Specification-Guided Learning of Nash Equilibria with High Social Welfare

Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, 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 objective 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 [35]. Our focus is on non-cooperative systems, where the agents are trying to achieve their own goals [27]; for such systems, the goal is typically to learn a policy for each agent such that the joint strategy forms a Nash equilibrium.

A key challenge facing existing approaches is how tasks are specified. 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 [22] and cooperative [35] 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 social 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

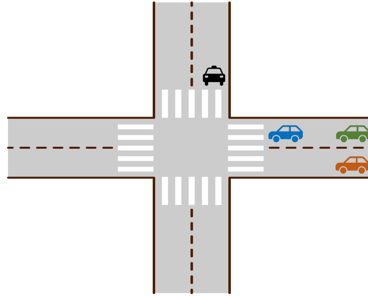

Fig. 1: Intersection Example

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 [4] 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:

- *Black car:* Cross the intersection before the green and orange cars.
- *Blue car:* Cross the intersection before the black car and stay a car length ahead of the green and orange cars.
- *Green car:* Cross the intersection before the black car.
- *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.

1.1 Related Work

Multi-agent RL. There has been work on learning Nash equilibria in the multi-agent RL setting [1, 20, 21, 32, 36, 37]; however, these approaches focus on learning an arbitrary equilibrium and do not optimize social welfare. There has also been work on studying weaker notions of equilibria in this context [13, 43], as well as work on learning Nash equilibria in two agent zero-sum games [4, 31, 40].

RL from high-level specifications. There has been recent work on using specifications based on temporal logic for specifying RL tasks in the single agent setting [2, 6, 8, 12, 14, 16, 17, 23, 25, 30, 33, 41, 42].A comprehensive survey may be found at [3]. There has also been recent work on using temporal logic specifications for multi-agent RL [15, 35], but these approaches focus on cooperative scenarios in which there is a common objective that all agents are trying to achieve.

Equilibrium in Markov games. There has been work on computing Nash equilibrium in Markov games [27, 38], including work on computing ϵ-Nash equilibria from logical specifications [9, 10], as well as recent work focusing on computing welfare-optimizing Nash equilibria from temporal specifications [28, 29]; however, all these works focus on the planning setting where the transition probabilities are known. Checking for existence of Nash equilibrium, even in deterministic games, has been shown to be NP-complete for reachability objectives [5].

Social welfare. There has been work on computing welfare maximizing Nash equilibria for bimatrix games, which are two-player one-step Markov games with known transitions [11, 18]; in contrast, we study this problem in the context of general Markov games.

2 Preliminaries

2.1 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 [31]. A *trajectory* $\zeta \in \mathcal{Z} = (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S}$ is a finite sequence $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t$ 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 a_k.

For any $i \in [n]$, let $\mathcal{D}(A_i)$ denote the set of distributions over A_i—i.e., $\mathcal{D}(A_i) = \{\Delta : A_i \to [0,1] \mid \sum_{a_i \in A_i} \Delta(a_i) = 1\}$. 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 every $\zeta \in \mathcal{Z}$, 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 $(\pi_1, \ldots, \pi_n)$ 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 assume we can obtain future samples starting at s_1.

Assumption 1 *We can obtain samples from $P(\cdot \mid s, a)$ for any previously observed state s and any action a.*

2.2 Specification Language

We consider the specification language Spectrl to express agent specifications. We choose Spectrl since there is existing work on leveraging the structure of Spectrl specifications for single-agent RL [26]. However, we believe our algorithm can be adapted to other specification languages as well.

Formally, a Spectrl specification is defined over a set of *atomic predicates* $\mathcal{P}_0$, where every $p \in \mathcal{P}_0$ is associated with a function $[\![p]\!] : \mathcal{S} \to \mathbb{B} = \{\texttt{true}, \texttt{false}\}$; we say a state s *satisfies* p (denoted $s \models p$) if and only if $[\![p]\!](s) = \texttt{true}$. The set of *predicates* $\mathcal{P}$ consists of conjunctions and disjunctions of atomic predicates. The syntax of a predicate $b \in \mathcal{P}$ is given by the grammar $b ::= p \mid (b_1 \wedge b_2) \mid (b_1 \vee b_2)$, where $p \in \mathcal{P}_0$. Similar to atomic predicates, each predicate $b \in \mathcal{P}$ corresponds to a function $[\![b]\!] : \mathcal{S} \to \mathbb{B}$ defined naturally over Boolean logic. Finally, the syntax of Spectrl is given by [1]

$$\phi ::= \texttt{achieve}\ b \mid \phi_1\ \texttt{ensuring}\ b \mid \phi_1; \phi_2 \mid \phi_1\ \texttt{or}\ \phi_2,$$

where $b \in \mathcal{P}$. Each specification ϕ corresponds to a function $[\![\phi]\!] : \mathcal{Z} \to \mathbb{B}$, and we say $\zeta \in \mathcal{Z}$ satisfies ϕ (denoted $\zeta \models \phi$) if and only if $[\![\phi]\!](\zeta) = \texttt{true}$. Letting ζ

[1] Here, `achieve` and `ensuring` correspond to the "eventually" and "always" operators in temporal logic.

be a finite trajectory of length t, this function is defined by

$$
\begin{aligned}
&\zeta \models \texttt{achieve}\ b && \text{if } \exists\, i \le t,\ s_i \models b \\
&\zeta \models \phi\ \texttt{ensuring}\ b && \text{if } \zeta \models \phi \text{ and } \forall\, i \le t,\ s_i \models b \\
&\zeta \models \phi_1; \phi_2 && \text{if } \exists\, i < t,\ \zeta_{0:i} \models \phi_1 \text{ and } \zeta_{i+1:t} \models \phi_2 \\
&\zeta \models \phi_1\ \texttt{or}\ \phi_2 && \text{if } \zeta \models \phi_1 \text{ or } \zeta \models \phi_2.
\end{aligned}
$$

Intuitively, the first clause means that the trajectory should eventually reach a state that satisfies the predicate b. The second clause says that the trajectory should satisfy specification ϕ while always staying in states that satisfy b. The third clause says that the trajectory should sequentially satisfy ϕ_1 followed by ϕ_2. The fourth clause means that the trajectory should satisfy either ϕ_1 or ϕ_2.

2.3 Abstract Graphs

Spectrl specifications can be represented by *abstract graphs* as defined below:

Definition 1. An *abstract graph* $\mathcal{G} = (U, E, u_0, F, \beta, \mathcal{Z}_{\text{safe}})$ is a directed acyclic graph (DAG) with vertices U, (directed) edges $E \subseteq U \times U$, initial vertex $u_0 \in U$, final vertices $F \subseteq U$, subgoal region map $\beta : U \to 2^S$ such that for each $u \in U$, $\beta(u)$ is a subgoal region,[2] and *safe trajectories* $\mathcal{Z}_{\text{safe}} = \bigcup_{e \in E} \mathcal{Z}_{\text{safe}}^e \cup \bigcup_{f \in F} \mathcal{Z}_{\text{safe}}^f$, where $\mathcal{Z}_{\text{safe}}^e \subseteq \mathcal{Z}$ denotes the safe trajectories for edge $e \in E$ and $\mathcal{Z}_{\text{safe}}^f \subseteq \mathcal{Z}$ denotes the safe trajectories for final vertex $f \in F$.

Intuitively, (U, E) is a standard DAG, and u_0 and F define a graph reachability problem for (U, E). Furthermore, β and $\mathcal{Z}_{\text{safe}}$ connect (U, E) back to the original MDP $\mathcal{M}$; in particular, for an edge $e = u \to u'$, $\mathcal{Z}_{\text{safe}}^e$ is the set of safe trajectories in $\mathcal{M}$ that can be used to transition from $\beta(u)$ to $\beta(u')$.

Definition 2. A trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t$ in $\mathcal{M}$ satisfies the abstract graph $\mathcal{G}$ (denoted $\zeta \models \mathcal{G}$) if there is a sequence of indices $0 = k_0 \le k_1 < \cdots < k_\ell \le t$ and a path $\rho = u_0 \to u_1 \to \cdots \to u_\ell$ in $\mathcal{G}$ such that

- $u_\ell \in F$,
- for all $z \in \{0, \ldots, \ell\}$, we have $s_{k_z} \in \beta(u_z)$,
- for all $z < \ell$, letting $e_z = u_z \to u_{z+1}$, we have $\zeta_{k_z:k_{z+1}} \in \mathcal{Z}_{\text{safe}}^{e_z}$, and
- $\zeta_{k_\ell:t} \in \mathcal{Z}_{\text{safe}}^{u_\ell}$.

The first two conditions state that the trajectory should visit a sequence of subgoal regions corresponding to a path from the initial vertex to some final vertex, and the last two conditions state that the trajectory should be composed of subtrajectories that are safe according to $\mathcal{Z}_{\text{safe}}$.

Prior work shows that for every Spectrl specification ϕ, we can construct an abstract graph $\mathcal{G}_\phi$ such that for every trajectory $\zeta \in \mathcal{Z}$, $\zeta \models \phi$ if and only if $\zeta \models \mathcal{G}_\phi$ [26]. Finally, the number of states in the abstract graph is linear in the size of the specification.

[2] We do not require that the subgoal regions partition the state space or that they be non-overlapping.

2.4 Nash Equilibrium and Social Welfare

Given a Markov game $\mathcal{M}$ with unknown transitions and SPECTRL 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', $J_i(\pi) \geq J_i((\pi_{-i}, \pi_i')) - \epsilon$. Our goal is to compute a joint policy π that maximizes the social welfare given by

$$\texttt{welfare}(\pi) = \frac{1}{n}\sum_{i=1}^{n} J_i(\pi)$$

subject to the constraint that π is an ϵ-Nash equilibrium.

3 Overview

Our framework for computing a high-welfare ϵ-Nash equilibrium consists of 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 equilibrium by adding punishment strategies. A policy is returned if it passes the verification check in the second phase. Algorithm 1 summarizes our framework.

For the enumeration phase, it is impractical to enumerate all joint policies even for small environments, since the total number of deterministic joint policies is $\Omega(|\mathcal{A}|^{|\mathcal{S}|^{H-1}})$, which is $\Omega(2^{n|\mathcal{S}|^{H-1}})$ if each agent has atleast two actions. Thus, in the prioritized enumeration phase, we apply a specification-guided heuristic to reduce the number of joint policies considered. The resulting search space is independent of $|\mathcal{S}|$ and H, depending only on the specifications $\{\phi_i\}_{i\in[n]}$. Since the transition probabilities are unknown, these joint policies are trained using an efficient compositional RL approach.

Since the joint policies are trained cooperatively, they are typically not ϵ-Nash equilibria. Hence, in the verification phase, we use a probably approximately correct (PAC) procedure (Algorithm 2) to determine whether a given joint policy can be modified by adding *punishment strategies* to form an ϵ-Nash equilibrium. Our approach is to reduce this problem to solving two-agent zero-sum games. The key insight is that for a given 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

Algorithm 1 HighNashSearch
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: PrioritizedPolicies $\leftarrow$ PrioritizedEnumeration$(\mathcal{M}, \phi_1, \ldots, \phi_n)$
2: **for** joint policy $\pi \in$ PrioritizedPolicies **do**
3: // Can π be extended to an ϵ-NE?
4: `isNash`, $\tau \leftarrow$ VerifyNash$(\mathcal{M}, \pi, \phi_1, \cdots, \phi_n, \epsilon, \delta, p)$
5: **if** `isNash` **then return** $\pi \bowtie \tau$ // Add punishment strategies
6: **return** No ϵ-NE found

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 its score by a margin $\geq \epsilon$, then the joint policy (augmented with punishment strategies) is an ϵ-Nash equilibrium. Thus, checking if a joint policy can be converted to an ϵ-Nash equilibrium reduces to solving a two-agent zero-sum game for each agent. Each punishment game is solved using a self-play RL algorithm for learning policies in min-max games with unknown transitions [4], after converting specification-based scores to reward-based scores. While the initial joint policy is deterministic, the punishment strategies can be probabilistic.

Overall, we provide the guarantee that with high probability, if our algorithm returns a joint policy, it will be an ϵ-Nash equilibrium.

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 1; details are in Appendix A. 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* $B \subseteq [n]$, we construct a product abstract graph, from the abstract graphs of all active agents' specifications. The property of this product is that if a trajectory ζ in $\mathcal{M}$ corresponds to a path in the product that ends in a final state then ζ satisfies the specification of all active agents. 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. This enables 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

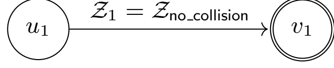

Fig. 2: Abstract Graph of black car.

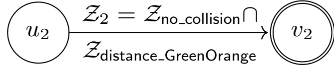

Fig. 3: Abstract Graph of blue car.

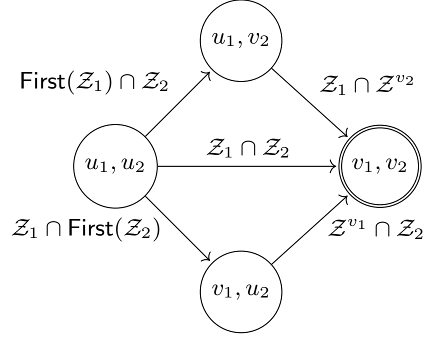

Fig. 4: Product Abstract Graph of black and blue cars. $\mathcal{Z}^{v_1}$ and $\mathcal{Z}^{v_2}$ refer to safe trajectories after the black and blue cars have reached their final states, respectively.

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 Product Abstract Graph

Let $\phi_1, \dots, \phi_n$ be the specifications for the n-agents, respectively, and let $\mathcal{G}_i = (U_i, E_i, u_0^i, F_i, \beta_i, \overline{\mathcal{Z}}_{\text{safe},i})$ 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 $B \subseteq [n]$ is used to learn joint policies which satisfy the specification of all agents in B with high probability.

Definition 3. *Given a set of agents $B = \{i_1, \dots, i_m\} \subseteq [n]$, the product graph $\mathcal{G}_B = (\overline{U}, \overline{E}, \overline{u}_0, \overline{F}, \overline{\beta}, \overline{\mathcal{Z}}_{safe})$ is the asynchronous product of $\mathcal{G}_i$ for all $i \in B$, with*

- *$\overline{U} = \prod_{i \in B} U_i$ is the set of product vertices,*
- *An edge $e = (u_{i_1}, \dots, u_{i_m}) \to (v_{i_1}, \dots, v_{i_m}) \in \overline{E}$ if at least for one agent $i \in B$ the edge $u_i \to v_i \in E_i$ and for the remaining agents, $u_i = v_i$,*
- *$\overline{u}_0 = (u_0^{i_1}, \dots, u_0^{i_m})$ is the initial vertex,*
- *$\overline{F} = \Pi_{i \in B} F_i$ is the set of final vertices,*
- *$\overline{\beta} = (\beta_{i_1}, \dots, \beta_{i_m})$ is the collection of concretization maps, and*
- *$\overline{\mathcal{Z}}_{safe} = (\overline{\mathcal{Z}}_{safe,i_1}, \dots, \overline{\mathcal{Z}}_{safe,i_m})$ is the collection of safe trajectories.*

We denote the i-th component of a product vertex $\overline{u} \in \overline{U}$ by u_i for agent $i \in B$. Similarly, the i-th component in an edge $e = \overline{u} \to \overline{v}$ is denoted by $e_i = u_i \to v_i$ for $i \in B$; note that e_i can be a self loop which is not an edge in

$\mathcal{G}_i$. For an edge $e \in \overline{E}$, we denote the set of agents $i \in B$ for which $e_i \in E_i$, and not a self loop, by $\mathsf{progress}(e)$.

Abstract graphs of the black car and the blue car from the motivating example are shown in Figures 2 and 3 respectively. The vertex v_1 denotes the subgoal region $\beta_{\text{black}}(v_1)$ consisting of states in which the black car has crossed the intersection but the orange and green cars have not. The subgoal region $\beta_{\text{blue}}(v_2)$ is the set of states in which the blue car has crossed the intersection but the black car has not. $\mathcal{Z}_1$ denotes trajectories in which the black car does not collide and $\mathcal{Z}_2$ denotes trajectories in which the blue car does not collide and stays a car length ahead of the orange and green cars. The product abstract graph for the set of active agents $B = \{\text{black}, \text{blue}\}$ is shown in Fig 4. The safe trajectories on the edges reflect the notion of *achieving* a product edge which we discuss below.

A trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t$ *achieves* an edge $e = \overline{u} \rightarrow \overline{v}$ in $\mathcal{G}_B$ if all progressing agents $i \in \mathsf{progress}(e)$ reach their target subgoal region $\beta_i(v_i)$ along the trajectory and the trajectory is safe for all agents in B. For a progressing agent $i \in \mathsf{progress}(e)$, the initial segment of the rollout until the agent reaches its subgoal region 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 $\mathcal{G}_B$. 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 $[n] \setminus B$. In order to formally define this notion, we need to setup some notation.

For a predicate $b \in \mathcal{P}$, let the set of safe trajectories w.r.t. b be given by $\mathcal{Z}_b = \{\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t \in \mathcal{Z} \mid \forall\, 0 \leq k \leq t, s_k \models b\}$. It is known that safe trajectories along an edge in an abstract graph constructed from a SPECTRL specification is either of the form $\mathcal{Z}_b$ or $\mathcal{Z}_{b_1} \circ \mathcal{Z}_{b_2}$, where $b, b_1, b_2 \in \mathcal{P}$ and $\circ$ denotes concatenation [26]. In addition, for every final vertex f, $\mathcal{Z}_{\text{safe}}^{f}$ is of the form $\mathcal{Z}_b$ for some $b \in \mathcal{P}$. We define First as follows:

$$\mathsf{First}(\mathcal{Z}') = \begin{cases} \mathcal{Z}_b, & \text{if } \mathcal{Z}' = \mathcal{Z}_b \\ \mathcal{Z}_{b_1}, & \text{if } \mathcal{Z}' = \mathcal{Z}_{b_1} \circ \mathcal{Z}_{b_2} \end{cases}$$

We are now ready to define the notion of satisfiability of a product edge.

Definition 4. A rollout $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_k$ *achieves an edge* $e = \overline{u} \rightarrow \overline{v}$ in $\mathcal{G}_B$ (denoted $\zeta \models_B e$) if

1. for all progressing agents $i \in \mathsf{progress}(e)$, there exists an index $k_i \leq k$ such that $s_{k_i} \in \beta_i(v_i)$ and $\zeta_{0:k_i} \in \mathcal{Z}_{\text{safe},i}^{e_i}$. If $v_i \in F_i$ then $\zeta_{k_i:k} \in \mathcal{Z}_{\text{safe},i}^{v_i}$. Otherwise, $\zeta_{k_i:k} \in \mathsf{First}(\mathcal{Z}_{\text{safe},i}^{v_i \rightarrow w_i})$ for all $w_i \in \mathsf{outgoing}(v_i)$. Furthermore, we require $k_i > 0$ if $u_i \neq u_0^i$.
2. for all non-progressing agents $i \in B \setminus \mathsf{progress}(e)$, if $u_i \notin F_i$, $\zeta \in \mathsf{First}(\mathcal{Z}_{\text{safe},i}^{u_i \rightarrow w_i})$ for all $w_i \in \mathsf{outgoing}(u_i)$. Otherwise (if $u_i \in F_i$), $\zeta \in \mathcal{Z}_{\text{safe},i}^{u_i}$

We can now define what it means for a trajectory to achieve a path in the product graph $\mathcal{G}_B$.

Definition 5. Given $B \subseteq [n]$, a rollout $\zeta = s_0 \to \cdots \to s_t$ *achieves* a path $\rho = \overline{u}_0 \to \cdots \to \overline{u}_\ell$ in $\mathcal{G}_B$ (denoted $\zeta \models_B \rho$) if there exists indices $0 = k_0 \leq k_1 \leq \cdots \leq k_\ell \leq t$ such that (i) $\overline{u}_\ell \in \overline{F}$, (ii) $\zeta_{k_z:k_{z+1}}$ achieves $\overline{u}_z \to \overline{u}_{z+1}$ for all $0 \leq z < \ell$, and (iii) $\zeta_{k_\ell:t} \in \mathcal{Z}^{u_{\ell,i}}_{\text{safe},i}$ for all $i \in B$.

Theorem 2. *Let $\rho = \overline{u_0} \to \overline{u_1} \to \cdots \to \overline{u_\ell}$ be a path in the product abstract graph $\mathcal{G}_B$ for $B \subseteq [n]$. Suppose trajectory $\zeta \models_B \rho$. Then $\zeta \models \phi_i$ for all $i \in B$.*

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

4.2 Compositional RL Algorithm

Our compositional RL algorithm learns joint policies corresponding to paths in product abstract graphs. For every $B \subseteq [n]$, it learns a joint policy π_e for each edge in the product abstract graph $\mathcal{G}_B$, which is the (deterministic) policy that maximizes the probability of achieving e from a given initial state distribution. We assume all agents are acting cooperatively; thus, we treat the agents as one and use single-agent RL to learn each edge policy. We will check whether any deviation to this co-operative behaviour by any agent can be punished by the coalition of other agents in the verification phase. The reward function is designed to capture the reachability objective of progressing agents and the safety objective of all active agents.

The edges are learned in topological order, allowing us to learn an induced state distribution for each product vertex $\overline{u}$ prior to learning any edge policies from $\overline{u}$; this distribution is used as the initial state distribution when learning outgoing edge policies from $\overline{u}$. In more detail, the distribution for the initial vertex of $\mathcal{G}_B$ 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. This is possible because the product graph is a DAG.

Given edge policies Π along with a path $\rho = \overline{u}_0 \to \overline{u}_1 \to \cdots \to \overline{u}_\ell = \overline{u} \in \overline{F}$ in $\mathcal{G}_B$, we define a *path policy* π_ρ to navigate from $\overline{u}_0$ to $\overline{u}$. In particular, π_ρ executes $\pi_{e[z]}$, where $e[z] = \overline{u}_z \to \overline{u}_{z+1}$ (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 $B \subseteq [n]$. 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

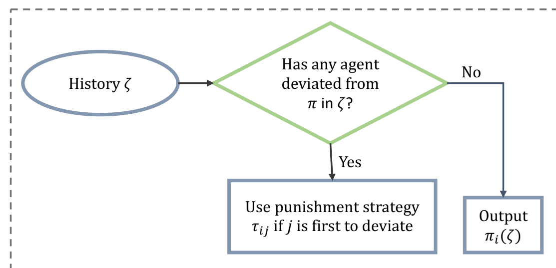

Fig. 5: π_i augmented with punishment strategies.

finite state. Since the joint policies are trained cooperatively, they are typically not ϵ-Nash equilibria. Thus, our verification algorithm not only tries to prove that a given joint policy is a ϵ-Nash equilibrium, but also tries to modify it so it satisfies this property. In particular, our verification algorithm attempts to modify a given joint policy by adding *punishment strategies* so that the resulting policy is an ϵ-Nash equilibrium.

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 5, 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).

Given a finite-state deterministic joint policy π, the verification problem is to check if there exists a set of punishment strategies $\tau = \bigcup_i \tau_i$ such that the joint policy $\tilde{\pi} = \pi \bowtie \tau = (\pi_1 \bowtie \tau_1, \ldots, \pi_n \bowtie \tau_n)$ 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 an ϵ-Nash equilibrium.

5.2 High-Level Procedure

Our approach is to compute the best set of punishment strategies τ^* w.r.t. π 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^* w.r.t. π is one that minimizes the value of $\mathrm{br}_j(\pi \bowtie \tau)$ for all $j \in [n]$. To be precise, define $\tau[j] = \{\tau_{ij} \mid i \neq j\}$ to be the set of punishment strategies *against* agent j. Then, we want to compute τ^* such that for all j,

$$\tau^* \in \arg\min_{\tau} J_j((\pi \bowtie \tau)_{-j}, \mathrm{br}_j(\pi \bowtie \tau)). \quad (1)$$

We observe that for any two sets of punishment strategies τ, τ' with $\tau[j] = \tau'[j]$ and any policy π'_j, we have $J_j((\pi \bowtie \tau)_{-j}, \pi'_j) = J_j((\pi \bowtie \tau')_{-j}, \pi'_j)$. This is because, for any τ, punishment strategies in $\tau \setminus \tau[j]$ do not affect the behaviour of the joint policy $((\pi \bowtie \tau)_{-j}, \pi'_j)$, since no agent other than agent j will deviate from π. Hence, $\mathrm{br}_j(\pi \bowtie \tau)$ as well as $J_j((\pi \bowtie \tau)_{-j}, \mathrm{br}_j(\pi \bowtie \tau))$ are independent of $\tau \setminus \tau[j]$; therefore, we can separately compute $\tau^*[j]$ (satisfying Equation 1) for each j and take $\tau^* = \bigcup_j \tau^*[j]$. The following theorem follows from the definition of τ^*; see Appendix B.1 for a proof.

Theorem 3. *Given a finite-state deterministic 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

$$\mathtt{dev}_j^{\pi} = \min_{\tau[j]} \max_{\pi'_j} J_j((\pi \bowtie \tau)_{-j}, \pi'_j). \quad (2)$$

5.3 Reduction to Min-Max Games

Next, we describe how to reduce the computation of optimal deviation scores to a standard self-play RL setting. We first translate the problem from the specification setting to a reward-based setting using *reward machines*.

Reward Machines. A *reward machine (RM)* [22] 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, $\delta_r : \mathcal{S} \times Q \to [-1, 1]$ is the reward function and q_0 is the initial RM state. Given a trajectory $\zeta = s_0 \xrightarrow{a_0} \dots \xrightarrow{a_{t-1}} s_t$, the reward assigned 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 construct an RM such that the reward assigned to a trajectory ζ indicates whether ζ satisfies ϕ; a proof can be found in Appendix B.2.

Theorem 4. *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. 2 are now min-max values of expect reward, except that it is not in a standard min-max setting since the policy of every non-deviating agent $i \neq j$ is constrained to be of the form $\pi_i \bowtie \tau_i$. 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; details are in Appendix B.3.

Theorem 5. *Given a finite-state deterministic joint policy $\pi = (M, \alpha, \sigma, m_0)$, for any agent j, we can construct a simulator for an augmented two-player zero-sum Markov game $\mathcal{M}_j^\pi$ (with rewards) which has the following properties.*

- *The number of states in $\mathcal{M}_j^\pi$ is at most $2|S||M||Q_j|$.*
- *The actions of player 1 is A_j, and the actions of player 2 is $\mathcal{A}_{-j} = \prod_{i \neq j} A_i$.*
- *The min-max value of the two player game corresponds to the deviation cost of j, i.e.,*

$$\mathtt{dev}_j^\pi = \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2),$$

  where $\bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2) = \mathbb{E}\big[\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) \mid \bar{\pi}_1, \bar{\pi}_2\big]$ is the expected sum of rewards w.r.t. the distribution over $(H+1)$-length trajectories generated by using the joint policy $(\bar{\pi}_1, \bar{\pi}_2)$ in $\mathcal{M}_j^\pi$.
- *Given any policy $\bar{\pi}_2$ for player 2 in $\mathcal{M}_j^\pi$, we can construct a set of punishment strategies $\tau[j] = $ PUNSTRAT$(\bar{\pi}_2)$ against agent j in $\mathcal{M}$ such that*

$$\max_{\bar{\pi}_1} \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2) = \max_{\pi_j'} J_j((\pi \bowtie \tau[j])_{-j}, \pi_j').$$

Given an estimate $\tilde{\mathcal{M}}$ of $\mathcal{M}$, we can also construct an estimate $\tilde{\mathcal{M}}_j^\pi$ of $\mathcal{M}_j^\pi$.

We omit the superscript π from $\mathcal{M}_j^\pi$ when there is no ambiguity. We denote by CONSTRUCTGAME$(\tilde{\mathcal{M}}, j, \mathcal{R}_j, \pi)$ the product construction procedure that constructs and returns $\tilde{\mathcal{M}}_j^\pi$.

Algorithm 2 VerifyNash
Inputs: Finite-state deterministic joint policy π, specifications ϕ_j for all j, Nash factor ϵ, precision δ, failure probability p.
Outputs: `True` or `False` along with a set of punishment strategies τ.

1: $\texttt{existsNE} \leftarrow \texttt{True}$
2: $\tau \leftarrow \emptyset$
3: $\tilde{\mathcal{M}} \leftarrow$ BFS-Estimate$(\mathcal{M}, \delta, p)$ // Only run if $\mathcal{M}$ has not been estimated before.
4: **for** agent $j \in \{1, \ldots, n\}$ **do**
5: $\quad \mathcal{R}_j \leftarrow$ ConstructRM(ϕ_j)
6: $\quad \tilde{\mathcal{M}}_j \leftarrow$ ConstructGame$(\tilde{\mathcal{M}}, j, \mathcal{R}_j, \pi)$
7: $\quad \tilde{\texttt{dev}}_j \leftarrow \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$
8: $\quad \bar{\pi}_2^* \leftarrow \arg\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$
9: $\quad \texttt{existsNE} \leftarrow \texttt{existsNE} \wedge (\tilde{\texttt{dev}}_j \leq J_j(\pi) + \epsilon - \delta)$
10: $\quad \tau \leftarrow \tau \cup$ PunStrat$(\bar{\pi}_2^*)$
11: **return** `existsNE`, τ

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 in [4], 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 [24], but this approach requires estimating each $\mathcal{M}_j$ separately. Under Assumption 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 can directly construct an estimate of $\mathcal{M}_j^\pi$ for any π and j when required. Letting $|Q| = \max_j |Q_j|$ and $|M|$ denote the size of the largest finite-state policy output by our enumeration algorithm, we get the following guarantee; see Appendix B.4 for a proof.

Theorem 6. *For any $\delta > 0$ and $p \in (0, 1]$,* BFS-Estimate$(\mathcal{M}, \delta, p)$ *computes an estimate $\tilde{\mathcal{M}}$ of $\mathcal{M}$ 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 any finite-state deterministic joint policy π and any agent j,*

$$\left| \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j^\pi}(\bar{\pi}_1, \bar{\pi}_2) - \mathit{dev}_j^\pi \right| \leq \delta,$$

where $\bar{J}^{\tilde{\mathcal{M}}_j^\pi}(\bar{\pi}_1, \bar{\pi}_2)$ is the expected reward over length $H+1$ trajectories generated by $(\bar{\pi}_1, \bar{\pi}_2)$ in $\tilde{\mathcal{M}}_j^\pi$. Furthermore, letting $\bar{\pi}_2^* \in \arg\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$ and

$\tau[j] = \textsc{PunStrat}(\bar{\pi}_2^*)$, *we have*

$$\Big| \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j^\pi}(\bar{\pi}_1, \bar{\pi}_2^*) - \max_{\pi_j'} J_j((\pi \bowtie \tau[j])_{-j}, \pi_j') \Big| \leq \delta. \tag{3}$$

The min-max value of $\tilde{\mathcal{M}}_j^\pi$ as well as $\bar{\pi}_2^*$ can be computed using value iteration. Our full verification algorithm is summarized in Algorithm 2. It checks if $\tilde{\texttt{dev}}_j \leq J_j(\pi) + \epsilon - \delta$ for all j, and returns `True` if so and `False` otherwise. It also simultaneously computes the punishment strategies τ using the optimal policies for player 2 in the punishment games. Note that BFS-Estimate is called only once (i.e., the first time VerifyNash is called) and the obtained estimate $\tilde{\mathcal{M}}$ is stored and used for verification of every candidate policy π. The following soundness guarantee follows from Theorem 6; proof in Appendix B.5.

Corollary 1 (Soundness). *For any $p \in (0, 1]$, $\varepsilon > 0$ and $\delta \in (0, \varepsilon)$, with probability at least $1 - p$, if* HighNashSearch *returns a joint policy $\tilde{\pi}$ then $\tilde{\pi}$ is an ϵ-Nash equilibrium.*

6 Complexity

In this section, we analyze the time and sample complexity of our algorithm in terms of the number of agents n, size of the specification $|\phi| = \max_{i\in[n]} |\phi_i|$, number of states in the environment $|\mathcal{S}|$, number of joint actions $|\mathcal{A}|$, time horizon H, precision δ and the failure probability p.

Sample Complexity. It is known [26] that the number of edges in the abstract graph $\mathcal{G}_i$ corresponding to specification ϕ_i is $O(|\phi_i|^2)$. Hence for any set of active agents B, the number of edges in the product abstract graph $\mathcal{G}_B$ is $O(|\phi|^{2|B|})$. Hence total number of edge policies learned by our compositional RL algorithm is $\sum_{B\subseteq[n]} O((|\phi|^2)^{|B|}) = O((|\phi|^2 + 1)^n)$. We learn each edge using a fixed number of sample steps C, which is a hyperparameter.

The number of samples used in the verification phase is the same as the number used by BFS-Estimate. The maximum size of a candidate policy output by the enumeration algorithm $|M|$ is at most the length of the longest path in a product abstract graph. Since the maximum path length in a single abstract graph $\mathcal{G}_i$ is bounded by $|\phi_i|$ and at least one agent must progress along every edge in a product graph, the maximum length of a path in any product graph is at most $n|\phi|$. Also, the number of states in the reward machine $\mathcal{R}_j$ corresponding to $|\phi_j|$ is $O(2^{|\phi_j|})$. Hence, from Theorem 6 we get that the total number of sample steps used by our algorithm is $O\big((|\phi|^2+1)^n C + \frac{2^{4|\phi|}|\mathcal{S}|^3 n^2 |\phi|^2 |\mathcal{A}| H^4}{\delta} \log\big(\frac{|\mathcal{S}||\mathcal{A}|}{p}\big)\big)$.

Time Complexity. As with sample complexity, the time required to learn all edge policies is $O((|\phi|^2 + 1)^n(C + |\mathcal{A}|))$ where the term $|\mathcal{A}|$ is added to account for the time taken to select an action from $\mathcal{A}$ during exploration (we use Q-learning with ε-greedy exploration for learning edge policies). Similarly,

time taken for constructing the reward machines and running BFS-Estimate is $O(\frac{2^{4|\phi|}|\mathcal{S}|^3n^2|\phi|^2|\mathcal{A}|H^4}{\delta}\log\big(\frac{|\mathcal{S}||\mathcal{A}|}{p}\big))$.

The total number of path policies considered for a given set of active agents B is bounded by the number of paths in the product abstract graph $\mathcal{G}_B$ that terminate in a final product state. First, let us consider paths in which exactly one agent progresses in each edge. The number of such paths is bounded by $(|B||\phi|)^{|B||\phi|}$ since the length of such paths is bounded by $|B||\phi|$ and there are at most $|B||\phi|$ choices at each step—i.e., progressing agent j and next vertex of the abstract graph $\mathcal{G}_{\phi_j}$. Now, any path in $\mathcal{G}_B$ can be constructed by merging adjacent edges along such a path (in which at most one agent progresses at any step). The number of ways to merge edges along such a path is bounded by the number of groupings of edges along the path into at most $|B||\phi|$ groups which is bounded by $(|B||\phi|)^{|B||\phi|}$. Therefore, the total number of paths in $\mathcal{G}_B$ is at most $2^{2|B||\phi|\log(n|\phi|)}$. Finally, the total number of path policies considered is at most $\sum_{B\subseteq[n]}2^{2|B||\phi|\log(n|\phi|)}\leq((n|\phi|)^{2|\phi|}+1)^n=O(2^{2n|\phi|\log(2n|\phi|)})$.

Now, for each path policy π, the verification algorithm solves $\tilde{\mathcal{M}}_j^{\pi}$ using value iteration which takes $O(|\tilde{\mathcal{S}}||\mathcal{A}|Hf(|\mathcal{A}|))=O(2^{|\phi|}n|\phi||\mathcal{S}||\mathcal{A}|Hf(|\mathcal{A}|))$ time, where $f(|\mathcal{A}|)$ is the time required to solve a linear program of size $|\mathcal{A}|$. Also accounting for the time taken to sort the path policies, we arrive at a time complexity bound of $2^{O(n|\phi|\log(n|\phi|))}\text{poly}(|\mathcal{S}|,|\mathcal{A}|,H,\frac{1}{p},\frac{1}{\delta})$.

It is worth noting that the procedure halts as soon as our verification procedure successfully verifies a policy; this leads to early termination for cases where there is a high value ϵ-Nash equilibrium (among the policies considered). Furthermore, our verification algorithm runs in polynomial time and therefore one could potentially improve the overall time complexity by reducing the search space in the prioritized enumeration phase—e.g., by using domain specific insights.

7 Experiments

We evaluate our algorithm on finite state environments and a variety of specifications, aiming to answer the following:

- Can our approach be used to learn ϵ-Nash equilibria?
- Can our approach learn policies with high social welfare?

We compare our approach to two baselines described below, using 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 (specifically, Q-learning) to compute $\text{br}_i(\pi)$ for each agent i.

Environments and specifications. We show results 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. 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. We consider specifications similar to the ones in the motivating example. Details are in Appendix D, and results on two additional environments are in Appendix E.

Baselines. We compare our NE computation method (HighNashSearch) to two approaches for learning in non-cooperative games. The first, maqrm, is an adaption of the reward machine based learning algorithm proposed in [35]. maqrm was originally proposed for cooperative multi-agent RL where there is a single specification for all the agents. It proceeds by first decomposing the specification into individual ones for all the agents and then runs a Q-learning-style algorithm (qrm) in parallel for all the agents. We use the second part of their algorithm directly since we are given a separate specification for each agent. 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 [27]. To promote high social welfare, we select the highest value Nash solution for the matrix game at each stage of value iteration. Note that this greedy strategy may not maximize social welfare. Both maqrm and nvi learn from rewards as opposed to specification; thus, we supply rewards in the form of reward machines constructed from the specifications. nvi is guaranteed to return an ϵ-Nash equilibrium with high probability, but maqrm is not guaranteed to do so. Details are in Appendix C.

Results. Our results are summarized in Table 1. For each specification, we ran all algorithms 10 times with a timeout of 24 hours. Along with the average social welfare and $\epsilon_{\min}$, we also report the average number of sample steps taken in the environment as well as the number of runs that terminated before timeout. For a fair comparison, all approaches were given a similar number of samples from the environment.

Nash equilibrium. 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.

Social Welfare. 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

Spec.	Num. of agents	Algorithm	$\texttt{welfare}(\pi)$ (avg ± std)	$\epsilon_{\min}(\pi)$ (avg ± std)	Num. of terminated runs	Avg. num. of sample steps (in millions)
ϕ^1	3	HighNashSearch	**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	HighNashSearch	**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	HighNashSearch	**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	HighNashSearch	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	HighNashSearch	**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 for all specifications in Intersection Environment. Total of 10 runs per benchmark. Timeout = 24 hrs.

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).

8 Conclusions

We have proposed a framework for maximizing social welfare under the constraint that the joint policy should form an ϵ-Nash equilibrium. Our approach involves learning and enumerating a small set of finite-state deterministic policies in decreasing order of social welfare and then using a self-play RL algorithm to check if they can be extended with punishment strategies to form an ϵ-Nash equilibrium. Our experiments demonstrate that our approach is effective in learning Nash equilibria with high social welfare.

One limitation of our algorithm is that the policies considered by our framework are chosen based on the specifications, which may lead to scenarios where we miss high welfare solutions. For example, ϕ^2 corresponds to specifications in the motivating example except that the blue car is not required to stay a car length ahead of the other two cars. In this scenario, it is possible for three cars to achieve their goals in an equilibrium solution if the blue car helps the cars behind by staying in the middle of the intersection until they catch up. Such a joint policy is not among the set of policies considered; therefore, our approach learns a solution in which only two cars achieve their goals. We believe that such limitations can be overcome in future work by modifying the various components within our enumerate-and-verify framework.

Bibliography

[1] Akchurina, N.: Multi-agent reinforcement learning algorithm with variable optimistic-pessimistic criterion. In: ECAI. vol. 178, pp. 433–437 (2008)

[2] Aksaray, D., Jones, A., Kong, Z., Schwager, M., Belta, C.: Q-learning for robust satisfaction of signal temporal logic specifications. In: Conference on Decision and Control (CDC). pp. 6565–6570. IEEE (2016)

[3] Alur, R., Bansal, S., Bastani, O., Jothimurugan, K.: A framework for transforming specifications in reinforcement learning. arXiv preprint arXiv:2111.00272 (2021)

[4] Bai, Y., Jin, C.: Provable self-play algorithms for competitive reinforcement learning. In: Proceedings of the 37th International Conference on Machine Learning (2020)

[5] Bouyer, P., Brenguier, R., Markey, N.: Nash equilibria for reachability objectives in multi-player timed games. In: International Conference on Concurrency Theory. pp. 192–206. Springer (2010)

[6] Brafman, R., De Giacomo, G., Patrizi, F.: Ltlf/ldlf non-markovian rewards. In: Proceedings of the AAAI Conference on Artificial Intelligence (2018)

[7] Brafman, R.I., Tennenholtz, M.: R-max - a general polynomial time algorithm for near-optimal reinforcement learning. JMLR **3** (2003)

[8] Camacho, A., Toro Icarte, R., Klassen, T.Q., Valenzano, R., McIlraith, S.A.: Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In: International Joint Conference on Artificial Intelligence. pp. 6065–6073 (7 2019)

[9] Chatterjee, K.: Two-player nonzero-sum ω-regular games. In: International Conference on Concurrency Theory. pp. 413–427. Springer (2005)

[10] Chatterjee, K., Majumdar, R., Jurdziński, M.: On nash equilibria in stochastic games. In: International workshop on computer science logic. pp. 26–40. Springer (2004)

[11] Czumaj, A., Fasoulakis, M., Jurdzinski, M.: Approximate nash equilibria with near optimal social welfare. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)

[12] De Giacomo, G., Iocchi, L., Favorito, M., Patrizi, F.: Foundations for restraining bolts: Reinforcement learning with ltlf/ldlf restraining specifications. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 29, pp. 128–136 (2019)

[13] Greenwald, A., Hall, K., Serrano, R.: Correlated q-learning. In: ICML. vol. 3, pp. 242–249 (2003)

[14] Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-regular objectives in model-free reinforcement learning. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 395–412 (2019)

[15] Hammond, L., Abate, A., Gutierrez, J., Wooldridge, M.: Multi-agent reinforcement learning with temporal logic specifications. In: International

Conference on Autonomous Agents and MultiAgent Systems. p. 583–592 (2021)
[16] Hasanbeig, M., Kantaros, Y., Abate, A., Kroening, D., Pappas, G.J., Lee, I.: Reinforcement learning for temporal logic control synthesis with probabilistic satisfaction guarantees. In: Conference on Decision and Control (CDC). pp. 5338–5343 (2019)
[17] Hasanbeig, M., Abate, A., Kroening, D.: Logically-constrained reinforcement learning. arXiv preprint arXiv:1801.08099 (2018)
[18] Hazan, E., Krauthgamer, R.: How hard is it to approximate the best nash equilibrium? In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms. p. 720–727. SODA '09, Society for Industrial and Applied Mathematics (2009)
[19] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Acm Sigact News **32**(1), 60–65 (2001)
[20] Hu, J., Wellman, M.P.: Nash q-learning for general-sum stochastic games. Journal of machine learning research **4**(Nov), 1039–1069 (2003)
[21] Hu, J., Wellman, M.P., et al.: Multiagent reinforcement learning: theoretical framework and an algorithm. In: ICML. vol. 98, pp. 242–250. Citeseer (1998)
[22] Icarte, R.T., Klassen, T., Valenzano, R., McIlraith, S.: Using reward machines for high-level task specification and decomposition in reinforcement learning. In: International Conference on Machine Learning. pp. 2107–2116. PMLR (2018)
[23] Jiang, Y., Bharadwaj, S., Wu, B., Shah, R., Topcu, U., Stone, P.: Temporal-logic-based reward shaping for continuing learning tasks (2020)
[24] Jin, C., Krishnamurthy, A., Simchowitz, M., Yu, T.: Reward-free exploration for reinforcement learning. In: International Conference on Machine Learning. pp. 4870–4879. PMLR (2020)
[25] Jothimurugan, K., Alur, R., Bastani, O.: A composable specification language for reinforcement learning tasks. In: Advances in Neural Information Processing Systems. vol. 32, pp. 13041–13051 (2019)
[26] Jothimurugan, K., Bansal, S., Bastani, O., Alur, R.: Compositional reinforcement learning from logical specifications. Advances in Neural Information Processing Systems **34** (2021)
[27] Kearns, M., Mansour, Y., Singh, S.: Fast planning in stochastic games. In: Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence. pp. 309–316 (2000)
[28] Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Equilibria-based probabilistic model checking for concurrent stochastic games. In: International Symposium on Formal Methods. pp. 298–315. Springer (2019)
[29] Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Prism-games 3.0: Stochastic game verification with concurrency, equilibria and time. In: International Conference on Computer Aided Verification. pp. 475–487. Springer (2020)
[30] Li, X., Vasile, C.I., Belta, C.: Reinforcement learning with temporal logic rewards. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3834–3839. IEEE (2017)

[31] Littman, M.L.: Markov games as a framework for multi-agent reinforcement learning. In: Machine learning proceedings 1994, pp. 157–163. Elsevier (1994)

[32] Littman, M.L.: Friend-or-foe q-learning in general-sum games. In: ICML. vol. 1, pp. 322–328 (2001)

[33] Littman, M.L., Topcu, U., Fu, J., Isbell, C., Wen, M., MacGlashan, J.: Environment-independent task specifications via gltl (2017)

[34] McKelvey, R.D., McLennan, A.M., Turocy, T.L.: Gambit: Software tools for game theory (2014), http://www.gambit-project.org

[35] Neary, C., Xu, Z., Wu, B., Topcu, U.: Reward machines for cooperative multi-agent reinforcement learning (2021)

[36] Perolat, J., Strub, F., Piot, B., Pietquin, O.: Learning Nash Equilibrium for General-Sum Markov Games from Batch Data. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (2017)

[37] Prasad, H., LA, P., Bhatnagar, S.: Two-timescale algorithms for learning nash equilibria in general-sum stochastic games. In: Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems. pp. 1371–1379 (2015)

[38] Shapley, L.S.: Stochastic games. Proceedings of the national academy of sciences **39**(10), 1095–1100 (1953)

[39] Toro Icarte, R., Klassen, T.Q., Valenzano, R., McIlraith, S.A.: Reward machines: Exploiting reward function structure in reinforcement learning. arXiv preprint arXiv:2010.03950 (2020)

[40] Wei, C.Y., Hong, Y.T., Lu, C.J.: Online reinforcement learning in stochastic games. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. pp. 4994–5004 (2017)

[41] Xu, Z., Topcu, U.: Transfer of temporal logic formulas in reinforcement learning. In: International Joint Conference on Artificial Intelligence. pp. 4010–4018 (7 2019)

[42] Yuan, L.Z., Hasanbeig, M., Abate, A., Kroening, D.: Modular deep reinforcement learning with temporal logic specifications. arXiv preprint arXiv:1909.11591 (2019)

[43] Zinkevich, M., Greenwald, A., Littman, M.: Cyclic equilibria in markov games. Advances in Neural Information Processing Systems **18**, 1641 (2006)

A Prioritized Enumeration

A.1 Proof of Theorem 2

Proof. We show that if $\zeta \models_B \rho$ then every agent $i \in B$ achieves a path in the abstract graph $\mathcal{G}_i$ of its specification ϕ_i. Let $0 = k_0 \le k_1 \le \cdots \le k_\ell \le t$ be the indices that satisfy the criteria in Definition 5.

For agent $i \in B$, let indices $0 \le z_0 < \cdots < z_x < \ell$ be such that the agent makes progress along the edge $e_{z_y} = \overline{u}_{z_y} \to \overline{u}_{z_y+1}$ for $0 \le y \le x$. Note that agents can make progress only till they reach a final state in their abstract graph. So, $(u_{z_y})_i \notin F_i$ and $(u_{z_x+1})_i \in F_i$ for all $0 \le y \le x$. Further note, $(u_{z_0})_i = u_0^i$ and $(u_{z_{y+1}})_i = (u_{z_y+1})_i$ for $0 \le y < x$ since the agent i has not made any progress in between.

Let $\zeta_y = \zeta_{k_{z_y}:k_{z_y+1}}$ be the sub-trajectory that achieves the edge e_{z_y} for $0 \le y \le x$, i.e. $\zeta_y \models_B e_{z_y}$ for $0 \le y \le x$.

Since, agent i has made progress along e_{z_y}, using Definition 4, we can obtain indices $0 = p_0 \le \cdots < p_{x+1} \le t$ on the trajectory such that

- $p_y \le k_{z_y} \le p_{y+1}$ for $0 \le y < x$
- $s_{p_{y+1}} \in \beta_i((u_{z_y+1})_i)$ for all $0 \le y \le x$
- $\zeta_{k_{z_y}:p_{y+1}} \in \mathcal{Z}_{\text{safe},i}^{(e_{z_y})_i}$ for all $0 \le y \le x$
- $\zeta_{p_{y+1}:k_{z_y+1}} \in \mathsf{First}(\mathcal{Z}_{\text{safe},i}^{(u_{z_y+1})_i \to w_i})$ for $w_i \in \mathsf{outgoing}((u_{z_y+1})_i)$ for $0 \le y < x$ and $\zeta_{p_{x+1}:k_{z_x+1}} \models \mathcal{Z}_{\text{safe},i}^{(u_{z_x+1})_i}$
  (This is because $(u_{z_y})_i \notin F_i$ and $(u_{z_x+1})_i \in F_i$ for all $0 \le y \le x$, as observed earlier)

Additionally, by using, $(u_{z_y+1})_i = (u_{z_{y+1}})_i$ for $0 \le y < x$ since the agent i has not made any progress in between, the above is simplified to: there exists indices $0 = p_0 \le \cdots < p_{x+1} \le t$ on the trajectory such that

- $p_y \le k_{z_y} \le p_{y+1}$ for $0 \le y \le x$
- $s_{p_{y+1}} \in \beta_i((u_{z_{y+1}})_i)$ for all $0 \le y \le x$
- $\zeta_{k_{z_y}:p_{y+1}} \in \mathcal{Z}_{\text{safe},i}^{(e_{z_y})_i}$ for all $0 \le y \le x$
- $\zeta_{p_{y+1}:k_{z_y+1}} \in \mathsf{First}(\mathcal{Z}_{\text{safe},i}^{(e_{z_{y+1}})_i})$ for $0 \le y < x$ and $\zeta_{p_{x+1}:k_{z_x+1}} \models \mathcal{Z}_{\text{safe},i}^{(u_{z_x+1})_i}$
  (This is because $(e_{z_{y+1}})_i$ is an outgoing edge from $(u_{z_{y+1}})_i$)

Our goal is to show that $\zeta_{p_y:p_{y+1}} \in \mathcal{Z}_{\text{safe},i}^{(e_{z_y})_i}$ for $0 \le y \le x$. We already know that $\zeta_{k_{z_y}:p_{y+1}} \in \mathcal{Z}_{\text{safe},i}^{(e_{z_y})_i}$ for all $0 \le y \le x$. Observing the fact that for any edge e of $\mathcal{G}_i$ the set $\mathsf{First}(\mathcal{Z}_{\text{safe},i}^{e})$ is closed under concatenation, it is sufficient to show $\zeta_{p_y:k_{z_y}} \in \mathsf{First}(\mathcal{Z}_{\text{safe},i}^{(e_{z_y})_i})$ for $0 \le y \le x$. We do this in two cases.

- **Case** $y = 0$: In this case $\zeta_{p_0:k_{z_0}} = \zeta_{k_0:k_{z_0}}$. So, it has made no progress along the path till $s_{k_{z_0}}$. So, the non-progressing agent will ensure its trajectory is safe with respect to outgoing edges from vertex $(u_{z_0})_i = (u_0)_i$. Now, $(e_{z_0})_i$ is an outgoing edge from $(u_{z_0})_i$. So, we get that $\zeta_{0:k_{z_0}} \in \mathsf{First}(\mathcal{Z}_{\text{safe},i}^{(e_{z_0})_i})$.

– **Case** $0 < y \leq x$: Here, we know that $\zeta_{p_y:k_{z_{y-1}+1}} \in \mathsf{First}(\mathcal{Z}^{(e_{z_y})_i}_{\text{safe},i})$. So, it is sufficient to show that $\zeta_{k_{z_{y-1}+1}:k_{z_y}} \in \mathsf{First}(\mathcal{Z}^{(e_{z_y})_i}_{\text{safe},i})$. This is true because agent i is has not progressed on any of the edges between $\overline{u}_{z_{y-1}+1}$ and $\overline{u}_{z_y}$. The ith component of all vertices between these states is $(u_{z_y})_i$, since agent i will not change its vertex. Now $(e_{z_y})_i$ is an outgoing edge from $(u_{z_y})_i$. So, in particular, we get that $\zeta_{k_{z_{y-1}+1}:k_{z_y}} \in \mathsf{First}(\mathcal{Z}^{(e_{z_y})_i}_{\text{safe},i})$.

So, consider the path $(u_0)_i \to (u_{z_0})_i \to (u_{z_1})_i \to (u_{z_x})_i \to (u_{z_x+1})_i \in F_i$ in graph $\mathcal{G}_i$. There exists indices $0 = p_0 \leq p_1 < ... < p_{x+1} \leq t$ such that

– $s_{p_0} = s_0 \in \beta_i((u_0)_i)$ (from Definition 5), $s_{p_y} \in \beta_i((u_{z_y})_i)$ for $0 < y \leq x$ and $s_{p_{x+1}} \in \beta_i((u_{z_x+1})_i)$ (from inferences made above)
– $\zeta_{p_y:p_{y+1}} \in \mathcal{Z}^{(e_{z_y})_i}_{\text{safe},i}$ for $0 \leq y \leq x$
– $\zeta_{p_{x+1}:t} \in \mathcal{Z}^{(u_{z_x+1})_i}_{\text{safe},i}$ because agent i cannot progress any further after visiting state $(u_{z_x+1})_i \in F_i$.

Thus, for agent i, $\zeta \models \mathcal{G}_i$, which implies $\zeta \models \phi_i$. □

A.2 Algorithm

The complete algorithm for prioritized enumeration is outlined in Algorithm 3. The details of non-standard functions are given below.

$\mathsf{AverageDistribution}$*:* The set $\Gamma(\overline{u})$ consists of as many state distributions as there are incoming edges into the state when $\overline{u} \neq \overline{u_0}$. When $\overline{u} = \overline{u}_0$, $\Gamma(\overline{u})$ only contains the distribution corresponding to the initial state distribution of the underlying environment. The function $\mathsf{AverageDistribution}$ computes an initial state distribution for the input abstract product vertex $\overline{u}$ by taking an average of all of the distributions in $\Gamma(\overline{u})$.

$\mathsf{LearnEdgePolicy}$*:* Use (single agent) RL to learn a co-operative joint policy π_e so that π_e achieves the edge $e = \overline{u} \to \overline{v}$ from the given initial state distribution $\eta_{\overline{u}}$. We use single-agent RL, specifically Q-learning, to learn a co-operative joint policy to achieve an edge with the reward $\mathbb{1}(\zeta \models_B e)$. Precisely, we learn π_e such that

$$\pi_e \in \arg\max_{\pi} \Pr_{s_0 \sim \eta_{\overline{u}}, \zeta \sim \mathcal{D}_{\pi,s_0}} [\zeta \models_B e]$$

where $\zeta \sim \mathcal{D}_{\pi_e,s_0}$ is the trajectory sampled from executing policy π_e from state s_0.

$\mathsf{ReachDistribution}$*:* Given an edge $e = \overline{u} \to \overline{v}$, edge policy π_e, and initial state distribution $\eta_{\overline{u}}$, this function evaluates the state distribution induced on $\overline{v}$ upon executing policy π_e with an initial state distribution $\eta_{\overline{u}}$. Formally, for any $s \in \mathcal{S}$

$$\Pr_{s' \sim \eta_{\overline{v},e}} [s = s'] = \Pr_{s_0 \sim \eta_{\overline{u}}, \zeta \sim \mathcal{D}_{\pi_e,s_0}} [s = s_k \mid \zeta_{0:k} \models_B e \text{ and } \forall k' < k, \zeta_{0:k'} \not\models_B e]$$

where $\zeta \sim \mathcal{D}_{\pi_e,s_0}$ is the trajectory sampled from executing policy π_e from state s_0 and k is the length of the smallest prefix of ζ that achieves e.

Algorithm 3 PRIORITIZEDENUMERATION
Inputs: n-agent Environment $\mathcal{M}$ and agent specifications $\phi_1, \ldots, \phi_n$
Output: Ranking scheme

```
1: Initialize pathPolicyWelfareMaxHeap ← ∅
2: for i ∈ [n] do G_i ← AbstractGraph(φ_i)
3: for B ∈ 2^[n] do
4:    G_B ← ProductAbstractGraph(G_1, ... G_n, B)
5:    Initialize path policies Π(ū_0) ← {ε} and Π(ū) ← ∅ if ū ≠ ū_0
6:    Initialize state distribution Γ(ū_0) ← {At(s_0)} and Γ(ū) ← ∅ if ū ≠ ū_0
7:    topoSortedVertexStack ← TopologicalSort(Ū, Ē)
8:    while topoSortedVertexStack ≠ ∅ do
9:       ū ← topoSortedVertexStack.pop()
10:      η_ū ← AverageDistribution(Γ(ū))
11:      for e = ū → v̄ ∈ outgoing(ū) do
12:         π_e ← LearnEdgePolicy(e, η_ū)
13:         η_{v̄,e} ← ReachDistribution(e, π_e, η_ū)
14:         Add η_{v̄,e} to Γ(v̄)
15:         for π_ρ ∈ Π(ū) do Add π_ρ ∘ π_e to Π(v̄)
16:      if ū ∈ F then
17:         for π_ρ ∈ Π(ū) do
18:            c ← EstimatePolicyWelfare(π_ρ, φ_1, ..., φ_n)
19:            Add (π_ρ, c) to pathPolicyWelfareMaxHeap
20: return pathPolicyWelfareMaxHeap
```

EstimatePolicyWelfare*:* Once a path policy π_ρ is learnt, we estimate the probability of satisfaction of the specifications $J_i(\pi_\rho)$ for all agents i using Monte-Carlo sampling. Then `welfare`(π_ρ) is computed by taking the mean of the probabilities of satisfaction of agent specifications.

B Nash Equilibria Verification

B.1 Best Punishment Strategy

Our verification procedure involves computing the best punishment strategies $\tau^*[j]$ against agent j for each $j \in [n]$. We give a proof of Theorem 3 below from which it follows that computing the best punishment strategies is sufficient to decide whether a given finite-state deterministic joint policy π can be extended to an ϵ-Nash equilibrium.

Proof (Proof of Theorem 3). It follows from the definition of $\pi \bowtie \tau$ that the distribution over H-length trajectories induced by $\pi \bowtie \tau$ in $\mathcal{M}$ is the same as the one induced by π since τ is never triggered when all agents are following π. Therefore, $J_j(\pi \bowtie \tau) = J_j(\pi)$ for all j and τ. Now, suppose there is a τ such that $\pi \bowtie \tau$ is an ϵ-Nash equilibrium. Then for all j,

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

But $\tau^*[j]$ minimizes the LHS of the above equation which is independent of $\tau \setminus \tau[j]$. Therefore, for all j,

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

Hence, $\pi \bowtie \tau^*$ is an ϵ-Nash equilibrium. The rest of the Theorem follows from the definition of ϵ-Nash equilibrium. □

B.2 Reward Machine Construction

In this section, we detail the construction of reward machines from SPECTRL specifications such that the reward of any finite-length trajectory is 1 if the trajectory satisfies the specification and 0 otherwise.

We proceed by constructing deterministic finite-state automata (DFA) that accepts all trajectories which satisfy the specification. Next, we will convert the DFA into a reward machine with the desired reward function.

DFA Construction. A finite-state automaton is a tuple $D = (Q, \mathcal{B}, \delta, q_{\mathsf{init}}, F)$ where Q is a finite-set of states, $\mathcal{B}$ is a finite-set of propositions, q_{init} is the initial state, and $F \subseteq Q$ is the set of accepting states. The transition relation is defined as $\delta \subseteq Q \times \mathsf{Formula}(\mathcal{B}) \times Q$ where $\mathsf{Formula}(\mathcal{B})$ is the set of boolean formulas over propositions $\mathcal{B}$. A finite-state automaton is *deterministic* if every assignment $\sigma \in 2^{\mathcal{P}}$ can transition to a unique state from every state, i..e, if for all states $q \in Q$ and assignments $\sigma \in 2^{\mathcal{B}}$, $|\{q' \mid (q, b, q') \in \delta \text{ and } \sigma \models b\}| \leq 1$. Otherwise, it is *non-deterministic*. Every non-deterministic finite-state automata (NFA) can be converted to a deterministic finite-state automata (DFA). A *run* of a *word* (sequence of assignments over $\mathcal{B}$) given by $w = w_0 \dots w_m \in (2^{\mathcal{B}})^*$ is a sequence of states $\rho = q_0 \dots q_{m+1}$ such that $q_0 = q_{\mathsf{init}}$ and there exists $(q_i, b_i, q_{i+1}) \in \delta$ such that $w_i \models b_i$ for all $0 \leq i < m$. A run $q_0 \dots q_{m+1}$ is accepting if $q_{m+1} \in F$. A word w is accepted by D if it has an accepting run.

Let SPECTRL specifications be defined over the set of basic predicates $\mathcal{P}_0$.We define a labelling function $L : \mathcal{S} \to 2^{\mathcal{P}_0}$ such that $L(s) = \{p \mid [\![p]\!](s) = \mathtt{true}\}$. Given a trajectory $\zeta = s_0, \dots, s_t$ in the environment $\mathcal{M}$, let its proposition sequence $\mathcal{L}(\zeta)$ be given by $\mathcal{L}(s_0), \dots, \mathcal{L}(s_t)$.

Lemma 1. *Given* SPECTRL *specification ϕ, we can construct a DFA D_ϕ such that a trajectory $\zeta \models \phi$ iff $\mathcal{L}(\zeta)$ is accepted by D_ϕ.*

Proof. We use structural induction on SPECTRL specifications to construct the desired DFA. The construction is very similar to the construction of finite-state automata from regular expressions, and hence details of proof have been skipped [19]. The construction is given below:

Eventually (ϕ ::= `achieve` b). Construct finite-state automata $D_\phi = (\{q_{\mathsf{init}}, q\}, \mathcal{P}_0, \delta, q_{\mathsf{init}}, \{q\})$ where

$$\delta = \{(q_{\mathsf{init}}, \neg b, q_{\mathsf{init}}), (q_{\mathsf{init}}, b, q), (q, \mathsf{True}, q)\}.$$

Clearly, D_ϕ is deterministic because the only state from which more that two transitions emanate is q_{init} is defined on functions that negate one another (b and $\neg b$), hence they will not have any common assignment.

Always (ϕ ::= ϕ_1 `ensuring` b.) Let the DFA for ϕ_1 be $D_1 = (Q, \mathcal{P}_0, \delta_1, q_{\mathsf{init}}, F)$. Then the DFA for ϕ is given by $D_\phi = (Q, \mathcal{P}_0, \delta, q_{\mathsf{init}}, F)$ where

$$\delta = \{(q, b \wedge b', q') \mid (q, b', q') \in \delta_1\}.$$

D_ϕ is deterministic because D_1 is deterministic.

Sequencing ($\phi_1; \phi_2$). Let the DFA for ϕ_1 and ϕ_2 be $D_1 = (Q_1, \mathcal{P}_0, \delta_1, q^1_{\mathsf{init}}, F_1)$ and $D_2 = (Q_2, \mathcal{P}_0, \delta_2, q^2_{\mathsf{init}}, F_2)$, respectively. Construct the NFA $N_\phi = (Q_1 \sqcup Q_2, \mathcal{P}_0, \delta, q^1_{\mathsf{init}}, F_2)$ where

$$\delta = \delta_1 \cup \delta_2 \cup \bigcup_{f \in F_1} \mathsf{DivertAwayFrom}(f)$$

where $\mathsf{DivertAwayFrom}(f) = \{(q^1_1, b, q^2_{\mathsf{init}})\} \mid (q^1_1, b, f) \in \delta_1\}$. Essentially, transitions in $\mathsf{DivertAwayFrom}(f)$ divert all incoming transitions to $f \in F_1$ to the initial state of the second DFA.

Then, the DFA D_ϕ is obtained from determinization of N_ϕ.

Choice (ϕ ::= ϕ_1 `or` ϕ_2). Let the DFA for ϕ_1 and ϕ_2 be $D_1 = (Q_1, \mathcal{P}_0, \delta_1, q^1_{\mathsf{init}}, F_1)$ and $D_2 = (Q_2, \mathcal{P}_0, \delta_2, q^2_{\mathsf{init}}, F_2)$, respectively. Construct NFA $N_\phi = (Q_1 \sqcup Q_2 \setminus \{q^2_{\mathsf{init}}\}, \mathcal{P}_0, \delta, q^1_{\mathsf{init}}, F_1 \sqcup F_2)$ where

$$\begin{aligned} \delta = \;& \delta_1 \cup \delta_2 \setminus \{(q^2_1, b, q^2_2) \mid (q^2_1, b, q^2_2) \in \delta_2 \text{ and } q^2_1 = q^2_{\mathsf{init}}\} \\ & \cup \{(q^1_{\mathsf{init}}, b, q^2_2) \mid (q^2_1, b, q^2_2) \in \delta_2 \text{ and } q^2_1 = q^2_{\mathsf{init}}\} \end{aligned}$$

Then, the DFA D_ϕ is obtained from determinization of N_ϕ.

Lastly, we can extend DFA D_ϕ to make it *complete*, i.e., if for all states $q \in Q$ and assignments $\sigma \in 2^{\mathcal{P}_0}$, $|\{q' \mid (q, b, q') \in \delta \text{ and } \sigma \models b\}| = 1$.

Reward Machine. Given a SPECTRL specification ϕ, let $D_\phi = (Q, \mathcal{P}_0, \delta, q_{\mathsf{init}}, F)$ be the DFA such that a trajectory $\zeta \models \phi$ iff $L(\zeta)$ is accepted by the DFA D_ϕ, where $L : \mathcal{S} \to 2^{\mathcal{P}_0}$ is the labelling function. WLOG, assume $\mathcal{D}_\phi$ is complete.

Construct a reward machine $\mathcal{R}_\phi = (Q \cup \{\mathsf{dead}\}, \delta_u, \delta_r, q_{\mathsf{init}})$ where the state transition function $\delta_u : \mathcal{S} \times \mathcal{A} \times Q \to Q$ is defined as

$$\delta_u(s, -, q) = q' \text{where } (q, b, q') \in \delta \text{ s.t. } L(s) \models b$$

and the reward function $\delta_r : \mathcal{S} \times Q \to [-1, 1]$ is given by

$$\delta_r(s, q) = \begin{cases} 1 & \text{if } q \notin F, q' = \delta_u(s, -, q') \text{ and } q' \in F \\ -1 & \text{if } q \in F, q' = \delta_u(s, -, q') \text{ and } q' \notin F \\ 0 & \text{otherwise} \end{cases}$$

Observe that the above functions are well defined since the DFA is deterministic and complete.

Proof (Proof of Theorem 4). Let $\mathcal{R}_\phi$ be as constructed above. Let $\zeta = s_0, s_1, \ldots s_t, s_{t+1}$. Then, by construction a run $\rho = q_0, q_1 \ldots q_{t+1}$ of $L(\zeta_{0:t})$ in D_ϕ is also a run of ζ in $\mathcal{R}_\phi$. Then, the reward function is design so that (a) each time the run visits a state in F from a non-accepting state, it will receive a reward of 1, (b) each time the run visits a state in $Q \setminus F$ from a state in F, and it receives -1, and (c) 0 otherwise.

Suppose $\zeta_{0:t}$ does not satisfy ϕ. Then ρ is not an accepting run in D_ϕ. Then, each time the run visits a state in F, the run will exit states in F after a finite amount of time. Thus, either ζ receives a reward of 0 or it receives a reward of 1 and -1 an equal number of times. In this case, $\mathcal{R}_\phi(\zeta) = 0$ since the +1s and -1s will cancel each other out.

Suppose $\zeta \models \phi$. Then, ρ is an accepting run in D_ϕ. Let k be the largest index such that $q_k \notin F$ and $q_\ell \in F$ for all $k < \ell \leq t + 1$. So, $\delta_r(s_k, q_k) = 1$ and $\delta_r(s_\ell, q_\ell) = 0$ for all $k < \ell \leq t$. Additionally, the run $q_0, \ldots, q_k$ is not an accepting run in $\mathcal{D}_\phi$. Thus, the trajectory $\zeta_{0:k-1}$ does not satisfy ϕ, thus $\mathcal{R}_\phi(\zeta_{0:k-1}) = 0$. So, $\mathcal{R}_\phi(\zeta) = \Sigma_{z=0}^{t} \delta_r(s_z, q_z) = \Sigma_{z=0}^{k-1} \delta_r(s_z, q_z) + \delta_r(s_k, q_k) + \Sigma_{z=k+1}^{t} \delta_r(s_z, q_z)$. Since $\Sigma_{z=0}^{k-1} \delta_r(s_z, q_z) = \Sigma_{z=k+1}^{t} \delta_r(s_z, q_z) = 0$, we get that $\mathcal{R}_\phi(\zeta) = 1$. □

B.3 Simulating Punishment Game

The optimal deviation score of agent j w.r.t π, $\texttt{dev}_j^\pi$, is the min-max value of a two player zero-sum Markov game in which the max-agent is agent j of $\mathcal{M}$ and the min-agent is a coalition of punishing agents $\{i \in [n] \mid i \neq j\}$ whose combined policy is constrained to be of the form $(\pi \bowtie \tau)_{-j}$. Before solving this min-max game, we eliminate the constraint on the min-agent's policy by constructing a product of $\mathcal{M}$ with reward machine $\mathcal{R}_j = (Q_j, \delta_u^j, \delta_r^j, q_0^j)$ and finite-state deterministic joint policy $\pi = (M, \alpha, \sigma, m_0)$. We now describe the product construction.

Proof (Proof of Theorem 5). For an agent j, the two-player zero-sum game $\mathcal{M}_j$ is defined by $\mathcal{M}_j = (\mathcal{S}_j, A_j, \mathcal{A}_{-j}, P_j, H + 1, s_0^j, R_j)$ with rewards where,

- The set of states $\mathcal{S}_j$ is the product $\mathcal{S}_j = \mathcal{S} \times M \times Q_j \times \{\perp, \top\}$.
- The set of actions of the max-agent is A_j and the set of actions of the min-agent is $\mathcal{A}_{-j} = \prod_{i \neq j} A_i$. Given $a_j \in A_j$ and $a_{-j} \in \mathcal{A}_{-j}$, we denote the joint action by $(a_j, a_{-j}) \in \mathcal{A}$.

– The last component of a state denotes whether agent j has deviated from π_j in the past or not. Intuitively, $\bot$ implies that agent j has *not* deviated from π_j in the past and $\top$ implies that it has deviated from π_j in the past. We define an update function f_j which is used to update this information at every step. The deviation update function $f_j : \mathcal{S} \times \mathcal{A} \times M \times \{\bot, \top\} \to \{\bot, \top\}$ is defined by $f_j(s, a, m, \top) = \top$ and $f_j(s, a, m, \bot) = \bot$ if $a_j = \sigma(s, m)_j$ and $\top$ otherwise.

  The transitions of $\mathcal{M}_j$ are such that the action of the min-agent a_{-j} is ignored and replaced with the output of π_{-j} until agent j deviates from π_j (or equivalently, until the last component of the state is $\top$). The transition probabilities are given by
  - $P_j((s, m, q, b), a, (s', m', q', b')) = P(s, (a_j, \sigma(s, m)_{-j}), s')$ if $m' = \alpha(s, (a_i, \sigma(s, m)_{-j}), m)$, $q' = \delta_u(s, (a_i, \sigma(s, m)_{-j}), q)$, $b = \bot$ and $b' = f_j(s, a, m, b)$.
  - $P_j((s, m, q, b), a, (s', m', q', b')) = P(s, a, s')$ if $m' = \alpha(s, a, m)$, $q' = \delta_u(s, a, q)$, $b = \top$ and $b' = f_j(s, a, m_j, b)$.
  - $P_j((s, m, q, b), a, (s', m', q', b')) = 0$ otherwise.

– The initial state is $s_0^j = (s_0, m_0, q_0^j, \bot)$.
– The rewards are given by $R_j((s, m, q, b), a) = \delta_r^j(s, q)$.

Let us denote by $\bar{\pi}_1$ and $\bar{\pi}_2$ the policies of the max-agent and the min-agent respectively. Then the expected reward attained by the max-agent is

$$\bar{J}_j(\bar{\pi}_1, \bar{\pi}_2) = \mathbb{E}\Big[\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) \mid \bar{\pi}_1, \bar{\pi}_2\Big]$$

where the expectation is w.r.t. the distribution over trajectories of length $H+1$ generated by using $(\bar{\pi}_1, \bar{\pi}_2)$ in $\mathcal{M}_j$. Given a trajectory $\bar{\zeta} = \bar{s}_0 \xrightarrow{a_0} \bar{s}_1 \xrightarrow{a_1} \dots \xrightarrow{a_{t-1}} \bar{s}_t$ in $\mathcal{M}_j$, we denote by $\bar{\zeta} \downarrow_{\mathcal{M}}$ the trajectory projected to the state space of $\mathcal{M}$—i.e., $\bar{\zeta} \downarrow_{\mathcal{M}} = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{t-1}} s_t$. From Theorem 4 and the above definition of $\mathcal{M}_j$ it follows that for any trajectory $\bar{\zeta}$ in $\mathcal{M}_j$ of length $H+1$ we have

$$\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) = \mathcal{R}_j(\bar{\zeta} \downarrow_{\mathcal{M}}) = \mathbb{1}(\bar{\zeta}_{0:H} \downarrow_{\mathcal{M}} \models \phi_j).$$

Let $\mathcal{D}^{\mathcal{M}}[\pi]$ denote the distribution over length H trajectories in $\mathcal{M}$ generated by π and $\mathcal{D}^{\mathcal{M}_j}[\bar{\pi}]$ denote the distribution over length $H+1$ trajectories in $\mathcal{M}_j$ generated by $\bar{\pi}$. It is easy to see that any policy π'_j for agent j in $\mathcal{M}$ can be interpreted as a policy $g(\pi'_j)$ for the max-agent in $\mathcal{M}_j$ and any policy $\bar{\pi}_1$ for the max-agent in $\mathcal{M}_j$ can be interpreted as a policy $g'(\bar{\pi}_1)$ for agent j in $\mathcal{M}$. Since the actions of the min-agent in $\mathcal{M}_j$ are only taken into account after agent j deviates from π_j, we also have that any policy of the form $(\pi \bowtie \tau)_{-j}$ for the punishing agents in $\mathcal{M}$ corresponds to a policy $h(\tau)$ of the min-agent in $\mathcal{M}_j$ and any policy $\bar{\pi}_2$ of the min-agent in $\mathcal{M}_j$ corresponds to a policy $(\pi \bowtie h'(\bar{\pi}_2))_{-j}$ for the punishing agents in $\mathcal{M}$. Furthermore the mappings g, g', h, h' satisfy the property that for any trajectory $\bar{\zeta}$ of length $H+1$ in $\mathcal{M}_j$, we have

– for any τ and π'_j in $\mathcal{M}$, $\mathcal{D}^{\mathcal{M}}[(\pi \bowtie \tau)_{-j}, \pi'_j](\bar{\zeta} \downarrow_{\mathcal{M}}) = \mathcal{D}^{\mathcal{M}_j}[g(\pi'_j), h(\tau)](\bar{\zeta})$ and,
– for any $\bar{\pi}_1$ and $\bar{\pi}_2$ in $\mathcal{M}$, $\mathcal{D}^{\mathcal{M}}[(\pi \bowtie h'(\bar{\pi}_2))_{-j}, g'(\bar{\pi}_1)](\bar{\zeta} \downarrow_{\mathcal{M}}) = \mathcal{D}^{\mathcal{M}_j}[\bar{\pi}_1, \bar{\pi}_2](\bar{\zeta})$.

Therefore

$$
\begin{aligned}
\texttt{dev}_j^\pi &= \min_{\tau[j]} \max_{\pi'_j} J_j((\pi \bowtie \tau)_{-j}, \pi'_j) \\
&= \min_{\tau[j]} \max_{\pi'_j} \mathbb{E}_{\zeta \sim \mathcal{D}^{\mathcal{M}}[(\pi \bowtie \tau)_{-j}, \pi'_j]} \Big[\zeta \models \phi_i \Big] \\
&= \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \mathbb{E}_{\bar{\zeta} \sim \mathcal{D}^{\mathcal{M}_j}[\bar{\pi}_1, \bar{\pi}_2]} \Big[\sum_{k=0}^{H} R_j(\bar{s}_k, a_k) \Big] \\
&= \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}_j(\bar{\pi}_1, \bar{\pi}_2).
\end{aligned}
$$

It is easy to see that the function h' has the desired properties of PunStrat. Finally, we observe that given a simulator for $\mathcal{M}$ it is straightforward to construct a simulator for $\mathcal{M}_j$. Similarly, given an estimate $\tilde{\mathcal{M}}$ of $\mathcal{M}$ we can use the above definition of $\mathcal{M}_j$ to construct an estimate $\tilde{\mathcal{M}}_j$ of $\mathcal{M}_j$. □

B.4 Solving Punishment Games

Our sample efficient algorithm for solving the punishment game relies on Assumption 1. There are algorithms for solving min-max games (with unknown transition probabilities) without this assumption [4] albeit with slightly worse sample complexity. We outline the details of our algorithm below.

Estimating $\mathcal{M}$ under Assumption 1. The first step is to estimate the transition probabilities of $\mathcal{M}$ using BFS-Estimate which is outlined in Algorithm 4. BFS-Estimate performs a breadth-first-search on the transition graph of $\mathcal{M}$. In order to figure out all outgoing edges from a state s, multiple samples are collected by taking each possible action K-times from s. Newly discovered states are then added to the state space of $\tilde{\mathcal{M}}$ and the collected samples are used to estimate transition probabilities. The value K is defined in line 2 of the algorithm in which $|Q|$ denotes the maximum size of the state space of reward machine $\mathcal{R}_j$ for any agent j—i.e., $|Q| = \max_j |Q_j|$. BFS-Estimate has the following approximation guarantee.

Lemma 2. *With probability at least $1 - p$, for all $s \in \tilde{\mathcal{S}}$, $a \in \mathcal{A}$ and $s' \in \mathcal{S}$,*

$$
\left| \tilde{P}(s' \mid s, a) - P(s' \mid s, a) \right| \leq \varepsilon = \frac{\delta}{2|\mathcal{S}||M||Q|H^2}
$$

where $\tilde{P}(s' \mid s, a)$ is taken to be 0 if $s' \notin \tilde{\mathcal{S}}$.

Algorithm 4 BFS-Estimate
Inputs: Precision δ, failure probability p.
Outputs: Estimated model $\tilde{\mathcal{M}}$ of $\mathcal{M}$.

```
1: M̃ ← (S̃ = {s_0}, A, P̃ = ∅, H, s_0)
2: K ← ⌈ (2|S|^2 |M|^2 |Q|^2 H^4 / δ^2) log(2|S|^2|A| / p) ⌉
3: queue ← [s_0]
4: while ¬ queue.isempty() do
5:    s ← queue.pop()
6:    for a ∈ A do
7:       // Initialize number of visits to each state
8:       N ← empty-map()
9:       for s' ∈ S̃ do
10:         N[s'] ← 0
11:      // Obtain K samples for the state-action pair (s, a)
12:      for x ∈ {1, ..., K} do
13:         s' ~ P(· | s, a)
14:         // Add any newly discovered state to S̃ and the map N
15:         if s' ∉ S̃ then
16:            S̃ ← S̃ ∪ {s'}
17:            queue.add(s')
18:            N[s'] ← 0
19:         // Increment number of visits to s'
20:         N[s'] ← N[s'] + 1
21:      // Store estimated transition probabilities in P̃
22:      for s' ∈ S̃ do
23:         P̃(s' | s, a) ← N[s'] / K
24: return M̃
```

Proof. For any given $s \in \tilde{\mathcal{S}}$, $a \in \mathcal{A}$ and $s' \in \mathcal{S}$, the probability $\tilde{P}(s' \mid s, a)$ is estimated using K independent samples from $P(\cdot \mid s, a)$. Therefore, using Chernoff bounds, we get

$$\Pr\left[\left|\tilde{P}(s' \mid s, a) - P(s' \mid s, a)\right| > \varepsilon\right] \leq 2e^{-2K\varepsilon^2}$$

Applying union bound over all triples $(s, a, s') \in \tilde{\mathcal{S}} \times \mathcal{A} \times \mathcal{S}$ and substituting the values of K and ε, we get

$$\begin{aligned}\Pr\left[\bigcup_{s,a,s'} \left\{\left|\tilde{P}(s' \mid s, a) - P(s' \mid s, a)\right| > \varepsilon\right\}\right] &\leq 2|\mathcal{S}|^2|\mathcal{A}|e^{-2K\varepsilon^2} \\ &\leq 2|\mathcal{S}|^2|\mathcal{A}|e^{-\log(\frac{2|\mathcal{S}|^2|\mathcal{A}|}{p})} = p.\end{aligned}$$

Hence we obtained the desired bound. □

Obtaining estimates of $\mathcal{M}_j^\pi$. After estimating $\mathcal{M}$, for any finite-state deterministic joint policy π and any agent j, we perform the product construction outlined

in Section B.3 with the estimated model $\tilde{\mathcal{M}}$ to obtain an estimate $\tilde{\mathcal{M}}_j^\pi$ of the punishment game $\mathcal{M}_j^\pi$. The constructed model $\tilde{\mathcal{M}}_j^\pi$ can be used to estimate $\texttt{dev}_j^\pi$ as claimed in Theorem 6.

Proof (Proof of Theorem 6). Since the transition probabilities in $\tilde{\mathcal{M}}_j^\pi$ are inherited from $\tilde{\mathcal{M}}$, from Lemma B.4 we have that with probability at least $1-p$, for any π, j, $\bar{s}, \bar{s}' \in \mathcal{S}_j$ and $a \in \mathcal{A}$ such that $\bar{s}$ is in the state space of $\tilde{\mathcal{M}}_j^\pi$,

$$\left|\tilde{P}_j(\bar{s}' \mid \bar{s}, a) - P_j(\bar{s}' \mid \bar{s}, a)\right| \leq \varepsilon = \frac{\delta}{2|\mathcal{S}||M||Q|H^2}$$

where $\tilde{P}_j$ represents the transition probabilities of $\tilde{\mathcal{M}}_j$[3]. Consider the model $\tilde{\mathcal{M}}_j'$ which is a modification of $\tilde{\mathcal{M}}_j$ that has state space $\mathcal{S}_j$ (state space of $\mathcal{M}_j$) and for any $\bar{s}, \bar{s}' \in \mathcal{S}_j$ and $a \in \mathcal{A}$, its transition probabilities are defined by $\tilde{P}_j'(\bar{s'} \mid \bar{s}, a) = \tilde{P}_j(\bar{s'} \mid \bar{s}, a)$ if $\bar{s}$ is in the state space of $\tilde{M}_j$ and $P_j(\bar{s'} \mid \bar{s}, a)$ otherwise. We have $\left|\tilde{P}_j'(\bar{s}' \mid \bar{s}, a) - P_j(\bar{s}' \mid \bar{s}, a)\right| \leq \varepsilon$ for all $\bar{s}, \bar{s}' \in \mathcal{S}_j$ and $a \in \mathcal{A}$. For any two policies $\bar{\pi}_1$ and $\bar{\pi}_2$ of the max-agent and the min-agent in $\mathcal{M}_j$ respectively, we denote by $\bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$ and $\bar{J}^{\tilde{\mathcal{M}}_j'}(\bar{\pi}_1, \bar{\pi}_2)$ the expected reward over $H+1$ length trajectories generated by $(\bar{\pi}_1, \bar{\pi}_2)$ in $\tilde{\mathcal{M}}_j$ and $\tilde{\mathcal{M}}_j'$ respectively. Then $\bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) = \bar{J}^{\tilde{\mathcal{M}}_j'}(\bar{\pi}_1, \bar{\pi}_2)$ because both the models assign the same probability to all runs as any run that leaves the state space of $\tilde{\mathcal{M}}_j$ has probability zero in both the models. Now we can apply Lemma 4 of [7] to conclude that $|\bar{J}^{\tilde{\mathcal{M}}_j'}(\bar{\pi}_1, \bar{\pi}_2) - \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2)| \leq \delta$ and hence $|\bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2)| \leq \delta$ for any $\bar{\pi}_1$ and $\bar{\pi}_2$. This implies that for any $\bar{\pi}_2$ we have

$$|\max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \max_{\bar{\pi}_1} \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2)| \leq \delta \tag{4}$$

and therefore can conclude that

$$|\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2)| \leq \delta.$$

Applying Theorem 5 we get

$$|\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2) - \texttt{dev}_j^\pi| \leq \delta.$$

Now, let $\bar{\pi}_2^* = \arg\min_{\bar{\pi}_2} \max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2)$ and $\tau[j] = \textsc{PunStrat}(\bar{\pi}_2^*)$. Then from Equation 4 we can conclude that $|\max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j}(\bar{\pi}_1, \bar{\pi}_2^*) - \max_{\bar{\pi}_1} \bar{J}_j^\pi(\bar{\pi}_1, \bar{\pi}_2^*)| \leq \delta$. Using Theorem 5 we get

$$\left|\max_{\bar{\pi}_1} \bar{J}^{\tilde{\mathcal{M}}_j^\pi}(\bar{\pi}_1, \bar{\pi}_2^*) - \max_{\pi_j'} J_j((\pi \bowtie \tau[j])_{-j}, \pi_j')\right| \leq \delta.$$

Finally the total number of samples used is at most $|\mathcal{S}||\mathcal{A}|K = O\left(\frac{|\mathcal{S}|^3|M|^2|Q|^4|\mathcal{A}|H^4}{\delta^2} \log\left(\frac{|\mathcal{S}||\mathcal{A}|}{p}\right)\right)$. □

[3] Omitting the superscript π in $\tilde{\mathcal{M}}_j^\pi$

Algorithm 5 Nash Value Iteration
Inputs: n-agent Markov game $\mathcal{M}$ with rewards, horizon H.
Outputs: Nash equilibrium joint policy $\pi = (\pi_1, \ldots, \pi_n)$.

1: Initialize joint policy $\pi = (\pi_1, \ldots, \pi_n)$
2: Initialize value function $V : \mathcal{S} \times [H+1] \to \mathbb{R}^n$ to be the zero map
3: **for** $t \in \{H, H-1, \ldots, 1\}$ **do**
4: **for** $s \in \mathcal{S}$ **do**
5: Initialize step game $G_s^t : \mathcal{A} \to \mathbb{R}^n$
6: **for** $a = (a_1, \ldots, a_n) \in \mathcal{A}$ **do**
7: $G_s^t(a_1, \ldots, a_n) = R(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)}[V(s', t+1)]$
8: $(d_1, d_2, \ldots, d_n) \leftarrow$ Best-Nash-General-Sum$(G_s^t) \in \mathcal{D}(A_1) \times \cdots \times \mathcal{D}(A_n)$
9: $V(s,t) \leftarrow \mathbb{E}_{a_1 \sim d_1, a_2 \sim d_2, \ldots, a_n \sim d_n}[G_s^t(a_1, \ldots, a_n)]$
10: $\pi(s,t) \leftarrow (d_1, d_2, \ldots, d_n)$
11: **return** π

For each j, the min-max game $\tilde{\mathcal{M}}_j$ is solved in polynomial time using value iteration [4] to compute an estimate $\tilde{\texttt{dev}}_j$ of $\texttt{dev}_j^\pi$ which is used in Line 9 of Algorithm 2 to check whether agent j can successfully deviate from π_j.

B.5 Soundness Guarantee

The soundness guarantee of HighNashSearch follows from Theorem 6.

Proof (Proof of Corollary 1). From Theorem 6 we get that with probability at least $1-p$, if a policy $\tilde{\pi} = \pi \bowtie \tau$ is returned by our algorithm, then for all j we have

$$\begin{aligned} \max_{\pi_j'} J_j((\pi \bowtie \tau)_{-j}, \pi_j') &\leq \tilde{\texttt{dev}}_j + \delta && \text{(Equation 3)} \\ &\leq J_j(\pi) + \epsilon && \text{(Line 9 of Algorithm 2)} \\ &= J_j(\pi \bowtie \tau) + \epsilon. \end{aligned}$$

Therefore, $\pi \bowtie \tau$ is an ϵ-Nash equilibrium. □

C Baselines

C.1 Nash Value Iteration

This baseline first computes an estimate $\tilde{\mathcal{M}}$ of $\mathcal{M}$ using BFS-Estimate (Algorithm 4) 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 5 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` [34] for solving the step games.

Algorithm 6 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 $\pi = (\pi_1, \dots, \pi_n)$.

1: **for** $i \in [n]$ **do** $(U_i, \delta_u^i, \delta_r^i, u_0^i) \leftarrow \mathsf{RewardMachine}(\phi_i)$
2: // Initialize environment state, reward machines state, and Q-functions
3: Current state $s \leftarrow s_0$
4: **for** $i \in [n]$ **do** $u_i \leftarrow u_0^i$
5: **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 $a_i \in \mathcal{A}_i$
6: **for** $l \in \{0, \dots, N\}$ **do**
7: // Sample actions from policy derived from Q-functions
8: **for** $i \in [n]$ **do** choose action $a_i \in \mathcal{A}_i$ at $(s, (u_1, \dots, u_n))$ using exploration policy derived from Q_i (e.g., ε-greedy)
9: // Take a step in environment and the reward machines
10: Take action $a = (a_1, \dots a_n)$ in $\mathcal{M}$ and observe the next state s'
11: **for** $i \in [n]$ **do** compute the reward $r_i \leftarrow \delta_r^i(s, u_i)$ and next RM state $u_i' \leftarrow \delta_u^i(s, a, u_i))$
12: // Update all Q-functions
13: **if** s' is terminal **then**
14: **for** $i \in [n]$ **do** $Q_i(s, (u_1, \dots, u_n), a) \xleftarrow{\alpha} r_i$
15: **else**
16: **for** $i \in [n]$ **do** $Q_i(s, (u_1, \dots, u_n), a_i) \xleftarrow{\alpha} r_i + \gamma \cdot \max_{a_i' \in \mathcal{A}_i} Q_i(s', (u_1', \dots, u_n'), a_i')$
17: **if** s' is terminal **then**
18: // Reset environment state and reward machines state
19: $s \leftarrow s_0$ and **for** $i \in [n]$ **do** $u_i \leftarrow u_0^i$
20: **else**
21: $s \leftarrow s'$ and **for** $i \in [n]$ **do** $u_i \leftarrow u_i'$
22: **for** $i \in [n]$ **do** $\pi_i \leftarrow$ Best action policy derived from Q_i
23: **return** $(\pi_1, \dots, \pi_n)$

C.2 Multi-agent QRM

The second baseline (Algorithm 6) is a multi-agent variant of QRM [22, 39]. We derive reward machines from agent specifications using the procedure described in Appendix B.2.

We learn one Q-function for each 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 $q \leftarrow (1 - \alpha) \cdot q + \alpha \cdot q'$.

D Benchmark Details

D.1 Intersection

The specifications and the corresponding initial states for the intersection environment are described below.

- 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.
- 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.

D.2 Single Lane Environment

The environment consists of k agents along a straight track of length l. All agents are initially placed at the 0th location and the destination is at the lth location. In a single step, each agent can either move forward one location (with a failure probability of 0.05) or remain in its current position. We create competitive and co-operative scenarios through agent specifications. For example, we can create competitive scenarios in which an agent meets its specification only if it reaches the final location before all other or a set of other agents. We can also create co-operative scenarios in which the i-th agent must reach its goal before the j-th agent, for various pairs of agents (i, j). In these cases, the highest social welfare would occur when the agents manage to coordinate so that all ordering constraints are satisfied (we ensure that there are no cycles in the ordering constraints). The specifications are described below and results are in Table 2.

Specifications

- ϕ^1 All agents should reach the final state
- ϕ^2 Agent 1 should reach its destination before Agent 0. Agent 2 must reach the destination.
- ϕ^3 Agent 1 should reach its destination before Agent 0. All agents must reach their destination.
- ϕ^4 Agent 0 and Agent 1 are competing to reach the destination first. Only the agent reaching first meets the specification. Agent 2's goal is to reach the destination.

ϕ^5 Agent 0 should reach the mid-point before Agent 1. However, Agent 1 should reach the final destination before Agent 0. All agents should eventually reach the final destination.

ϕ^6 Agent 0 should reach the mid-point before Agent 1. However, Agent 1 and Agent 2 should reach the final destination before Agent 0. All agents should reach the final destination.

Spec.	Num. of agents	Algorithm	welfare(π) (avg ± std)	$\epsilon_{\min}(\pi)$ (avg ± std)	Num. of runs terminated	Avg. num. of sample steps (in millions)
ϕ^1	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.02
		NVI	1.00 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	1.00 ± 0.00	0.01 ± 0.00	10	4.00
ϕ^2	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.01
		NVI	1.00 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	0.66 ± 0.00	0.99 ± 0.00	10	4.00
ϕ^3	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.50
		NVI	1.00 ± 0.00	0.00 ± 0.00	8	5.00
		MAQRM	0.81 ± 0.01	0.56 ± 0.03	10	4.00
ϕ^4	3	HighNashSearch	0.67 ± 0.00	0.00 ± 0.00	10	3.03
		NVI	0.33 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	0.63 ± 0.00	0.57 ± 0.01	10	4.00
ϕ^5	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.60
		NVI	0.33 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	0.62 ± 0.01	0.47 ± 0.02	10	4.00
ϕ^6	3	HighNashSearch	1.00 ± 0.00	0.00 ± 0.00	10	3.68
		NVI	0.00 ± 0.00	0.00 ± 0.00	10	5.00
		MAQRM	0.44 ± 0.01	0.55 ± 0.02	10	4.00

Table 2: Results for all specifications in Single Lane Environment. Total of 10 runs per benchmark. Timeout = 24 hrs.

D.3 Gridworld

The environment is a 4×4 discrete grid with 2-agents. The agents are initially placed at opposite corners of the grid. In every step, each agent can either move in one of four cardinal directions (with a failure probability) or remain in position. The task of each agent is to visit a series of locations on the grid while ensuring no collision between the agents. The agents must learn to coordinate between themselves to accomplish their tasks. As an example, each agent's task is to visit any one of the other two corners in the grid. In this case, the agents must learn to choose and navigate to different corners to minimize their risk of collision. This specification can be increased in length (thus, in complexity) by sequencing visits to more locations on the grid. All scenarios are cooperative. The specifications are described below and results are in Table 3.

Specifications

ϕ^1 Swap the positions of both agents without collision.
ϕ^2 Both agents should choose to visit one of the other two corners of the grid without collision.
ϕ^3 Append ϕ^3 with the specification to reach the corner that is diagonally opposite the initial position of the agent without collision.
ϕ^4 Append ϕ^4 with the agents swapping their positions without collision.
ϕ^5 Append ϕ^5 with the agents swapping their positions again without collision.

Spec.	Num. of agents	Algorithm	welfare(π) (avg ± std)	$\epsilon_{\min}(\pi)$ (avg ± std)	Num. of runs terminated	Avg. num. of sample steps (in millions)
ϕ^1	2	HighNashSearch	0.95 ± 0.02	0.00 ± 0.00	10	18.86
		nvi	1.00 ± 0.00	0.00 ± 0.00	10	22.40
		maqrm	Timeout	Timeout	10	Timeout
ϕ^2	2	HighNashSearch	0.99 ± 0.01	0.00 ± 0.00	10	29.74
		nvi	1.00 ± 0.00	0.00 ± 0.00	8	38.40
		maqrm	Timeout	Timeout	10	Timeout
ϕ^3	2	HighNashSearch	0.98 ± 0.02	0.00 ± 0.00	10	48.40
		nvi	1.00 ± 0.00	0.00 ± 0.00	4	57.60
		maqrm	Timeout	Timeout	10	Timeout
ϕ^4	2	HighNashSearch	0.94 ± 0.02	0.00 ± 0.00	10	65.91
		nvi	0.94 ± 0.01	0.00 ± 0.00	7	92.80
		maqrm	Timeout	Timeout	10	Timeout
ϕ^5	2	HighNashSearch	0.86 ± 0.05	0.00 ± 0.00	10	87.05
		nvi	0.13 ± 0.01	0.00 ± 0.00	10	128.00
		maqrm	Timeout	Timeout	10	Timeout

Table 3: Results for all specifications in Gridworld Environment. Total of 10 runs per benchmark. Timeout = 24 hrs.

E Experimental Details

Hyperparameters. In our implementation of HighNashSearch, we used Q-learning with ε-greedy exploration to learn edge policies with $\varepsilon = 0.15$, learning rate of 0.1 and discount factor $\gamma = 0.9$. In the verification phase, we used Nash factor $\epsilon = 0.06$ and precision value $\delta = 0.01$. In our verification algorithm, the failure probability p is only used within BFS-Estimate to compute the number of samples K to collect for each state-action pair. In the experiments, we directly set the value of K to 1000.

Platform. Our implementation of HighNashSearch and the baselines is in Python3. All experiments were run on a 80-core machine with processor speed 1.2GHz and Ubuntu 18.04.