Specification-Guided Learning of Nash Equilibria with High Social Welfare

Kishor Jothimurugan
University of Pennsylvania

Suguman Bansal
University of Pennsylvania

Osbert Bastani
University of Pennsylvania

Rajeev Alur
University of Pennsylvania

Abstract

Reinforcement learning has been shown to be an effective strategy for automatically training policies for challenging control problems. Focusing on non-cooperative multi-agent systems, we propose a novel reinforcement learning framework for training joint policies that form a Nash equilibrium. In our approach, rather than providing low-level reward functions, the user provides high-level specifications that encode the goal of each agent. Then, guided by the structure of the specifications, our algorithm searches over policies to identify one that provably forms an ϵ-Nash equilibrium (with high probability). Importantly, it prioritizes policies in a way that maximizes social welfare across all agents. Our empirical evaluation demonstrates that our algorithm computes equilibrium policies with high social welfare, whereas state-of-the-art baselines either fail to compute Nash equilibria or compute ones with comparatively lower social welfare.

1 Introduction

Reinforcement learning (RL) is an effective strategy for automatically synthesizing controllers for challenging control problems. As a consequence, there has been interest in applying RL to multi-agent systems. For example, RL has been used to coordinate agents in cooperative systems to accomplish a shared goal [Neary et al., 2021]. Our focus is on non-cooperative systems, where the agents are trying to achieve their own goals [Littman, 1994]; for such systems, the goal is typically to learn a policy for each agent such that the joint strategy profile forms a Nash equilibrium.

A key challenge facing existing approaches is task specification. First, they typically require that the task for each agent is specified as a reward function. However, reward functions tend to be very low-level, making them difficult to manually design; furthermore, they often obfuscate high-level structure in the problem known to make RL more efficient in the single-agent [Icarte et al., 2018] and cooperative [Neary et al., 2021] settings. Second, they typically focus on computing an arbitrary Nash equilibrium. However, in many settings, the user is a social planner trying to optimize the overall social welfare of the system, and most existing approaches are not designed to optimize welfare.

We propose a novel multi-agent RL framework for learning policies from high-level specifications (one specification per agent) such that the resulting joint policy (i) has high social welfare, and (ii) is an ϵ-Nash equilibrium (for a given ϵ). We formulate this problem as a constrained optimization problem where the goal is to maximize social welfare under the constraint that the joint policy is an ϵ-Nash equilibrium. Our algorithm for solving this optimization problem uses an enumerative search strategy. First, it enumerates candidate policies in decreasing order of social welfare. To ensure a tractable search space, it restricts to policies that conform to the structure of the user-provided specification. Then, for each candidate policy, it uses an explore-then-exploit self-play RL algorithm [Bai and Jin,

Presented at Workshop on Safe and Robust Control of Uncertain Systems, NeurIPS 2021.

2020] to compute *punishment strategies* that are triggered when some agent deviates from the original joint policy. It also computes the maximum benefit each agent derives from deviating, which can be used to determine whether the joint policy augmented with punishment strategies forms an ϵ-Nash equilibrium; if so, it returns the joint policy.

Intuitively, the enumerative search tries to optimize social welfare, whereas the self-play RL algorithm checks whether the ϵ-Nash equilibrium constraint holds. Since this RL algorithm comes with PAC guarantees, our algorithm is guaranteed to return an ϵ-Nash equilibrium with high probability.

Motivating example. Consider the road intersection scenario in Figure 1. There are four cars; three are traveling east to west and one is traveling north to south. At any stage, each car can either move forward one step or stay in place. Suppose each car's specification is as follows: (a) *Black car:* Cross the intersection before the green and orange cars. (b) *Blue car:* Cross the intersection before the black car and stay a car length ahead of the green and orange cars. (c) *Green car:* Cross the intersection before the black car. (d) *Orange car:* Cross the intersection before the black car. We also require that the cars do not crash into one another.

Clearly, not all agents can achieve their goals. The next highest social welfare is for three agents to achieve their goals. In particular, one possibility is that all cars except the black car achieve their goals. However, the corresponding joint policy requires that the black car does not move, which is not a Nash equilibrium—there is always a gap between the blue car and the other two cars behind, so the black car can deviate by inserting itself into the gap to achieve its own goal. Our algorithm uses self-play RL to optimize the policy for the black car, and finds that the other agents cannot prevent the black car from improving its outcome in this way. Thus, it correctly rejects this joint policy. Eventually, our algorithm computes a Nash equilibrium in which the black and blue cars achieve their goals whereas the green and the orange cars do not achieve their goals.

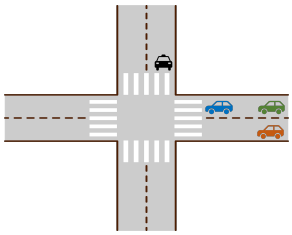

Figure 1: Intersection

2 Problem Formulation

Markov Game We consider an n-agent Markov game $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, H, s_0)$ with a finite set of states $\mathcal{S}$, actions $\mathcal{A} = A_1 \times \cdots \times A_n$ where A_i is a finite set of actions available to agent i, transition probabilities $P(s' \mid s, a)$ for $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$, finite horizon H, and initial state s_0 [Littman, 1994]. A *trajectory* $\zeta \in \mathcal{Z}$ is a finite sequence $\zeta = s_0 a_0 \dots a_{t-1} s_t$ where $s_k \in \mathcal{S}$, $a_k \in \mathcal{A}$; we use $|\zeta| = t$ to denote the length of the trajectory ζ and $a_k^i \in A_i$ to denote the action of agent i in the joint action a_k.

For any $i \in [n]$, let $\mathcal{D}(A_i)$ denote the set of distributions over A_i. A *policy* for agent i is a function $\pi_i : \mathcal{Z} \to \mathcal{D}(A_i)$ mapping trajectories to distributions over actions. A policy π_i is *deterministic* if for any $\zeta \in \mathcal{Z}$, there is an action $a_i \in A_i$ such that $\pi_i(\zeta)(a_i) = 1$; in this case, we also use $\pi_i(\zeta)$ to denote the action a_i. A *joint policy* $\pi : \mathcal{Z} \to \mathcal{D}(A)$ maps finite trajectories to distributions over joint actions. We use $(\pi_1, \dots, \pi_n)$ to denote the joint policy in which agent i chooses its action in accordance to π_i. We denote by $\mathcal{D}_\pi$ the distribution over H-length trajectories in $\mathcal{M}$ induced by π.

We consider the reinforcement learning setting in which we do not know the probabilities P but instead only have access to a simulator of $\mathcal{M}$. Typically, we can only sample trajectories of $\mathcal{M}$ starting at s_0. Some parts of our algorithm are based on an assumption which allows us to obtain sample trajectories starting at any state that has been observed before. For example, if taking action a_0 in s_0 leads to a state s_1, we can store s_1 and obtain future samples starting at s_1.

Assumption 2.1. *We can obtain samples from $P(\cdot \mid s, a)$ for any previously observed state s and action a.*

Specification Language Agent specifications are expressed in SPECTRL. Every SPECTRL specification ϕ can be represented by an *abstract graph* [Jothimurugan et al., 2021].

Algorithm 1 HIGHNASH

Inputs: Markov game (with unknown transition probabilities) $\mathcal{M}$ with n-agents, agent specifications $\phi_1, \ldots, \phi_n$, Nash factor ϵ, precision δ, failure probability p.

Outputs: ϵ-NE, if found.

1: Policies $\leftarrow$ LearnPolicies$(\phi_1, \ldots, \phi_n, \mathcal{M})$
2: PrioritizedPolicies $\leftarrow$ RankPolicies(Policies)
3: **for** joint policy $\pi \in$ PrioritizedPolicies **do**
4: // Can π be extended to an ϵ-NE?
5: Initialize ϵ-NE $\pi_N \leftarrow \pi$
6: **for** agent $j \in \{1, \ldots, n\}$ **do**
7: // Solve min-max game against agent j
8: $\mathcal{M}_j \leftarrow$ SimulatePunishmentGame$(\mathcal{M}, \pi, \phi_j, j)$
9: $\widetilde{\texttt{dev}}_j, \bar{\pi}_{-j} \leftarrow$ SolveMinMaxGame$(\mathcal{M}_j, \delta, p)$
10: **if** $\widetilde{\texttt{dev}}_j \leq J_j(\pi) + \epsilon$ **then**
11: $\pi_N \leftarrow \pi_N \bowtie \bar{\pi}_{-j}$ // Add punishment strat.
12: **else**
13: **goto** Line 3 // π cannot be extended to an ϵ-NE; try next policy
14: **return** π_N
15: **return** No ϵ-NE found

Nash Equilibrium and Social Welfare Given a Markov game $\mathcal{M}$ with unknown transitions and specifications $\phi_1, \ldots, \phi_n$ for the n agents respectively, the score of agent i from a joint policy π is given by $J_i(\pi) = \Pr_{\zeta \sim \mathcal{D}_\pi}[\zeta \models \phi_i]$.

Our goal is to compute a *high-value* ϵ-Nash equilibrium in $\mathcal{M}$ w.r.t these scores. Given a joint policy $\pi = (\pi_1, \ldots, \pi_n)$ and an alternate policy π_i' for agent i, let (π_{-i}, π_i') denote the joint policy $(\pi_1, \ldots, \pi_i', \ldots, \pi_n)$. Then, a joint policy π is an *ϵ-Nash equilibrium* if for all agents i and all alternate policies π_i', $J_i(\pi) \geq J_i((\pi_{-i}, \pi_i')) - \epsilon$. Then, our goal is to compute a joint policy π that maximizes the social welfare $\texttt{welfare}(\pi) = \frac{1}{n}\sum_{i=1}^{n} J_i(\pi)$ subject to the constraint that π is an ϵ-Nash equilibrium.

3 Algorithm Overview

Our algorithm for computing a high-welfare ϵ-Nash equilibrium proceeds in two phases. The first phase is a *prioritized enumeration* procedure that learns deterministic joint policies in the environment and ranks them in decreasing order of social welfare. The second phase is a *verification phase* that checks whether a given joint policy can be extended to an ϵ-Nash by adding punishment strategies. Our algorithm returns once an ϵ-Nash equilibrium is found. Algorithm 1 summarizes our framework; lines 1-2 and 5-12 are the prioritized enumeration and verification phases, respectively.

For enumeration it is impractical to enumerate all joint policies, since the total number of deterministic joint policies may be exponential in $|\mathcal{S}|^H$. Thus, the prioritized enumeration phase applies a specification-guided heuristic to reduce the number of joint policies considered (Section 4). The resulting search space is independent of $|\mathcal{S}|$ and H, depending only on the specifications ϕ_i. Since the environment is unknown, these joint policies are trained using compositional RL.

For verification (Section 5), we propose a probably approximately correct (PAC) procedure to determine whether a given joint policy can be extended to an ϵ-Nash equilibrium. Our approach is to reduce checking if a joint policy is an ϵ-Nash equilibrium to solving a two-agent zero-sum game for each agent. The key insight is that for a joint policy to be an ϵ-Nash equilibrium, unilateral deviations by any agent must be successfully punished by the coalition of all other agents. In such a *punishment game*, the deviating agent attempts to maximize its score while the coalition of other agents attempts to minimize its score, leading to a competitive min-max game between the agent and the coalition. If the deviating agent can improve its score by a margin $\geq \epsilon$, then the joint policy cannot be extended to an ϵ-Nash equilibrium. Alternatively, if no agent can increase their score by a margin $\geq \epsilon$, then the joint policy (augmented with punishment strategies) is an ϵ-Nash equilibrium. Each punishment game is solved using a self-play RL algorithm for learning policies in min-max games with unknown transitions [Bai and Jin, 2020], after converting specification-based scores to reward-based scores.

While the initial joint policy is deterministic, the punishment strategies can be probabilistic. Our algorithm is guaranteed to return an ϵ-Nash equilibrium with high probability, when it returns.

4 Prioritized Enumeration

We summarize our specification-guided compositional RL algorithm for learning a finite number of deterministic joint policies in an unknown environment under Assumption 2.1 These policies are then ranked in decreasing order of their (estimated) social welfare.

Our learning algorithm harnesses the structure of specifications, exposed by their abstract graphs, to curb the number of joint policies to learn. For every set of *active agents* $P \subseteq [n]$, we construct a product abstract graph, from the abstract graphs of all active agents' specifications, such that if a trajectory ζ in $\mathcal{M}$ corresponds to a path in the product that ends in a final state then ζ satisfies all active agents' specifications. That is, our procedure learns one joint policy for every path in the product graph that reaches a final state. By learning joint policies for every set of active agents, we are able to learn policies under which some agents may not satisfy their specifications, which is required for learning joint policies in non-cooperative settings. Note that the number of paths (and hence the number of policies considered) is independent of $|\mathcal{S}|$ and H, and depends only on the number of agents and their specifications.

One caveat is that the number of paths may be exponential in the number of states in the product graph. It would be impractical to naïvely learn a joint policy for every path. Instead, we design an efficient compositional RL algorithm that learns a joint policy for each edge in the product graph; these edge policies are then composed together to obtain joint policies for paths in the product graph.

4.1 Abstract Graph

Every SPECTRL specification ϕ can be represented by an *abstract graph* $\mathcal{G} = (U, E, u_0, F, \beta, \mathcal{Z}_{\text{safe}})$, where the vertices U and directed edges $E \subseteq U \times U$ form a DAG, $u_0 \in U$ is the initial state, and $F \subseteq U$ are the final states [Jothimurugan et al., 2021]. The concretization map $\beta : U \to 2^S$ maps abstract vertices to subgoal regions in the environment, The set of safe trajectories $\mathcal{Z}_{\text{safe}}$ is the union of $\mathcal{Z}^e_{\text{safe}}$ and $\mathcal{Z}^f_{\text{safe}}$ for all $e \in E$ and $f \in F$, where $\mathcal{Z}^e_{\text{safe}}$ and $\mathcal{Z}^f_{\text{safe}}$ are the set of safe trajectories in the environment taken to traverse an edge $e \in E$ and after reaching a final vertex $f \in F$, respectively.

Intuitively, the vertices of $\mathcal{G}$ are subgoal regions (i.e., subsets of the state space where an intermediate goal is satisfied), and edges of $\mathcal{G}$ are transitions between subgoal regions.

A finite trajectory $\zeta = s_0 a_0 s_1 a_1 \dots a_{t-1} s_t$ in $\mathcal{M}$ *satisfies* $\mathcal{G}$ (denoted $\zeta \models \mathcal{G}$) if there is a sequence of indices $0 = k_0 \leq k_1 < \cdots < k_\ell \leq t$ and a path $\rho = u_0 \to u_1 \to \cdots \to u_\ell$ in $\mathcal{G}$ such that (i) $u_\ell \in F$, (ii) for all $z \in \{0, \dots, \ell\}$, we have $s_{k_z} \in \beta(u_z)$, (iii) for all $z < \ell$, letting $e_z = u_z \to u_{z+1}$, we have $\zeta_{k_z:k_{z+1}} \in \mathcal{Z}^{e_z}_{\text{safe}}$, and (iv) $\zeta_{k_\ell:t} \in \mathcal{Z}^{u_\ell}_{\text{safe}}$. Then, for any specification ϕ and any trajectory ζ it has been shown that $\zeta \models \phi$ if and only if $\zeta \models \mathcal{G}_\phi$. Finally, the number of states in an abstract graph is linear in the size of the specification. Therefore, WLOG, the specification of each agent can be assumed to be an abstract graph.

4.2 Product Abstract Graph

Let $\phi_1, \dots, \phi_n$ be the specifications for the n-agents, respectively, let $\mathcal{G}_i = (U_i, E_i, u^i_0, F_i, \beta_i, \mathcal{Z}_{\text{safe},i})$ be the abstract graph of specification ϕ_i in the environment $\mathcal{M}$. We construct a product abstract graph for every set of active agents in $[n]$. The product graph for a set of active agents $P \subseteq [n]$ is used to learn joint policies that satisfy the specification of all agents in P with high probability. The specifications of non-active agents in $[n] \setminus P$ may not be satisfied by runs generated by these joint policies. For the set of agents $P = \{i_1, \dots, i_m\} \subseteq [n]$, the product graph $\mathcal{G}_P = (U, E, u_0, F, \beta, \mathcal{Z}_{\text{safe}})$ is an asynchronous product of $\mathcal{G}_i$ for all $i \in P$. Here, $U = \Pi_{i \in P} U_i$ is the set of product vertices, $u_0 = (u^{i_1}_0, \dots, u^{i_m}_0)$ is the initial vertex, and $F = \Pi_{i \in P} F_i \subseteq U$ is the set of final vertices. A vertex $(u_{i_1}, \dots u_{i_m}) \in F$ if $u_i \in F_i$ for all $i \in P$. The edge relation E accounts for the asynchronous product. An edge $e = (u_{i_1}, \dots, u_{i_m}) \to (v_{i_1}, \dots, v_{i_m}) \in E$ if at least for one agent $i \in P$ the edge $u_i \to v_i \in E_i$ and for the remaining agents, $u_i = v_i$. For an edge $e \in E$, we denote the set of agents $i \in P$ for which $u_i \to v_i \in E_i$ by $\texttt{Progress}(e)$. Finally, β and $\mathcal{Z}_{\text{safe}}$ are the collections of concretization maps and safe trajectories of agents in P. We denote the i-th component of a product

vertex $u \in U$ by u_i for agent $i \in P$. Similarly, the i-th component in an edge e is denoted by e_i for $i \in P$.

A trajectory $\zeta = s_0 \to \cdots \to s_t$ *achieves* edge $e = u \to v$ in $\mathcal{G}_P$ if all progressing agents $i \in \texttt{Progress}(e)$ reach their subgoal region $\beta_i(v_i)$ along the trajectory and the trajectory is safe for all agents in P. For a progressing agent $i \in \texttt{Progress}(e)$, the initial segment of the rollout until the agent reaches its target subgoal region $\beta_i(v_i)$ should be safe with respect to the edge e_i. After that, the rollout should be safe with respect to every future possibility for the agent. This is required to ensure continuity of the rollout into adjacent edges in the product graph $\mathcal{G}_P$. For the same reason, we require that the entire rollout is safe with respect to all future possibilities for non-progressing agents. Note that we are not concerned with non-active agents in $[n] \setminus P$.

Definition 4.1. Given $P \subseteq [n]$, a rollout $\zeta = s_0 \to \cdots \to s_t$ *achieves* a path $\rho = u_0 \to \cdots \to u_\ell$ in $\mathcal{G}_P$ (denoted $\zeta \models_P \rho$) if there exists indices $0 = k_0 \le k_1 \le \cdots \le k_\ell \le t$ such that (i) $u_\ell \in F$, (ii) $\zeta_{k_z:k_{z+1}}$ achieves $u_z \to u_{z+1}$ for all $0 \le z < \ell$, and (iii) $\zeta_{k_\ell:t} \in \mathcal{Z}_{\text{safe},i}^{u_{\ell,i}}$ for all $i \in P$.

Theorem 4.2. *Let $\rho = u_0 \to u_1 \to \cdots \to u_\ell$ be a path in the product abstract graph $\mathcal{G}_P$ for $P \subseteq [n]$. Suppose trajectory $\zeta \models_P \rho$. Then $\zeta \models \phi_i$ for all $i \in P$.*

That is, joint policies that maximize the probability of achieving paths in the product abstract graph $\mathcal{G}_P$ have high social welfare w.r.t. the active agents P.

4.3 Compositional RL Algorithm

Our compositional RL algorithm learns joint policies corresponding to paths in product abstract graphs. For every $P \subseteq [n]$, it learns a joint policy π_e for each edge in the product abstract graph $\mathcal{G}_P$, which is the (deterministic) policy that maximizes the probability of achieving e from a given initial state distribution. We use single-agent RL to learn edge policies cooperatively. The reward function is designed to capture the reachability objective of progressing agents and the safety objective of all agents. The edges are learned in topological order, allowing us to learn an induced state distribution for each product vertex u prior to learning any edge policies from u; this distribution is used as the initial state distribution when learning outgoing edge policies from u. In more detail, the distribution for the initial vertex of $\mathcal{G}_P$ is taken to be the initial state distribution of the environment; for every other product vertex, the distribution is the average over distributions induced by executing edge policies for all incoming edges.

Given edge policies Π along with a path $\rho = u_0 \to u_1 \to \cdots \to u_\ell = u \in F$ in $\mathcal{G}_P$, we define a *path policy* π_ρ to navigate from u_0 to u. In particular, π_ρ executes $\pi_{e[z]}$, where $e[z] = u_z \to u_{z+1}$ (starting from $z = 0$) until the resulting trajectory achieves $e[z]$, after which it increments $z \leftarrow z + 1$ (unless $z = \ell$). That is, π_ρ is designed to achieve the sequence of edges in ρ. Note that π_ρ is a finite-state deterministic joint policy in which vertices on the path correspond to the memory states that keep track of the index of the current policy. This way, we obtain finite-state joint policies by learning edge policies only. This process is repeated for all sets of active agents $P \subseteq [n]$. These finite-state joint policies are then ranked by estimating their social welfare on several simulations.

5 Nash Equilibria Verification

The prioritized enumearation phase produces a list of path policies which are ranked by the total sum of scores. Each path policy is deterministic and also finite state. Since the joint policies are trained cooperatively, they are typically not an ϵ-Nash equilibrium. Our verification algorithm attempts to modify a given joint policy by adding *punishment strategies* so that the resulting policy is an ϵ-Nash equilibrium.

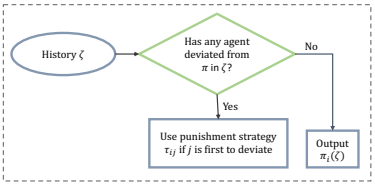

Figure 2: π_i augmented with punishment strategies.

Concretely, it takes as input a finite-state deterministic joint policy $\pi = (M, \alpha, \sigma, m_0)$ where M is a finite set of *memory states*, $\alpha : \mathcal{S} \times \mathcal{A} \times M \to M$ is the memory update function, $\sigma : \mathcal{S} \times M \to \mathcal{A}$ maps states to (joint) actions and m_0 is the initial policy state. The *extended memory update function* $\hat{\alpha} : \mathcal{Z} \to M$ is given by $\hat{\alpha}(\epsilon) = m_0$ and $\hat{\alpha}(\zeta s_t a_t) = \alpha(s_t, a_t, \hat{\alpha}(\zeta))$. Then, π is given by $\pi(\zeta s_t) = \sigma(s_t, \hat{\alpha}(\zeta))$. The policy π_i of agent i simply chooses the ith component of $\pi(\zeta)$ for any history ζ.

The verification algorithm learns one punishment strategy $\tau_{ij} : \mathcal{Z} \to \mathcal{D}(A_i)$ for each pair (i, j) of agents. As outlined in Figure 2, the modified policy for agent i uses π_i if every agent j has taken actions according to π_j in the past. In case some agent j' has taken an action that does not match the output of $\pi_{j'}$, then agent i uses the punishment strategy τ_{ij}, where j is the agent that deviated the earliest (ties broken arbitrarily). The goal of verification is to check if there is a set of punishment strategies $\{\tau_{ij} \mid i \neq j\}$ such that after modifying each agent's policy to use them, the resulting joint policy is an ϵ-Nash equilibrium.

5.1 Problem Formulation

We denote the set of all punishment strategies of agent i by $\tau_i = \{\tau_{ij} \mid j \neq i\}$. We define the composition of π_i and τ_i to be the policy $\tilde{\pi}_i = \pi_i \bowtie \tau_i$ such that for any trajectory $\zeta = s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{t-1}} s_t$, we have

- $\tilde{\pi}_i(\zeta) = \pi_i(\zeta)$ if for all $0 \leq k < t$, $a_k = \pi(\zeta_{0:k})$—i.e., no agent has deviated so far,
- $\tilde{\pi}_i(\zeta) = \tau_{ij}(\zeta)$ if there is a k such that (i) $a_k^j \neq \pi_j(\zeta_{0:k})$ and (ii) for all $\ell < k$, $a_\ell = \pi(\zeta_{0:\ell})$. If there are multiple such j's, an arbitrary but consistent choice is made (e.g., the smallest such j).

Then, the verification problem is to check if there exists a set of punishment strategies $\tau = \bigcup_i \tau_i$ such that the joint policy $\tilde{\pi} = \pi \bowtie \tau = (\tilde{\pi}_1, \ldots, \tilde{\pi}_n)$ is an ϵ-Nash equilibrium. In other words, the problem is to check if there exists a policy $\tilde{\pi}_i$ for each agent i such that (i) $\tilde{\pi}_i$ follows π_i as long as no other agent j deviates from π_j and (ii) the joint policy $\tilde{\pi} = (\tilde{\pi}_1, \ldots, \tilde{\pi}_n)$ is in ϵ-Nash equilibrium.

5.2 High-Level Procedure

Our approach is to compute the best set of punishment strategies τ^* and check if $\pi \bowtie \tau^*$ is an ϵ-Nash equilibrium. The best punishment strategy against agent j is the one that minimizes its incentive to deviate. To be precise, we define the best response of j with respect to a joint policy $\pi = (\pi_1, \ldots, \pi_n)$ to be $\mathrm{br}_j(\pi) \in \arg\max_{\pi'_j} J_j(\pi_{-j}, \pi'_j)$. Then, the best set of punishment strategies τ^* is one that minimizes the value of $\mathrm{br}_j(\pi \bowtie \tau)$ for all $j \in [n]$. To be precise, define $\tau[j] = \{\pi_{ij} \mid i \neq j\}$ to be the set of punishment strategies *against* agent j. Then, we have

$$\tau^*[j] \in \arg\min_{\tau[j]} J_j((\pi \bowtie \tau)_{-j}, \mathrm{br}_j(\pi \bowtie \tau)).$$

Note that both $\mathrm{br}_j(\pi \bowtie \tau)$ and $J_j((\pi \bowtie \tau)_{-j}, \pi'_j)$ are independent of $\tau \setminus \tau[j]$; therefore, we can separately compute $\tau^*[j]$ for each j and take $\tau^* = \bigcup_j \tau^*[j]$. The following theorem follows by definition of τ^*.

Theorem 5.1. *Given joint policy $\pi = (\pi_1, \ldots, \pi_n)$, if there is a set of punishment strategies τ such that $\pi \bowtie \tau$ is an ϵ-Nash equilibrium, then $\pi \bowtie \tau^*$ is an ϵ-Nash equilibrium, where τ^* is the set of best punishment strategies w.r.t. π. Furthermore, $\pi \bowtie \tau^*$ is an ϵ-Nash equilibrium iff for all j,*

$$J_j((\pi \bowtie \tau^*)_{-j}, \mathrm{br}_j(\pi \bowtie \tau^*)) - \epsilon \leq J_j(\pi \bowtie \tau^*) = J_j(\pi).$$

Thus, to solve the verification problem, it suffices to compute (or estimate), for all j, the optimal deviation scores

$$\texttt{dev}_j = \min_{\tau[j]} \max_{\pi'_j} J_j((\pi \bowtie \tau)_{-j}, \pi'_j). \tag{1}$$

5.3 Reduction to Min-Max Games

Next, we describe how to reduce the computation of punishment strategies to a standard self-play RL algorithm. Given a deterministic, finite-state joint policy $\pi = (M, \alpha, \sigma, m_0)$, we first translate the problem from the specification setting to the reward-based setting using *reward machines*.

Reward Machines. A *reward machine (RM)* [Icarte et al., 2018] is a tuple $\mathcal{R} = (Q, \delta_u, \delta_r, q_0)$, where Q is a finite set of states, $\delta_u : \mathcal{S} \times \mathcal{A} \times Q \to Q$ is the state transition function, $\delta_r : \mathcal{S} \times Q \to [-1, 1]$ and q_0 is the initial RM state. Given a trajectory $\zeta = s_0 a_0 \ldots a_{t-1} s_t$, the reward assigned

Algorithm 2 Verify Nash

Inputs: Finite state deterministic joint policy π, RMs $\mathcal{R}_j$ for all j, Nash factor ϵ, precision δ, failure probability p.

Outputs: `True` or `False`.

```
existsNE ← True
M̃ ← BFS-ESTIMATE(M, δ, p)
for agent j ∈ {1, ..., n} do
    M̃_j ← CONSTRUCTGAME(M̃, j, R_j, π)
    deṽ_j ← min_{π̄2} max_{π̄1} J̄^{M̃_j}(π̄1, π̄2)
    existsNE ← existsNE ∧ (deṽ_j ≤ J_j(π) + ε)
return existsNE
```

by $\mathcal{R}$ to ζ is $\mathcal{R}(\zeta) = \sum_{k=0}^{t-1} \delta_r(s_k, q_k)$, where $q_{k+1} = \delta_u(s_k, a_k, q_k)$ for all k. For any SPECTRL specification ϕ, we can define an RM such that the reward assigned to a trajectory ζ indicates whether ζ satisfies ϕ.

Theorem 5.2. *Given any* SPECTRL *specification ϕ, we can construct an RM $\mathcal{R}_\phi$ such that for any trajectory ζ of length $t+1$, $\mathcal{R}_\phi(\zeta) = \mathbb{1}(\zeta_{0:t} \models \phi)$.*

For an agent j, let $\mathcal{R}_j$ denote $\mathcal{R}_{\phi_j} = (Q_j, \delta_u^j, \delta_r^j, q_0^j)$. Letting $\tilde{\mathcal{D}}_\pi$ be the distribution over length $H+1$ trajectories induced by using π, we have $\mathbb{E}_{\zeta \sim \tilde{\mathcal{D}}_\pi}[\mathcal{R}_j(\zeta)] = J_j(\pi)$. The deviation values defined in Eq. 1 are now min-max values of expect reward, except that it is not in a standard min-max setting since the optimized policies of agents other that j appear in the augmented policy $\pi \bowtie \tau$. This issue can be handled by considering a product of $\mathcal{M}$ with the reward machine $\mathcal{R}_j$ and the finite state joint policy π. The following theorem follows naturally.

Theorem 5.3. *For any agent j, we can construct a simulator for an augmented two-player zero-sum Markov game $\mathcal{M}_j$ (with rewards) which has the following properties.*

- *The number of states in $\mathcal{M}_j$ is at most $2|S||M||Q_j|$.*
- *The actions of player 1 is A_j, and the actions of player 2 is $\mathcal{A}_{-j}$.*
- *The min-max value of the two player game corresponds to the deviation cost of j, i.e., $\texttt{dev}_j = \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \mathbb{E}\Big[\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) \mid \bar{\pi}_1, \bar{\pi}_2\Big]$, where the expectation is w.r.t. the distribution over length $H+1$ trajectories generated by using policy $(\bar{\pi}_1, \bar{\pi}_2)$ in $\mathcal{M}_j$.*

5.4 Solving Min-Max Games

The min-max game $\mathcal{M}_j$ can be solved using self-play RL algorithms. Many of these algorithms provide probabilistic approximation guarantees for computing the min-max value of the game. We use a model-based algorithm, similar to the one proposed by Bai and Jin [2020], that first estimates the model $\mathcal{M}_j$ and then solves the game in the estimated model.

One approach is to use existing algorithms for reward-free exploration to estimate the model [Jin et al., 2020], but this approach requires estimating each $\mathcal{M}_j$ separately. Under Assumption 2.1, we provide a simpler and more sample-efficient algorithm, called BFS-ESTIMATE, for estimating $\mathcal{M}$. BFS-ESTIMATE performs a search over the transition graph of $\mathcal{M}$ by exploring previously seen states in a breadth first manner. When exploring a state s, multiple samples are collected by taking all possible actions in s several times and the corresponding transition probabilities are estimated. After obtaining an estimate of $\mathcal{M}$, we construct estimates of $\mathcal{M}_j$ for all j with the following guarantee.

Theorem 5.4. *Under Assumption 2.1, for any $\delta > 0$ and any $p \in (0, 1]$,* BFS-ESTIMATE *computes estimates $\tilde{\mathcal{M}}_j$ of $\mathcal{M}_j$ for all j using $O\left(\frac{|\mathcal{S}|^3|M|^2|Q|^4|\mathcal{A}|H^4}{\delta^2} \log\left(\frac{|\mathcal{S}||\mathcal{A}|}{p}\right)\right)$ sample steps such that with probability at least $1-p$, for all j, $\left|\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \texttt{dev}_j\right| \leq \delta$, where $\bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$ is the expected reward over length $H+1$ trajectories generated by $(\bar{\pi}_1, \bar{\pi}_2)$ in $\tilde{\mathcal{M}}_j$.*

Spec.	Num. of agents	Algorithm	$\texttt{welfare}(\pi)$ (avg ± std)	$\epsilon_{\min}(\pi)$ (avg ± std)	Num. of runs terminated	Avg. num. of sample steps (in millions)
ϕ^1	3	HIGHNASH	**0.33 ± 0.00**	**0.00 ± 0.00**	10	1.78
		NVI	0.32 ± 0.00	**0.00 ± 0.00**	10	1.92
		MAQRM	0.18 ± 0.01	0.51 ± 0.01	10	2.00
ϕ^2	4	HIGHNASH	**0.55 ± 0.10**	**0.01 ± 0.02**	10	11.53
		NVI	0.04 ± 0.01	0.02 ± 0.01	10	12.60
		MAQRM	0.12 ± 0.01	0.20 ± 0.03	10	15.00
ϕ^3	4	HIGHNASH	**0.49 ± 0.01**	**0.00 ± 0.01**	10	11.26
		NVI	0.45 ± 0.01	**0.00 ± 0.01**	10	12.60
		MAQRM	0.11 ± 0.01	0.22 ± 0.02	10	15.00
ϕ^4	3	HIGHNASH	0.90 ± 0.15	**0.00 ± 0.00**	10	2.16
		NVI	**0.98 ± 0.00**	**0.00 ± 0.00**	4	2.18
		MAQRM	0.23 ± 0.01	0.39 ± 0.04	10	2.00
ϕ^5	5	HIGHNASH	**0.58 ± 0.02**	**0.00 ± 0.00**	10	62.17
		NVI	0.05 ± 0.01	0.01 ± 0.01	7	80.64
		MAQRM	Timeout	Timeout	0	Timeout

Table 1: Results in Intersection Environment. Total of 10 runs per benchmark. Timeout = 24 hrs.

The min-max value of $\tilde{\mathcal{M}}_j$ is computed using value iteration. Our full verification algorithm is summarized in Algorithm 2. It checks if $\tilde{\texttt{dev}}_j \leq J_j(\pi) + \epsilon$ for all j, and returns `True` if so and `False` otherwise. The following guarantees follow from Theorem 5.4.

Corollary 5.5 (Soundness)**.** *If a policy π cannot be converted to an $(\epsilon + \delta)$-Nash equilibrium by adding punishment strategies, then* VERIFYNASH *returns* `False` *with probability at least $1 - p$.*

Corollary 5.6 (Completeness)**.** *If a policy π can be converted to an $(\epsilon - \delta)$-Nash equilibrium by adding punishment strategies, then* VERIFYNASH *returns* `True` *with probability at least $1 - p$.*

6 Experiments

We evaluate our algorithm on the *Intersection environment* illustrated in Figure 1, which consists of k-cars (agents) at a 2-way intersection of which k_1 and k_2 cars are placed along the N-S and E-W axes, respectively. We aim to answer the following: (a). Can our approach be used to learn ϵ-Nash equilibria? (b). Can our approach learn policies with high social welfare?

We compare our NE computation method (HIGHNASH) to two approaches for learning in non-cooperative games. The first, MAQRM, is an adaption of the reward machine based learning algorithm proposed by Neary et al. [2021]. The second baseline, NVI, is a model-based approach that first estimates transition probabilities, and then computes a Nash equilibrium in the estimated game using value iteration for stochastic games [Kearns et al., 2000]. More details about the environment and the baselines can be found in the Appendix. The comparison uses two metrics: (i) the social welfare $\texttt{welfare}(\pi)$ of the learned joint policy π, and (ii) an estimate of the minimum value of ϵ for which π forms an ϵ-Nash equilibrium: $\epsilon_{\min}(\pi) = \max\{J_i(\pi_{-i}, \text{br}_i(\pi)) - J_i(\pi) \mid i \in [n]\}$. Here, $\epsilon_{\min}(\pi)$ is computed using single agent RL to compute the best response $\text{br}_i(\pi)$ for each i.

Observations (Summarized in Table 1) For each specification, we ran all algorithms 10 times with a timeout of 24 hours. Our approach learns policies that have low values of $\epsilon_{\min}$, indicating that it can be used to learn ϵ-Nash equilibria for small values of ϵ. NVI also has similar values of ϵ, which is expected since NVI provides guarantees similar to our approach w.r.t. Nash equilibria computation. On the other hand, MAQRM learns policies with large values of $\epsilon_{\min}$, implying that it fails to converge to a Nash equilibrium in most cases. Our experiments show that our approach consistently learns policies with high social welfare compared to the baselines. For instance, ϕ^3 corresponds to the specifications in the motivating example for which our approach learns a joint policy that causes both blue and black cars to achieve their goals. Although NVI succeeds in learning policies with high social welfare for some specifications (ϕ^1, ϕ^3, ϕ^4) it fails to do so for others (ϕ^2, ϕ^5).

References

Yu Bai and Chi Jin. Provable self-play algorithms for competitive reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

Rodrigo Toro Icarte, Toryn Klassen, Richard Valenzano, and Sheila McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, pages 2107–2116. PMLR, 2018.

Chi Jin, Akshay Krishnamurthy, Max Simchowitz, and Tiancheng Yu. Reward-free exploration for reinforcement learning. In *International Conference on Machine Learning*, pages 4870–4879. PMLR, 2020.

Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. Compositional reinforcement learning from logical specifications. *arXiv preprint arXiv:2106.13906*, 2021.

Michael Kearns, Yishay Mansour, and Satinder Singh. Fast planning in stochastic games. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 309–316, 2000.

Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

Richard D. McKelvey, Andrew M. McLennan, and Theodore L. Turocy. Gambit: Software tools for game theory, 2014.

Cyrus Neary, Zhe Xu, Bo Wu, and Ufuk Topcu. Reward machines for cooperative multi-agent reinforcement learning, 2021.

Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *arXiv preprint arXiv:2010.03950*, 2020.

Algorithm 3 Nash Value Iteration
Inputs: n-agent Markov game $\mathcal{M}$ with rewards, horizon H.
Outputs: Nash equilibrium joint policy $\pi = (\pi_1, \ldots, \pi_n)$.

Initialize joint policy $\pi = (\pi_1, \ldots, \pi_n)$
Initialize value function $V : \mathcal{S} \times [H+1] \to \mathbb{R}^n$ to be the zero map
for $t \in \{H, H-1, \ldots, 1\}$ **do**
  for $s \in \mathcal{S}$ **do**
    Initialize step game $G_s^t : \mathcal{A} \to \mathbb{R}^n$
    for $a = (a_1, \ldots, a_n) \in \mathcal{A}$ **do**
      $G_s^t(a_1, \ldots, a_n) = R(s, a) + \mathbb{E}_{s' \sim P(\cdot|s,a)}[V(s', t+1)]$
    $(d_1, d_2, \ldots, d_n) \leftarrow$ BEST-NASH-GENERAL-SUM$(G_s^t) \in \mathcal{D}(A_1) \times \cdots \times \mathcal{D}(A_n)$
    $V(s,t) \leftarrow \mathbb{E}_{a_1 \sim d_1, a_2 \sim d_2, \ldots, a_n \sim d_n}[G_s^t(a_1, \ldots, a_n)]$
    $\pi(s,t) \leftarrow (d_1, d_2, \ldots, d_n)$
return π

A Baseline

A.1 NVI

This baseline first computes an estimate $\tilde{\mathcal{M}}$ of $\mathcal{M}$ using BFS-ESTIMATE and then computes a product of $\tilde{\mathcal{M}}$ with the reward machines corresponding to the agent specifications in order to define rewards at every step. It then solves the resulting general sum game $\tilde{\mathcal{M}}'$ using value iteration. The value iteration procedure is outlined in Algorithm 3 which uses BEST-NASH-GENERAL-SUM to solve n-player general-sum strategic games (one-step games) at each step. When there are multiple Nash equilibria for a step game, BEST-NASH-GENERAL-SUM chooses one with the highest social welfare (for that step). In our experiments, we use the library `gambit` [McKelvey et al., 2014] for solving the step games.

A.2 MAQRM

The baseline (Algorithm 4) performs a multi-agent variant of QRM [Icarte et al., 2018, Toro Icarte et al., 2020]. We derive reward machines from agent specifications.

We learn one Q-function for every agent. The Q-function for the i-th agent, denoted $Q_i : S \times \Pi_{i \in [n]} U_i \to \mathcal{A}_i$, can be used to derive the best action for the i-th agent from the current state of the environment and reward machines of all agents. In every step, Q_i is used to sample an action a_i for the i-th agent. The joint action $(a_i)_{i \in [n]}$ is used to take a step in the environment and all reward machines. Finally, each Q_i is individually updated according to the reward gained by the i-th agent.

For notational convenience, we let $q \xrightarrow{\alpha} q'$ denote $q \leftarrow (1-\alpha) \cdot q + \alpha \cdot q'$.

B Intersection Environment

The state consists of the location of all cars where the location of a single car is a non-negative integer. 1 corresponds to the intersection, 0 corresponds to the location one step towards the south or west of the intersection (depending on the car) and locations greater than 1 are to the east or north of the intersection. Each agent has two actions. STAY stays at the current position. MOVE decreases the position value by 1 with probability 0.95 and stays with probability 0.05.

Specifications

ϕ^1 Two N-S cars both starting at 3 and one E-W car starting at 2. N-S cars' goal is to reach 0 before the E-W car without collision. E-W car's goal is to reach 0 before both N-S agents without collision.

ϕ^2 Same as motivating example except that blue car is not required to stay a car length away from green and orange cars.

ϕ^3 Same as motivating example.

Algorithm 4 Multi-agent QRM

Inputs: n-agent Markov game $\mathcal{M} = (\mathcal{S}, \mathcal{A} = \Pi_{i\in[n]}\mathcal{A}_i, P, H, s_0)$, agent specifications $\phi_1, \dots \phi_n$, learning rate $\alpha \in (0, 1]$, discount factor $\gamma \in (0, 1]$, $\varepsilon \in (0, 1]$

Outputs: Joint policy $\pi = (\pi_1, \dots, \pi_n)$.

for $i \in [n]$ **do** $(U_i, \delta_u^i, \delta_r^i, u_0^i) \leftarrow \mathsf{RewardMachine}(\phi_i)$
// Initialize environment state, reward machines state, and Q-functions
$s \leftarrow s_0$ and **for** $i \in [n]$ **do** $u_i \leftarrow u_0^i$
for $i \in [n]$ **do** Initialize $Q_i(s, (u_1, \dots, u_n), a_i)$ for all states $s \in S, u_i \in U_i$, and actions $a_i \in \mathcal{A}_i$
for $l \leftarrow 0$ to num_steps **do**
  // Sample actions from policy derived from Q-functions
  for $i \in [n]$ **do** choose action $a_i \in \mathcal{A}_i$ at $(s, (u_1, \dots, u_n))$ using policy derived from Q_i (e.g., ε-greedy)
  // Take a step in environment and the reward machines
  Take action $a = (a_1, \dots a_n)$ in $\mathcal{M}$ and observe the next state s'
  for $i \in [n]$ **do** compute the reward $r_i \leftarrow \delta_r^i(s, u_i)$ and next RM state $u_i' \leftarrow \delta_u^i(s, a, u_i))$
  // Update all Q-functions
  if s' is terminal **then**
    for $i \in [n]$ **do** $Q_i(s, (u_1, \dots, u_n), a) \xleftarrow{\alpha} r_i$
  else
    for $i \in [n]$ **do** $Q_i(s, (u_1, \dots, u_n), a_i) \xleftarrow{\alpha} r_i + \gamma \cdot \max_{a_i' \in \mathcal{A}_i} Q_i(s', (u_1', \dots, u_n'), a_i')$
  if s' is terminal **then**
    // Reset environment state and reward machines state
    $s \leftarrow s_0$ and **for** $i \in [n]$ **do** $u_i \leftarrow u_0^i$
  else
    $s \leftarrow s'$ and **for** $i \in [n]$ **do** $u_i \leftarrow u_i'$
for $i \in [n]$ **do** $\pi_i \leftarrow$ Best action policy derived from Q_i
return $(\pi_1, \dots, \pi_n)$

ϕ^4 Two N-S agents (0 and 1) both starting at 3 and one E-W agent (2) starting at 3. Agent 0's task is to reach 0 before other two agents. Agent 1's task is to reach 0. Agent 2's task is to reach 0 before agent 1. All agents must avoid collision.

ϕ^5 Two N-S cars starting at 2 and 3 and three E-W cars all starting at 2, 3 and 4, respectively. N-S cars' goal is to reach 0 before the E-W cars without collision. E-W cars' goal is to reach 0 before both N-S cars without collision.