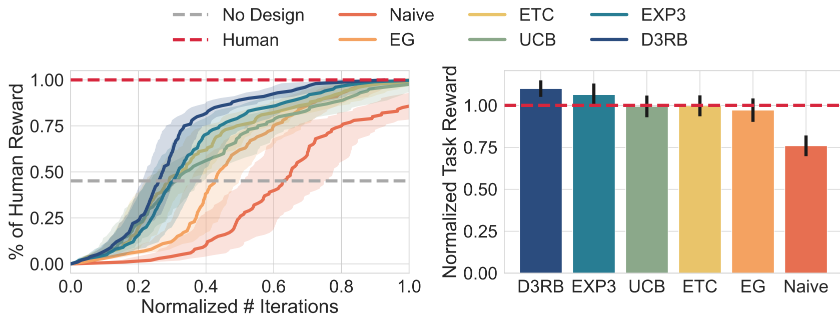


Figure 4: Comparison of different rewards selection algorithms for ORSO. **Left**: Number of iterations necessary for human-level performance. **Right**: Average normalized task reward for different selection algorithms. We provide a more granular breakdown in Appendix H.

metric for understanding how much performance is lost due to suboptimal reward selection over time. Lower regret indicates that ORSO quickly identifies high-quality reward functions, reducing the number of iterations wasted on poorly performing ones. Figure 5 shows the normalized cumulative regret for different selection algorithms. Notably, ORSO's regret can become negative, indicating that it finds reward functions that outperform the human baseline.

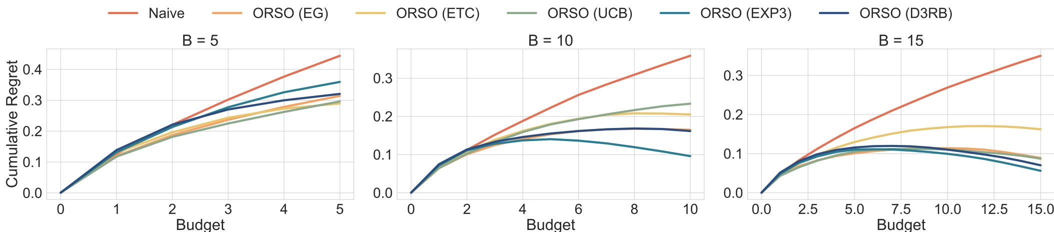


Figure 5: Regret of different selection algorithms with varying budgets. We recall that a budget B indicates that the ORSO has been run for $B \times \texttt{n_iters}$ iterations.

ORSO is Effective with Large Reward Sets We also evaluate ORSO with different selection algorithms when we are provided with a fixed but large set of reward functions. Specifically, we conduct experiments on the ANT task using ORSO with a budget $B = 15$ and candidate shaping reward sets of sizes $K \in \{48, 96\}$.

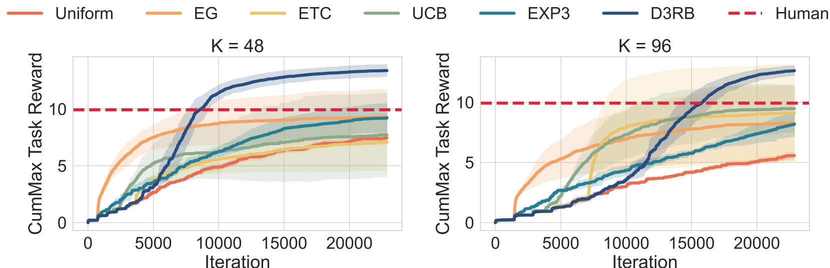


Figure 6: Comparison of multiple selection algorithms for the ANT task with a high number of reward function candidates. The shaded areas represent 95% confidence intervals over 5 different seeds. The order of the reward functions is randomized for each seed.

In this setting – with a fixed budget, but no iterative improvement – algorithms that commit to a selection earlier can allocate more iterations to training on the chosen reward functions. On the

other hand, exploring for longer may allow us to find the optimal reward function but potentially leave insufficient time for training.

As illustrated in Figure 6, D^{3}RB consistently identifies and selects an effective reward function from the set. In contrast, "greedier" methods such as ε-greedy, explore-then-commit, and UCB can depend more on the stochasticity of training and on average do not surpass human-designed reward functions. Exp3 and uniform exploration, while more exploratory, may overemphasize exploration at the expense of exploiting promising reward functions, leading to suboptimal performance.

6 Related Work

Traditionally, researchers manually specified reward components and tuned their coefficients (Ng et al., 1999; Margolis & Agrawal, 2022; Liu et al., 2024), a method that often requires significant domain expertise and involves numerous iterations of trial and error.

Recent work has increasingly explored the potential of foundation models in reward design. Approaches like L2R (Yu et al., 2023) leverage large language models to generate reward functions by converting natural language descriptions into code using predefined reward API primitives, though this requires notable effort in manual template design. Other works such as Eureka (Ma et al., 2024) and Text2Reward (Xie et al., 2024) use language models to generate dense reward functions based on task descriptions and environment codes.

Foundation models have also been directly employed as reward models. Researchers have used cosine similarity of CLIP embeddings (Rocamonde et al., 2024), vision language models for trajectory preference labeling (Wang et al., 2024), and large language models for constructing preference datasets and intrinsic reward modeling (Klissarov et al., 2024; Kwon et al., 2023).

In parallel, research on online model selection has addressed the challenge of dynamically choosing suitable models in sequential decision-making environments (Agarwal et al., 2017; Foster et al., 2019; Pacchiano et al., 2020; Lee et al., 2021).

A more comprehensive review of related work is provided in Appendix A.

7 Conclusion

In this paper, we introduce Orso, a novel approach for reward design in reinforcement learning that significantly accelerates the design of shaped reward functions. We find that even simple strategies like ε-greedy and explore-then-commit yield substantial improvements over naive selection, suggesting that reward design can be effectively framed as a sequential decision problem. Orso reduces both time and computational costs by more than half compared to earlier methods, making reward design accessible to a wider range of researchers. What once required a larger amount of computational resources can not be done on a single desktop in a reasonable time. By formalizing the reward design problem and providing a theoretical analysis of Orso's regret when using the D^{3}RB algorithm, we also contribute to the theoretical understanding of reward design in RL.

Looking ahead, our work opens several promising directions for future research, including the development of more sophisticated exploration strategies tailored for reward design, and the application of our approach to more complex, real-world RL problems.

7.1 Limitations and Future Work

A key limitation of Orso is its reliance on a predefined task reward, which is typically straightforward to construct for simpler tasks but can be challenging for more complex ones or for tasks that include a qualitative element to them, e.g., making a quadruped walk with a "nice" gait. Future work could explore eliminating the need for such hand-crafted task rewards by leveraging techniques that translate natural language instructions directly into evaluators, potentially using vision-language models, similarly to Wang et al. (2024); Rocamonde et al. (2024). Another alternative is to use preference data to learn a task reward model (Christiano et al., 2017; Zhang & Ramponi, 2023) and use the latter as a signal for the model selection algorithm.

540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593

REFERENCES

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, pp. 1, 2004.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Alekh Agarwal, Haipeng Luo, Behnam Neyshabur, and Robert E Schapire. Corralling a band of bandit algorithms. In *Conference on Learning Theory*, pp. 12–38. PMLR, 2017.

Pulkit Agrawal. The task specification problem. In *5th Annual Conference on Robot Learning, Blue Sky Submission Track*, 2021. URL https://openreview.net/forum?id=cBdnThrYkV7.

Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

Eric R Chen, Zhang-Wei Hong, Joni Pajarinen, and Pulkit Agrawal. Redeeming intrinsic rewards via constrained policy optimization. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=36Yz37cEN_Q.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

Chris Dann, Claudio Gentile, and Aldo Pacchiano. Data-driven online model selection with regret guarantees. In *International Conference on Artificial Intelligence and Statistics*, pp. 1531–1539. PMLR, 2024.

Dylan J Foster, Akshay Krishnamurthy, and Haipeng Luo. Model selection for contextual bandits. *Advances in Neural Information Processing Systems*, 32, 2019.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and JoÃĢo GM AraÃšjo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

Martin Klissarov, Pierluca D'Oro, Shagun Sodhani, Roberta Raileanu, Pierre-Luc Bacon, Pascal Vincent, Amy Zhang, and Mikael Henaff. Motif: Intrinsic motivation from artificial intelligence feedback. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=tmBKIecDE9.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=10uNUgI5Kl.

Jonathan Lee, Aldo Pacchiano, Vidya Muthukumar, Weihao Kong, and Emma Brunskill. Online model selection for reinforcement learning with function approximation. In *International Conference on Artificial Intelligence and Statistics*, pp. 3340–3348. PMLR, 2021.

Minghuan Liu, Zixuan Chen, Xuxin Cheng, Yandong Ji, Ri-Zhao Qiu, Ruihan Yang, and Xiaolong Wang. Visual whole-body control for legged loco-manipulation. In *8th Annual Conference on Robot Learning*, 2024. URL https://openreview.net/forum?id=cT2N3p1AcE.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IEduRUO55F.

A Rupam Mahmood, Dmytro Korenkevych, Gautham Vasan, William Ma, and James Bergstra. Benchmarking reinforcement learning algorithms on real-world robots. In *Conference on robot learning*, pp. 561–591. PMLR, 2018.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance GPU based physics simulation for robot learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=fgFBtYgJQX_.

Gabriel B. Margolis and Pulkit Agrawal. Walk these ways: Tuning robot control for generalization with multiplicity of behavior. In *6th Annual Conference on Robot Learning*, 2022. URL https://openreview.net/forum?id=52c5e73SlS2.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pp. 278–287, 1999.

Aldo Pacchiano, My Phan, Yasin Abbasi Yadkori, Anup Rao, Julian Zimmert, Tor Lattimore, and Csaba Szepesvari. Model selection in contextual stochastic bandit problems. *Advances in Neural Information Processing Systems*, 33:10328–10337, 2020.

Younghyo Park, Gabriel B. Margolis, and Pulkit Agrawal. Position: Automatic environment shaping is the next frontier in RL. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=dslUyy1rN4.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Juan Rocamonde, Victoriano Montesinos, Elvis Nava, Ethan Perez, and David Lindner. Vision-language models are zero-shot reward models for reinforcement learning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=N0I2RtD8je.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Yufei Wang, Zhanyi Sun, Jesse Zhang, Zhou Xian, Erdem Biyik, David Held, and Zackory Erickson. RL-VLM-f: Reinforcement learning from vision language foundation model feedback. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=YSoMmNWZZx.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=tUM39YTRxH.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montserrat Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to rewards for robotic skill synthesis. In *7th Annual Conference on Robot Learning*, 2023. URL https://openreview.net/forum?id=SgTPdyehXMA.

Chen Bo Calvin Zhang and Giorgia Ramponi. HIP-RL: Hallucinated inputs for preference-based reinforcement learning in continuous domains. In *ICML 2023 Workshop The Many Facets of Preference-Based Learning*, 2023. URL https://openreview.net/forum?id=PRm1KxRrWI.

Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, Anind K Dey, et al. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pp. 1433–1438. Chicago, IL, USA, 2008.

A RELATED WORK

Reward Design for RL Designing effective reward functions for reinforcement learning has been a long-standing challenge. Several approaches have been proposed to tackle it.

Traditionally, researchers manually specify reward components and tune their coefficients (Ng et al., 1999; Margolis & Agrawal, 2022; Liu et al., 2024). This method often demands significant domain expertise and can be highly resource-intensive, involving numerous iterations of trial and error in designing reward functions, training policies, and adjusting reward parameters.

Another approach is to learn reward functions from expert demonstrations via methods like apprenticeship learning (Abbeel & Ng, 2004) and maximum entropy inverse RL (Ziebart et al., 2008). While these methods can capture complex behaviors, they often rely on high-quality demonstrations and may struggle in environments where such data is scarce or noisy.

Preferences can also be used to learn reward functions (Zhang & Ramponi, 2023; Christiano et al., 2017). This approach involves collecting feedback in the form of preferences between different trajectories, which are then used to infer a reward function that aligns with the desired behavior. This method is particularly useful in scenarios where it is difficult to explicitly define a reward function or obtain expert demonstrations, as it allows for more intuitive and accessible feedback from users.

Foundation Models and Reward Functions Recent work has explored the use of large language/vision models (LL/VMs) to aid in the reward design process. L2R (Yu et al., 2023) leverages large language models to generate reward functions for RL tasks by first creating a natural language "motion description" and then converting it into code using predefined reward API primitives. While innovative, L2R has notable limitations: it requires significant manual effort in designing templates and primitives and is constrained by the latter. EUREKA (Ma et al., 2024) and Text2Reward (Xie et al., 2024) use LLMs to generate dense reward functions for RL given the task description in natural language and the environment code.

Foundation models have also been directly used as reward models. Rocamonde et al. (2024) uses the cosine similarity of CLIP embeddings of language instructions and renderings of the state as a state-only reward model. Similarly, Wang et al. (2024) automatically generates reward functions for RL using a vision language model to label pairs of trajectories with preference, given a task description. Motif (Klissarov et al., 2024) first constructs a pair-wise preferences dataset using a large language model (LLM), learns a preference-based intrinsic reward model with the Bradley-Terry (Bradley & Terry, 1952) model, and then uses this reward model to train a reinforcement learning agent. Kwon et al. (2023) uses a similar approach, where an LLM is used during training to evaluate an RL policy, given a few examples of successful behavior or a description of the desired behavior.

Online Model Selection The problem of model selection in sequential decision-making environments has gained significant attention in recent years (Agarwal et al., 2017; Foster et al., 2019; Pacchiano et al., 2020; Lee et al., 2021). This area of research addresses the challenge of dynamically choosing the most suitable model or algorithm from a set of candidates while learning.

Agarwal et al. (2017) introduced CORRAL, a method to combine multiple bandit algorithms in a master algorithm. Foster et al. (2019) proposed model selection guarantees for linear contextual bandits. Pacchiano et al. (2020) extend the CORRAL algorithm and propose Stochastic CORRAL. Lastly, Lee et al. (2021) propose Explore-Commit-Eliminate (ECE), an algorithm for model selection in RL with function approximation. A common requirement across all these approaches is the need to know the regret guarantees of the base algorithms.

Our work is closely related to Dann et al. (2024), which removes the need for known regret guarantees and instead uses *realized* regret bounds for the base learners. In our setting, the set of models comprises the reward functions set and their corresponding policies.

702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755

B ONLINE MODEL SELECTION

In this section, we introduce the model selection problem and some necessary notation modified from Dann et al. (2024) for our analysis.

We consider a general sequential decision-making process consisting of a *meta learner* interacting with an environment over $T \in \mathbb{N}$ rounds via a set of *base learners*. At each round of interaction $t = 1, 2, \ldots, T$, the meta learner selects a base learner b_t and after executing b_t, the environment returns a model selection reward $R_t \in \mathbb{R}$. The objective of the meta learner is to sequentially choose base learners $b_1, \ldots, b_T$ to maximize the expected cumulative sum of model selection rewards, i.e., $\max \mathbb{E}\left[\sum_{t=1}^T R_t\right]$. We denote by $v^b = \mathbb{E}[R \mid b]$ the expected model selection reward, given that the learner chooses base learner b, i.e., the value of base learner b. The total model selection reward accumulated by the algorithm over T rounds is denoted by $u_T = \sum_{t=1}^T v^{b_t}$. The objective is to minimize the cumulative regret after T rounds of interaction,

$$\mathrm{Reg}(T) \coloneqq \sum_{t=1}^{T} \mathrm{reg}(b_t) = \sum_{t=1}^{T} v^\star - v^{b_t}, \tag{5}$$

where $v^\star$ is the value of the optimal base learner.

In our setting, each base learner corresponds to a reward function r and its associated policy π, i.e., $b = (f, \pi)$. In this case, choosing to execute base learner b means training with algorithm $\mathfrak{A}$ starting from checkpoint π and using RL reward function f. The model selection reward R is then the evaluation of the trained policy under the task reward r, i.e., $\mathcal{J}(\pi)$. The regret of base learner b can therefore be written as $\mathrm{reg}(b) = v^\star - v^b = \mathcal{J}(\pi^\star) - \mathcal{J}(\pi)$, where $\pi^\star$ is the optimal policy. Therefore the objective becomes minimizing

$$\mathrm{Reg}(T) = \sum_{t=1}^{T} \mathcal{J}(\pi^\star) - \mathcal{J}\left(\pi_{n_t^i}^{f^{i_t}}\right). \tag{6}$$

We note an important difference between the online model selection problem and the multi-armed bandit (MAB) problem. In model selection, the meta learner interacts with an environment over T rounds, selecting from K base learners. In each round t, the meta learner picks a base learner $i_t \in [K]$ (index of base learner chosen at step t) and follows its policy, updating the base learner's state with new data. Unlike MAB problems, where mean rewards are stationary, the mean rewards here are non-stationary due to the stateful nature of base learners (the base learners are learning as they see more data), making the design of effective model selection algorithms challenging.

Notation The policy associated with base learner i at round t is denoted by π_t^i, so that $\pi_t = \pi_t^{i_t}$. We denote the number base learner i has been played up to round t as $n_t^i = \sum_{\ell=1}^t \mathbb{1}\{i_\ell = i\}$ and the total cumulative reward for learner i as $u_t^i = \sum_{\ell=1}^t \mathbb{1}\{i_\ell = i\} v^{\pi_t^i}$, where we use $v^{\pi_t^i} = v^{b_t^i}$ to highlight that the policy associated with base learner i changes over time, but the reward function used for RL does not. We denote the internal clock for each base learner with a subscript (k) such that $\pi_{(k)}^i$ is the policy of learner i when chosen for the k-th time, i.e., $\pi_t^i = \pi_{(n_t^i)}^i$.

Remark 1. *The cumulative regret in Equation* (6) *is an upper bound for the model selection cumulative regret.*

Proof. This is straightforward to see. Let us first note that, by definition, for all $t \in [T]$, we have

$$\mathcal{J}(\pi_t^\star) \geq \mathcal{J}\left(\pi_{n_t^i}^{f^{i_t}}\right). \tag{7}$$

Therefore,

$$\sum_{t=1}^{T} \mathcal{J}(\pi^\star) - \mathcal{J}(\pi_t^\star) \leq \sum_{t=1}^{T} \mathcal{J}(\pi^\star) - \mathcal{J}\left(\pi_{n_t^i}^{f^{i_t}}\right), \tag{8}$$

i.e.,

$$\mathrm{MReg}(T) \leq \mathrm{Reg}(T), \tag{9}$$

concluding the proof. $\square$

C ORSO WITH DOUBLING DATA-DRIVEN REGRET BALANCING

Here, we present the complete ORSO algorithm with Doubling Data-Driven Regret Balancing (D^3RB) as the model selection algorithm.

D^3RB is built upon the idea of *regret balancing*, which aims to optimize the performance of multiple models by balancing their respective regrets. Imagine weighing two models on a balance scale where the "weight" corresponds to their regret; the goal is to keep the regret of both models balanced. This approach ensures that models with higher regret rates are selected less frequently, while those with lower regret rates are favored.

Concretely, regret balancing involves associating each learner with a candidate regret bound. The model selection algorithm then competes against the regret bound of the best-performing learner among those that are well-specified – meaning their realized regret stays within their candidate bounds. Traditional approaches often rely on known *expected* regret bounds. In contrast, D^3RB focuses on *realized* regret, allowing the model selection algorithm to compete based on the actual regret outcomes of each base learner. The algorithm dynamically adjusts the regret bounds in a data-driven manner, adapting to the realized regret of the best-performing learner over time. This approach overcomes the limitation of needing known regret bounds, which are often unavailable for complex problems.

D^3RB maintains three estimators for each base learner: regret coefficients $\widehat{d}_t^i$, average rewards $\widehat{u}_t^i / n_t^i$ and balancing potentials ϕ_t^i. At each step t, D^3RB selects the base learner with the lower balancing potential and executes it. Then it performs the misspecification test in Equation (10) to check if the estimated regret coefficient for base learner i_t is consistent with the observed data. If the test triggers, i.e., the $\widehat{d}^{i_t}$ is too small, then the algorithm doubles it. Lastly, D^3RB sets the balancing potential ϕ_t^i to $\widehat{d}_t^{i_t} \sqrt{n_t^{i_t}}$.

$$\frac{\widehat{u}_t^{i_t}}{n_t^{i_t}} + \frac{\widehat{d}_{t-1}^{i_t} \sqrt{n_t^{i_t}}}{n_t^{i_t}} + c\sqrt{\frac{\ln \frac{K \ln n_t^{i_t}}{\delta}}{n_t^{i_t}}} < \max_{j \in [K]} \frac{\widehat{u}_t^j}{n_t^j} - c\sqrt{\frac{\ln \frac{K \ln n_t^j}{\delta}}{n_t^j}} \tag{10}$$

Algorithm 2 ORSO with D^{3}RB

Require: MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \rho_0)$, algorithm $\mathfrak{A}$, generator G, minimum regret coefficients $d_{\min}$, failure probability δ

1: Sample K reward functions $\mathcal{R}^K = \left\{f^1, \ldots, f^K\right\} \sim G$
2: Initialize K policies $\left\{\pi^1, \ldots, \pi^K\right\}$
3: Initialize balancing potentials $\phi_1^i = d_{\min}$ for all $i \in [K]$
4: Initialize regret coefficients $\widehat{d}_0^i = d_{\min}$ for add $i \in [K]$
5: Initialize counts $n_0^i = 0$ and total values $\widehat{u}_0^i = 0$ for all $i \in [K]$
6: **for** $t = 1, 2, \ldots, T$ **do**
7: Select a base learner $i_t \in [K] \in \arg\min_{i \in [K]} \phi_t^i$
8: Update $\pi^{i_t} \leftarrow \mathfrak{A}_{f^{i_t}}(\mathcal{M}_{i_t}, N, \pi^{i_t})$
9: Evaluate $R_t = \mathcal{J}\left(\pi^{i_t}\right) \leftarrow \mathtt{Eval}(\pi^{i_t})$
10: `// Update necessary variables`
11: Set $n_t^i = n_{t-1}^i, \widehat{u}_t^i = \widehat{u}_{t-1}^i, \widehat{d}_t^i = \widehat{d}_{t-1}^i$, and $\phi_{t+1}^i = \phi_t^i$ for all $i \in [K] \setminus \{i_t\}$
12: Update statistics for current learner $n_t^{i_t} = n_{t-1}^{i_t} + 1$ and $\widehat{u}_t^{i_t} = \widehat{u}_{t-1}^{i_t} + R_t$
13: Perform misspecification test

$$\frac{\widehat{u}_t^{i_t}}{n_t^{i_t}} + \frac{\widehat{d}_{t-1}^{i_t}\sqrt{n_t^{i_t}}}{n_t^{i_t}} + c\sqrt{\frac{\ln \frac{K \ln n_t^{i_t}}{\delta}}{n_t^{i_t}}} < \max_{j \in [K]} \frac{\widehat{u}_t^j}{n_t^j} - c\sqrt{\frac{\ln \frac{K \ln n_t^j}{\delta}}{n_t^j}} \tag{11}$$

14: **if** test is triggered **then**
15: Double the regret coefficient $\widehat{d}_t^{i_1} \leftarrow 2\widehat{d}_{t-1}^{i_t}$
16: **else**
17: Keep the regret coefficient unchanged $\widehat{d}_t^{i_1} \leftarrow \widehat{d}_{t-1}^{i_t}$
18: **end if**
19: Update the balancing potential $\phi_{t+1}^{i_t} \leftarrow \widehat{d}_t^{i_t}\sqrt{n_t^{i_t}}$
20: **end for**
21: `// Best policy and reward function under the task reward`
22: **return** $\pi_T^\star, f_T^\star = \arg\max_{i \in [K]} \mathcal{J}(\pi^i)$

D PROOF OF LEMMA 4.4

In this section, we present the complete proof of Lemma 4.4. We will start by showing that when Assumption 4.2 holds, then with probability at least $1-\delta$, the estimated regret coefficient of learner $i_\star$ will never double provided that $d_{\min} \geq c$, where c is the confidence multiplier in D^{3}RB.

Lemma D.1 (Non-doubling regret coefficient)**.** *When $\mathcal{E}$ holds, and algorithm D^{3}RB is in use*

$$\widehat{d}_t^{i_\star} = d_{\min} \quad \textit{and} \quad n_T^i \leq n_T^{i_\star} + 1 \quad \textit{for all } i \in [K] \tag{12}$$

for all $t \in \mathbb{N}$.

Proof. In order to show this result it is sufficient to show that when $\mathcal{E}$ holds, algorithm $i_\star$ does not undergo any doubling event. Doubling of the regret coefficients only happens when the misspecification test triggers for algorithm $i_\star$.

We will show this by induction.

Base Case ($t = 1$) At $t = 1$, for all algorithms $i \in [K]$:

- $\widehat{d}_1^i = d_{\min}$ (by initialization)
- $n_1^i = 1$ if i is the first algorithm chosen, 0 otherwise

Therefore $n_1^i \leq n_1^{i_\star} + 1$ holds

Inductive Step Inductive hypothesis: assume that for some $t \geq 1$:

- $\widehat{d}_{t-1}^{i_\star} = d_{\min}$
- $n_{t-1}^i \leq n_{t-1}^{i_\star} + 1$ for all $i \in [K]$

We need to show these properties hold for t. Let $i_t = i_\star$. When $\mathcal{E}$ holds, the left-hand side (LHS) of D^{3}RB's misspecification test satisfies

$$\begin{aligned}
\frac{\widehat{u}_t^{i_t}}{n_t^{i_t}} + \frac{\widehat{d}_{t-1}^{i_t}\sqrt{n_t^{i_t}}}{n_t^{i_t}} + c\sqrt{\frac{\ln\frac{K\ln n_t^{i_t}}{\delta}}{n_t^{i_t}}} &= \frac{\widehat{u}_t^{i_\star}}{n_t^{i_\star}} + \frac{\widehat{d}_{t-1}^{i_\star}\sqrt{n_t^{i_\star}}}{n_t^{i_\star}} + c\sqrt{\frac{\ln\frac{K\ln n_t^{i_\star}}{\delta}}{n_t^{i_\star}}} && (i_t = i_\star)\\
&\geq \frac{u_t^{i_\star}}{n_t^{i_\star}} + \frac{\widehat{d}_{t-1}^{i_\star}\sqrt{n_t^{i_\star}}}{n_t^{i_\star}} && (\text{event } \mathcal{E})\\
&\overset{(i)}{=} \frac{u_t^{i_\star}}{n_t^{i_\star}} + \frac{d_{\min}\sqrt{n_t^{i_\star}}}{n_t^{i_\star}} && (13)
\end{aligned}$$

where (i) holds because by the induction hypothesis $\widehat{d}_{t-1}^{i_\star} = d_{\min}$. We will now show that $n_t^{i_\star} \geq n_t^j$ for all $j \in [K]$. Since by the inductive hypothesis $\widehat{d}_\ell^{i_\star} = d_{\min}$ for all $\ell \leq t-1$, the potential $\phi_\ell^{i_\star} = d_{\min}\sqrt{n_{\ell-1}^{i_\star}}$ for all $\ell \leq t$.

For $i \in [K]$ let $t(i)$, be the last time – before time t – algorithm i was played. For $i \neq i_\star$ we have $t(i) < t$. Since i was selected at time $t(i)$, by definition of the potentials,

$$\widehat{d}_{t(i)-1}^{i_\star}\sqrt{n_{t(i)-1}^{i_\star}} = d_{\min}\sqrt{n_{t(i)-1}^{i_\star}} \geq \widehat{d}_{t(i)-1}^{i}\sqrt{n_{t(i)-1}^{i}} \geq d_{\min}\sqrt{n_{t(i)-1}^{i}}$$

so that $n_{t(i)-1}^{i_\star} \geq n_{t(i)-1}^i$. Since both $n_t^{i_\star} = n_{t(i)-1}^{i_\star} + 1$ and $n_t^i = n_{t(i)-1}^i + 1$ we conclude that $n_t^{i_\star} \geq n_t^i$.

We now turn our attention to the right-hand side (RHS) of D[3]RB's misspecification test. When $\mathcal{E}$ holds, the RHS of D[3]RB's misspecification test satisfies,

$$\max_{j\in[K]} \frac{\widehat{u}_t^j}{n_t^j} - c\sqrt{\frac{\ln \frac{K \ln n_t^j}{\delta}}{n_t^j}} \leq \max_{j\in[K]} \frac{u_t^j}{n_t^j}$$

$$\stackrel{(i)}{\leq} \max_{j\in[K]} \frac{u^{i_\star}_{(n_t^j)}}{n_t^j}$$

$$\stackrel{(ii)}{\leq} \frac{u_t^{i_\star}}{n_t^{i_\star}} \tag{14}$$

where inequalities (i) and (ii) hold because of Assumption 4.2. Combining inequalities 13 and 14 we conclude the misspecification test of algorithm D[3]RB will not trigger. Thus, $\widehat{d}_t^{i_\star}$ remains at $d_{\min}$ and for all $i \in [K]$, $n_t^i \leq n_t^{i_\star} + 1$ continues to hold. This finalizes the proof. ☐

We are now ready to prove the regret bound on the base learners given in Lemma 4.4.

Lemma 4.4. *Under event $\mathcal{E}$ and Assumption 4.2, with probability $1-\delta$, the regret of all learners i is bounded in all rounds T as*

$$\sum_{t=1}^{n_T^i} \mathrm{reg}(\pi^i_{(t)}) \leq 6 d_T^{i_\star} \sqrt{n_T^{i_\star} + 1} + 5c\sqrt{(n_T^{i_\star} + 1) \ln \frac{K \ln T}{\delta}}, \tag{4}$$

where $d_T^{i_\star} = d^{i_\star}_{(n_T^{i_\star})}$.

Proof. Consider a fixed base learner i and time horizon T, and let $t \leq T$ be the last round where i was played but the misspecification test did not trigger. If no such round exists, then set $t = 0$. By Corollary 9.1 in Dann et al. (2024), i can be played at most $1 + \log_2 \frac{\bar{d}_T^i}{d_{\min}}$ times between t and T, where $\bar{d}_T^i = \max_{\ell \leq T} d_\ell^i$. Thus,

$$\sum_{k=1}^{n_T^i} \mathrm{reg}\left(\pi^i_{(k)}\right) \leq \sum_{k=1}^{n_t^i} \mathrm{reg}\left(\pi^i_{(k)}\right) + 1 + \log_2 \frac{\bar{d}_T^i}{d_{\min}}.$$

If $t = 0$, then the desired statement holds. Thus, it remains to bound the first term in the RHS above when $t > 0$. Since $i = i_t$ and the test did not trigger we have, for any base learner j with $n_t^j > 0$,

$$\begin{aligned}
\sum_{k=1}^{n_t^i} \mathrm{reg}\left(\pi^i_{(k)}\right) &= n_t^i v^\star - u_t^i && \text{(definition of regret)}\\
&= n_t^i v^\star - \frac{n_t^i}{n_t^j} u_t^j + \frac{n_t^i}{n_t^j} u_t^j - u_t^i \\
&= \frac{n_t^i}{n_t^j}\left(n_t^j v^\star - u_t^j\right) + \frac{n_t^i}{n_t^j} u_t^j - u_t^i \\
&= \frac{n_t^i}{n_t^j}\left(\sum_{k=1}^{n_t^j} \mathrm{reg}\left(\pi^j_{(k)}\right)\right) + \frac{n_t^i}{n_t^j} u_t^j - u_t^i && \text{(definition of regret)}\\
&\leq \frac{n_t^i}{n_t^j}\left(d_t^j \sqrt{n_t^j}\right) + \frac{n_t^i}{n_t^j} u_t^j - u_t^i && \text{(definition of regret rate)}\\
&= \sqrt{\frac{n_t^i}{n_t^j}} d_t^j \sqrt{n_t^i} + \frac{n_t^i}{n_t^j} u_t^j - u_t^i.
\end{aligned}$$

We now focus on $j = i_\star$ and use the balancing condition in Lemma 9.2 in Dann et al. (2024) to bound the first factor $\sqrt{n_t^i / n_t^{i_\star}}$. This condition gives that $\phi_{t+1}^i \leq 3\phi_{t+1}^{i_\star}$. Since both $n_t^{i_\star} > 0$ and $n_t^i > 0$, we have $\phi_{t+1}^i = \widehat{d}_t^i \sqrt{n_t^i}$ and $\phi_{t+1}^{i_\star} = \widehat{d}_t^{i_\star} \sqrt{n_t^{i_\star}}$. Thus, we get

$$\sqrt{\frac{n_t^i}{n_t^{i_\star}}} = \sqrt{\frac{n_t^i}{n_t^{i_\star}}} \cdot \frac{\widehat{d}_t^i}{\widehat{d}_t^{i_\star}} \cdot \frac{\widehat{d}_t^{i_\star}}{\widehat{d}_t^i} = \frac{\phi_{t+1}^i}{\phi_{t+1}^{i_\star}} \cdot \frac{\widehat{d}_t^{i_\star}}{\widehat{d}_t^i} \leq 3 \frac{\widehat{d}_t^{i_\star}}{\widehat{d}_t^i} \leq 3, \tag{15}$$

where the last inequality holds because of Lemma D.1 and because $\widehat{d}_t^i \geq d_{\min}$.

Plugging this back into the expression above and setting $j = i_\star$, we have

$$\sum_{k=1}^{n_t^i} \operatorname{reg}\left(\pi_{(k)}^i\right) \leq 3 d_t^{i_\star} \sqrt{n_t^i} + \frac{n_t^i}{n_t^{i_\star}} u_t^{i_\star} - u_t^i.$$

To bound the last two terms, we use the fact that the misspecification test did not trigger in round t. Therefore,

$$\begin{aligned}
u_t^i &\geq \widehat{u}_t^i - c\sqrt{n_t^i \ln \frac{K \ln n_t^i}{\delta}} && \text{(event } \mathcal{E}\text{)} \\
&= n_t^i \left(\frac{\widehat{u}_t^i}{n_t^i} + c\sqrt{\frac{\ln \frac{K \ln n_t^i}{\delta}}{n_t^i}} + \frac{\widehat{d}_t^i \sqrt{n_t^i}}{n_t^i} \right) - 2c\sqrt{n_t^i \ln \frac{K \ln n_t^i}{\delta}} - \widehat{d}_t^i \sqrt{n_t^i} \\
&\geq \frac{n_t^i}{n_t^{i_\star}} \widehat{u}_t^{i_\star} - \sqrt{\frac{n_t^i}{n_t^{i_\star}}} c\sqrt{n_t^i \ln \frac{K \ln n_t^{i_\star}}{\delta}} - 2c\sqrt{n_t^i \ln \frac{K \ln n_t^i}{\delta}} - \widehat{d}_t^i \sqrt{n_t^i}. && \text{(test not triggered)}
\end{aligned}$$

Rearranging terms and plugging this expression in the bound above gives

$$\begin{aligned}
\sum_{k=1}^{n_t^i} \operatorname{reg}(\pi_{(k)}^i) &\leq 3 d_t^{i_\star} \sqrt{n_t^i} + \sqrt{\frac{n_t^i}{n_t^{i_\star}}} c\sqrt{n_t^i \ln \frac{K \ln n_t^{i_\star}}{\delta}} + 2c\sqrt{n_t^i \ln \frac{K \ln n_t^i}{\delta}} + \widehat{d}_t^i \sqrt{n_t^i} \\
&\leq 3 d_t^{i_\star} \sqrt{n_t^i} + 3c\sqrt{n_t^i \ln \frac{K \ln n_t^{i_\star}}{\delta}} + 2c\sqrt{n_t^i \ln \frac{K \ln n_t^i}{\delta}} + \widehat{d}_t^i \sqrt{n_t^i} && \text{(Equation (15))} \\
&\leq 3 d_t^{i_\star} \sqrt{n_t^i} + 3c\sqrt{n_t^i \ln \frac{K \ln n_t^{i_\star}}{\delta}} + 2c\sqrt{n_t^i \ln \frac{K \ln n_t^i}{\delta}} + 3\widehat{d}_t^{i_\star} \sqrt{n_t^{i_\star}} \\
& && \text{(Equation (15))} \\
&\leq 3 d_t^{i_\star} \sqrt{n_t^i} + 3\widehat{d}_t^{i_\star} \sqrt{n_t^{i_\star}} + 5c\sqrt{n_t^i \ln \frac{K \ln t}{\delta}} && (\max(n_t^i, n_t^{i_\star}) \leq t) \\
&\leq 3 d_t^{i_\star} \sqrt{n_t^i} + 3 d_{\min} \sqrt{n_t^{i_\star}} + 5c\sqrt{n_t^i \ln \frac{K \ln t}{\delta}} && \text{(Lemma D.1)}
\end{aligned}$$

Finally, Lemma D.1 also implies $n_t^i \leq n_t^{i_\star} + 1$ and since $d_{\min} \leq d_t^{i_\star}$,

$$\sum_{k=1}^{n_t^i} \operatorname{reg}(\pi_{(k)}^i) \leq 6 d_t^{i_\star} \sqrt{n_t^{i_\star} + 1} + 5c\sqrt{(n_t^{i_\star} + 1) \ln \frac{K \ln t}{\delta}}.$$

The statement follows by setting $t = T$. $\square$

1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079

E REWARD FUNCTIONS DEFINITIONS

In this section, we present the definition of the human-engineered reward functions and the task reward functions used to evaluate the generated reward in Table 1. The task reward functions are the same as the ones used in Ma et al. (2024).

Table 1: Task reward functions definitions.

ENVIRONMENT	TASK REWARD
CARTPOLE	$\sum \mathbb{1}\{\texttt{agent is alive}\}$
BALLBALANCE	$\sum \mathbb{1}\{\texttt{agent is alive}\}$
ANT	$\texttt{current_distance - previous_distance}$
HUMNAOID	$\texttt{current_distance - previous_distance}$
ALLEGROHAND	$\sum \mathbb{1}\{\texttt{rotation_distance < 0.1}\}$
SHADOWHAND	$\sum \mathbb{1}\{\texttt{rotation_distance < 0.1}\}$

The human-designed reward functions from (Makoviychuk et al., 2021) are

- CARTPOLE

$$r = \left(1.0 - \texttt{pole_angle}^2 - 0.01 \cdot |\texttt{cart_vel}| - 0.005 \cdot |\texttt{pole_vel}|\right).$$

The reward is additionally multiplied by -2.0 if $|\texttt{cart_pos}| > \texttt{reset_dist}$ and multiplied by -2.0 once again if $\texttt{pole_angle} > \frac{\pi}{2}$.

- BALLBALANCE

$$r = \texttt{pos_reward} \times \texttt{speed_reward} = \frac{1}{1 + \texttt{ball_dist}} \times \frac{1}{1 + \texttt{ball_speed}},$$

where

$$\texttt{ball_dist} = \sqrt{\texttt{ball_pos_x}^2 + \texttt{ball_pos_y}^2 + (\texttt{ball_pos_z} - 0.7)^2},$$

where 0.7 is the desired height above the ground, and

$$\texttt{ball_speed} = \|\texttt{ball_velocity}\|_2 .$$

- ANT and HUMNAOID

$$\begin{aligned} r = r_{\texttt{progress}} &+ r_{\texttt{alive}} \times \mathbb{1}\{\texttt{torso_height} \geq \texttt{termination_height}\} + r_{\texttt{upright}} \\ &+ r_{\texttt{heading}} + r_{\texttt{effort}} + r_{\texttt{act}} + r_{\texttt{dof}} \\ &+ r_{\texttt{death}} \times \mathbb{1}\{\texttt{torso_height} \leq \texttt{termination_height}\}, \end{aligned}$$

where

$$\begin{aligned} r_{\texttt{progress}} &= \texttt{current_potential - previous_potential} \\ r_{\texttt{upright}} &= \langle \texttt{torso_up_vector}, \texttt{up_vector} \rangle > 0.93 \\ r_{\texttt{heading}} &= \texttt{heading_vector} \times \begin{cases} 1.0, & \text{if } \texttt{norm_angle_to_target} \geq 0.8 \\ \frac{\texttt{norm_angle_to_target}}{0.8}, & \text{otherwise} \end{cases} \\ r_{\texttt{act}} &= -\sum \|\texttt{actions}\|^2 \\ r_{\texttt{effort}} &= \sum_{i=1}^{N} \texttt{actions}_i \times \texttt{normalized_motor_strength}_i \times \texttt{dof_velocity}_i \end{aligned}$$

$$\texttt{potential} = -\frac{\|p_{\texttt{target}} - p_{\texttt{torso}}\|_2}{dt}$$

- ALLEGROHAND and SHADOWHAND

$$r = -10 r_{\texttt{dist}} + r_{\texttt{rot}} - 2 \times 10^{-4} r_{\texttt{act}}$$

where

$$r_{\texttt{dist}} = \|p_{\texttt{obj}} - p_{\texttt{target}}\|_2$$
$$r_{\texttt{rot}} = \frac{1}{|\texttt{rot_dist}| + 0.1}$$
$$r_{\texttt{act}} = \sum \|\texttt{actions}\|^2$$
$$\texttt{rot_dist} = 2 \times \arcsin\left(\max\left(1, \|q_{\texttt{obj}}, \overline{q_{\texttt{target}}}\|_2\right)\right)$$

where q is the quaternion and $\overline{q}$ is its conjugate.

E.1 Reward Functions Selected by ORSO

We report the best reward function selected by ORSO below. The reward functions are reported as is, with only the formatting of comments and spacing changed to fit within the box.

Reward Function for Allegro Hand

```
def compute_gpt_reward(
    object_rot: torch.Tensor,
    goal_rot: torch.Tensor,
    shadow_hand_dof_pos: torch.Tensor,
    shadow_hand_dof_vel: torch.Tensor,
    actions: torch.Tensor
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:

    # Configurable parameters
    dist_reward_scale = float(2.0)
    action_penalty_scale = float(0.05)
    success_tolerance = float(0.05)
    reach_goal_bonus = float(20.0)

    # Compute distance to goal rotation using Quaternion distance
    q_diff = object_rot - goal_rot
    dist_to_goal = torch.norm(q_diff, dim=-1)

    # Rotation distance reward (scaled)
    rot_reward = torch.exp(-dist_reward_scale * dist_to_goal)

    # Action penalty (scaled)
    action_penalty = torch.sum(actions**2, dim=-1)
    action_penalty_scaled = action_penalty_scale * action_penalty

    # Check if the goal has been reached within the tolerance
    success_mask = dist_to_goal < success_tolerance
    goal_bonus = torch.where(
        success_mask,
        torch.tensor(reach_goal_bonus, device=dist_to_goal.device),
        torch.tensor(0.0, device=dist_to_goal.device)
    )

    # Total reward
    reward = rot_reward - action_penalty_scaled + goal_bonus

    # Dictionary of individual reward components
    reward_components = {
        "rot_reward": rot_reward,
        "action_penalty": action_penalty_scaled,
        "goal_bonus": goal_bonus
    }

    return reward, reward_components
```

Reward Function for Ant

```
def compute_gpt_reward(
    root_states: torch.Tensor,
    actions: torch.Tensor,
    dt: float
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Device
    device = root_states.device

    # Extract necessary information from the root states
    velocity = root_states[:, 7:10]  # [vx, vy, vz]
    torso_position = root_states[:, 0:3]  # [px, py, pz]

    # Forward velocity along the x-axis
    forward_velocity = velocity[:, 0]

    # Reward component: scaled forward velocity
    # Retain existing scaling factor
    forward_reward = forward_velocity * 2.0

    # Penalty for large actions
    # (to avoid unnecessary or jerky movements)
    action_penalty = torch.sum(actions**2, dim=-1)
    # Increased scaling factor for more impact
    action_penalty_scaled = action_penalty * 1.0

    # Desired height range (e.g., 0.45 to 0.55)
    target_height = torch.tensor(0.5, device=device)
    height_diff = torch.abs(torso_position[:, 2] - target_height)
    # Adjusted temperature parameter to increase contribution
    balance_temperature = 0.1
    # Retain existing scaling
    balance_reward = torch.exp(-height_diff/balance_temperature) * 5.0

    # Additional penalty for deviation from target angle
    # (to encourage running straight)
    target_angle = torch.tensor(0.0, device=device)
    # Assuming index 5 is yaw angle
    angle_diff = torch.abs(root_states[:, 5] - target_angle)
    angle_penalty = -torch.exp(-angle_diff / balance_temperature)

    # Survival bonus to encourage longer episode lengths
    # Reduced overall magnitude
    survival_bonus = torch.ones_like(forward_velocity) * 0.5

    # Total reward calculation
    reward = forward_reward + balance_reward +
        angle_penalty - action_penalty_scaled +
        survival_bonus

    # Dictionary of individual reward components for debugging
    reward_components = {
        'forward_reward': forward_reward,
        'action_penalty_scaled': -action_penalty_scaled,
        'balance_reward': balance_reward,
        'angle_penalty': angle_penalty,
        'survival_bonus': survival_bonus
    }

    return reward, reward_components
```

1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295

Reward Function for Ball Balance

```
def compute_gpt_reward(
    ball_positions: torch.Tensor,
    ball_linvels: torch.Tensor
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    """
    Compute the reward for keeping the ball on the table top
    without falling.

    Args:
    - ball_positions: torch.Tensor of shape (N, 3) giving the
        positions of the balls.
    - ball_linvels: torch.Tensor of shape (N, 3) giving the
        linear velocities of the balls.

    Returns:
    - reward: the total reward as a torch.Tensor of shape (N,)
    - reward_components: dictionary with individual reward components.
    """

    # Assume ball_positions[:, 2] is the height z of the ball.
    target_height = torch.tensor(0.5, device=ball_positions.device)

    # Reward for staying close to the target height
    height_diff = torch.abs(ball_positions[:, 2] - target_height)
    # Decreased temperature for larger impact
    height_temp = torch.tensor(5.0, device=ball_positions.device)
    height_reward = torch.exp(-height_diff * height_temp)

    # Reward for having low linear velocity
    ball_linvels_norm = torch.linalg.norm(ball_linvels, dim=1)
    # Increased scale for more significant impact
    vel_scale = torch.tensor(10.0, device=ball_positions.device)
    vel_reward = torch.exp(-ball_linvels_norm * vel_scale)

    # Penalty for being far from the center (in xy-plane)
    center_xy = torch.tensor([0, 0], device=ball_positions.device)
    xy_diff = torch.linalg.norm(
        ball_positions[:, :2] - center_xy,
        dim=1
    )
    # Some threshold distance
    xy_threshold = torch.tensor(0.5, device=ball_positions.device)
    xy_penalty = torch.where(
        xy_diff > xy_threshold,
        -torch.exp(xy_diff - xy_threshold),
        torch.tensor(0.0, device=ball_positions.device)
    )

    # Combine the rewards
    total_reward = height_reward + vel_reward + xy_penalty

    # Compile individual components into a dictionary
    reward_components = {
        "height_reward": height_reward,
        "vel_reward": vel_reward,
        "xy_penalty": xy_penalty
    }

    return total_reward, reward_components
```

1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349

Reward Function for Cartpole

```
def compute_gpt_reward(
    dof_pos: torch.Tensor,
    dof_vel: torch.Tensor,
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:

    # Extract pole angle and angular velocity
    pole_angle = dof_pos[:, 1]
    pole_ang_vel = dof_vel[:, 1]

    # Reward components
    # Reward for keeping the pole upright
    upright_bonus_t = 10.0
    upright_bonus = torch.exp(-upright_bonus_t*(pole_angle**2))

    # Penalty for pole's angular velocity (to encourage stability)
    ang_vel_penalty_t = 0.1
    ang_vel_penalty = torch.exp(-ang_vel_penalty_t*(pole_ang_vel**2))

    # Sum the rewards and penalties
    reward = upright_bonus + ang_vel_penalty

    # Create a dictionary of individual reward components for
    # debugging or further analysis
    reward_components = {
        'upright_bonus': upright_bonus,
        'ang_vel_penalty': ang_vel_penalty,
    }

    return reward, reward_components
```

Reward Function for Humanoid

```
def compute_gpt_reward(
    root_states: torch.Tensor,
    targets: torch.Tensor,
    dt: float
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract relevant components
    velocity = root_states[:, 7:10]

    # Vector pointing to the target
    torso_position = root_states[:, 0:3]
    to_target = targets - torso_position
    to_target[:, 2] = 0

    # Normalize to_target to get direction
    direction_to_target = torch.nn.functional.normalize(
        to_target,
        p=2.0,
        dim=-1
    )

    # Project velocity onto direction to target to get velocity
    # component in the right direction
    velocity_towards_target = torch.sum(
        velocity * direction_to_target,
        dim=-1,
        keepdim=True
    )

    # Reward for moving towards the target quickly
    speed_reward = velocity_towards_target.squeeze()

    # Apply an exponential transformation to encourage higher speeds
    temp_speed = 0.1
    speed_reward_transformed = torch.exp(speed_reward/temp_speed)-1.0

    # Combine rewards (single component in this case)
    total_reward = speed_reward_transformed

    # Reward components in a dictionary form
    rewards = {"speed_reward": speed_reward_transformed}

    return total_reward, rewards
```

Reward Function for Shadow Hand

```
def compute_gpt_reward(
    object_rot: torch.Tensor,
    goal_rot: torch.Tensor,
    actions: torch.Tensor,
    success_tolerance: float,
    reach_goal_bonus: float,
    rot_reward_scale: float,
    action_penalty_scale: float
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:

    # Rotation Distance Reward with adjusted scaling
    rot_dist = torch.norm(object_rot - goal_rot, dim=-1)
    # New temperature parameter for rotational reward
    rot_reward_temp = 3.0
    rot_reward = torch.exp(-rot_dist*rot_reward_scale/rot_reward_temp)

    # Goal Achievement Bonus
    goal_reached = rot_dist < success_tolerance
    goal_bonus = reach_goal_bonus * goal_reached.float()

    # Action Penalty with increased scale
    # Increasing the action penalty scale
    increased_aps = 2.0 * action_penalty_scale
    action_penalty = torch.sum(actions**2, dim=-1) * increased_aps

    # Intermediate Reward for making progress towards rotating to goal
    interm_steps_temp = 0.5
    intermediate_steps_reward = torch.exp(-rot_dist/interm_steps_temp)

    # Penalty for large deviations from goal orientation
    deviation_scale = 0.2
    deviation_penalty = rot_dist * deviation_scale

    # Calculate total reward
    total_reward = rot_reward + goal_bonus +
        intermediate_steps_reward - action_penalty -
        deviation_penalty

    # Create a dictionary of individual rewards for monitoring
    reward_dict = {
        "rot_reward": rot_reward,
        "goal_bonus": goal_bonus,
        "intermediate_steps_reward": intermediate_steps_reward,
        "action_penalty": action_penalty,
        "deviation_penalty": deviation_penalty
    }

    return total_reward, reward_dict
```

1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511

F IMPLEMENTATION DETAILS

Rejection Sampling While the LLM produces seemingly good code, this does not guarantee that the sampled code is bug-free and runnable. In ORSO, we employ a simple rejection sampling technique to construct sets of only valid reward functions with high probability, such that reward functions that cannot be compiled or produce $\pm\infty$ or `NaN` values are discarded.

Given criteria ϕ to be satisfied, our rejection sampling scheme repeats the steps in Algorithm 3 until we have sampled the desired number, K, of valid reward functions.

Algorithm 3 Rejection Sampling in ORSO

1: Sample a candidate reward function $f \sim G$
2: **if** $\phi(f)$ is satisfied **then**
3: Add f to the set of candidate reward functions
4: **else**
5: Reject reward function f
6: **end if**

In our practical implementation, checking if criteria ϕ are satisfied consists of instantiating an environment with the generated reward function, running a random policy on it, and checking the values produced by the reward function. If the environment cannot be instantiated or if the values returned by the reward function are $\pm\infty$ or `NaN`, the reward function is rejected. It is worth making two important observations. First, this is much computationally cheaper than instantiating the environment for training because one does not need to initialize large neural networks and can use fewer parallel environments than the number necessary for training. Moreover, we note that the rejection sampling mechanism only guarantees a higher probability of a valid reward function code as the policy used to evaluate the function is random and the optimization process used during the training of an RL algorithm could still induce undesirable values.

Iterative Improvement of the Reward Function Set In the initial phase of ORSO, the algorithm generates a set of candidate reward functions $\mathcal{R}^K$ for the online reward selection and policy optimization step. While this approach is effective if $\mathcal{R}^K$ contains an effective reward function, any selection process will fail to achieve a high task reward if the set does not contain a good reward function. To address this limitation, we introduce a mechanism for improving the reward function set through iterative resampling and in-context evolution. This is similar to Ma et al. (2024), however, we introduce some important changes to prevent the in-context evolution from overfitting to initially suboptimal reward functions.

Resampling is triggered when at least one reward function has been used to train a policy for the number of iterations specified in the environment configuration or if all the reward functions in the set incurred too large a regret compared to the previous best policy if the algorithm has undergone at least one resampling step.

There are several strategies for resampling reward functions, each with its trade-offs. The simplest approach is to sample new reward functions from scratch, using the same generator G that was used in the initial phase. However, this method may not provide significant improvement, as it essentially restarts the search process without leveraging the information gained from the previous iterations of training.

A more sophisticated approach is to greedily in-context evolve the reward function from the best-performing candidate so far as is done in Ma et al. (2024). This involves making incremental adjustments to the reward function that has shown the most promise, potentially moving it closer to an optimal reward function. However, while this greedy strategy can lead to improvements, it also has the risk of overfitting to an initially suboptimal reward function if, for example, the initial set does not contain effective reward functions.

To mitigate the risk of overfitting, we introduce a simple strategy that allows the algorithm to be more exploratory. Specifically, we combine greedy evolution with random sampling: half of the reward functions are evolved in context from the best-performing candidate, while the other half is sampled from scratch. This approach allows the algorithm to explore new regions of the reward

function space while still exploiting the knowledge gained from previous iterations. We provide the full pseudo-code for ORSO with rejection sampling and iterative improvement in Algorithm 4.

Algorithm 4 ORSO with Rejection Sampling and Iterative Improvement

Require: MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \rho_0)$, algorithm $\mathfrak{A}$, generator G, budget T, threshold `n_iters`
1: Sample K valid reward functions $\mathcal{R}^K = \{f^1, \ldots, f^K\} \sim G$ using Algorithm 3
2: Initialize K policies $\{\pi^1, \ldots, \pi^K\}$
3: Initialize selection counts $N^K = \{0, \ldots, 0\}$
4: Set $t \leftarrow 1$
5: **while** $t \leq T$ **do**
6: Select a model $i_t \in [K]$ according to a selection strategy
7: Update $\pi^{i_t} \leftarrow \mathfrak{A}_{f^{i_t}}(\mathcal{M}, N, \pi^{i_t})$
8: Evaluate $\mathcal{J}(\pi^{i_t}) \leftarrow \texttt{Eval}(\pi^{i_t})$
9: Update selection counts: $N^{i_t} \leftarrow N^{i_t} + 1$
10: Update variables (e.g., reward estimates and confidence intervals)
11: **if** $N^{i_t} \geq$ `n_iters` or regret w.r.t. previous best is too high **then**
12: Resample $\mathcal{R}^K \sim G$ (half in-context evolution, half from scratch)
13: Sample a new set of reward functions $\mathcal{R}^K = \{f^1, \ldots, f^K\}$ using rejection sampling
14: Reset policies $\{\pi^1, \ldots, \pi^K\}$
15: Reset selection counts $N^K = \{0, \ldots, 0\}$
16: **end if**
17: $t \leftarrow t + 1$
18: **end while**
19: **return** $\pi^\star_T, f^\star_T = \arg\max_{i \in [K]} \mathcal{J}(\pi^i)$

1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619

G SELECTION ALGORITHMS AND HYPERPARAMETERS

In this section, we present the pseudocode for all reward selection algorithms used in our experiments with their associated hyperparameters in Table 2.

Algorithm 5 ε-Greedy

Require: Number of arms K, total time T, exploration probability ε

1: Initialize counts $n_0^i = 0$ and total values $\widehat{u}_0^i = 0$ for all $i \in [K]$
2: **for** $t = 1, \dots, T$ **do**
3: Select arm

$$i_t = \begin{cases} \arg\max_i(\widehat{u}_t^i / n_t^i), & \text{with probability } 1 - \varepsilon \\ i \sim \text{Uniform}([K]), & \text{with probability } \varepsilon \end{cases}$$

4: Play arm i_t and observe reward r_t
5: Set $n_t^i = n_{t-1}^i$, and $\widehat{u}_t^i = \widehat{u}_{t-1}^i$ for all $i \in [K] \setminus \{i_t\}$
6: Update statistics for current learner $n_t^{i_t} = n_{t-1}^{i_t} + 1$ and $\widehat{u}_t^{i_t} = \widehat{u}_{t-1}^{i_t} + r_t$
7: **end for**

Algorithm 6 Explore-then-Commit

Require: Number of arms K, total time T, exploration phase length T_0

1: Initialize counts $n_0^i = 0$ and total values $\widehat{u}_0^i = 0$ for all $i \in [K]$
2: `// Explore`
3: **for** $t = 1, \dots, T_0$ **do**
4: Select arm $i_t = (t \mod K) + 1$
5: Play arm i_t and observe reward r_t
6: Set $n_t^i = n_{t-1}^i$, and $\widehat{u}_t^i = \widehat{u}_{t-1}^i$ for all $i \in [K] \setminus \{i_t\}$
7: Update statistics for current learner $n_t^{i_t} = n_{t-1}^{i_t} + 1$ and $\widehat{u}_t^{i_t} = \widehat{u}_{t-1}^{i_t} + r_t$
8: **end for**
9: `// Commit`
10: $i_\star = \arg\max_i(u_t^i / n_t^i)$
11: **for** $t = T_0 + 1$ to T **do**
12: Play arm $i_\star$ and observe reward r_t
13: **end for**

Algorithm 7 UCB (Upper Confidence Bound)

Require: Number of arms K, total time T, confidence multiplier c

1: Initialize counts $n_0^i = 0$ and total values $\widehat{u}_0^i = 0$ for all $i \in [K]$
2: **for** $t = 1, \dots, K$ **do**
3: Select arm $i_t = t$
4: Play arm i_t and observe reward r_t
5: Update statistics for current learner $n_t^{i_t} = n_{t-1}^{i_t} + 1$ and $\widehat{u}_t^{i_t} = \widehat{u}_{t-1}^{i_t} + r_t$
6: **end for**
7: **for** $t = K + 1, \dots, T$ **do**
8: Select arm

$$i_t = \arg\max_i \left(\frac{\widehat{u}_t^i}{n_t^i} + c\sqrt{2\frac{\ln t}{n_t^i}} \right)$$

9: Play arm i_t and observe reward r_t
10: Set $n_t^i = n_{t-1}^i$, and $\widehat{u}_t^i = \widehat{u}_{t-1}^i$ for all $i \in [K] \setminus \{i_t\}$
11: Update statistics for current learner $n_t^{i_t} = n_{t-1}^{i_t} + 1$ and $\widehat{u}_t^{i_t} = \widehat{u}_{t-1}^{i_t} + r_t$
12: **end for**

Algorithm 8 Exp3 (Exponential-weight algorithm for Exploration and Exploitation)

Require: Number of arms K, total time T, learning rate η
1: Initialize weights $w_0^i = 1$ and probabilities $p_0^i = 1/K$ for all $i \in [K]$
2: **for** $t = 1, \ldots, T$ **do**
3: Select arm i_t according to distribution $P_t = [p_t^1, \ldots, p_t^K]$
4: Play arm i_t and observe reward r_t
5: Estimate reward $\widehat{r}_t = r_t / p_t^{i_t}$
6: Update weight $w_t^{i_t} = w_{t-1}^{i_t} \exp(\eta \widehat{r}_t / K)$
7: Update probabilities

$$p_t^i = (1-\eta)\frac{w_t^i}{\sum_{j=1}^K w_t^j} + \frac{\eta}{K} \qquad \text{for all } i \in [K]$$

8: **end for**

Table 2: Hyperparameters for MAB Algorithms

ALGORITHM	PARAMETER	VALUE
EPSILON-GREEDY	ε	0.1
EXPLORE-THEN-COMMIT	T_0	$5 \cdot K$
UCB	c	1.0
EXP3	η	0.1

H Additional Experimental Results

In this section, we report additional experimental evaluations. In particular, we show how different configurations of budget constraints B and sizes K of the reward function set perform with different reward selection algorithms in different environments in Figures 7 to 10.

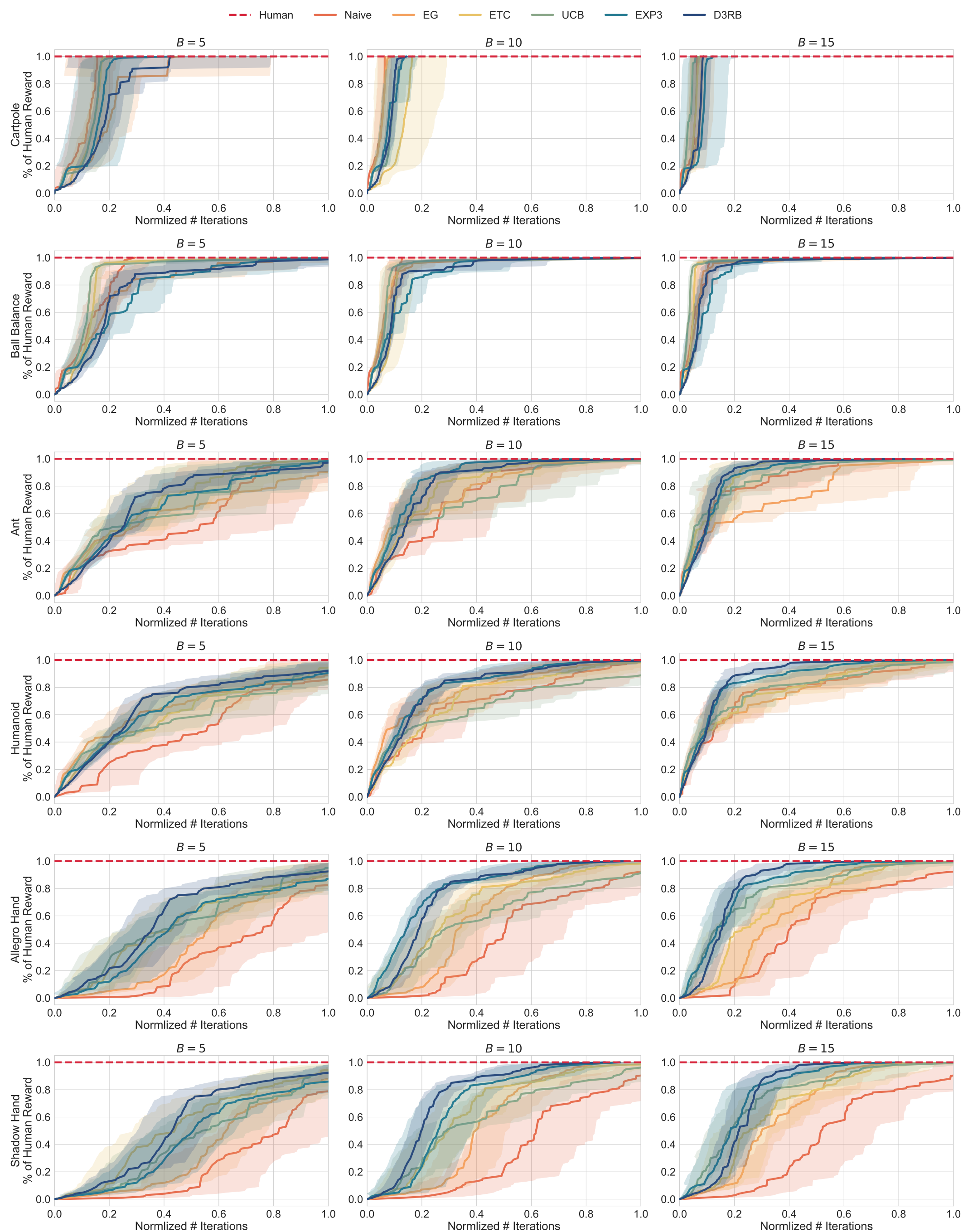


Figure 7: Number of iterations necessary to reach human-engineered reward function performance with different computation budgets and tasks. The shaded areas represent 95% confidence intervals. To construct this plot, we sample the first index when the performance reaches each percentage point from 1% to 100% of human performance during training.

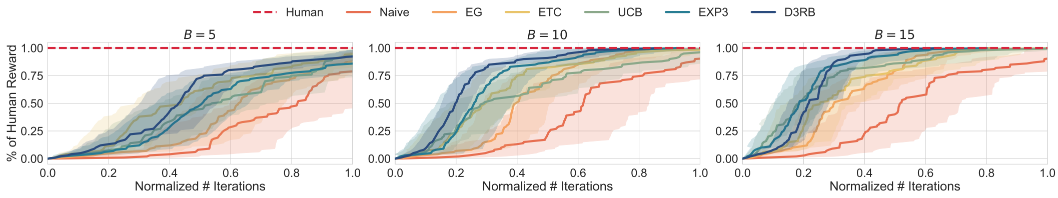


Figure 8: Number of iterations necessary to reach human-engineered reward function performance with different computation budgets. The shaded areas represent 95% confidence intervals.

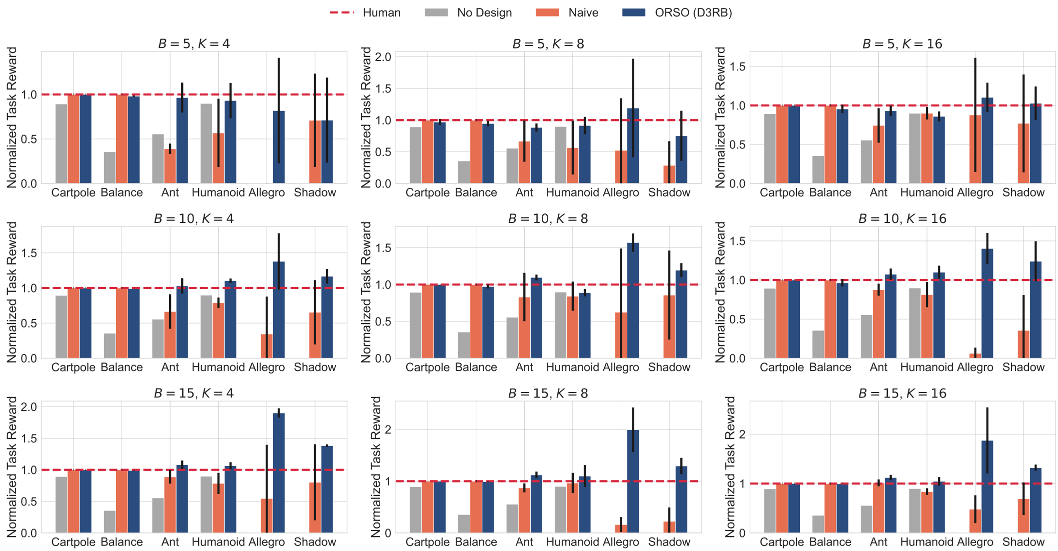


Figure 9: Average performance with 95% confidence intervals for ORSO with different budget constraints and reward function set size. The red horizontal dashed line represents the policies trained with the human-engineered reward function.

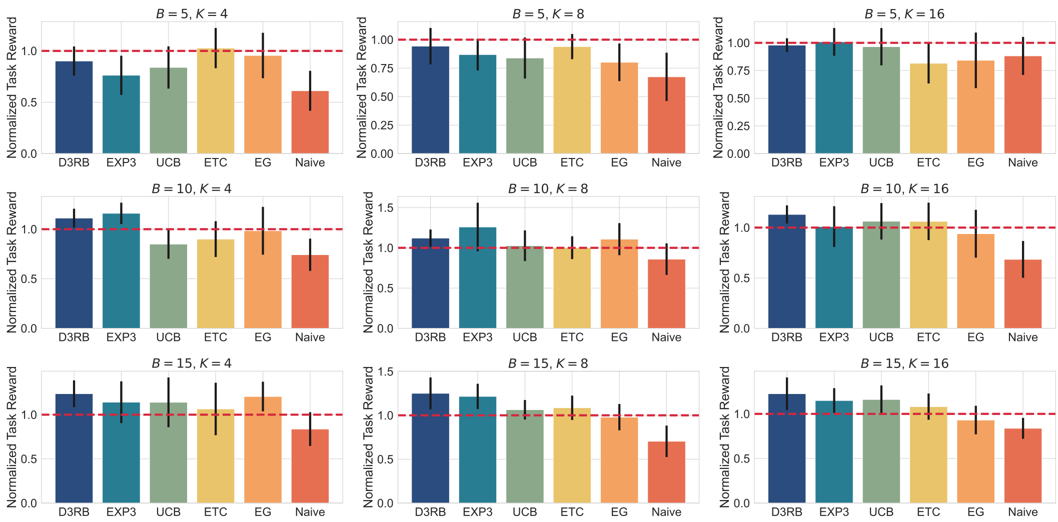


Figure 10: Comparison of different reward selection algorithms for ORSO with different budget constraints and reward function set size.

We also plot in Figure 11 the time necessary to achieve the same performance as policies trained with human-designed reward functions as a function of the number of parallel GPUs available for all budget constraints and all tasks considered.

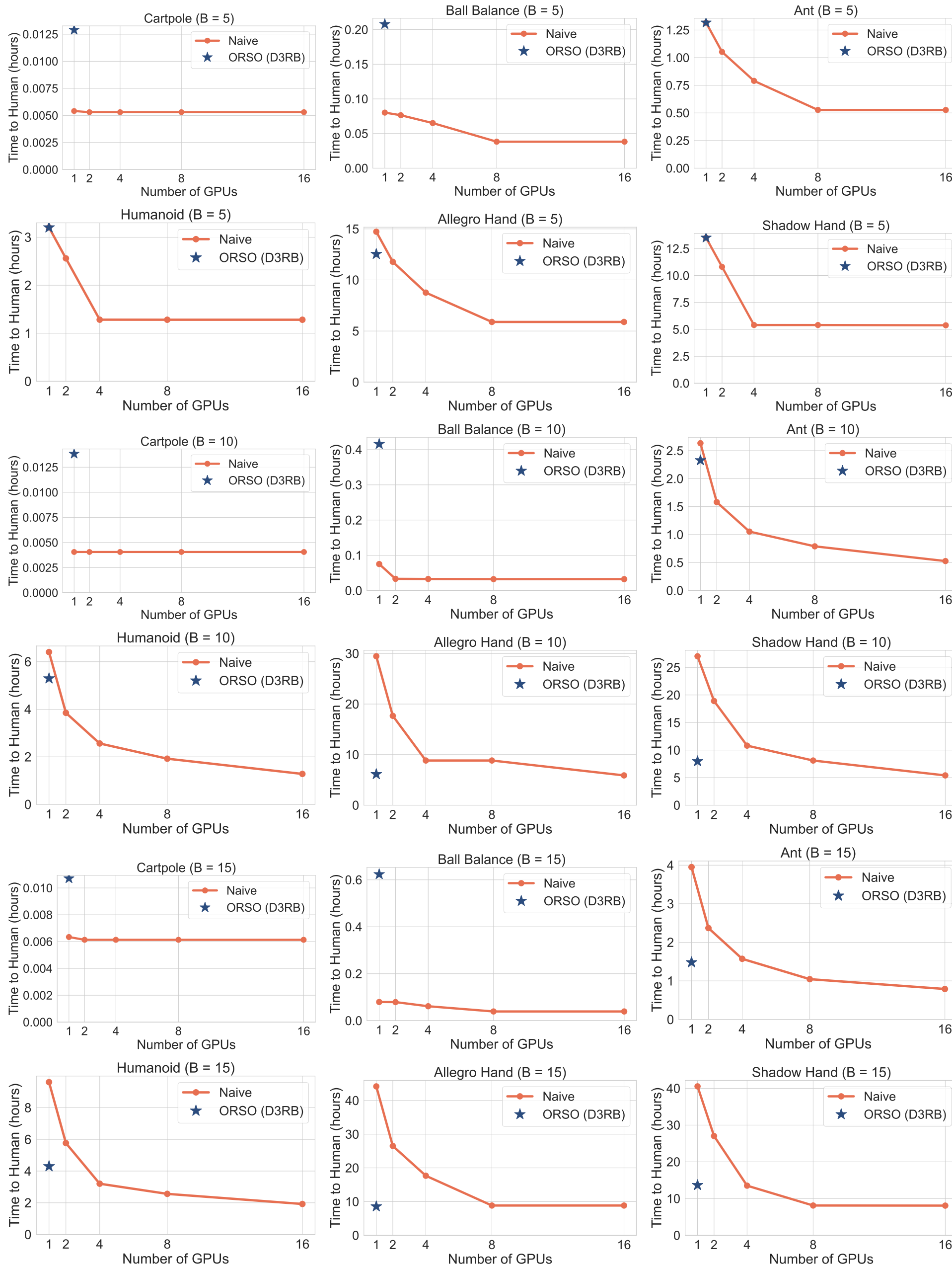


Figure 11: Time necessary to achieve the same performance as policies trained with human-designed reward functions as a function of the number of parallel GPUs.

H.1 Chosen Reward Functions for Large Reward Set

In order to validate that ORSO with D^{3}RB indeed chooses the optimal reward function, we train a policy for each of the $K = 96$ reward functions for the ANT task in Figure 6. In Table 3, we report the mean task reward with 95% confidence intervals over five seeds. Rewards are ordered from best to worst, with those within one confidence interval of the best reward underlined. Bolded values indicate the reward functions selected by ORSO across the seeds we ran.

Table 3: Mean task reward for each Reward ID with 95% confidence intervals (CI).

Reward ID	Mean ($\pm$ CI)
34	**10.24 $\pm$ 0.36**
18	10.01 $\pm$ 0.63
71	9.98 $\pm$ 0.37
79	9.88 $\pm$ 0.73
21	**9.77 $\pm$ 0.22**
94	9.70 $\pm$ 0.38
66	**9.67 $\pm$ 0.27**
81	**9.55 $\pm$ 0.80**
70	**9.51 $\pm$ 0.63**
37	9.46 $\pm$ 0.55
33	9.34 $\pm$ 0.83
95	9.27 $\pm$ 0.33
47	9.24 $\pm$ 0.68
63	9.21 $\pm$ 0.76
54	9.20 $\pm$ 0.80
80	9.16 $\pm$ 0.25
62	8.88 $\pm$ 0.37
38	8.81 $\pm$ 0.45
49	8.81 $\pm$ 0.77
35	8.69 $\pm$ 1.07
5	8.61 $\pm$ 0.56
52	8.35 $\pm$ 1.43
67	8.32 $\pm$ 0.85
46	8.30 $\pm$ 0.89
68	8.20 $\pm$ 1.22
75	8.09 $\pm$ 0.40
84	8.05 $\pm$ 1.25
85	7.77 $\pm$ 0.97
72	7.64 $\pm$ 1.27
55	7.43 $\pm$ 1.46
20	7.26 $\pm$ 0.18
23	7.26 $\pm$ 1.12
86	7.15 $\pm$ 0.42
36	7.06 $\pm$ 0.68
91	6.93 $\pm$ 1.45
1	6.50 $\pm$ 1.17
31	6.36 $\pm$ 0.80
61	6.06 $\pm$ 0.93
19	5.78 $\pm$ 1.43
25	5.67 $\pm$ 1.41
48	5.65 $\pm$ 1.54
59	5.59 $\pm$ 1.02
26	5.50 $\pm$ 0.89
60	5.47 $\pm$ 1.17
44	5.47 $\pm$ 1.36
40	5.34 $\pm$ 1.73
73	5.33 $\pm$ 1.77
0	5.30 $\pm$ 1.38
56	5.05 $\pm$ 0.62
39	4.91 $\pm$ 0.64
74	4.88 $\pm$ 0.49
30	4.83 $\pm$ 0.78
78	4.83 $\pm$ 0.35
6	4.76 $\pm$ 0.91
2	4.69 $\pm$ 0.92
28	4.66 $\pm$ 1.21
8	4.65 $\pm$ 0.44
16	4.57 $\pm$ 1.08
29	4.53 $\pm$ 0.81
65	4.44 $\pm$ 0.62
50	4.23 $\pm$ 1.94
58	3.89 $\pm$ 0.43
53	3.86 $\pm$ 0.44
32	3.79 $\pm$ 0.73
22	3.74 $\pm$ 0.54
3	3.48 $\pm$ 1.66
69	3.22 $\pm$ 0.42
4	3.18 $\pm$ 0.42
88	3.18 $\pm$ 0.43
64	3.12 $\pm$ 0.16
9	3.11 $\pm$ 0.39
17	3.10 $\pm$ 0.15
93	3.02 $\pm$ 0.21
14	2.99 $\pm$ 0.52
45	2.89 $\pm$ 0.29
83	2.72 $\pm$ 0.82
27	2.50 $\pm$ 0.72
10	2.15 $\pm$ 0.43
57	1.69 $\pm$ 0.80
7	1.67 $\pm$ 1.01
82	1.03 $\pm$ 0.35
42	0.63 $\pm$ 0.80
41	0.37 $\pm$ 0.30
43	0.33 $\pm$ 0.16
76	0.22 $\pm$ 0.08
89	0.22 $\pm$ 0.07
24	0.21 $\pm$ 0.03
11	0.21 $\pm$ 0.08
12	0.19 $\pm$ 0.14
15	0.19 $\pm$ 0.14
92	0.14 $\pm$ 0.07
77	0.13 $\pm$ 0.04
13	0.05 $\pm$ 0.00
51	0.05 $\pm$ 0.00
87	0.05 $\pm$ 0.02
90	0.00 $\pm$ 0.00

H.2 VISUALIZING ORSO

To better visualize how ORSO selects the best reward function, discards suboptimal ones efficiently, and thanks to this, explores more reward functions, we provide further visualizations in this section.

Figures 12 and 13 show a full training of ORSO (D^3RB) and EUREKA with a budget $B = 15$ and $K = 16$ on the ALLEGROHAND task, respectively. In both figures, the top plot shows the task reward during training. The colors indicate the reward functions currently in use. The middle plot more clearly shows the reward function being currently used. The vertical axis contains the reward function indices. In both plots, the dashed vertical lines indicate that a resampling has been triggered. Lastly, the bottom plot shows the unnormalized cumulative regret during training.

Comparing the two figures, we can see that ORSO initially explores all reward functions near-uniformly, but quickly finds a policy that surpasses the policy from the human-engineered reward function, leading to a decrease in regret. On the other hand, EUREKA uniformly trains on each reward function leading the algorithm to explore fewer reward functions. Moreover, we see that the lack of rejection sampling can result in initial reward function sets that contain many invalid reward functions – indicated by a $\times$ in the figures.

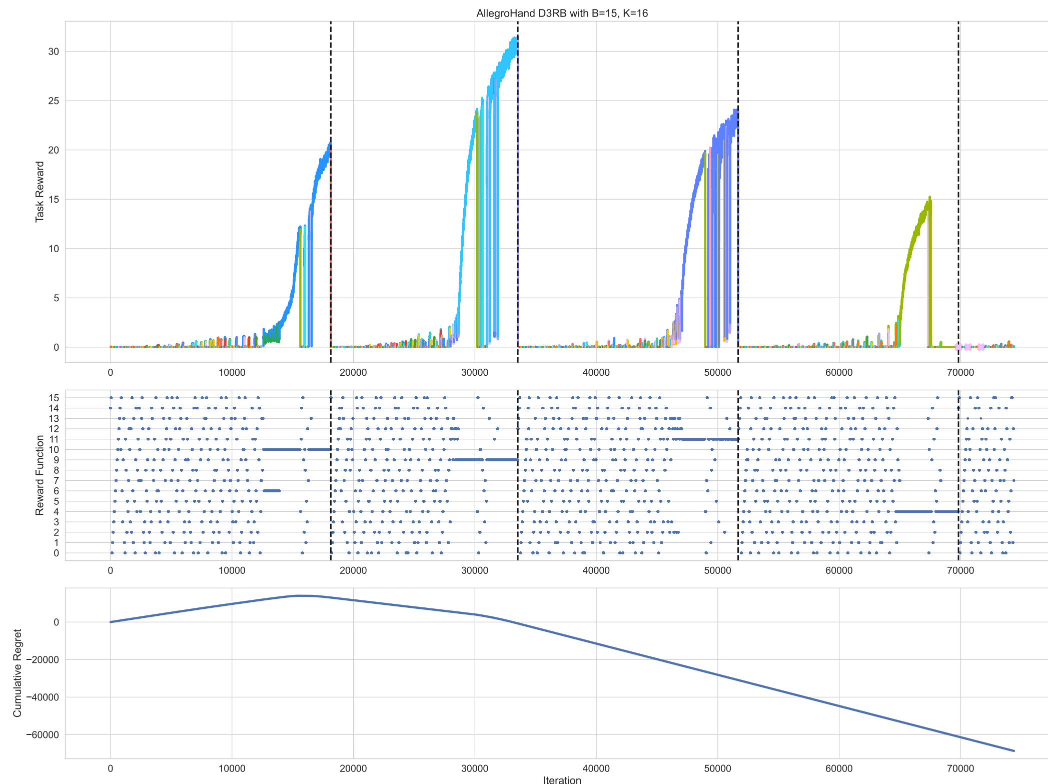


Figure 12: ORSO (D^3RB) on ALLEGROHAND with $B = 15$ and $K = 16$.

1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970

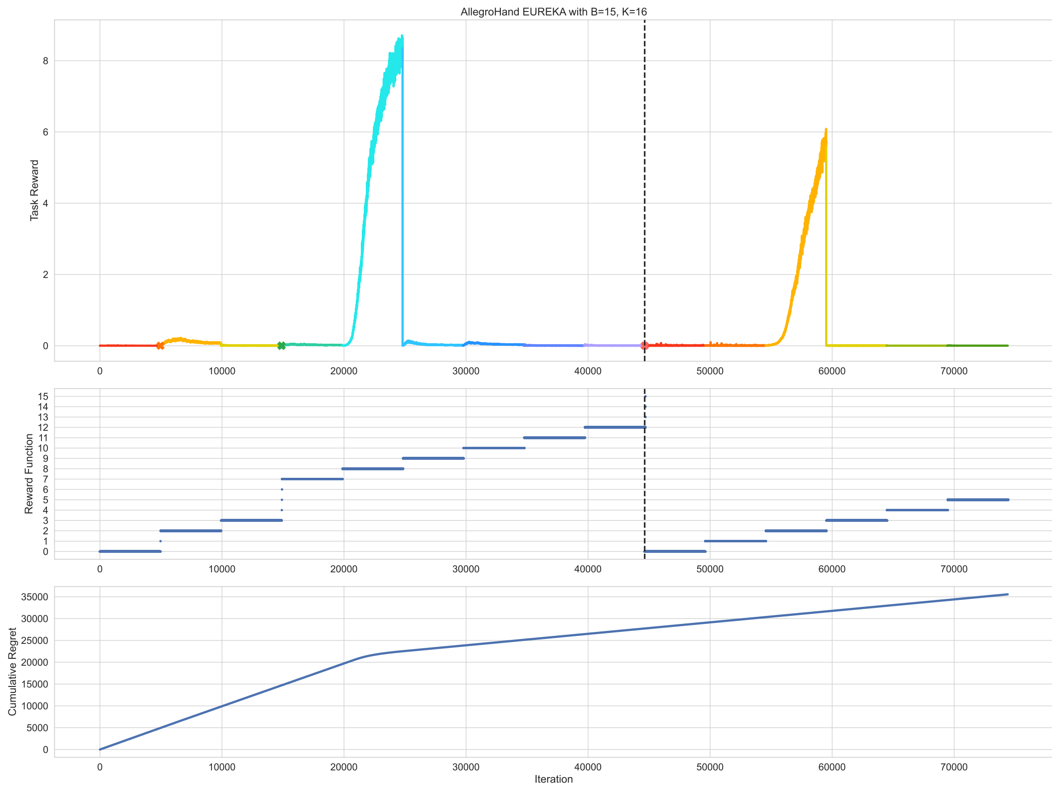


Figure 13: EUREKA on ALLEGROHAND with $B = 15$ and $K = 16$.

1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997

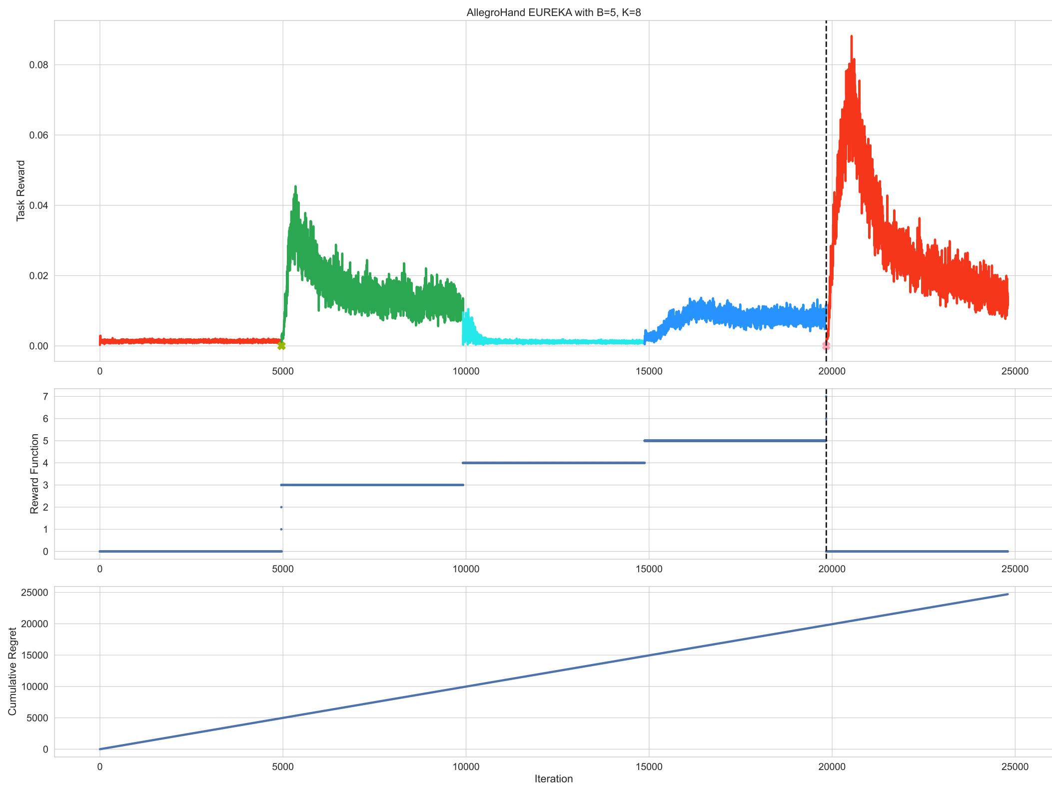


Figure 14: EUREKA on ALLEGROHAND with $B = 5$ and $K = 8$.

1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051

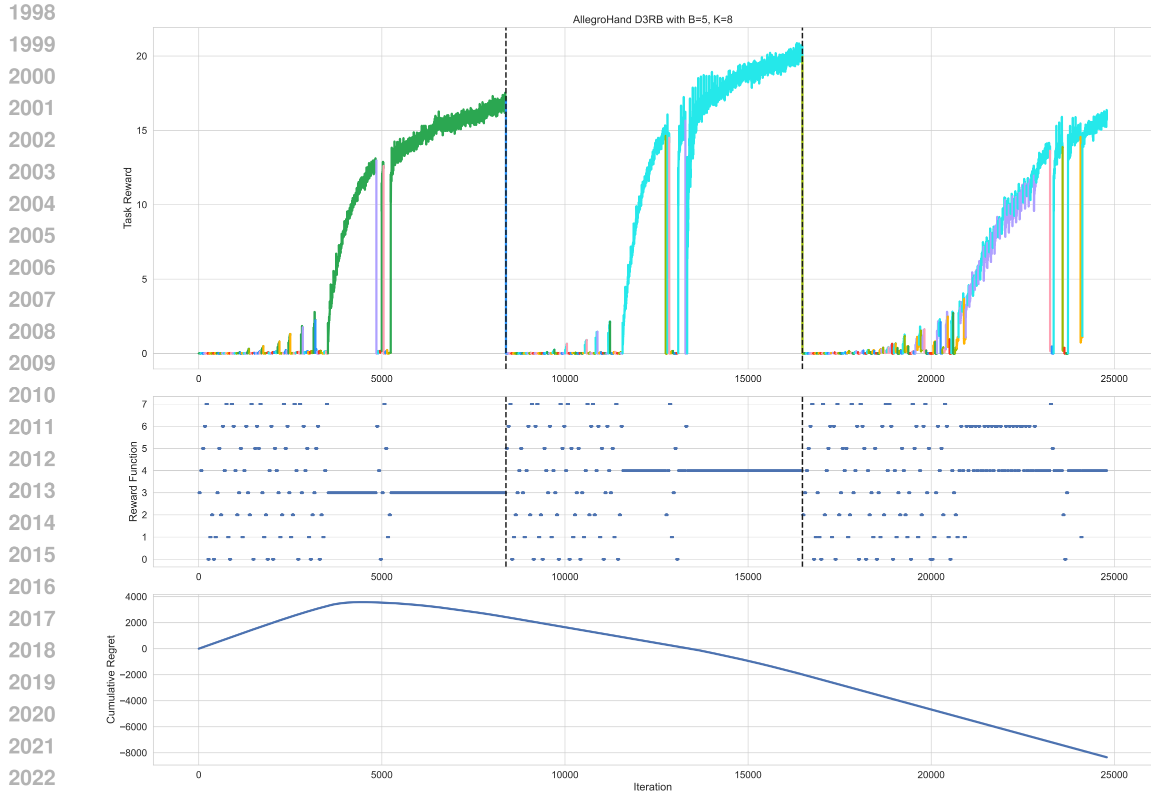


Figure 15: ORSO (D^{3}RB) on ALLEGROHAND with $B = 5$ and $K = 8$.

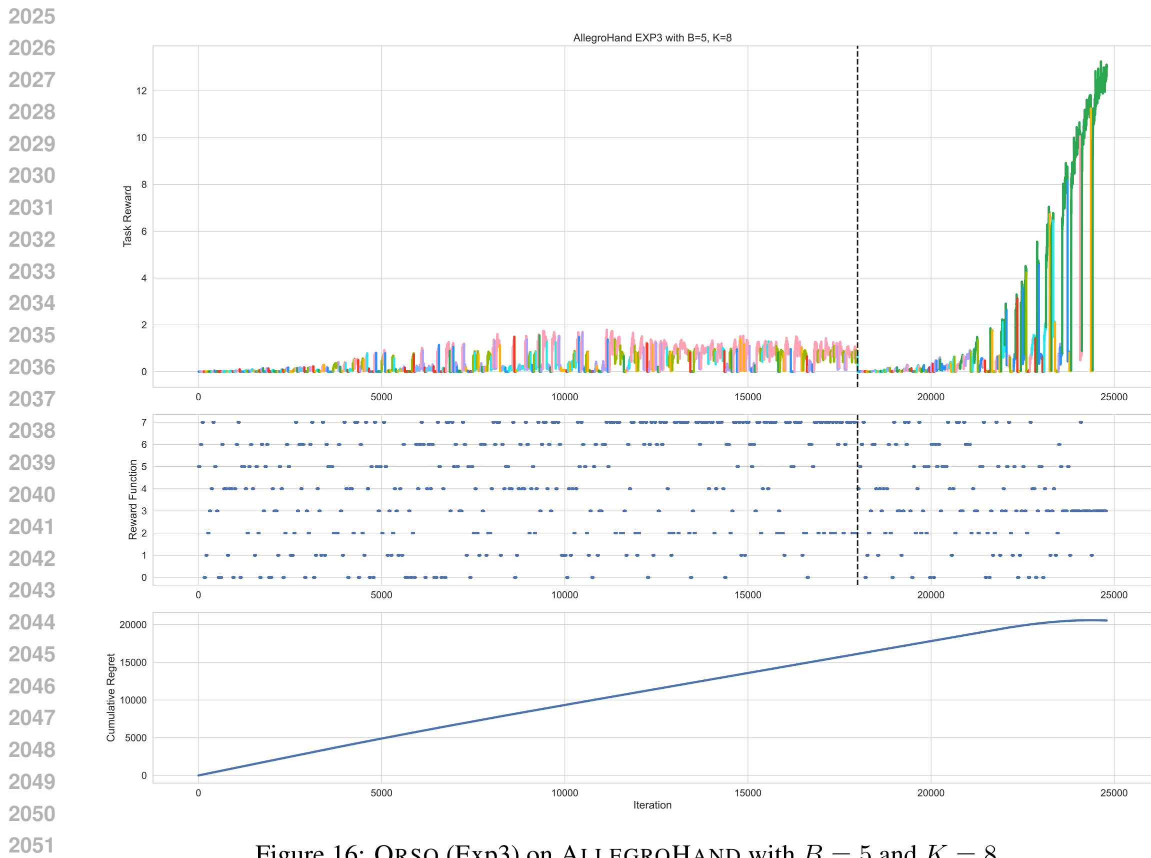


Figure 16: ORSO (Exp3) on ALLEGROHAND with $B = 5$ and $K = 8$.

2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079

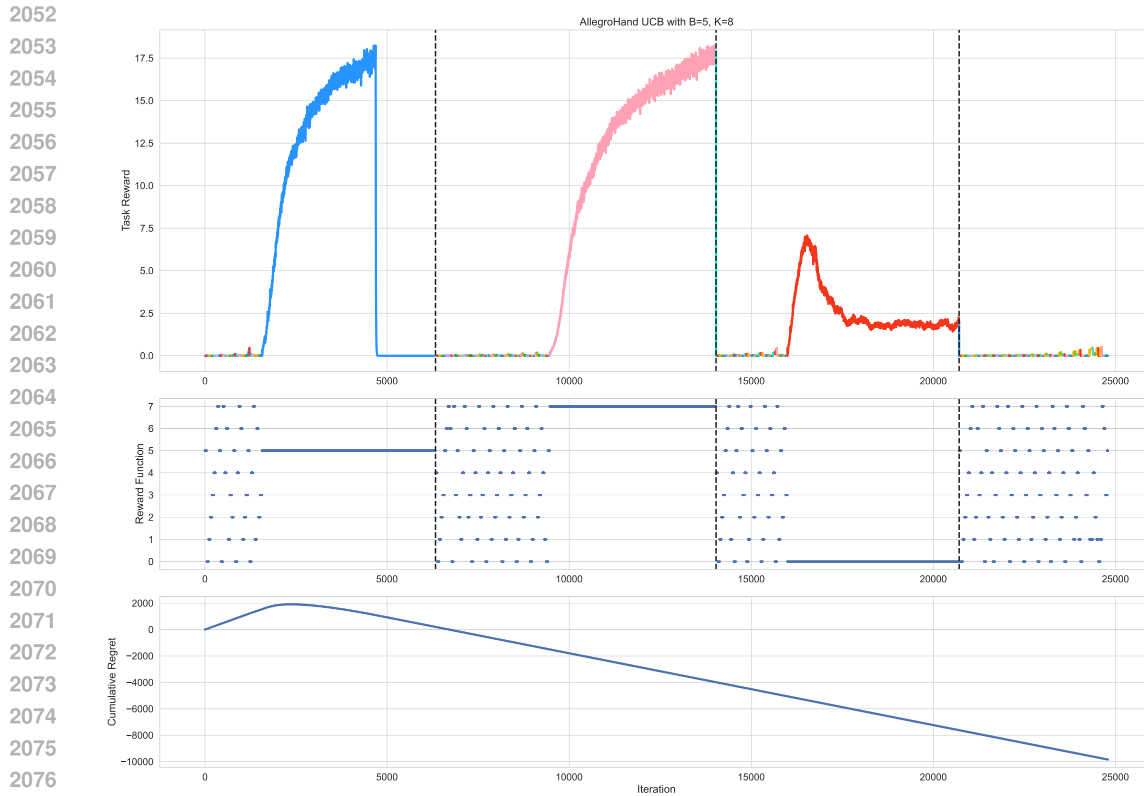


Figure 17: ORSO (UCB) on ALLEGROHAND with $B = 5$ and $K = 8$.

2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095 2096 2097 2098 2099 2100 2101 2102 2103 2104 2105

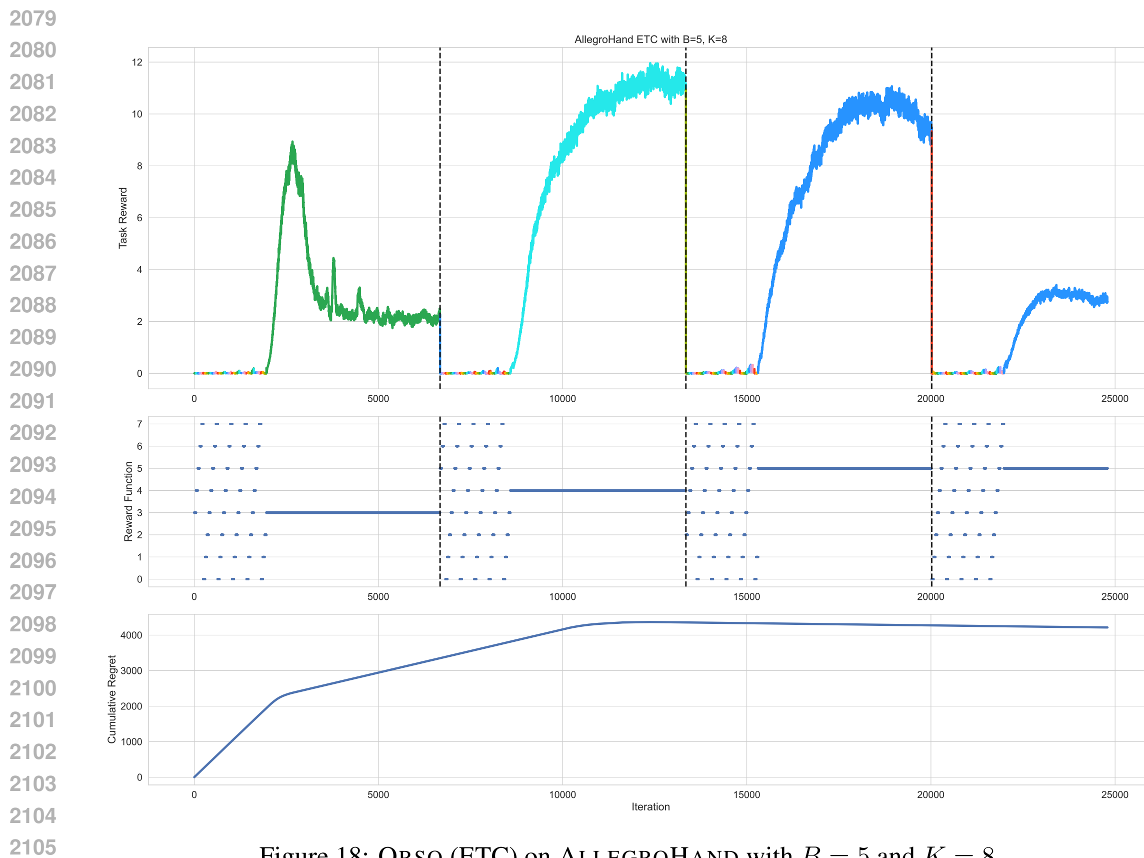


Figure 18: ORSO (ETC) on ALLEGROHAND with $B = 5$ and $K = 8$.

2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159

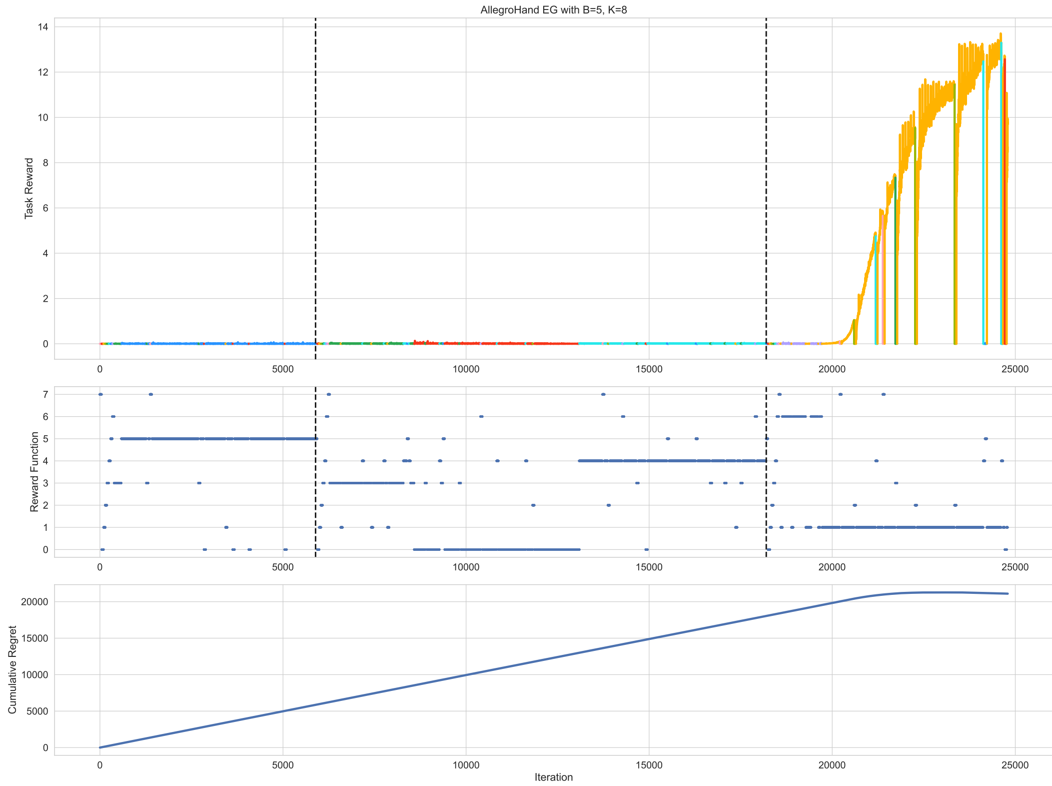


Figure 19: ORSO (EG) on ALLEGROHAND with $B = 5$ and $K = 8$.

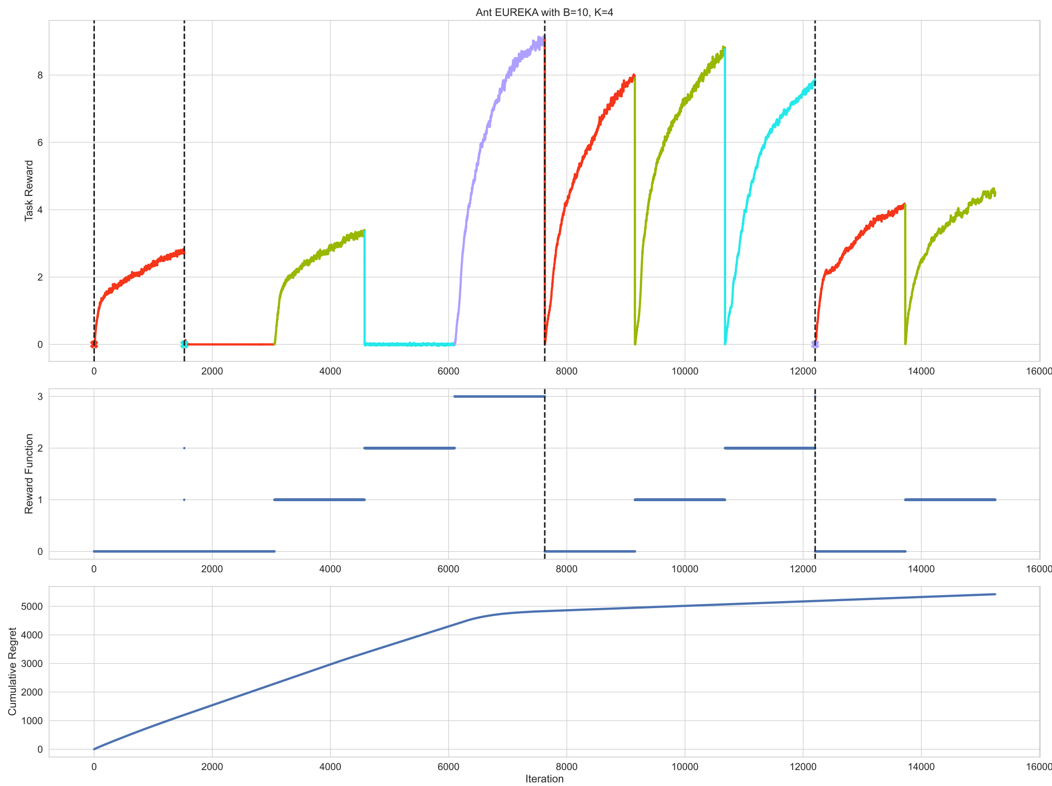


Figure 20: EUREKA on ANT with $B = 10$ and $K = 4$.

2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187

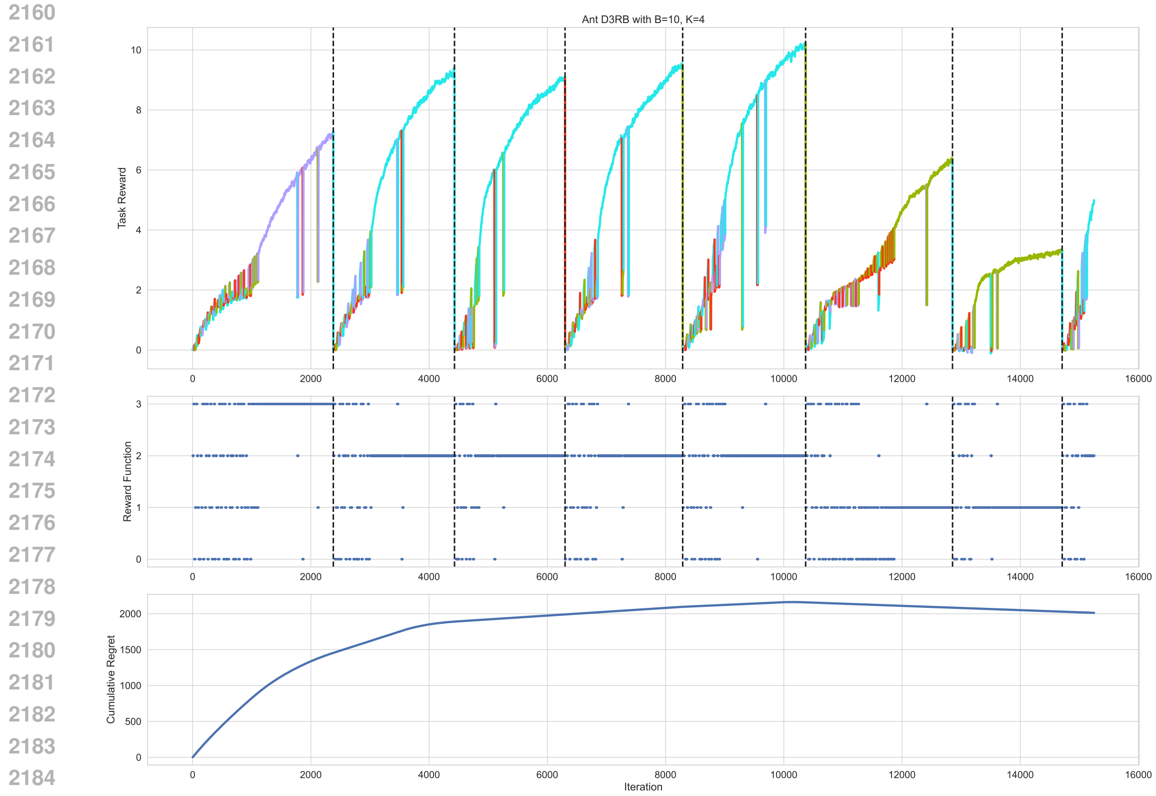


Figure 21: ORSO (D^{3}RB) on ANT with $B = 10$ and $K = 4$.

2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213

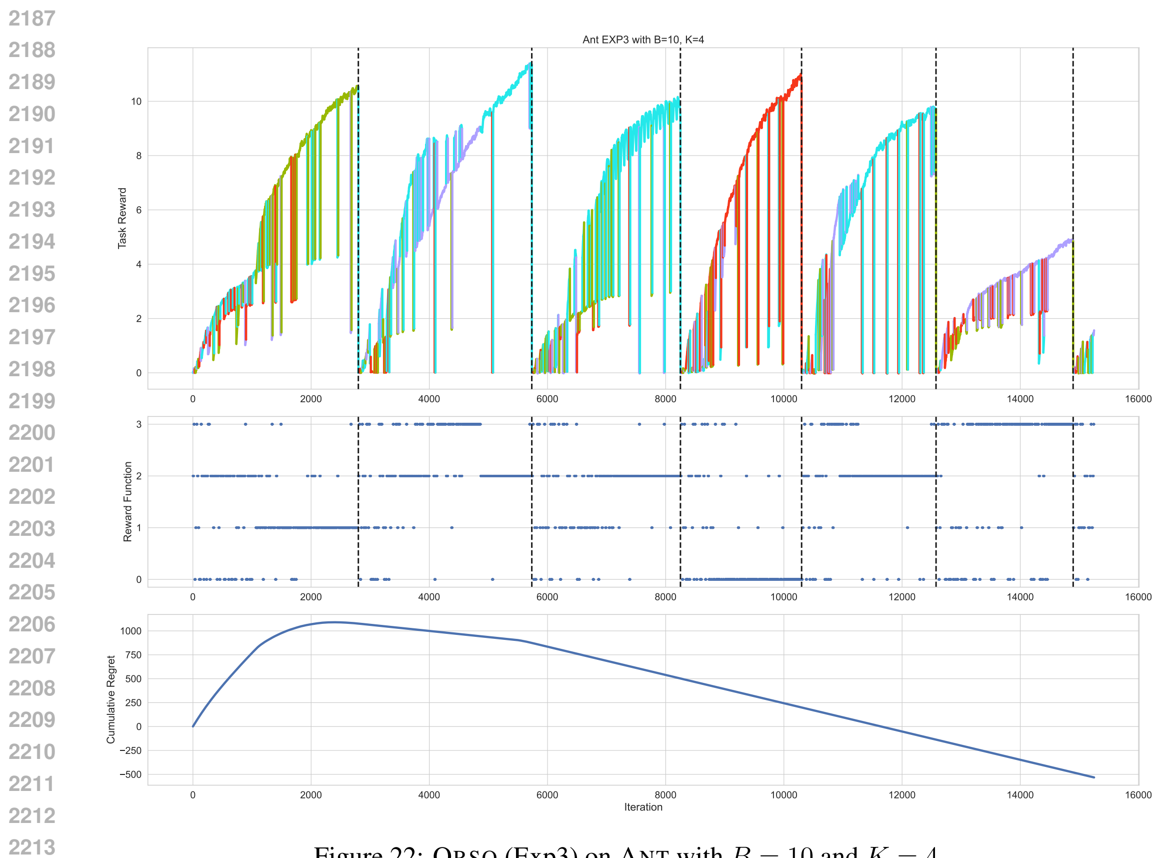


Figure 22: ORSO (Exp3) on ANT with $B = 10$ and $K = 4$.

2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241

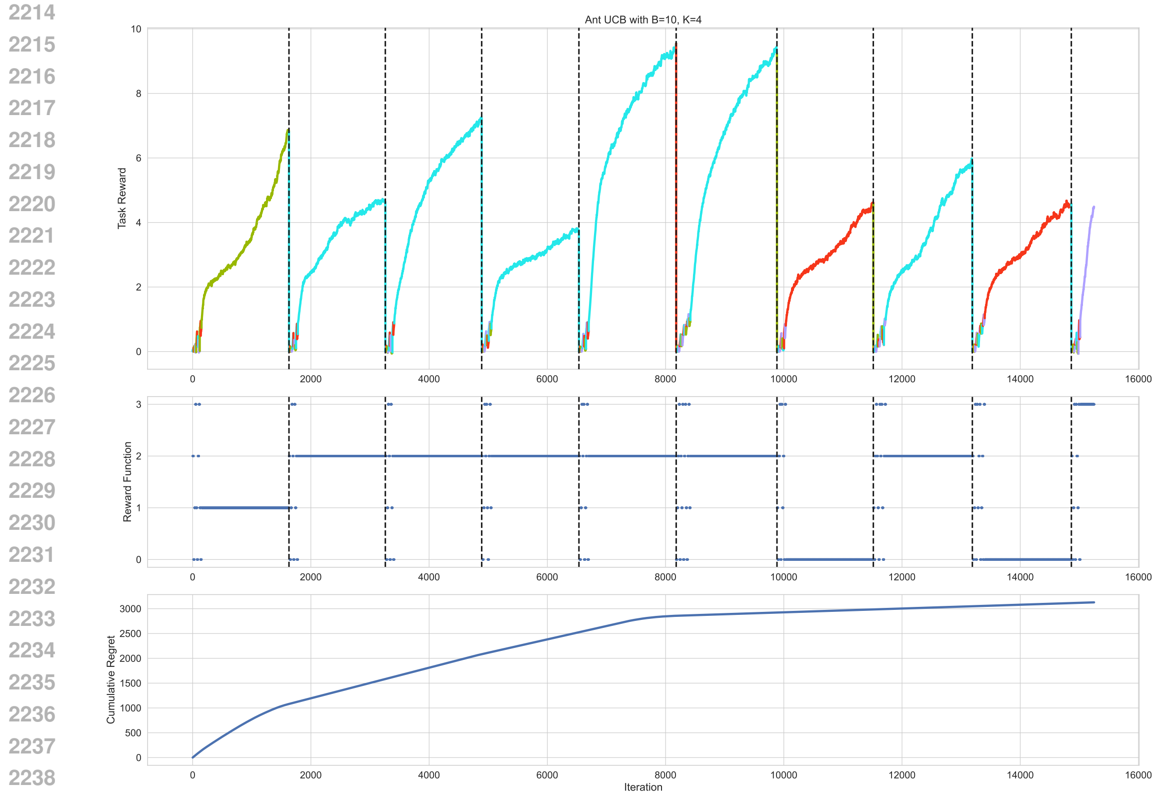


Figure 23: ORSO (UCB) on ANT with $B = 10$ and $K = 4$.

2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267

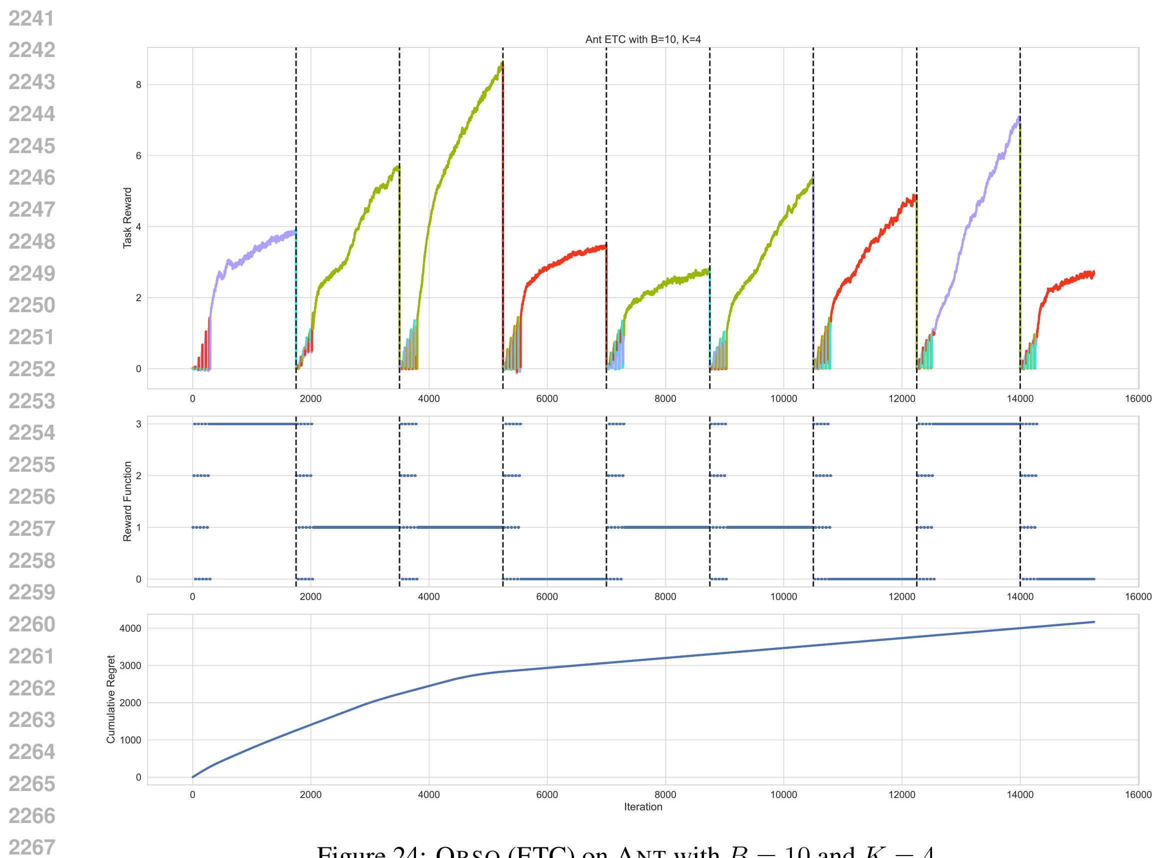


Figure 24: ORSO (ETC) on ANT with $B = 10$ and $K = 4$.

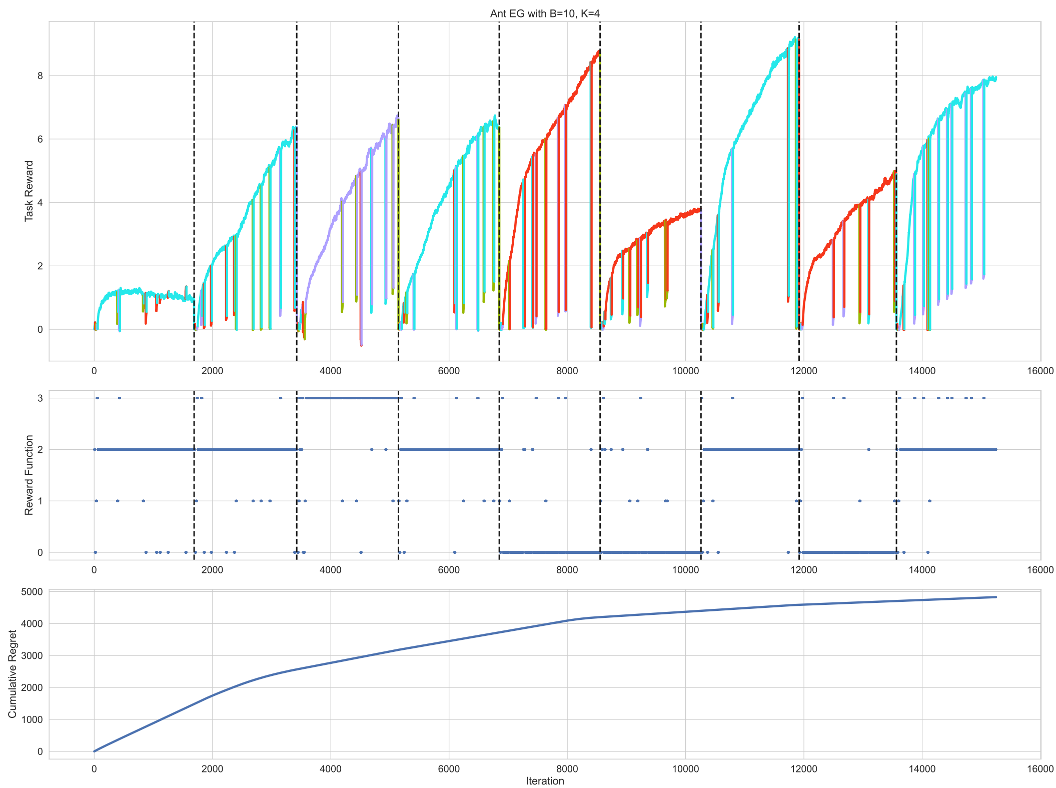


Figure 25: ORSO (EG) on ANT with $B = 10$ and $K = 4$.