DESIGN PRINCIPLES FOR TD-BASED MULTI-POLICY MORL IN INFINITE HORIZONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-Objective Reinforcement Learning (MORL) addresses problems with multiple, often conflicting goals by seeking a set of trade-off policies rather than a single solution. Existing approaches that learn many policies at once have shown promise in deep settings, but they depend on supervised retraining and carefully curated data, making them ill-suited for online and infinite-horizon tasks. Temporal-Difference (TD) methods offer a natural alternative, as they update policies incrementally during interaction, but current TD-based approaches are limited to small, episodic problems. In this work, we present design principles for extending TD-based multi-policy MORL to both predictable (stationary) and flexible (non-stationary) policies, to avoid spurious dominance relations, and to detect cycles. Through ablation studies, we show how each principle contributes to recovering diverse and reliable policies, providing a principled path toward scalable TD-based multi-policy methods in deep MORL.

1 INTRODUCTION

Multi-Objective Reinforcement Learning (MORL) is an RL setting with multiple, often conflicting objectives. MORL algorithms aim to approximate the Pareto Front (PF) — the set of value vectors (and corresponding policies) for which no objective can be improved without degrading another — capturing trade-offs to support preference-based decision-making (Hayes et al., 2021). In standard RL, no single policy can accommodate all preference trade-offs; objectives are typically aggregated into a single scalar reward function via scalarization (combining multiple rewards into one) (Roijers et al., 2013). When preferences are unknown beforehand, scalarization is insufficient, as it requires retraining for each preference and often provides limited PF coverage (Moffaert & Nowé, 2014). Such challenges motivate methods that learn multiple policies to represent the full preference space.

Approaches for learning multiple policies fall broadly into two categories. Outer-loop methods iteratively derive new policies from previously learned ones (Roijers et al., 2015; Parisi et al., 2017; Röpke et al., 2024), while inner-loop methods aim to learn the entire policy set in a single run (Moffaert & Nowé, 2014; Ruiz-Montiel et al., 2017; Reymond & Nowé, 2019; Reymond et al., 2022a). Pareto Conditioned Networks (PCNs) (Reymond et al., 2022a) extend inner-loop methods to deep RL by conditioning policies on desired return vectors, enabling scalability to high-dimensional settings. However, PCNs are not based on Temporal-Difference (TD) updates and rely on costly supervised-style retraining and careful trajectory curation, which limits scalability and online adaptation for infinite-horizon tasks. TD methods, in contrast, learn incrementally at each interaction and naturally support infinite-horizon settings. Yet, existing TD-based multi-policy approaches remain largely restricted to tabular, episodic settings (Moffaert & Nowé, 2014; Ruiz-Montiel et al., 2017).

To apply tabular inner-loop methods to the infinite-horizon setting via TD learning, we introduce a series of design choices to tackle the challenges of this setting: allow tracking and following policies when multiple options exist; learning solutions for tasks requiring predictable behavior vs. discovering a broader range of solutions; ensuring policies are not prematurely discarded; computing undiscounted returns to give the user an accurate sense of performance when selecting the policy; and detecting cycles, which offer well-defined returns for infinite horizons. Together, these elements provide the basis for a systematic analysis of tabular multi-policy RL in infinite-horizon settings, providing a principled foundation for future deep-RL extensions.

Problem Setting: MORL is essential for tasks with conflicting objectives. Multi-policy methods like PCNs can produce solution sets in deep RL but rely on costly supervised retraining and trajectory curation, limiting scalability and online adaptation in infinite-horizon settings. TD methods support incremental online learning but remain mostly tabular and episodic. As a result, no TD-based multi-policy approach effectively handles infinite-horizon MORL, forcing a trade-off between supervised and simplistic TD methods. We address this gap by adapting tabular TD inner-loop methods to infinite horizon problems, laying a principled foundation for future deep-RL extensions.

Our contributions include: **(i)** A systematic analysis of tabular multi-policy MORL in infinite-horizon settings, outlining design principles guiding TD-based deep-RL methods. **(ii)** A novel trajectory-based framework to track and execute both stationary and non-stationary policies, handling cycles and supporting robust policy selection and trade-off analysis by the user. **(iii)** Identification and resolution of spurious domination from reward horizon mismatches, reward estimate frequency, and incomplete structural information, with mechanisms for managing trajectories and cycles efficiently, supporting reliable learning and retrieval of policies.

2 Related Work on MORL

MORL seeks to optimize several, typically conflicting, objectives, with preferences often unknown beforehand. The goal is to find a set of non-dominated policies, where no objective can be improved without sacrificing another. Such policies represent trade-offs subject to preference rather than an objectively best solution, forming what is known as the Pareto Front (PF) (Hayes et al., 2021).

Multi-objective problems can be converted into single-objective ones through *scalarization*; however, choosing weights is difficult, and even exhaustive searches often miss parts of the PF where non-isomorphic weight–objective mappings cause similar weights to yield very different trade-offs (Das & Dennis, 1997; Moffaert & Nowé, 2014). Inspired by Reward-Free Exploration (RFE) (Jin et al., 2020), where agents first explore (learning) and later optimize for any reward formulation (planning), Preference-Free Exploration (PFE) (Wu et al., 2020) allows agents to learn with rewards and then search based on specific preferences. However, PFE also relies on weights to define the preference and requires a new search for every new preference set. Multi-policy methods approximate the PF directly, allowing users to analyze trade-offs without predefined preference weights. Outer-loop approaches (Roijers et al., 2015; Parisi et al., 2017; Röpke et al., 2024) iteratively derive new policies from those learned in previous runs. In contrast, inner-loop methods (Moffaert & Nowé, 2014; Ruiz-Montiel et al., 2017; Mandow & de-la Cruz, 2018; Reymond & Nowé, 2019; Reymond et al., 2022a;b), learn the entire policy set simultaneously within a single run.

In tabular approaches, Moffaert & Nowé (2014) keeps track of policies by separating returns into average immediate and future discounted rewards, while Ruiz-Montiel et al. (2017); Mandow & de-la Cruz (2018) links each policy to the next action in a state. These methods are limited to episodic, acyclic settings, restricting use in infinite-horizon problems. In deep RL, early single-run multi-policy work (Reymond & Nowé, 2019) struggled to scale even in simple domains. More recently, Reymond et al. (2022a) introduced Pareto Conditioned Networks (PCNs), training a supervised model on an iteratively updated trajectory dataset to produce actions conditioned on a target return and horizon. While PCNs yield flexible policies, they inherit key limitations of supervised/imitation-style training: costly periodic retraining, adaptation constrained by dataset refresh rate, and reliance on careful trajectory curation and exploration. In contrast, Temporal-Difference (TD) methods update policies online at each interaction, using reward feedback for immediate improvement and more reliable credit assignment — crucial in infinite-horizon settings. In this paper, we extend inner loop tabular methods to infinite-horizon settings, laying a foundation for TD deep RL extensions. For a broader MORL survey, see Hayes et al. (2021).

3 Methodology

We introduce a TD-based multi-policy MORL framework (Figure 1) that color-labels transitions (Section 3.1) and structures them into stationary or non-stationary trajectories (Section 3.2). Our method accounts for spurious dominations (Section 3.3) and enables policies with well-defined undiscounted returns in extended infinite-horizon settings (Section 3.4). It further provides explicit tracking of stationary segments (Section 3.5) and cycle detection (Section 3.6), ensuring stationarity.

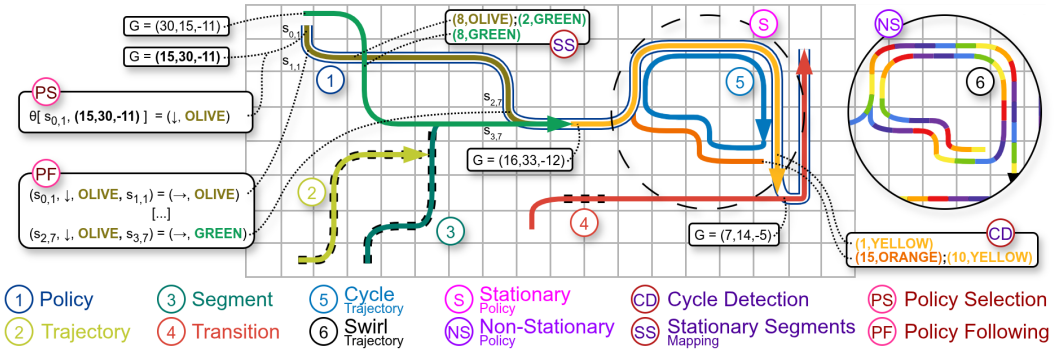

Figure 1: **Overview of the components underpinning our analysis.** A uniquely-colored segment forest: a policy (1) unfolds as a sequence of trajectories (2), each decomposed into segments (3) made of transitions (4). Trajectories may form cycles (5) or swirls (6), and policies can be stationary (S) or non-stationary (NS). We highlight cycle detection (CD), the Stationary Segments Mapping (SS), and the policy selection (PS) and policy following (PF).

3.1 Following a policy through trajectories

In RL, a policy π guides the agent along trajectories τ (each trajectory is a sequence of transitions δ), mapping states to action distributions that maximize cumulative reward. Since the agent can start from any state, a policy can also be seen as a forest of independent or converging trajectories. In multi-policy settings, multiple non-dominated actions may exist per state, producing a set of non-dominated trajectories and making it non-trivial to determine which action continues a policy. Without careful tracking, the agent may deviate from its intended trajectory and follow a dominated policy, even if each action it takes remains non-dominated (Moffaert & Nowé, 2014).

Our method follows a policy tracking approach similar to Ruiz-Montiel et al. (2017), where we maintain multiple non-dominated trajectories and update their returns via TD learning. Building on this foundation, we formalize tracking with a trajectory-centric approach where a color-label mechanism and a *Policy-Transition Table* (PTT) explicitly and intuitively tracks policies: we define an extended Q-Table $Q(s, a, c)$ mapping state s, action a, and a *color label* c, and then map, via the PTT, which transitions are part of the same trajectory (e.g., $(s, a, c) \Rightarrow (s', a', c') \Rightarrow (s'', a'', c'')$), providing a clear structure for executing policies in complex multi-policy scenarios while supporting extensions for the infinite horizon setting. With the trajectory-centric mechanism in place, the agent learns under a standard exploratory policy (e.g., ϵ-greedy), updating value estimates via TD learning. A complete algorithmic description is provided in Appendix A.

Once training is complete, policy execution proceeds as follows. The *user selects a policy* based on preferred trade-offs. The initial action a and color c follow Equation 1, where the Θ *selection operator* represents the user's choice, and ND (*non-dominated operator*) filters out dominated returns; A is the action set. To *execute the selected policy*, we use the PTT: after transitioning to next state s' (Equation 2), we query the table with (s, a, c, s') to obtain the next action a' and color c' (Equation 3). Alternating between Equations 2 and 3 keeps the agent on the intended trajectory τ.

$$\textit{(Policy Selection)} \quad (a_0, c_0) \sim \Theta\left(ND \bigcup_{a \in A;\, c \textit{ from } \delta_\tau^{(l,c)} \in \mathcal{T}(s_0,a)} Q(s_0, a, c)\right) \tag{1}$$

$$\textit{(Transition)} \quad s' \sim P(s' \mid s, a) \tag{2}$$

$$\textit{(Policy Following)} \quad (a', c') \sim \pi(s, a, c, s') \tag{3}$$

3.2 Learning stationary and non-stationary policies

A stationary policy selects actions from a fixed distribution for each state (i.e., the action does not vary with time: $\pi_t(a|s) = \pi_{t'}(a|s)$), whereas a non-stationary policy explicitly depends on time ($\pi_t(a|s) = \pi(a|s, t)$). Most works focus on non-stationary, as they can outperform stationary policies and recover the entire Pareto Front rather than just the convex part (White, 1982; Roijers

et al., 2013; Hayes et al., 2021). Still, stationary policies can be applied to problems requiring completely predictable behavior (i.e., always taking the same action in a given state), and stationary policies may be faster to learn, since stationary policies are a subset of non-stationary policies. We examine constructing both stationary and non-stationary policies and how they relate to each other.

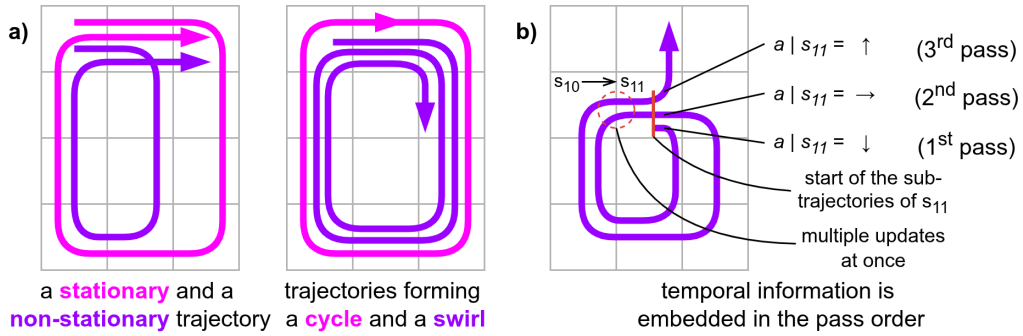

Figure 2: **Trajectory types and temporal encoding:** **(a)** Stationary trajectory and cycle (outer paths) vs. non-stationary trajectory and swirl (inner paths); **(b)** Time is encoded by pass order.

Whether a policy is strictly stationary or generally non-stationary depends on whether trajectories are allowed to form cycles (Figure 2a, left). Because the decision is made locally, it can capture only limited distinctions: forming cycles produces closed, repeating behavior, ensuring strict stationarity (outer, pink trajectories), while avoiding cycles allows the broader set of non-stationary policies (inner, purple trajectories). To enforce this distinction, we define that stationary policies assign the same color label to transitions whose trajectories end in the same state, forming single-colored trajectory trees, whereas non-stationary policies assign each transition a distinct color label (Equation 4, refined from Equation 3). A key consequence is that stationary trajectories can limit each other's expansions: distinct branches of the same color cannot overlap since there cannot be a transition with a repeated color leading to two distinct paths. This issue is addressed in Section 3.5. Cycles are thus possible only by connecting matching-colored transitions. On the other hand, non-stationary policies produce *swirls*: trajectories that revisit the same states yet treat transitions as distinct, enabling cyclic-like behavior without committing to strict stationarity (Figure 2a, right).

$$\textit{(Policy Following)} \qquad (a', c') \sim \pi(s, a, c, s') \quad , \begin{cases} c = c' & \text{, for stationary policies} \\ c \neq c' & \text{, for non-stationary policies} \end{cases} \tag{4}$$

In our trajectory-based formulation, there is no need to track time — time is implicit in the structure of the trajectory itself. For instance, in Figure 2b, an agent visiting state s_{11} three times (t=0, 6, 14) the position of each occurrence in the trajectory (first, second, third) determines the appropriate action (down, right, and up, respectively), allowing the agent to take different actions at each occurrence of s without referencing an explicit time step t. This implicit encoding offers a key advantage: the agent can update the return estimates for all occurrences of a state in all trajectories at once (e.g., when moving from s_{10} to s_{11}, update all transitions), rather than revisiting that state at different time steps as required in time-dependent policies, leading to a greater sample efficiency. Moreover, each transition in a given trajectory also marks the start of a well-defined (smaller) trajectory, capturing the information from that point forward. In Figure 2b, state s_{11} appears in three "sub-trajectories": one covering all three visits, one for the last two, and one with the final visit (i.e., different tail sizes of the trajectory).

3.3 Spurious Domination in Cycles, Swirls, and Regular Trajectories

Domination occurs when a trajectory τ_1 yields a higher return than τ_2. *Spurious* (false) dominations arise when there is: (i) *differences in horizon length*: when a longer trajectory may seem better simply by having more opportunity to accumulate reward, even if a shorter one would eventually yield a higher return (Figure 3a, left). This problem is exacerbated with cycles or swirls, which repeatedly dominate the very transitions that support it (Figure 3a, right); (ii) *reward estimate frequency* (also known as reward-frequency): when frequently visited transitions converge faster to

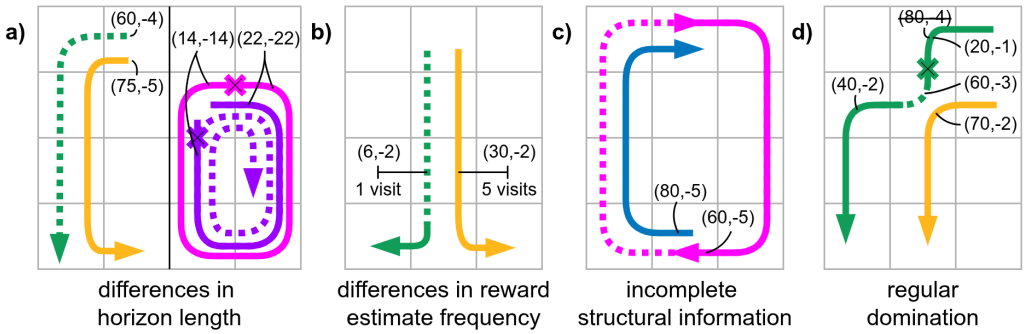

Figure 3: **(Spurious) Dominations.** Dotted lines: dominated paths; 'x': early termination; tuples: return and length (simplified). Spurious domination due to **(a)** differences in length of trajectories (left), and cycles and swirls (right); **(b)** differences in reward estimate frequency; **(c)** undetected cycle (outer path). **(d)** Regular domination: dominated trajectory gets split and later updated.

stable and higher-valued reward estimates, dominating less visited ones (Figure 3b); or (iii) *incomplete structural information*: when undetected cycles may be incorrectly dominated by higher-return trajectories, despite their structural ability to yield arbitrarily longer paths and greater returns (Figure 3c).

To address (i), we include a step-based reward component encoding *trajectory length* l (Equation 5). In strictly positive-reward settings, a negative length term prevents trivial domination by penalizing unnecessarily long trajectories; in strictly negative settings, a positive term allows the agent to learn longer trajectories that would otherwise be dominated due to compounding penalties; while in mixed rewards, both cases may occur. We handle all cases uniformly by treating length as a negative reward and applying a min-shift to all rewards. This normalization, along with the uniform growth of the trajectory length, supports fair comparisons and consistent treatment of trajectory length. To address (ii), we use *average rewards* (Equation 6), normalizing cumulative rewards r_{sum} by visit count r_{count} for balanced convergence. Together, (i) and (ii) address spurious domination in non-stationary policies, where trajectories and swirls extend naturally. We address (iii) in Section 3.6, where cycle information is explicitly managed.

$$\textit{(Trajectory Length)} \qquad l(s,a,c) = \begin{cases} l(s',a',c') + 1 & \text{, if } (s',a',c') \in \tau \\ 1 & \text{, if } (s',a',c') \notin \tau \end{cases} \tag{5}$$

$$\textit{(Average Reward)} \qquad \bar{r}(s,a) = r_{sum}(s,a)/r_{count}(s,a) \tag{6}$$

Ultimately, domination arises only when a shorter trajectory τ_s yields higher returns than an existing trajectory τ with the same start and end states. When domination occurs, the τ gets split at the start state into a valid sub-trajectory, connected to the trajectory's end, and to dangling, outdated transitions, each of which must be re-learned — as in Figure 3d (this issue is mitigated in Section 3.5). If left outdated, the agent eventually reaches a state missing the corresponding PTT entry (from Section 3.1) despite the trajectory being unfinished from the agent's perspective. Outdated transitions can also be pruned. Furthermore, after learning, the length component can be omitted from policy selection, leaving for execution only the trajectories that maximize the original task rewards.

3.4 Infinite Horizon Behavior and Undiscounted Returns

With the inclusion of positive cycles and swirls, the problem naturally shifts from an episodic to an infinite horizon setting, where cumulative returns may grow unbounded. Although we define trajectories with well-defined return G and length l, the issue remains as the trajectories can still grow unbounded in length. A common solution is to apply discounting to ensure returns converge. However, this undermines interpretability: the final return no longer reflects the total reward collected, but rather a weighted sum that is harder to interpret. Additionally, discounting the trajectory length component has the same effect as bounding the trajectories by a *maximum length*, which raises the question of whether finite trajectories can approximate infinite-horizon behavior.

We define a policy not as a single trajectory but as a sequence of trajectories, allowing the agent to run indefinitely while letting users choose a learned return or a custom one, with the agent adjusting collected returns (across subsequent trajectories) to preserve the chosen *average return* (Equation 7). This modeling also allows swirls to emulate cycles: once a trajectory ends, the agent can repeatedly reuse overlapping parts of the swirl. Infinite trajectories are thus unnecessary, and maximum length can be tuned, particularly when longer trajectories add no further gain in maintaining the average return. A planning phase can optionally complement learning by pre-checking the feasibility of custom returns or verifying that learned trajectories suffice to guarantee the average returns.

$$\textit{(Average Return)} \qquad \bar{G}(s,a,c) = \bar{G}(s,a,c) + \frac{\bar{r}(s,a) - \bar{G}(s',a',c')}{l(s',a',c')+1} \tag{7}$$

With the average return defined, we introduce the Q-function (Equation 8), capturing the return G as the product of the average return $\bar{G}$ and trajectory length l. Together, the components introduced so far suffice for non-stationary policies; the following sections continue to address stationary ones.

$$\textit{(Q-Function)} \qquad Q(s,a,c) = G(s,a,c) = \bar{G}(s,a,c) \times l(s,a,c) \tag{8}$$

3.5 Identifying and Managing Stationary Trajectories

As noted in Section 3.2, stationary trajectories may block one another, since a transition cannot belong to multiple trajectories of the same color. For example, in Figure 4a (left), two trajectories diverge and later converge. Only one can retain the shared color, forcing the other to be recolored. Recoloring, however, breaks the stationarity guarantee presented in Equation 4, even though the trajectories remain stationary, which can cause the agent to miss many stationary trajectories.

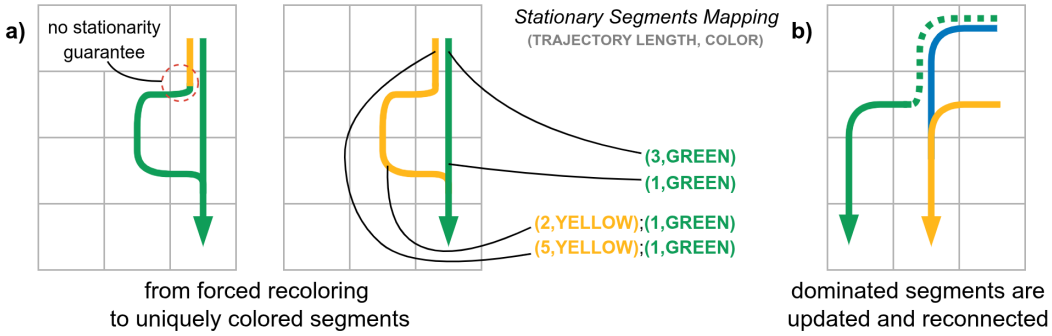

Figure 4: **Managing stationary trajectories.** Dotted lines show dominated paths. **(a, left)** Stationary trajectory must be recolored; **(a, right)** We propose uniquely-colored segments and a Stationary Segments Mapping. **(b)** Revisited domination: the segment is updated and reconnected (in blue).

We propose (i) decomposing each trajectory into *segments* σ — defined as a sequence of transitions that forms a sub-partition of trajectories — and assigning a unique color to each segment, $\sigma^{(c)}$, such that whenever a trajectory branches into a new path, a new color is assigned to the forming segment; (ii) introducing a *Stationary Segments Mapping* (SSM) that tracks such colored segments along with where they start in the trajectory (by the remaining length) — depicted in Figure 4a (right). To preserve stationarity across trajectories, the agent must then track the order of visited segments during execution; and (iii) applying cycle detection (discussed in Section 3.6) to prevent trajectories from extending past a cycle and becoming non-stationary. This approach replaces single-colored trajectory trees with uniquely-colored segment trees, where each new branching introduces a new color, allowing stationary trees to extend and change colors flexibly while preserving stationarity. The policy-following rule is accordingly updated from Equation 4 to Equation 9:

$$\textit{(Policy Following)} \qquad (a',c') \sim \pi(s,a,c,s'), \begin{cases} c = c' \text{ or } c \neq c' & \text{for stationary policies (w/ SSM)} \\ c \neq c' & \text{for non-stationary policies} \end{cases} \tag{9}$$

To preserve the structure of these segment trees, no two segments may share the same color. This requires revisiting trajectory domination (Section 3.3). When a domination occurs, the head segment

of the split now receives a new color c, updated return G and length l, and is reconnected to the dominant trajectory (Figure 4b). In addition to the trajectory return G and length l, we also store the segment-level return G and length l, which under these domination dynamics, will remain up-to-date. If a length mismatch is detected when switching segments, the entire segment can be updated without revisiting every state (as observed in Section 3.3), improving sample efficiency.

3.6 Performing Cycle Detection

Cycles offer a clear benefit of representing an infinite horizon with a well-defined return while remaining bounded in length. However, detecting cycles is nontrivial due to an inherent ambiguity. As illustrated in Figure 5a, when connecting transitions from s_{10} to s_{11}, in faded green, it is unclear whether this connection forms a cycle (in orange) — that is, whether the trajectory eventually leads back to itself — or merely encounters a separate trajectory and forms a longer path (in blue), since both cases yield the same return G and length l. In addition to the ambiguity, cycles often lack a well-defined end state (Section 3.3), causing them to dominate their tail transitions and rendering incoming trajectories repeatedly outdated. If a maximum trajectory length is imposed, the cycle eventually exhausts the length budget, preventing any incoming trajectory from reaching the cycle.

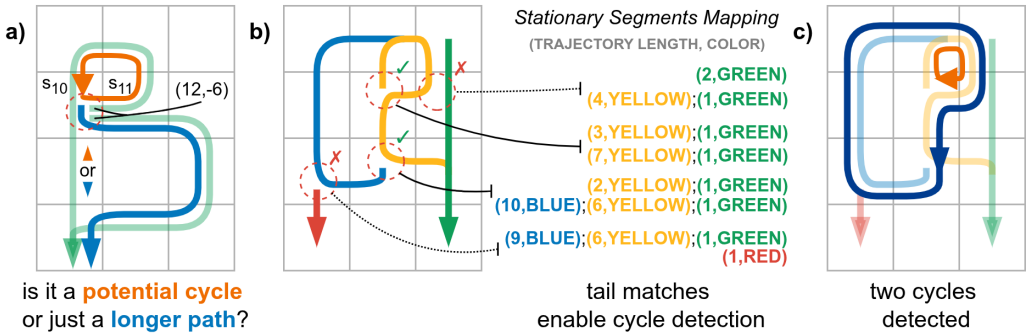

Figure 5: **Performing cycle detection.** Tuples denote return and length (simplified). **(a)** Ambiguity: cycle or longer path? **(b)** Tail matches confirm a cycle. **(c)** Resulting detected cycles.

Building on the segment tree structure defined in Section 3.5, we propose the following cycle detection mechanism. If the potential connection occurs between transitions with the same color c (e.g., yellow to yellow, Figure 5b), they must belong to the same segment, and a cycle can be safely formed. If the colors differ, we must determine whether both transitions lie on the same trajectory. This determination is done by checking whether the shorter trajectory is a suffix of the longer one, that is, whether one is the tail of the other. To verify this, we use the SSM (Section 3.5) to compare the segments by color c and length l. If one is indeed a suffix of the other, the transitions belong to the same trajectory (e.g., blue to yellow); if not, the paths must have diverged and represent distinct trajectories (e.g., red to blue and yellow to green), as illustrated in Figure 5b by the red *Xs*.

Once a cycle is detected (two resulting cycles in Figure 5c), we store a copy of its transitions for two reasons. First, all cycle transitions receive a new color c, with each transition marked as part of the cycle and sharing the same return G and length l. This copy ensures the original unmodified trajectories remain stationary. With cycle detection in place, the agent avoids exhausting maximum trajectory length budgets and repeatedly dominating its own transitions (discussed in Section 3.3).

4 Ablation Studies

We analyze key design choices through ablations on the well-known DeepSeaTreasure (Vamplew et al., 2011) problem, implemented in MO-Gymnasium (Alegre et al., 2022) and adapted to an infinite-horizon setting (agent goes to the initial state after the end of an episode). DeepSeaTreasure is a gridworld that tests the trade-off between treasure value and time (i.e., choosing between shallow, low-value treasures and deeper, higher-value ones). Agents are trained for 10,000 steps, averaging results over 20 seeds. Each *ablation study* (AS1-AS8) evaluates a specific aspect from Section 3, with Figures 6a–6d presenting the main results, and hypothesis, experiments, metrics, and results summarized in Table 1, and further ablation explanations are below the table.

Table 1: **Ablation plan.** Each row tests a design choice with metrics and expected outcomes.

S - Stationary; NS - Non-stationary; NPS - Naive Policy Selection; PF - Pareto Front; ND - non dominated.

Id	Ablation	Hypothesis / Experiment / Metrics / Results
AS1	Policy tracking *vs.* state-wise ND selection	Lack of trajectory-level tracking causes agents to follow dominated policies. **Experiment:** NPS (i.e., random non-dominated actions) vs. S and NS policies. **Metrics:** Policy coverage. **Results:** NPS gets less treasure value because it combines state-wise ND actions into trajectories that often yield dominated solutions (Figure 6a).
AS2	Learning the full/convex PF with NS/S policies	Allowing NS policies enables full PF recovery; restricting to S policies limits recovery to its convex subset (thus lower hypervolume). **Experiment:** S vs. NS policies. **Metrics:** Policy coverage; convergence speed. **Results:** NS polices recover the full PF, whereas S ones recover only the convex subset (Figure 6a and 6b).
AS3	Horizon-length bias in domination	Longer trajectories spuriously dominate others without normalization. **Experiment:** Disable trajectory length. **Metrics:** Hypervolume over time. **Results:** No observable difference, as the environment already incorporates a length-like component, time penalty (Figure 6b).
AS4	Reward-frequency bias in domination	Reward frequency biases domination, causing frequent low-reward policies to appear superior to sparse high-reward ones. **Experiment:** Disable average reward. **Metrics:** Hypervolume over time. **Results:** Imbalanced reward frequencies destabilize learning and degrade performance (Figure 6b).
AS5	Structural-info gaps in domination	Failure to detect cycles biases Pareto comparisons, making cycles appear dominated by high-return trajectories. **Experiment:** S without cycle detection. **Metrics:** Hypervolume over time. **Results:** Failing to detect cycles sharply degrades performance (Figure 6b).
AS6	Maximum trajectory-length sensitivity and policy selection	Maximum trajectory length affects whether policies sustain stable returns. **Experiment:** Vary max length (2, 5, 10, 20, 30). **Metrics:** Hypervolume from promised (max_length steps) vs. realized (1,000 steps). **Results:** Promised and realized returns align, indicating sustained return over following multiple trajectories (Figure 6c).
AS7	Blocking segments and SSM	SSM prevents blocking and coloring conflicts, enabling stationary policies to be recovered correctly. **Experiment:** Disable SSM. **Metrics:** Hypervolume from promised (max_length steps) vs realized (1,000 steps). **Results:** Promised and realized returns not only align but also improve, confirming correct recovery with SSM (Figure 6c).
AS8	Split-and-reconnect segment updates	Applying segment updates improves sample efficiency and accelerates recovery after dominated trajectories. **Experiment:** Disable reconnection. **Metrics:** # of splits; # of mismatches. **Results:** Split-and-reconnect yields substantially fewer trajectory-length mismatches (Figure 6d).

Figure 6a (AS1, AS2) reports returns for non-stationary (NS), stationary (S), and naive policy selection (NPS, i.e., selecting random non-dominated actions). We tested whether policies achieved the learned returns on average, with NPS using the same target returns as NS. NS policies recover the full Pareto Front, whereas S policies cover only the convex subset, and NPS yields many dominated solutions. Note that NPS only performs better for small time penalties (-1 and -2) because it captures better treasure/time penalty on average, while NS and S obtain their learned return.

Figure 6b (AS2–AS5) shows hypervolume[1] convergence for S and NS policies, around 1,000 time steps. The figure also highlights spurious domination, represented by 3 biases: horizon length bias (red), reward-frequency bias (purple), and structural information (pink). Note that square markers are all for NS and triangles for S policies. Disabling trajectory-length (AS3) does not affect horizon-

[1]A larger hypervolume indicates a better-performing algorithm for *convergence* (solutions are close to the true PF) and *diversity* (solutions are well-spread along the PF, that is, a good range of trade-offs).

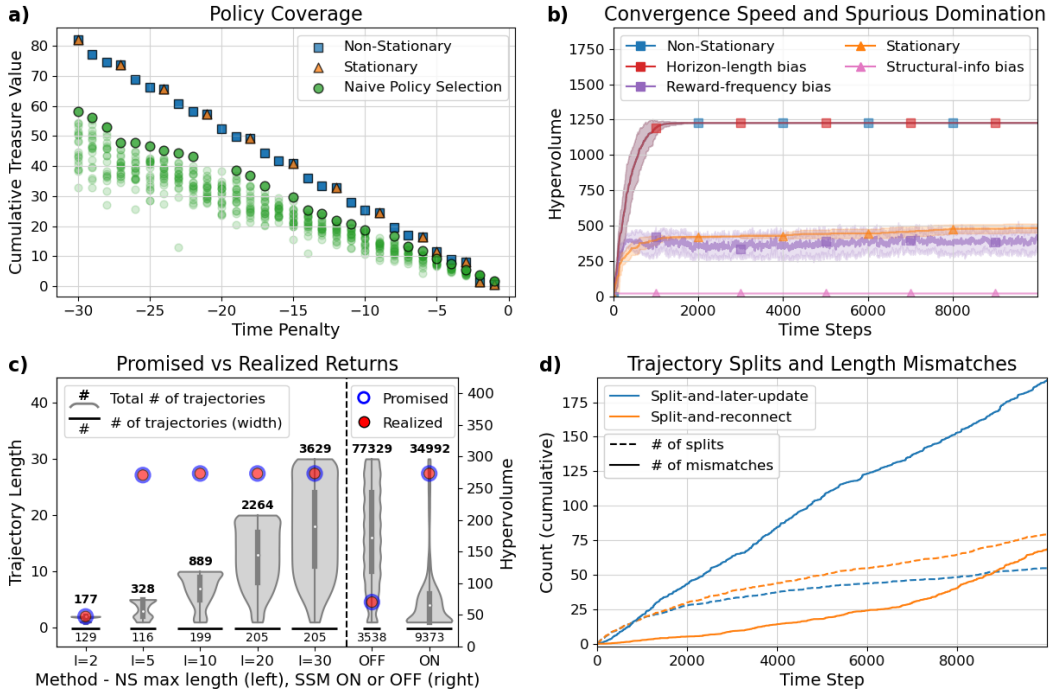

Figure 6: **Ablation study results.** Each plot shows different aspects of policy evaluation: (a) returns for Stationary, Non-Stationary, and naive policies; (b) convergence and spurious domination; (c) trajectory lengths and promised vs realized returns; (d) trajectory splitting and length mismatches.

length bias (red line), as the time penalty already acts as a trajectory length, balancing the returns. In contrast, the purple line shows that the reward-frequency bias (AS4) case causes slight instability and lower performance. Disabling cycle detection in S (AS5) removes structural guidance, further limiting performance (pink line with almost no hypervolume).

Figure 6c (AS6, AS7) examines maximum-length sensitivity in NS policies and the effect of enabling SSM in S policies. Violin plots show trajectory length distributions, with *promised* returns representing values presented to the user and *realized* returns representing values obtained after 1,000 steps. The match between promised and realized returns (right Y axis shows the hypervolume) indicates the agent sustains average returns over longer horizons. In DeepSeaTreasure, the return $(8.2, -3)$ yields the best average among all treasures, explaining why $l = 2$ underperforms, whereas $l \geq 5$ achieves greater hypervolume. Enabling SSM is crucial for stationary policies: without it, cycle detection is disabled, limiting long-horizon performance. The figure also shows that SSM causes trajectories to be shorter, as cycle detection halts growth once a cycle forms.

Figure 6d (AS8) reports splits (dashed lines) and mismatches (solid lines). Mismatches occur when trajectories end prematurely, while splits indicate domination. Since splits are similar, lower mismatches for Split-and-Reconnect (orange lines) demonstrate the agent quickly reconnects and updates segments, leading to a higher sample efficiency.

5 Conclusion

We introduced design principles that extend TD-based multi-policy MORL from episodic to infinite-horizon settings. Our trajectory-centric framework enables reliable tracking and execution of both stationary and non-stationary policies, supports cycle detection, and prevents spurious domination from horizon mismatches or incomplete information. Through ablation studies, we showed how each principle contributes to recovering diverse, interpretable, and stable policies. Looking ahead, these principles provide a foundation for extending TD-based methods to deep settings, where scalable representation and online adaptation remain open challenges. By ensuring well-defined behavior in infinite horizons, our framework provides a path toward practical multi-policy MORL systems that can support preference-driven decision-making in complex domains.

Reproducibility statement

We have taken several measures to ensure reproducibility. A detailed description of our algorithm, training procedures, and evaluation protocols is provided in Section 3 and 4 of the main text, with further implementation details in Appendix A. We also provide our code and scripts in the supplementary materials, which enable end-to-end reproduction of all experiments.

References

Lucas N. Alegre, Florian Felten, El-Ghazali Talbi, Grégoire Danoy, Ann Nowé, Ana L. C. Bazzan, and Bruno C. da Silva. MO-Gym: A library of multi-objective reinforcement learning environments. In *Proceedings of the 34th Benelux Conference on Artificial Intelligence BNAIC/Benelearn 2022*, 2022.

Indraneel Das and John E. Dennis. A closer look at drawbacks of minimizing weighted sums of objectives for pareto set generation in multicriteria optimization problems. *Structural optimization*, 14:63–69, 1997.

Conor F. Hayes, Roxana Ruadulescu, Eugenio Bargiacchi, Johan Kallstrom, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M. Zintgraf, Richard Dazeley, Fredrik Heintz, Enda Howley, Athirai Aravazhi Irissappane, Patrick Mannion, Ann Now'e, Gabriel de Oliveira Ramos, Marcello Restelli, Peter Vamplew, and Diederik M. Roijers. A practical guide to multi-objective reinforcement learning and planning. *Autonomous Agents and Multi-Agent Systems*, 36, 2021.

Chi Jin, Akshay Krishnamurthy, Max Simchowitz, and Tiancheng Yu. Reward-free exploration for reinforcement learning. *ArXiv*, abs/2002.02794, 2020.

Lawrence Mandow and José-Luis Pérez de-la Cruz. Pruning dominated policies in multiobjective pareto q-learning. In *Conferencia de la Asociación Española para la Inteligencia Artificial*, 2018.

Kristof Van Moffaert and Ann Nowé. Multi-objective reinforcement learning using sets of pareto dominating policies. *J. Mach. Learn. Res.*, 15:3483–3512, 2014.

Simone Parisi, Matteo Pirotta, and Jan Peters. Manifold-based multi-objective policy search with sample reuse. *Neurocomputing*, 263:3–14, 2017.

Mathieu Reymond and Ann Nowé. Pareto-dqn: Approximating the pareto front in complex multi-objective decision problems. 2019.

Mathieu Reymond, Eugenio Bargiacchi, and Ann Now'e. Pareto conditioned networks. In *Adaptive Agents and Multi-Agent Systems*, 2022a.

Mathieu Reymond, Conor F. Hayes, Lander Willem, Roxana Rădulescu, Steven Abrams, Diederik M. Roijers, Enda Howley, Patrick Mannion, Niel Hens, Ann Now'e, and Pieter J. K. Libin. Exploring the pareto front of multi-objective covid-19 mitigation policies using reinforcement learning. *Expert Syst. Appl.*, 249:123686, 2022b.

Diederik M. Roijers, Peter Vamplew, Shimon Whiteson, and Richard Dazeley. A survey of multi-objective sequential decision-making. *ArXiv*, abs/1402.0590, 2013.

Diederik M. Roijers, Shimon Whiteson, and Frans A. Oliehoek. Computing convex coverage sets for faster multi-objective coordination. *J. Artif. Intell. Res.*, 52:399–443, 2015.

Willem Röpke, Mathieu Reymond, Patrick Mannion, Diederik M Roijers, Ann Nowé, and Roxana Rădulescu. Divide and conquer: Provably unveiling the pareto front with multi-objective reinforcement learning. *arXiv preprint arXiv:2402.07182*, 2024.

Manuela Ruiz-Montiel, Lawrence Mandow, and José-Luis Pérez de-la Cruz. A temporal difference method for multi-objective reinforcement learning. *Neurocomputing*, 263:15–25, 2017.

Peter Vamplew, Richard Dazeley, Adam Berry, Rustam Issabekov, and Evan Dekker. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine Learning*, 84: 51–80, 2011.

Douglas John White. Multi-objective infinite-horizon discounted markov decision processes. *Journal of Mathematical Analysis and Applications*, 89:639–647, 1982.

Jingfeng Wu, Vladimir Braverman, and Lin F. Yang. Accommodating picky customers: Regret bound and exploration complexity for multi-objective reinforcement learning. In *Neural Information Processing Systems*, 2020.

A Algorithmic details and analysis

Tabular RL in the infinite-horizon setting follows the structure in Algorithm 1. The algorithm maintains a Q-Table of state–action values, updated incrementally as the agent interacts with the environment. At each step, the agent selects an action via an exploration policy (e.g., ϵ-greedy), observes the reward and next state, and applies the TD update on the Q-Table. As usual, the discount factor γ weights future rewards, while the learning rate α controls how quickly new information overrides old estimates. Repeated interactions refine the Q-values, which can then define a greedy policy. In Sections A.1 and A.2, the *Q-Table update* is replaced by our non-stationary and stationary *Multi-Policy Update*, respectively, and Section A.3 discusses their time and space complexity.

Algorithm 1 Tabular RL for Infinite-Horizon

Require: State space $\mathcal{S}$, action space $\mathcal{A}$, learning rate α, discount factor γ, exploration policy π_ϵ, maximum steps T

1: Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}$
2: **for** $t = 1$ **to** T **do**
3: Observe current state s_t
4: Select action a_t according to $\pi_\epsilon(Q, s_t)$
5: Execute a_t, observe reward r_t and next state s_{t+1}
6: **Update Q-Table:**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \Big]$$

7: $s_t \leftarrow s_{t+1}$
8: **end for**
9: **return** $Q(s, a)$

A.1 Non-Stationary

Algorithm 2 outlines the update procedure for the non-stationary multi-policy. The method extends standard TD updates to maintain multiple policies simultaneously via the Q-Table and a Policy-Transition Table — PTT (from Section 3.1).

Step 1. Retrieve from Q-Table the non-dominated set at the next state s', consisting of tuples (s', a', c') whose Q-values are non-dominated.

Step 2. If the set is empty, then s' is being visited for the first time and $Q(s', a', c')$ has no entries. In this case, add a new Q-value estimate for (s, a, c) from reward r with trajectory length 1.

Step 3. Fetch all PTT entries (s, a, c, s', a', c') associated with the current transition (s, a, s'). Delete PTT entries whose successors (s', a', c') are no longer in the non-dominated set (obsolete). Their Q-value estimates (i.e., (s, a, c) restart with reward r and trajectory length 1. Add any entries from the non-dominated set to the PPT, associating them with the current (s, a) and generating a new color c for each association (s, a, c, s', a', c'), thereby extending trajectories from s' to s with distinct colors c_i and adding 1 to the trajectory length.

Step 4. Use each updated PTT entry to update the Q-Table according to Equations 7 and 8.

Step 5. Prune dominated entries from (s, a) in the Q-Table and then in the PTT, preserving only Pareto-optimal (s, a, c) entries for future updates.

Algorithm 2 Multi-Policy Update: Non-Stationary

Require: Current state s, action a, next state s', Q-Table, Policy Transition Table (PTT)

Step 1: Fetch ND entries
next_state_nd_set ← fetch_nd_set(Q, s') ▷ (s', a', c') entries
Step 2: Initialize Q-Table if no ND entries at s'
if next_state_nd_set is empty **then**
    $c \leftarrow$ generate_new_color()
    ADD entry (s, a, c) *to Q-Table*
end if
Step 3: Fetch PTT entries for transition (s, a, s'), add new and discard old transitions
ptt_entry_set ← fetch_ptt_entries(PTT, s, a, s') ▷ (s, a, c, s', a', c') entries
ptt_old_entries ← ptt_entries_set − next_state_nd_set ▷ Compare (s', a', c'), keep PTT entry
for (s, a, c, s', a', c') in ptt_old_entries **do**
    DELETE entry (s, a, c, s', a', c') *from PPT*
end for(s, a, c, s', a', c')
ptt_new_entries ← next_state_nd_set − ptt_entries_set ▷ Compare (s', a', c'), keep (s', a', c')
for (s', a', c') in ptt_new_entries **do**
    $c \leftarrow$ generate_new_color()
    ADD entry (s, a, c, s', a', c') *to PTT*
end for
Step 4: Update Q-Table for each PTT entry
for (s, a, c, s', a', c') in ptt_entries **do**
    ADD/UPDATE entry (s, a, c) to Q-Table ▷ Update done via Equation 7 and 8
end for
Step 5: Clean up dominated entries
state_action_dominated_set ← *fetch_dominated_set(Q, s, a)*
for (s, a, c) in state_action_dominated_set **do**
    DELETE entry (s, a, c) *from Q-Table*
    DELETE entry from PTT which starts with (s, a, c) *as in* (s, a, c, s', a', c')
end for

A.2 STATIONARY

Algorithm 3 outlines the update procedure for the stationary multi-policy. In addition to the Q-Table and Policy Transition Table (PTT), it incorporates the Stationary Segments Mapping — SSM (from Section 3.5).

Step 1. Retrieve from Q-Table the non-dominated set at the next state s', consisting of tuples (s', a', c') whose Q-values are non-dominated.

Step 2. If the set is empty, then s' is being visited for the first time and $Q(s', a', c')$ has no entries. In this case, add a new Q-value estimate for (s, a, c) from reward r with trajectory length 1.

Step 3. Fetch all PTT entries (s, a, c, s', a', c') associated with the current transition (s, a, s'). **Unlike the non-stationary case,** no entries are obsolete: dominated trajectories are reconnected. Add non-dominated successors from Step 1 to the PTT, linking them to (s, a) with the appropriate color c. Assign a new color if (s, a, c) is already reached by another entry; otherwise, reuse the existing color, thereby extending trajectories from s' to s with trajectory length $+1$.

Step 4. Update the Q-Table using each revised PTT entry, according to Equations 7 and 8.

Step 5. Check for cycles. If one is found, duplicate its transitions and assign them a new shared color; otherwise, update the SSM with the new PTT entry.

Step 6. Prune dominated entries from (s, a) in the Q-Table. Reconnect all prior associations from pruned entries to their dominant counterparts in the PTT, updating their Q-Values, and generating new colors as in Step 3. This preserves only Pareto-optimal (s, a, c) entries for future updates.

Algorithm 3 Multi-Policy Update: Stationary

Require: Current state s, action a, next state s', Q-Table, Policy Transition Table (PTT), Stationary Segments Mapping (SSM)

Step 1: Fetch ND entries
next_state_nd_set ← fetch_nd_set(Q, s') ▷ (s', a', c') entries
Step 2: Initialize Q-Table if no ND entries at s'
if next_state_nd_set is empty **then**
    $c \leftarrow$ generate_new_color()
    ADD entry (s, a, c) *to Q-Table*
end if
Step 3: Fetch PTT entries for transition (s, a, s') and add new transitions
ptt_entry_set ← fetch_ptt_entries(PTT, s, a, s') ▷ (s, a, c, s', a', c') entries
ptt_new_entries ← next_state_nd_set − ptt_entries_set ▷ Compare (s', a', c'), keep (s', a', c')
for (s', a', c') in ptt_new_entries **do**
    $c \leftarrow$ fetch_or_generate_color(s', a', c') ▷ Repeat color if first extension, new one otherwise
    ADD entry (s, a, c, s', a', c') *to PTT*
end for
Step 4: Update Q-Table for each PTT entry
for (s, a, c, s', a', c') in ptt_entries **do**
    ADD/UPDATE entry (s, a, c) to Q-Table ▷ Update done via Equation 7 and 8
end for
Step 5: Check for cycles and update the SSM
for (s, a, c, s', a', c') in ptt_entries **do**
    if cycle is detected in PTT **then**
        Identify all transitions in the cycle
        $c \leftarrow$ generate_new_color()
        for each transition in the cycle **do**
            DUPLICATE the transition
            UPDATE SSM with the new PTT entry
        end for
    else
        UPDATE SSM with the new PTT entry
    end if
end for
Step 6: Clean up dominated entries
dominated_set ← fetch_dominated_set(Q, s, a)
for each d:(s, a, c) in dominated_set **do**
    for each PTT entry ending in (s, a, c), i.e., $(s^d_{prev}, a^d_{prev}, c^d_{prev}, s, a, c)$ **do**
        $c_{\text{new}} \leftarrow$ fetch_or_generate_color() ▷ As in Step 3
        RECONNECT previous links with color c and update Q-Table and PTT with c_{new}
    end for
    DELETE Q-Table entry (s, a, c)
    DELETE PTT entries starting from (s, a, c)*, i.e.,* (s, a, c, s', a', c')
end for

A.3 Algorithmic Complexity

We summarize the worst-case computational complexity of the stationary and non-stationary multi-policy updates.

The **non-stationary update** is dominated by computing the non-dominated set for the next state (Step 1 in Algorithm 2). Let M denote the number of objectives and $N_{s'}$ the total number of Q-value estimates for all actions a' in the next state s'. The worst-case complexity is $O(MN_{s'}^2) \sim O(N_{s'}^2)$, as the number of objectives is constant. This arises from pairwise comparisons of all Q-value vectors in s' across objectives.

In addition to doing the same work as the non-stationary update, the **stationary update** also iterates over stationary segments (from SSM) when checking for cycles (Step 5 in Algorithm 3) and over

outdated segments to update and reconnect them (Step 6 in Algorithm 3). Let S denote the number of stationary segments, N_{sa} the number of Q-value estimates for the current state-action pair (s, a) used for cycle detection, E_D and E_d the numbers of dominant and dominated entries, respectively, and Ld the dominated trajectory length. The worst-case complexity is $O(MN_{s'}^2 + SN_{sa}N_{s'} + L_dE_dE_D) \sim O(N_{s'}^2 + SN_{sa}N_{s'} + L_dE_dE_D)$, where the first term corresponds to computing the non-dominated set, the second to iterating over stationary segments for cycle detection, and the third to updating dominated segments. Here, $N_{s'}^2 \geq N_{sa}N_{s'}$ since N_{sa} counts only the estimates for the current action. In the worst case, S can be as large as the trajectory length of the smaller trajectory in each pair comparison.

For both stationary and non-stationary updates, **space complexity** is dominated by the Q-Table and PTT. The Q-Table stores one entry per non-dominated trajectory for each state-action pair, requiring at most $O(|S||A|N_{\max})$ space, where $N_{\max}$ is the maximum number of non-dominated trajectories per state-action. The PTT stores the transitions leading to these trajectories, with a worst-case bound of $O(|S|^2|A|N_{\max})$, though in practice it is much smaller. Similarly, the SSM stores stationary segments, with space proportional to the number of non-dominated segments. Because the number of stationary segments is smaller than the number of transitions, the SSM is smaller than the PTT.

B USE OF LARGE LANGUAGE MODELS (LLMS)

We used Large Language Models (LLMs) to support writing and research. Specifically, they were employed to polish and refine phrasing for clarity and conciseness, to assist in exploring alternative framings of ideas (e.g., refining how concepts and contributions are presented, not generating new research directions), to complement traditional tools (e.g., Google Scholar) in identifying related work, and to provide limited coding support, such as boilerplate code for plotting or debugging assistance. The models did not contribute at the level of a coauthor, and all research design, implementation, analysis, and conclusions are our own.