Auton Agent Multi-Agent Syst (2025) 39:35
https://doi.org/10.1007/s10458-025-09716-6

POSGGym: a library for decision-theoretic planning and learning in partially observable, multi-agent environments

Jonathon Schwartz[1] · **Rhys Newbury**[2] · **Dana Kulić**[2] · **Hanna Kurniawati**[1]

Accepted: 19 June 2025 / Published online: 21 July 2025

Abstract
Seamless integration of Planning Under Uncertainty and Reinforcement Learning (RL) promises to bring the best of both model-driven and data-driven worlds to multi-agent decision-making, resulting in an approach with assurances on performance that scales well to more complex problems. Despite this potential, progress in developing such methods has been hindered by the lack of adequate evaluation and simulation platforms. Researchers have had to rely on creating custom environments, which reduces efficiency and makes comparing new methods difficult. In this paper, we introduce POSGGym : a library for facilitating planning and RL research in partially observable, multi-agent domains. It provides a diverse collection of discrete and continuous environments, complete with their dynamics models and a reference set of policies that can be used to evaluate generalization to novel co-players. Leveraging POSGGym, we empirically investigate existing state-of-the-art planning methods and a method that combines planning and RL in the type-based reasoning setting. Our experiments corroborate that combining planning and RL can yield superior performance compared to planning or RL alone, given the model of the environment and other agents is correct. However, our particular setup also reveals that this integrated approach could result in worse performance when the model of other agents is incorrect. Our findings indicate the benefit of integrating planning and RL in partially observable, multi-agent domains, while serving to highlight several important directions for future research. Code available at: https://github.com/RDLLab/posggym.

Keywords Multi-agent · Planning under uncertainty · Reinforcement learning · Software

1 Introduction

Decision-theoretic planning, also known as *planning under uncertainty*, addresses the problem of using a dynamics model to find the optimal way to behave in uncertain environments [1, 2]. In scenarios involving a single-agent, this is formalized by the Partially Observable Markov Decision Process (POMDP) [3, 4], which incorporates uncertainty in the state of the environment and the stochastic outcomes of actions. Extending to the multi-

Extended author information available on the last page of the article

agent setting, various approaches exist [5], with the Partially Observable Stochastic Game (POSG) [6] being one of the most general. The appeal of decision-theoretic planning lies in its mathematical rigor and theoretical guarantees for optimal decision-making under uncertainty, vital for reliable autonomous systems in domains like robotics [7, 8], autonomous driving [9], and security [10, 11].

Unfortunately, applying planning under uncertainty to large, multi-agent environments has remained a challenge. Existing methods have been consistently limited by the high computational complexity of planning in partially observable multi-agent environments, primarily due to the exponential growth in problem size with planning horizon [5, 12]. Finding novel ways to improve the scaling of planning is essential if it is to be useful for many of the real-world problems we care about solving.

Advances in deep learning [13] offer promising solutions for improving the scaling of planning. Deep learning has been able to effectively leverage growing compute in order to solve increasingly more difficult problems [14]. Combining deep reinforcement learning (RL) with planning, in the form of search, has already led to remarkable achievements in multi-agent decision making in games such as Go, Chess, Shogi, Poker, and Hanabi [15–19]. Integrating learning with planning presents an exciting opportunity for the creation of autonomous agents capable of scaling to complex environments while robustly handling uncertainty.

This work aims to contribute to research at the intersection of learning and planning. Two key ingredients are essential for promoting progress in this direction. Firstly, high-quality benchmark domains are necessary for facilitating comparisons, reducing experiment times, and guiding research direction. Many recent achievements in multi-agent research have been driven by access to such benchmarks [15, 17, 20–23]. Secondly, clear baselines of the state-of-the-field are vital for understanding current limitations. While adjacent areas like multi-agent RL (MARL) and games have had substantial contributions to these components [24–28], multi-agent planning has received comparatively less attention.

In this paper, we introduce *POSGGym* (Fig. 1), a research library for planning and RL in partially observable, multi-agent environments. POSGGym models environments using the POSG framework and supports both discrete and continuous domains. It includes implementations of established and newer planning benchmarks, all under a unified API that is compatible with the existing ecosystem of MARL libraries [24, 29–32]. Additionally, POSGGym provides a reference set of policies, a crucial ingredient for multi-agent

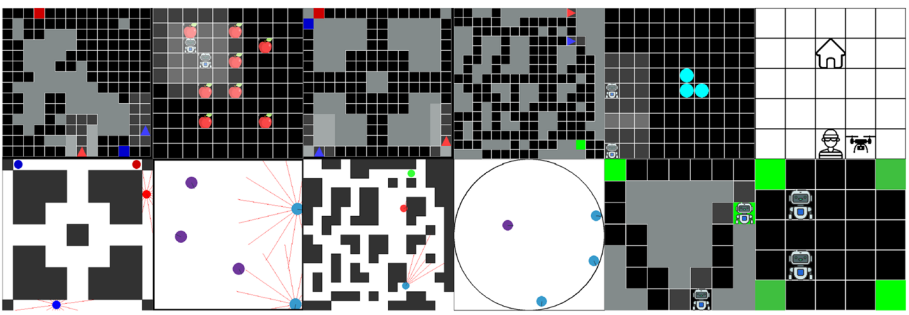


Fig. 1 POSGGym is a library for planning and RL research in partially observable, multi-agent domains. It includes a diverse set of discrete and continuous environments along with a collection of reference policies for reproducible evaluation

research. These policies and benchmark environments can help save researcher time and enable reproducible evaluations.

POSGGym enables extensive empirical investigation into current state-of-the-art decision-theoretic planning methods and their combination with RL. We evaluate four of the current state-of-the-art planners and combined planning with RL across diverse environments and populations of other agents. Our evaluation encompasses performance against both known and unknown co-player populations, allowing us to study various properties of each method including generalization to novel partners.

The key contributions of this paper are:

1. We introduce *POSGGym* , a new environment and agent library for planning and RL research in partially observable, multi-agent domains.
2. We use POSGGym to compare the performance of existing planning and RL methods across a range of environments and when paired with known and unknown partners.
3. Demonstrate how decision-theoretic planning and RL can be combined effectively to get better performance than either method alone.
4. Study and discuss some of the failure modes for existing planning and combined planning plus RL approaches, and propose future research directions.

2 Background

2.1 Partially observable stochastic games

Partially Observable Stochastic Games (POSG) are a foundational mathematical framework for modelling scenarios involving multiple agents interacting within a stochastic, partially observable environment. They generalize various other formal decision-making models. A Partially Observable Markov Decision Process (POMDP) is a POSG with a single agent [3, 4], while a decentralized POMDP (Dec-POMDP) [12] is a fully cooperative POSG where all agents share a reward function. A multi-agent Markov decision processes (MMDP) [33] is a fully cooperative, fully observable POSG. While a Markov game (MG), also known as a stochastic game, is a fully observable POSG [34]. The generality of POSGs has lead to their wide use in decision-theoretic planning and MARL.

Formally, a POSG is a tuple $\mathcal{M} = \langle \mathcal{I}, \mathcal{S}, S_0, \vec{\mathcal{A}}, \vec{\mathcal{O}}, \mathcal{T}, \mathcal{Z}, \mathcal{R} \rangle$ consisting of N agents indexed $\mathcal{I} = \{1, \dots, N\}$, a set of Markov states of the environment $\mathcal{S}$, an initial state distribution $S_0 \in \Delta(\mathcal{S})$[1], the joint action space $\vec{\mathcal{A}} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_N$, the joint observation space $\vec{\mathcal{O}} = \mathcal{O}_1 \times \cdots \times \mathcal{O}_N$, a state transition function $\mathcal{T} : \mathcal{S} \times \vec{\mathcal{A}} \times \mathcal{S} \to [0, 1]$ specifying the probability of transitioning to state s' given joint action $\vec{a}$ was performed in state s, the joint observation function $\mathcal{Z} : \mathcal{S} \times \vec{\mathcal{A}} \times \vec{\mathcal{O}} \to [0, 1]$ specifying the probability of joint observation $\vec{o}$ after performing joint action $\vec{a}$ and ending up in state s', and a reward function $\mathcal{R} : \mathcal{S} \times \vec{\mathcal{A}} \to \mathbb{R}^N$ defining the reward each agent receives when joint action $\vec{a}$ is performed in state s. Combining $\mathcal{T}, \mathcal{Z}, \mathcal{R}$ into a single function produces a generative model $\mathcal{G}$ which returns the next state, joint observation, and joint reward, given the current state and joint action $\langle s', \vec{o}, \vec{r} \rangle \sim \mathcal{G}(s, \vec{a})$.

[1] We use S_0 to denote the initial state distribution to distinguish it from the initial belief of an agent b_0 which in the multi-agent setting can be a distribution over more than just environment states.

At each step, each agent $i \in \mathcal{I}$ simultaneously performs an action $a_i \in \mathcal{A}_i$ and receives an observation $o_i \in \mathcal{O}_i$ and reward $r_i \in \mathbb{R}$. Each agent has no direct access to the environment state or knowledge of the other agent's actions and observations. Instead they must rely only on information in their *action-observation history* up to the current time step t: $h_{i,t} = \langle o_{i,0} a_{i,0} o_{i,1} a_{i,1} \dots o_{i,t-1} a_{i,t-1} o_{i,t} \rangle$. The set of all time t histories for agent i is denoted $\mathcal{H}_{i,t}$. Agents select their next action using their *policy* π_i which is a mapping from their history h_i(or belief, see Section 2.2) to a probability distribution over their actions, where $\pi_i(a_i|h_i)$ denotes the probability of agent i performing action a_i given history h_i. The goal of each agent i is to maximize its total expected *return* given by $J_i = \mathbb{E}\left[\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{i,t'}\right]$, where $\gamma \in [0, 1)$ is a discount factor. Importantly, each agent's reward at each step depends on the actions taken by the other agents present in the environment.

2.2 Decision-theoretic planning

Decision-theoretic planning is a broad term, in this paper we consider it as the field of research and methods concerned with finding the optimal way to behave in uncertain environments by explicitly modeling the uncertainty arising from partial observability and stochastic outcomes of actions [1, 2, 4]. For this reason it is often also referred to as *planning under uncertainty*. In the multi-agent setting, uncertainty about the other agent – their policy, actions, and internal state – must also be considered [5, 35]. The various methods differ primarily along the axis of whether they aim to find a policy for all agents in the environment [36] (e.g. a team of robots), or instead focus on controlling a single self-interested agent [37] (e.g. a robot that must operate around humans). The POSG framework, and thus POSGGym, accommodates research in both directions.

In this work, we concentrate on the problem of operating a self-interested agent in an environment shared with independent co-players. In cooperative environments this is analogous to the problem of *ad-hoc teamwork* [38, 39]. Throughout this paper, we use i to denote the planning agent, $j \neq i$ to denote a single other agent, and $-i = \mathcal{I} \setminus i$ to denote the set of all other agents.

Central to planning is the use of dynamics models of the environment. For this paper, we assume the agent has access to a generative model $\mathcal{G}$ of the underlying POSG, which it can use to sample next states, observations, and rewards. $\mathcal{G}$ is stochastic in nature, arising from the uncertainty in the environment dynamics, where multiple possible outcomes may be associated with each action, observation, and state transition. This is compared to having access to the full model which provides the full transition and observation probabilities and generally can be impractical to implement for environments with large state and observation spaces. Additionally, due to partial observability, the agent does not have direct access to the current state of the environment. Instead, it needs to maintain a belief state, representing a probability distribution over possible states based on past observations and actions.

2.2.1 Beliefs

At the core of planning under uncertainty lies an agent's *belief*, representing a distribution over possible states the agent could currently be in. Importantly, what constitutes a state varies depending on the context. In single-agent POMDPs, the belief b is a distribution over environment states $b \in \Delta(\mathcal{S})$ and encapsulates uncertainty in the physical state of the

environment $s \in \mathcal{S}$. In the multi-agent setting, the policies and internal states of the other agents must also be considered. Multi-agent frameworks differ based on how they represent the other agents' policies and internal states. For instance, the Interactive-POMDP (I-POMDP) [37] adopts a recursive reasoning approach, modeling other agents' policies as solutions to lower-level I-POMDPs and their internal state as lower level beliefs over environment states and possibly the beliefs of the other agents. More generally, it suffices to consider the space of possible policies $\pi_{-i} \in \Pi_{-i}$ and action-observation histories $h_{-i,t} \in \mathcal{H}_{-i,t}$ of the other agents [6, 37], where the space of policies could be a discrete set of policies [6, 37] or a distribution over continuous parameters [40]. For the purpose of this work, a belief $b_{i,t} \in \Delta(\mathcal{S} \times \Pi_{-i} \times \mathcal{H}_{-i,t})$ encompasses the environment state $s \in S$, other agents' policies $\pi_{-i} \in \Pi_{-i}$, and their histories $h_{-i,t} \in \mathcal{H}_{-i,t}$, where the other agents' policy space is a discrete set.

The planning agent uses its belief $b_{i,t}$ to select its next action according to its policy π_i and then updates its belief given the next observation from the environment. The initial belief $b_{i,0}$ is formed based on the initial environment state distribution and prior ρ over other agents' policies: $b_{i,0}(s, \pi_{-i}, h_{-i,0}) = S_0(s)\rho(\pi_{-i})$. In this work ρ is a uniform distribution, unless otherwise stated. Subsequent beliefs are computed based on the agent's most recent action $a_{i,t}$ and observation $o_{i,t}$ using the well-known Bayes filter: $b_{i,t+1}(s_{t+1}, \pi_{-i,t+1}, h_{-i,t+1}) = Pr(s_{t+1}, \pi_{-i,t+1}, h_{-i,t+1} | b_{i,t}, a_{i,t}, o_{i,t})$. Exact representation and computation of beliefs are intractable for all but small problems due to the curse of state dimensionality. Thus, modern planners rely on approximate methods.

As an example of how beliefs can be used to adapt online to the other agent, in a driving scenario, if agent i observes the other agent $-i$ barely slowing down when entering an intersection, this observation provides information about which type of policy π_{-i} they are using. If the planning agent's belief includes both an 'aggressive' policy, that tends to go fast through intersections, and a 'cautious' policy, that slows down before entering an intersection, observing the other agent speeding through an intersection would increase the probability assigned to the aggressive policy. The planning agent can then adapt its future actions accordingly, for example by slowing down to maintain a larger distance to the aggressive agent's vehicle.

2.2.2 Monte Carlo planning

Monte Carlo planning using Monte Carlo Tree Search (MCTS) [41, 42] and particle filters [43] is the dominant paradigm for planning in large problems. Rather than maintaining exact beliefs, beliefs are approximated using a particle filter with Monte Carlo updates. To compute the policy for the current belief, MCTS constructs a search tree of nodes online, where each node is a belief, effectively searching over the space of beliefs.

MCTS involves estimating the value of each action from the current belief via a series of simulated episodes. Each simulation starts from a state sampled from the current belief and proceeds in three stages: selection, expansion, and backpropagation.

1. **Selection:** the current search tree is traversed using an *exploration policy* until a leaf node of the tree is reached. The exploration policy balances exploring new regions of the search space with exploiting known high value regions. The two commonly used exploration policies are UCB (Upper Confidence Bound) [44], and PUCB ("Predictor"+

UCB) [45]. UCB selects actions as a function of the action's current estimated value and number of times it has been previously selected, while PUCB extends this to include an additional bias based on a *search policy*.

2. **Expansion:** upon reaching a leaf node, it is evaluated and expanded by adding it to the tree and adding edges for each action. Evaluation involves estimating the node's value. This is typically done through Monte Carlo rollouts or using pre-computed value functions.
3. **Backpropagation**: the nodes and edges visited along the simulated trajectory are updated by propagating the value estimate from the leaf node back-up to the root node of the tree. This includes incrementing the visit count for each node, for use by the exploration policy.

Upon completion of the search, the planning agent selects the action from the root node based on visits and values, typically by choosing the action with the largest value, most visits, or sampling from a distribution computed from these values. Following the action being performed in the environment and the planning agent receiving its next observation, the current belief of the agent is updated and set as the new root node of the tree. The search process is then repeated.

2.3 Reinforcement learning

Reinforcement Learning (RL) [46] is another approach for solving sequential-decision making problems. Whereas planning uses a dynamics model and explicit beliefs, RL focuses on learning a policy through interactions with the environment or a model. Deep RL in partially observable environments typically circumvent explicit beliefs by using recurrent networks (RNNs) [47] such as LSTMs [48] to represent the policy [49]. These RNNs can learn implicit beliefs, in that they learn representations of the sufficient statistics of the state of the world given the agents action-observation history. However, these learned implicit beliefs do not permit planning since search in planning requires explicitly sampling from beliefs in order to simulate possible future trajectories. Instead, prior work has combined RL with search by using a policy trained with deep RL as the *search policy* within MCTS using PUCB [15–17, 19].

An important question when applying RL in multi-agent environments is how to model the other agents in the environment? This question has given rise to a range of multi-agent training schemes being developed. One of the most well known is *self-play* [50] where the RL agent plays against itself, using the same policy for all agents in the environment. This has proven to be very effective, especially for two-player, zero-sum games [15, 16, 51, 52], or settings where there is centralized learning for decentralized execution [19]. While these results are very impressive, there is evidence that purely self-play agents can be prone to over-fitting, resulting in exploitable agents [53, 54].

Another approach is *population-based training* where there are multiple independent policies that are trained simultaneously and paired up according to some schema [53, 55, 56]. This type of training has been used to generate superhuman performance in large, complex, partially observable environments like StarCraft 2 [21] and Dota [20]. Over the years various schemas have been proposed including those based on cognitive hierarchies and nested reasoning [53, 57], playing against older versions of a policy [20, 50].

Finally, other methods instead use a fixed model of the opponent, typically either learned from data or handcrafted [58]. A good example of this is to generate a model of human behaviour from data and then use RL to learn a policy that acts well with respect to this model. This technique was crucial for generating the initial policies for superhuman agents in competitive domains Go [15] and Starcraft 2 [21]. More, recently it has been used to produce human level agents when playing against humans in the game of Diplomacy [59], which requires both cooperation and competition. This is the approach we use in this work, where the RL agent has access to a set of possible policies for the other agent during training, and tasked with learning a best-response (BR) to this set.

3 Related work

3.1 Multi-agent libraries

In recent years there has been a proliferation of MARL research libraries. This includes general suites such as PettingZoo [24], Melting Pot [25] and JaxMARL [60] which provide a standard API and a large collection of environments. Other libraries instead focus on specific domains such as StarCraft [26, 27], massively multi-agent online games [61], environments with hundreds to millions of agents [62], drones [63], autonomous driving [64, 65] and robotics [66–68]. While all the aforementioned libraries offer unique challenges they are designed specifically with deep MARL in mind and so have no or limited model support and/or focus on limited domains, making them hard to use for research in planning under uncertainty. In contrast, POSGGym explicitly decouples the model and environment in its API, making it easier for researchers to use the environments for planning and/or MARL. This decoupling also enables experiments evaluating the impact of model inaccuracy, by allowing researchers to easily use models that are different from the environment.

A number of libraries focusing on turn-based games have also been developed, including OpenSpiel [28], rlcard for card-games [69], and Pgx for accelerator-supported board games [70]. Unlike most MARL libraries, these libraries support search by exposing the environment model in their APIs. However, each library is based on the Extensive Form Games formalism [71] with a focus on turn-based games, such as classic card and board games.

Finally, there have been a couple of libraries designed for multi-agent planning. The Multi-Agent Decision Process Toolbox (MADPToolbox) [72] focuses on facilitating planning research and includes a collection of planning algorithms and benchmarks domains. However, the primary focus of MADPToolbox has been on planning algorithms for discrete Dec-POMDPs. Additionally, the design of the library makes it difficult to integrate with the modern ecosystem of deep learning and RL libraries. AdLeap-MAS [73] is a more recent library which focuses on ad-hoc reasoning, however it includes a very limited set of environments.

3.2 Multi-agent planning

In this work we empirically evaluate existing state-of-the-art methods for planning under uncertainty in large partially observable environments where the task is to control a single

self-interested agent. A closely related problem is that of *ad-hoc teamwork* [39] in cooperative environments where the goal is to design an agent that can adapt to novel teammates. Proposed methods in this area include those based on stage games [74], Bayesian beliefs [75, 76], types with parameters [40], and for the many agent setting [77]. All these methods use MCTS but are limited to environments where the state and actions of the other agents are fully observed, so are not applicable in our settings. For the the more general setting of type-based reasoning, [78] propose *POTMMCP* that incorporates a meta-policy for guiding search. POTMMCP was shown to outperform related methods across a range of cooperative, competitive, and mixed environments. A number of other works have focused on planning in more restricted settings. This includes strictly cooperative [79, 80] and competitive [81] environments, and settings where there is centralized control [82].

Several Monte Carlo planning methods have been proposed for solving large I-POMDPs. This includes methods based on finite-state automata [83] and for systems with communication [84]. However, most relevant for our setting are **IPOMCP** [85], which extends the single-agent POMCP [42] to solving I-POMDPS, and **INTMCP** [86] which uses nested-MCTS. These two methods represent the current state-of-the-art planning methods when it comes to solving large, general I-POMDPs.

3.3 Combined planning and learning

Combining search with RL has been an important part of superhuman performance in games. Self-play RL and MCTS have been combined to achieve beyond expert performance in two-player fully-observable zero-sum games with both a known [15, 16] and learned [87] environment model. Similar methods have been applied to zero-sum imperfect-information games [17, 18, 52, 88], as well as cooperative games where there is prior coordination for decentralized execution [19, 89]. Methods combining MCTS and RL in games for training a best-response policy against a distribution over policies [90] have also been proposed.

While combining RL with planning has shown success in fully observable settings [15, 16]) and partially observable settings with exact [17, 19, 52] or learned belief representations [88, 89], to the best of the authors' knowledge our work provides the first comprehensive evaluation of combining RL with particle-based planning methods. This distinction is important as particle-based methods are currently a widely used approach for planning in large partially observable environments where exact belief computation is intractable. Our results reveal both the potential benefits and limitations of this combination, particularly when dealing with model inaccuracy.

4 POSGGym

POSGGym has been developed with the following goals in mind:

- Provide a general API for environments, models, and reference agents that supports both planning and RL
- Provide implementations of a range of established and newer planning benchmark environments
- Provide a diverse set of co-player policies (reference agents) for implemented environ-

ments
- Be similar to Gym and PettingZoo APIs and compatible with the main RL algorithm libraries

The aim of POSGGym is to streamline planning and learning research in POSGs, with a particular emphasis on planning, since this is currently lacking in existing research libraries. To accomplish this, POSGGym uses a general yet user-friendly API, and includes a diverse collection of environments and reference agents. POSGGym 's API is based on those of the Gym [91, 92] and PettingZoo [24] libraries, as these are widely used by the RL community. It has the additional benefit of making it simple to integrate POSGGym with the many MARL algorithm libraries that are currently compatible with PettingZoo, e.g. [29–31].

4.1 API design

The POSGGym API has three main components: *environment*, *model*, and *agents* (Fig. 2).

4.1.1 Environment API

The POSGGym environment API, depicted in Fig. 3a, closely follows the structure of PettingZoo's parallel environment API (Fig. 3b), however, with certain aspects aligning more closely with the Gymnasium API (Fig. 3c).

At each timestep, each agent provides an `action`, which collectively form a joint action passed to the `step` function. This function updates the state of the environment and returns `observations`, `rewards`, `terminations`, `truncations`, `all_done`, `infos`. Each of these return values (except `all_done`) is a mapping from the ID of the agent to their respective return value. The `step` function mirrors PettingZoo's step function, with the addition of `all_done`, which indicates when all agents in the environment have reached a terminal state. Similarly, the `reset` method mirrors PettingZoo's and resets the environment to a starting state and returns an `observation` and `info` for each active agent. The state, observation, and action spaces utilize the same underlying classes as Gymnasium and PettingZoo, greatly simplifying compatibility across libraries. The `render` and `close` functions operate identically to Gymnasium: `render` provide visual representation

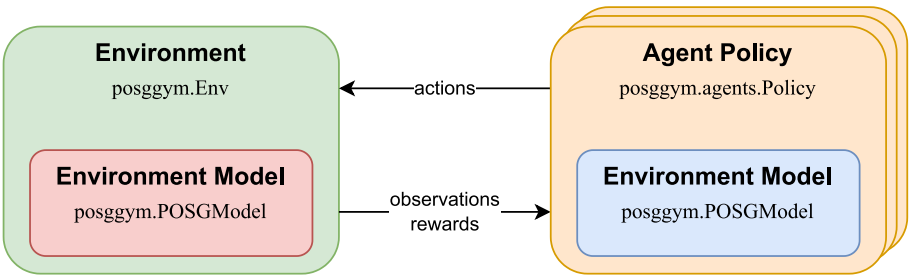


Fig. 2 POSGGym 's high-level architecture includes environment, model, and agent policy APIs. Agents interact with the environment by selecting actions according to their policy. The environment maintains a state and uses a model internally to update this state and generate observations and rewards for the agents. Each agent policy has access to its own model for planning. Researchers have full control over which model the agent has access to, so the agent's model may be the same or different to the environment's model

```python
import posggym

env = posggym.make("PursuitEvasion-v0", render_mode="human")
observations, infos = env.reset(seed=42)

for _ in range(1000):
    actions = {i: policies[i](observations[i]) for i in env.agents}
    observations, rewards, terminations, truncations, all_done, infos = env.step(actions)
    if all_done:
        observations, infos = env.reset()

env.close()
```

(a) POSGGym

```python
from pettingzoo.butterfly import pistonball_v6

env = pistonball_v6.parallel_env(render_mode="human")
observations = env.reset(seed=42)

for _ in range(1000):
    actions = {i: policies[i](observations[i]) for i in env.agents}
    observations, rewards, terminations, truncations, infos = env.step(actions)
    if not env.agents:
        observations = env.reset()

env.close()
```

(b) PettingZoo

```python
import gymnasium as gym

env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)

for _ in range(1000):
    action = policy(observation)
    observation, reward, terminated, truncated, info = env.step(action)
    if terminated or truncated:
        observation, info = env.reset()

env.close()
```

(c) Gymnasium

Fig. 3 Environment APIs

of the current state of the environment, while `close` closes the environment and performs any necessary clean-up.

POSGGym 's environment API differs from the PettingZoo parallel environment API in two key aspects: the `make` function for environment initialization, and the inclusion of `all_done` in the `step` method's return values. The use of the `make` function aligns more closely with the design of Gymnasium, offering users more convenience and control. For example, in Fig. 3 in the PettingZoo API the user has to explicitly import the `pistonball_v6` module, whereas in POSGGym and Gymnasium the `make` function only requires the name of the environment. This makes it more convenient to use the exact same code for different environments, since there is no need to explicitly change which code is imported. The inclusion of `all_done` differs from both PettingZoo and Gymnasium. This addition aims to simplify the tracking of agent termination during an episode, which can be non-trivial in open environments where the agents that are active may change over time. It also accounts for scenarios where agents can leave or join the environment within a single episode. For example, without the `all_done` output, the user has to track which agents are

done and then check at each step whether all agents are finished before ending the episode. POSGGym instead puts the burden of handling this logic on the environment developer where it can be written once and then shared by all users.

To make integration with existing MARL libraries easier, POSGGym provides a `PettingZoo` wrapper class that enables the conversion of any POSGGym environment into an equivalent PettingZoo parallel API environment. By using this wrapper, any POSGGym environment can be used seamlessly with any library that supports the PettingZoo API.

4.1.2 Model API

POSGGym 's model API serves as the distinguishing feature between POSGGym and existing environment libraries. The model API, shown in Fig. 4, provides access to a generative model of the environment, which can be utilized during planning. The design of the model API aims to closely resemble the environment API, with the exception that the majority of methods take an environment `state` as an additional input. The main methods are as follows:

- `sample_initial_state` – samples an initial environment state
- `sample_initial_obs` – samples initial observations for each agent given a state
- `get_agents` – returns the IDs of agents that are active in a given state
- `step` – similar to the environment `step` function, but also returns the next state. It takes both a state and joint actions as arguments
- `seed` – sets the random seed for the model

Models in POSGGym are stateless, except for their random seed. This is the key difference when comparing the model and environment APIs. Models have no internal state and thus require a state as input to perform a step, while the environment maintains an internal state which it evolves with each step. In fact, the default implementation of the Environment API

```
import posggym

env = posggym.make("PredatorPrey-v0")
model = env.model
model.seed(seed=42)

state = model.sample_initial_state()
observations = model.sample_initial_obs(state)

for t in range(50):
    actions = {i: policies[i].step(observations[i]) for i in model.get_agents(state)}
    timestep = model.step(state, actions)

    # timestep attribute can be accessed individually:
    state = timestep.state
    observations = timestep.observations

    # Or unpacked fully
    # state, observations, rewards, terminations, truncations, all_done, infos = timestep

    if timestep.all_done:
        state = model.sample_initial_state()
        observations = model.sample_initial_obs(state)
```

Fig. 4 POSGGym Model API

in POSGGym is merely a wrapper that manages the state on top of an underlying model class.

Separating out the underlying model from the environment is important for a number of reasons. Firstly, planning algorithms require a model. Having a clear separation in the API makes it clear what can be used for planning (model API) and what is the real environment outside of the planning agent's control (environment API). Secondly, it provides greater flexibility in the types of research that can be done. For example, to study planning with inaccurate models one can provide agents with models that are different from the environment, say with varying initial parameters or simplified dynamics (we cover this in greater detail in Section 4.4).

In addition to the generative model functionality described above, POSGGym defines the `POSGGym.POSGFullModel` API, which extends the `POSGGym.POSGModel` class to include all components of the formal POSG definition. This extension allows POSGGym to be used for defining models for algorithms that require the full model, rather than just a generative model. Specifically, the `POSGGym.POSGFullModel` includes the following additional methods:

- `get_initial_belief` – returns the initial state distribution S_0
- `transition_fn` – defines the state transition function $\mathcal{T} : \mathcal{S} \times \vec{\mathcal{A}} \times \mathcal{S} \to [0, 1]$
- `observation_fn` – defines the joint observation function $\mathcal{Z} : \mathcal{S} \times \vec{\mathcal{A}} \times \vec{\mathcal{O}} \to [0, 1]$
- `reward_fn` – defines the joint reward function $\mathcal{R} : \mathcal{S} \times \vec{\mathcal{A}} \to \mathbb{R}^N$

The full model definition is not included in the main `POSGGym.POSGModel` model class due to the difficultly of implementing full models in environments with very large state, action, or observation spaces and complex dynamics. In such cases, it is typically more practical and common to define a generative model that can be used for sample-based planning approaches like MCTS [42, 93].

4.1.3 Agent API

POSGGym includes a collection of *reference policies* for many of its environments. Access to these policies is done through the Agent API.

In this context it is important to distinguish between an agent and a policy. At a high-level, a reference policy π_i represents a fixed policy that maps from an agent's action-observation history to its actions. Within each environment, there exists a predefined number of agents N who interact within the environment. For each episode, each agent $i \in 1, \ldots, N$ has an associated policy π_i. In certain environments, such as symmetric environments where all agents are equivalent, it is possible for multiple agents to use the same policy. For example, in the game *Rock, Scissors, Paper* both agents could use the same policy of always playing"rock".

The goal of the Agent API is to offer a diverse collection of high-quality reference policies that can be leveraged for research, including for testing, training, and evaluation. As we demonstrate in Section 5, the policies can be used to measure generalization by holding-out the policies for evaluation only, with no use of the policies during RL training or within the planning algorithm (for other examples see [20, 21, 25, 55]). Alternatively, the policies can be used during planning, for example by using the policies to guide search [78]. Providing

a set of high-quality reference policies enables many possibilities for research, helps boost research efficiency, and makes it easier to perform standardized comparisons. This is particularly valuable in complex, partially-observable environments where generating policies can be especially challenging, requiring time, knowledge, and often access to substantial computing resources.

POSGGym's Agents API, depicted in Fig. 5, follows a similar design to the Environment API. Its key methods include `make`, `step`, and `reset`. The `make` function initializes a new instance of a policy, from the policy's unique ID, the environment model, and the ID of the agent the policy will be used for. It returns an instance of the `POSGGym.agents.Policy` class, the main Agent API class, which has two main methods: `reset` and `step`. `reset` sets the policy to its initial state, and optionally configures the random seed via the `seed` argument. `step` updates the policy with the latest observations for the policy's agent and returns the agent's next action.

When tackling POSGs, a critical consideration is the requirement for policies to maintain an internal state to handle the partial observability of the environment. As discussed in Sections 2.2.1 and 2.3, approaches vary: some utilize explicit beliefs where agents maintain and update a probabilistic representation of the unobservable features of the environment, while others rely on implicit beliefs that leverage learned representations or neural network architectures to capture relevant information from observations. Incorporating these crucial elements into decision-making allows policies to exhibit more sophisticated and adaptive behaviors within complex environments. To provide support for flexible internal states, the `POSGGym.agents.Policy` class API also includes a number of additional methods that provide information and finer-grained control over the policy. These methods include:

- `get_initial_state` - returns the initial state of the policy. For example, the initial belief or initial hidden RNN state.
- `get_next_state` - returns the next policy state, given the current policy state and

```python
import posggym
import posggym.agents as pga

env = posggym.make("PursuitEvasion-v0", grid="16x16")
policies = {
    "0": pga.make("PursuitEvasion-v0/grid=16x16/klr_k1_seed0_i0-v0", env.model, "0"),
    "1": pga.make("PursuitEvasion-v0/shortestpath-v0", env.model, "1"),
}

seed = 42
observations, infos = env.reset(seed=seed)
for policy in policies.values():
    seed += 1
    policy.reset(seed=seed)

for _ in range(1000):
    actions = {i: policies[i].step(observations[i]) for i in env.agents}
    observations, rewards, terminations, truncations, all_done, infos = env.step(actions)
    if all_done:
        observations, infos = env.reset()
        for policy in policies.values():
            policy.reset()

env.close()
for policy in policies.values():
    policy.close()
```

Fig. 5 POSGGym Agent API

the next observation.
- `sample_action` - sample an action given a policy state
- `get_pi` - get the distribution over actions given a policy state
- `set_state` - set the internal state of the policy
- `get_state` - get the internal state of the policy
- `get_state_from_history` - unrolls the policy to get its state given an action-observation history.

All together, the API provides enough control that the policy can be used for evaluation using the `step` method, or for planning using the finer-grained control methods like `get_next_state` and `sample_action`.

4.2 Environments

POSGGym currently offers a collection of 14 environments. These environments have been used in multi-agent research in various forms, with most existing in paper descriptions, across disparate programming languages, some in unmaintained research code, and others in MARL libraries with no model support. Table 1 shows the complete list of environments, along with some of their properties and the multi-agent concepts they involve. For detailed explanations of each environment, we refer the reader to POSGGym 's documentation[2]. Our intention is to expand the range of environments based on community demand, and actively encourage contributions from the community. To facilitate this, we have developed comprehensive documentation and tutorials to streamline the process of adding new environments.

POSGGym 's API is designed to support arbitrary numbers of agents, however the current version of the library focuses on environments with 2-10 agents. This focus is motivated by several factors. First, the few-agent setting presents distinct research challenges that differ from many-agent scenarios, particularly around agent modeling and belief updates. Agent modeling methods like recursive reasoning [35, 37] - which are crucial for many real-world applications - become intractable or require different approaches when scaled to hundreds or thousands of agents. Second, many practical applications, such as human-robot interaction [94], naturally involve a small number of agents. Finally, several high-quality libraries already exist for many-agent research, including MAgent [62], Neural MMO [61], and GigaStep [63]. POSGGym complements these libraries by providing comprehensive support for planning and learning research in the few-agent setting. Extending POSGGym to involve many-agent environments would simply require implementing such an environment using the general POSGGym API.

4.2.1 Classic

POSGGym includes several well-known problems that have been used extensively in planning and multi-agent research. These include Multi-Access Broadcast Channel (MABC) [6,

[2] Documentation available at: https://posggym.readthedocs.io/

Table 1 POSGGym environments, their properties, and the multi-agent concepts they involve. Related properties are grouped by row color

		Classic			Grid-World								Continuous			
		MABC	MA Tiger	RPS	CR	LBF	Two Paths	UAV	Driving	Predator Prey	Pursuit Evasion	Driving	Predator Prey	Pursuit Evasion	DTC
Properties	Cooperative	x			x	x				x			x		x
	Mixed		x			x			x	x		x	x		
	Competitive			x			x	x			x			x	
	Symmetric roles	x	x	x	x	x			x	x		x	x		x
	Asymmetric roles						x	x			x			x	
	Discrete Actions	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Continuous Actions											x	x	x	x
	Discrete Observations	x	x	x	x	x	x	x	x	x	x				
	Continuous Observations											x	x	x	x
	Pixel Observations					x	x	x	x	x	x				
Concepts	Temporal Coordination	x			x	x			x	x		x	x		x
	Spacial Coordination				x	x			x	x		x	x		x
	Reciprocity					x									
	Fair Resource Sharing					x				x		x	x		
	Deception				x		x	x			x			x	
	Convention following								x			x			
Concepts	Nested-Reasoning		x	x			x	x			x			x	

95–97], Multi-Agent Tiger [37, 83, 98–100], and Rock-Paper-Scissors. These problems encompass cooperative, mixed, and competitive scenarios, respectively, and support discrete actions and observations. Due to their smaller size and well-defined characteristics, some versions of these problems have known provably optimal solutions. This makes them useful for debugging and for fine-grained analysis of algorithms. POSGGym offers full model definitions for all the classic problems currently implemented in the library.

4.2.2 Grid-World

Seven widely used discrete grid-world problems are also provided by POSGGym. These problems encompass a variety of scenarios, including Cooperative Reaching (CR) [101, 102], Level Based Foraging (LBF) [68, 102–105], Two Paths [86], Unmanned Aerial Vehicle (UAV) [83, 100], Driving [78, 106, 107], Predatory Prey [78, 108–111], and Pursuit Evasion [78, 86, 112]. The current selection of problems was chosen to provide a diverse range of cooperative, mixed and competitive environments, as well as symmetric and asymmetric roles. Each environment is represented as a grid-world with discrete observations and actions.

4.2.3 Continuous

POSGGym also offers four 2D continuous problems. This includes adaptations of three grid-world environments: Driving, Predator Prey and Pursuit Evasion. For these environments the dynamics of the agents are modeled using a simple non-holonomic unicycle model, where agents are controlled by both angular and linear velocities. PyMunk [113] is employed as the physics engine to support these dynamics. Observations incorporate sensors that emit from an agents position in a circular pattern at a fixed distance. This approach aligns with the observation model used in PettingZoo's WaterWorld environment [114]. The fourth environment is the Drone Team Capture (DTC) environment [115, 116], which simulates a cooperative pursuit-evasion scenario. POSGGym 's implementation of DTC closely adheres to the original paper, with enhancements to accommodate partial observability, such as limited sight distance.

4.3 Reference agents

POSGGym currently offers reference agents for the majority of its environments. These agents encompass a combination of handcrafted heuristic policies and policies trained using various MARL algorithms. For instance, a number of grid-world and continuous environments include deep RL policies trained via self-play and K-Level Reasoning [57]. Additionally, handcrafted policies previously used in Level-Based Foraging [40, 104] and DTC [116–118] are included. For detailed information on the training methods for each policy, please refer to Appendix A. To access the comprehensive and up-to-date list of available policies, consult the library documentation.

4.4 Environment and model customization and control

POSGGym 's Environment and Model API's support customization via parameter-based modification of the underlying environment and model. Currently, all environments except for Rock-Paper-Scissors accept parameters which control various properties of the environment such as number of agents, observation distance, and environment layout. As developers of the library, we aimed to provide some reasonable default settings for each environment, but the API supports customization of these parameters to suit the user's requirements. The library documentation provides up-to-date information about the parameters available for each environment and what each of them does.

This flexibility enables researchers to test the generalization capabilities of planning and learning approaches under different environment configurations. For example, testing a planning method in larger and larger environments, or under different levels of observability. Additionally, by specifying different parameters for the true environment and the environment model that each policy has access to, POSGGym 's API enables users to evaluate the robustness of algorithms to model inaccuracy (Fig. 6).

4.5 Computational requirements and performance

POSGGym is designed to be lightweight, requiring only a single CPU for running environments at hundreds to thousands of steps per second (Table 2). In practice the performance bottleneck for experiments becomes policy computations of the planning or deep learning algorithm that is being tested. To help alleviate this in some cases, POSGGym provides utilities supporting vectorized execution of environments via the `posggym.vector.SyncVectorEnv` wrapper. This allows for batched computation which can greatly speed up training times when using deep learning based methods.

5 Experiments

We used POSGGym to empirically evaluate planning, RL, and combined planning plus RL methods across diverse environments. The goal of our experiments was to investigate current state-of-the-art planning under uncertainty methods and compare these with integrated

```
import posggym
import posggym.agents as pga

# true environment with blocks
true_env = posggym.make('PredatorPrey-v0', world="10x10Blocks")

# model of the environment without blocks
policy_model = posggym.make('PredatorPrey-v0', world="10x10").model

policies = {
    # Allow policy to interact with the policy model only
    "0": MyPolicy(model=policy_model, agent_id="0"),
    # Other agents can have correct (true_env.model) or incorrect (policy_model) model
    "1": pga.make("PredatorPrey-v0/H1-v0", model=policy_model, agent_id="1"),
}

observations, infos = true_env.reset()
for policy in policies.values():
    policy.reset()

for t in range(50):
    # agents can use policy model internally, but actually interact with true environment
    actions = {i: policies[i].step(observations[i]) for i in true_env.agents}
    observations, rewards, terminations, truncations, all_done, infos =
    true_env.step(actions)
    if all_done:
        observations, infos = true_env.reset()
        for policy in policies.values():
            policy.reset()

true_env.close()
```

Fig. 6 Model and environment customization in POSGGym. This example also shows how POSGGym can be used to evaluate a policy's robustness to model inaccuracy by giving the policy access to a model initialized using different parameters than the true environment

Table 2 Steps per second for different POSGGym environments running on a single Intel® Core™ i7-10750H CPU @ 2.60GHz with 16 GB RAM

Environment	Environment Type	Steps/s
MultiAccessBroadcastChannel-v0	Classic	59370.13
MultiAgentTiger-v0	Classic	63343.43
RockPaperScissors-v0	Classic	76395.78
Driving-v0	Grid-World	12128.70
DrivingGen-v0	Grid-World	1997.45
LevelBasedForaging-v2	Grid-World	20447.42
PredatorPrey-v0	Grid-World	9067.80
PursuitEvasion-v0	Grid-World	15427.85
TwoPaths-v0	Grid-World	27268.60
UAV-v0	Grid-World	43672.70
DrivingContinuous-v0	Continuous	551.53
DroneTeamCapture-v0	Continuous	1627.19
PredatorPreyContinuous-v0	Continuous	479.37
PursuitEvasionContinuous-v0	Continuous	481.14

RL and planning. Our experiments serve an additional purpose of providing baseline results and algorithm implementations for POSGGym to facilitate future research[3].

Some questions we hoped to answer in our experiments:

[3] Algorithm implementations and experiment code is available at: https://github.com/RDLLab/posggym-baselines

- How does the performance of planning and RL methods compare across various settings?
- How well does each method generalize when faced with novel partners?
- How effective is combining existing planning methods with RL?

5.1 Experiment setup

In our experiments we control a single planning agent in an environment with one other agent with unknown behavior. We focus on the Bayesian learning, also known as *type-based reasoning* [119], setting where the planning agent maintains a belief over a set of possible policies P, or "types", for the other agent. The goal of the planning agent is to choose actions that maximize its own total expected reward given its beliefs about the environment and the policy and internal state of the other agent. We make no assumptions about the reward structure of the environment and test across competitive, cooperative, and mixed incentive domains.

The high-level experimental design is shown in Fig. 7. For each environment there is a set of agent populations $\mathcal{P}$, where $P \in \mathcal{P}$ is a population containing a set of possible policies for the other agent. The planning agent is given access to the model of the environment as well as a known population of policies $P_{\text{plan}} \in \mathcal{P}$ which can be used for planning and training. Importantly, since our experiments focus on performance of the various methods when paired with an unknown other agent, the model of the environment is accurate and the same across populations, only the planning and test populations change. We assess each method against a separate population of policies $P_{\text{test}} \in \mathcal{P}$, which may be the same or different to the planning population. Experiments where the planning and test populations are the same, $P_{\text{plan}} = P_{\text{test}}$, are referred to as *in-distribution*. Conversely, experiments where $P_{\text{plan}} \neq P_{\text{test}}$ are referred to as *out-of-distribution*. By comparing in- and out-of-distribution performance, we are able to investigate each method's ability to generalize to novel co-players.

For each environment, we employ two distinct populations, denoted as P_0 and P_1 with $\mathcal{P} = \{P_0, P_1\}$. These are used for both the planning P_{plan} and test P_{test} populations. We conduct our experiments for each method by iterating over combinations of planning and

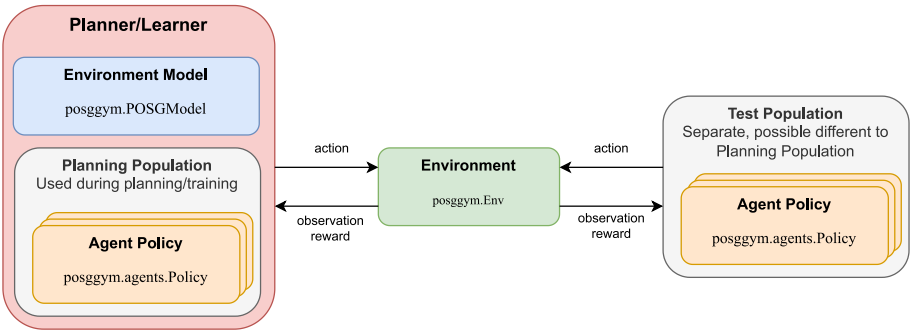


Fig. 7 High level experimental setup. During planning/training, each planning and RL method has access to the environment model and a planning population of policies for the other agent P_{plan}. Each method is then evaluated against a separate, possibly different population of policies P_{test}. In our experiments we have two populations P_0 and P_1 for each environment, each containing between five and six policies

test populations, $\langle P_{\text{plan}}, P_{\text{test}} \rangle \in \mathcal{P} \times \mathcal{P}$, resulting in four repetitions of each experiment per algorithm (each with a different $\langle P_{\text{plan}}, P_{\text{test}} \rangle$ combination). For every population $P \in \mathcal{P}$ we used a uniform distribution over policies as the prior, so the other agent had equal probability of using each policy in the set P.

A total of 10 to 12 policies were generated for the populations $\mathcal{P}$ in each environment. Depending on the environment the set was made up of heuristic, learned, or a mix of heuristic and learned policies. The 10 to 12 policies were divided roughly equally into the two populations P_0 and P_1 with each containing five to six policies. More details about the specific policies used in each population are provided in Appendix A.

5.2 Planning methods

We focus on planning methods designed for controlling a single agent within a general, multi-agent environment. This means we do not include a methods designed for environments with specific structures, e.g. with communication [84], centralized control [79, 80, 82] or full observability [77], although research in these areas is supported by POSGGym. Furthermore, since we are interested in studying planning in complex environments we restrict ourselves to methods that are capable of scaling to large environments with millions of states. In practice, this limits us to online planning approaches based on MCTS [41] as currently these have proven to be the most scalable, general methods. However, we highlight that POSGGym's general multi-agent API can be used for any planning method that is applicable to POSGs or any of its sub-classes, e.g. Dec-POMDP, POMDP, MG, etc.

For our experiments we compare the following planning methods:

- **Interactive Nested Tree Monte Carlo Planning** (INTMCP) [86]: Models the environment as an I-POMDP and uses nested-MCTS to solve the I-POMDP at each reasoning level online. We use a reasoning level of $l = 2$ for our experiments, UCB for the exploration policy, and Monte Carlo rollouts for node evaluation. Notably, it does not incorporate P_{plan} into its decision-making, instead relying on a recursive I-POMDP model for the other agent.
- **Interactive Partially Observable Monte Carlo Planning** (IPOMCP) [84, 85]: Models the environment as an I-POMDP with uncertainty over the other agent's internal state (history and policy). Uses P_{plan} for modelling the other agent[4]. Uses UCB for the exploration policy and Monte Carlo rollouts for node evaluation
- **Partially Observable Monte Carlo Planning** (POMCP) [42]: Models the environment as a POMDP with the other agent's actions selected uniformly at random. This approach acts as a baseline since it uses naive modeling for the other agent. Uses UCB for the exploration policy and Monte Carlo rollouts for node evaluation
- **Partially Observable Type-based Meta Monte Carlo Planning** (POTMMCP) [78]: Similar to IPOMCP except the set of other agent policies are additionally utilized to inform planning via a meta-policy. Uses PUCB for the exploration policy with the meta-policy as the search policy. Uses Monte Carlo rollouts and pre-computed value function for node evaluations, depending on the representation used for the other agent policies in P_{plan}.

[4] In the original IPOMCP [85] and CIPOMCP [84] papers a single policy for the other agent policy was generated using an I-POMDP solver, here we use the policies from the planning population.

A high-level comparison of the key differences between each planning algorithm is shown in Table 3. Noting that both INTMCP and POMCP utilize their own model for the other agent and thus do not incorporate the planning population into their decision-making. For these two methods, we only look at their out-of-distribution performance, since their internal models are always different from the test population. Our experiments help evaluate how beneficial the inductive biases of these two methods are.

We tested each method across a range of planning budgets $S \in [0.1, 1, 5, 10, 20]$, where S represents seconds of search time per step. Further details on each method and their hyperparameters are provided in Appendix B.

5.3 Learning method

For the learning method, we train a single deep RL policy as a best-response against each population $P \in \mathcal{P}$. We refer to the learning method as *Reinforcement Learning-Best Response* (RL-BR). To train each policy we use Proximal Policy Optimization (PPO) [120] as the specific RL algorithm because it is used extensively for MARL research [20, 121, 122] and worked well across all environments we tested. In each environment, a separate RL-BR policy $\pi_{BR,k}$ was trained for each population $P_k \in \mathcal{P}$. Each policy $\pi_{BR,k}$ was trained for a fixed number of episodes where at the start of each episode the policy for the other agent π_{-i} was sampled uniformly at random from the planning population, $\pi_{-i} \sim \mathcal{U}(P_k)$. In this way each policy was trained to be a best-response to the uniform mixture over the planning population.

To ensure reproducibility in our results, we trained five versions of the same policy using different random seeds for each planning population in each environment. Each individual RL-BR policy was trained until convergence as indicated by their learning curve. The learning curves for each policy and training hyperparameters are provided in Appendix C.

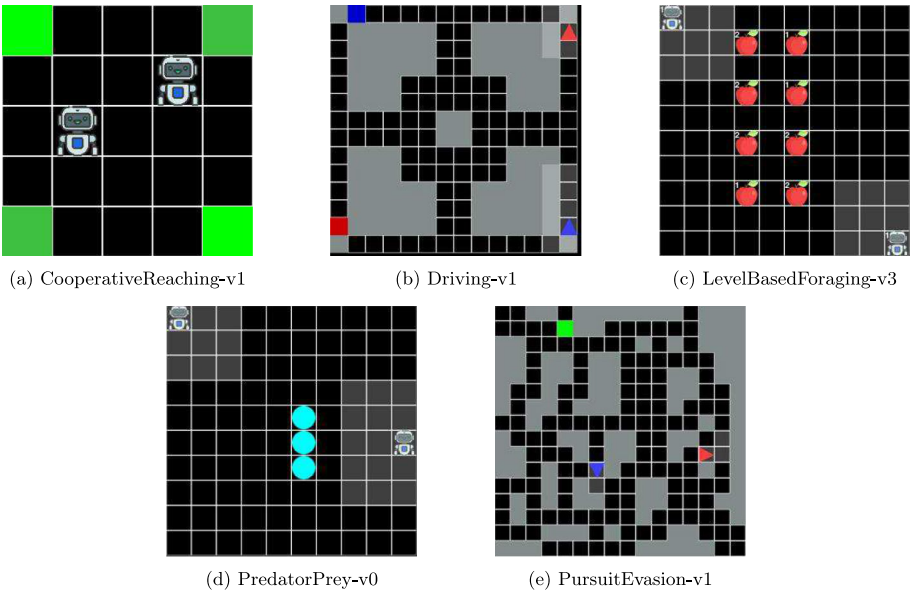
5.4 Combined planning and learning method

The combined planning and learning method incorporates the RL-BR policies from the previous section as a search policy within an MCTS based planner. POTMMCP [78] was used as the base planner with the search policy (normally the meta-policy) replaced with the RL-BR policy. We refer to this method simply as *Combined*. A side-by-side comparison of its properties with the other planning methods is shown in Table 3.

Combining a policy trained via RL with MCTS in this way has been applied in a number of previous works in varying ways [15–17, 52, 123]. Prior work has focused on settings

Table 3 Comparison of the components of the planning and combined algorithms used in our experiments. $\mathcal{U}(P_{\text{plan}})$ in the Other Agent Model column indicates the method models the other agent using a uniform distribution over the planning population of policies P_{plan}

Algorithm	Search Policy	Exploration Policy	Other Agent Model
INTMCP [86]	Random	UCB	Nested MCTS
IPOMCP [84, 85]	Random	UCB	$\mathcal{U}(P_{\text{plan}})$
POMCP [42]	Random	UCB	Random
POTMMCP [78]	Meta-Policy over P_{plan}	PUCB	$\mathcal{U}(P_{\text{plan}})$
COMBINED	π_{BR}	PUCB	$\mathcal{U}(P_{\text{plan}})$



(a) CooperativeReaching-v1 (b) Driving-v1 (c) LevelBasedForaging-v3

(d) PredatorPrey-v0 (e) PursuitEvasion-v1

Fig. 8 The environments used in our experiments. We used a diverse set of environments, including cooperative, mixed, and competitive scenarios. Since it is asymmetric we use two versions of PursuitEvasion-v1: *i0* and *i1* where planner controls the pursuer (blue) and evader (red)

where the environment is fully-observable or there is access to information-states. In this work we investigate combining RL plus planning using particle based beliefs. Using particle based beliefs is the dominant paradigm for planning under uncertainty in large environments. This work is the first, to the best of the authors knowledge, to provide a comprehensive evaluation of combining planning with RL using particle based beliefs in large multi-agent environments.

For all experiments using the combined learning and planning method we follow the same experiment protocol used for the planning methods, testing across a range of planning budgets $S \in [0.1, 1, 5, 10, 20]$. We also repeat each experiment using each of the five RL-BR policies trained for each environment and planning population.

5.5 Experiment environments

We tested each method on a range of environments from POSGGym. This included[5]: Cooperative-Reaching (cooperative) [101], Driving (mixed) [106, 107], Level-Based Foraging (mixed) [68, 103], Pursuit-Evasion (competitive) [86, 112], Predator-Prey (cooperative) [108, 109] (Fig. 8). These environments have all been previously studied in the RL and planning literature and span cooperative, mixed, and competitive settings, as well as a range of multi-agent concepts (see Table 1), allowing us to evaluate whether the benefits of combining planning and learning generalize across different types of multi-agent scenarios. We limited the selection to those with discrete actions and observations, as this was what the

[5] For further details on each environment please refer to the documentation at https://posggym.readthedocs.io

planning methods in our experiments were designed for. The size of the various components of each environment are shown in Table 4.

5.6 Experiment results

In this section, we present the main results of our experiments. We start with the key high-level findings looking at performance of each method when averaged across the entire set of environments. This gives us a view of how each method compares over a broad distribution. Next, we take a deeper dive into the results of each environment separately. Importantly, this allows us to gain some insights into where the methods do well, where they do not, and why.

5.6.1 Overall results

Figure 9 shows in- and out-of-distribution performance of each method averaged across environments and planning populations. Averaging across a distribution of environments allows us to look at how each algorithm performs more broadly, rather than for a single environment. To ensure all environments are weighted equally, we normalize the returns to be within [0, 1], so that 0 and 1 correspond to the minimum and maximum possible return within each environment. Below we discuss some of the key takeaways from our results.

For the in-distribution setting, combining planning and learning improves on either approach alone, given enough planning time We observe the benefits of incorporating planning alongside a trained RL policy when provided with an accurate model of the world and other agents. This finding aligns with previous research on combining search and RL in 2-player, zero-sum games [15, 16, 18] and cooperative games [19, 89]. Unlike previous studies that utilize exact or learned belief models with factored public and private observations, here we show that combining a RL policy with a planner is also an effective method when using particle based beliefs. We believe this is a promising result for efforts to scale up planning to more complex domains, showing that existing particle based planning methods can benefit from an RL trained policy, and vice versa.

However, this relative gain in performance does not occur in the out-of-distribution setting where the model of the other agent is inaccurate. In fact, performance appears to degrade slightly with increased search time. Subsequent sections will delve into the reasons behind this phenomenon.

Table 4 Properties of each experiment environment. $|S_0|$ denotes the number of environment states in the planning agent's initial belief which have non-zero support given the agent's initial observation. $|P_0|$ and $|P_1|$ are the number of policies within agent population

Environment	Reward Type	$\lvert\mathcal{S}\rvert$	$\lvert S_0\rvert$	$\lvert\mathcal{A}_i\rvert$	$\lvert\mathcal{O}_i\rvert$	$\lvert P_0\rvert$	$\lvert P_1\rvert$
CooperativeReaching-v1	Cooperative	625	1	5	625	5	6
Driving-v1	Mixed	8.9×10^{12}	9	5	6.6×10^{6}	5	5
LevelBasedForaging-v3	Mixed	1.5×10^{11}	5.0×10^{6}	6	30	5	5
PredatorPrey-v0	Cooperative	7.2×10^{10}	64	5	100	5	6
PursuitEvasion-v1_i0	Competitive	2.8×10^{8}	3	4	768	6	6
PursuitEvasion-v1_i1	Competitive	2.8×10^{8}	1	4	4608	6	6

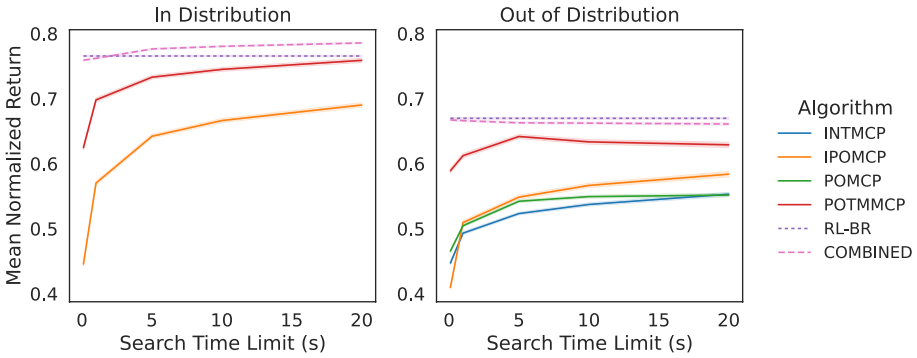


Fig. 9 In-distribution (left) and out-of-distribution (right) performance of learning (RL-BR), combined, and planning methods (INTMCP, IPOMCP, POMCP, POTMMCP) averaged across all environments. Each plot shows the mean return normalized to the interval [0, 1] from the $[\min, \max]$ possible returns for each environment. In-distribution shows results for when the planning and test populations are the same, out-of-distribution shows results when the planning and test populations are different. For the planning and combined methods we show results across search budgets (x-axis). Note, our implementations of INTMCP and POMCP do not use knowledge of the planning populations so we show only out-of-distribution performance for these methods. We find that combined planning and learning leads to improvements over each approach alone, but only in the in-distribution setting where the agent has an accurate model of the environment and the other agent. Shaded areas show 95% confidence intervals

The learning method (RL-BR) outperforms all pure planning methods in both in- and out-of-distribution settings This demonstrates a general benefit of RL over pure planning within our experimental setup. However, the RL performance does not come without some cost, requiring a larger amount of offline compute for training –up to 48 hours on 32 CPUs and 1 GPU per policy– compared to the online planners. Nonetheless, RL policies offer faster per-step execution time. Our observations highlight a key advantage of modern deep RL based methods, namely their ability to effectively leverage large amounts of compute.

For the in-distribution setting, planning methods exhibit improved performance with longer search time, with POTMMCP even matching RL-BR's performance with 20 s of search time. We observe that planning methods tends to converge towards optimality in environments where they posses a perfect model of the environment and the other agent. Notably, in some specific environments, planning methods outperform RL-BR, as shown in Fig. 11. We discuss this further in Section 5.6.2.

The most significant performance gap between learning and planning occurs in the out-of-distribution setting. Here, planning methods are prone to over-fitting to an inaccurate model of the other agent, leading to erroneous beliefs and subsequently bad policies. We discuss belief error during planning in greater detail in Section 5.6.3. Conversely, the RL-BR method likely benefits from its neural network policy representation, offering some degree of generalization to out-of-distribution settings. RL-BR also does not suffer due to belief approximation error stemming from Monte Carlo belief updates.

Monotonic improvement in performance with search time is evident for all planning and combined methods in the in-distribution setting, contrasting with the out-of-distribution setting While accurate models of the environment and the other agent lead to performance

gain with increased search time, this trend does not hold uniformly in multi-agent settings. Notably, when the test population differs from the planning population, we see performance plateau or even decline with planning budget. Further exploration into the underlying cause of this discrepancy is discussed in Section 5.6.3.

A large gap exists between in- and out-of-distribution performance across all methods Figure 10 illustrates this contrast in performance, highlighting the impact of inaccurate other agent models, regardless of whether methods are learning or planning based, or a combination of both. This observation indicates the general brittleness of the tested methods when encountering novel partners.

It is worth noting that our results are influenced by the specific test populations used. Adjusting the planning population, such as a larger or more diverse population based on some diversity metric, could potentially narrow this performance gap. The design of populations to enhance the robustness of autonomous agents in multi-agent settings is an active area of research [53, 101, 124, 125], and our finding emphasize its significance for both planning and RL.

Importantly, we observed similar overall trends between cooperative (Cooperative-Reaching-v1, PredatorPrey-v0), competitive (PursuitEvasion-v1), and mixed-incentive (Driving-v1, LevelBasedForaging-v3) environments. As we will discuss in the following sections, differences in performance between environments appear to be driven more by structural properties (e.g., observation and state space size, effective planning horizon) than by whether agents are cooperating or competing. This aligns with the decision-theoretic nature of the planning methods we tested, which are agnostic to reward structure and focus on maximizing the planning agent's reward while modeling other agents' behavior as part of the environment (Section 2.2).

Now we remind the reader that the above takeaways are based on averaging performance across all environments. Tn the next sections we zoom into the results for individual environments to explore these trends at a finer granularity.

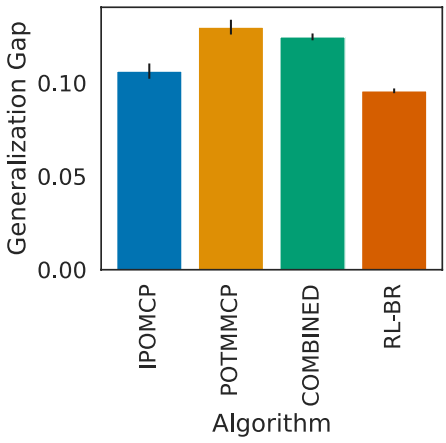


Fig. 10 Gap between in- and out-of-distribution mean normalized returns of planning (IPOMCP, POTMMCP), learning (RL-BR), and combined methods averaged across all environments. For planning and combined methods results using the maximum search time (20 s) are shown. Error bars show 95% confidence intervals

5.6.2 In-distribution performance

Here we analyze the in-distribution performance of each method in each environment individually. The primary findings are presented in Fig. 11. Contrary to the overall results, we observe more nuanced trends at the individual environment level.

In some cases, combining planning and learning can lead to deteriorating performance as the planning budget increases Specifically, in the LevelBasedForaging-v3 environment, the performance of the combined method decreases with longer search times. This unexpected outcome is notable because the combined method has access to an accurate model of the environment and the other agent, so we expect increasing performance with search time, as observed in all other environments. The result highlights a limitation of current particle-based planners, namely the introduction of error due to poor belief approximation.

From Table 4, we observe that the size of the initial state distribution for LevelBasedForaging-v3 is significantly larger than for any other environment we tested with $|S_0| = 5.0 \times 10^6$. In contrast, in our experiments the number of particles each planning method uses to represent its initial belief is at most 2320 (at the maximum planning budget). This is more than double what has been used in prior works on multi-agent decision theoretic planning, [84, 85], yet is still only a fraction of the initial belief size for LevelBasedForaging-v3. To make matters worse, in LevelBasedForaging-v3 many state features remain constant throughout an episode following initialization. Since all our planning methods rely on Bayesian updates for the beliefs, if the true value of a state features is not within the initial belief, the planner will never be aware of it, even with unlimited planning budget post-initial belief. This explanation is supported by examining the probability assigned to

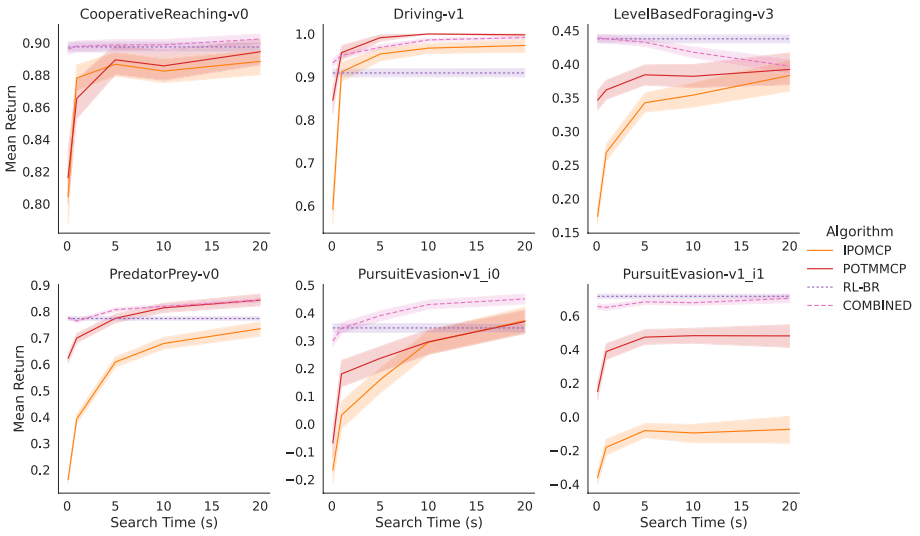


Fig. 11 In-distribution performance of planning, learning, and combined methods in each environment. The plots show the mean return of each method when the planning and test populations are the same across planning budgets (x-axis), with results averaged over the two experiment populations. Shaded areas show 95% confidence intervals

the true environment state by the combined agent's belief (Fig. 12), revealing a near-zero probability initially and reaching at most < 0.04. This is significantly lower than the initial belief accuracy in any of the other environments.

Poor belief accuracy leads to worsening performance as the planning budget increases due to how PUCB operates. Initially, action selection at the root of the tree is heavily biased by the search policy, in this case the RL-BR policy. However, as more time is spent searching, the visit counts of each action come to dominate, leading to less bias towards the search policy. Consequently, as planning time increases, the policy at the root of the tree diverges away from the search policy and towards the optimal action based on the environment model and agent's belief. In environments like LevelBasedForaging-v3, where belief accuracy is problematic, this means the agent biases more towards a policy based on an incorrect belief.

Continuing to increase planning time might eventually lead to performance improvement as belief approximation improves. However, this would require using significantly more particles for the belief representation, quickly becoming impractical for even larger environments. Our results in the LevelBasedForaging-v3 environment underscores a key limitation of current methods for belief-based planning. They either work only for relatively sparse belief spaces or require domain-specific knowledge to compute the true belief state [19] or learn a good approximation [89, 90]. Our result suggests that novel, general methods for producing robust approximate beliefs is an important direction for future planning under uncertainty research looking to scale to larger environments.

At least one planning method equals or exceeds the learning (RL-BR) method given sufficient planning time in four of six environments Specifically, POTMMCP outperforms RL-BR in Driving-v1 and PredatorPrey-v0, and matches the performance in CooperativeReaching-v0 and PursuitEvasion-v1_i0.

One explanation for this is that in these environments, each planning method is able to improve its prediction about the other agent's policy throughout an episode, leading to more accurate beliefs about the environment state and the other agent's future actions, as evi-

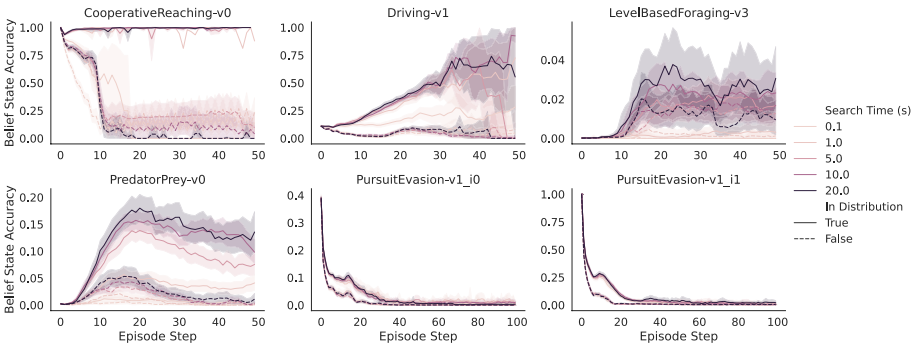


Fig. 12 Probability assigned by the combined method's belief to the true environment state during an episode. Each plot shows a different environment, with each line representing a different search budget (shaded areas show 95% confidence intervals). Solid lines show in-distribution accuracy; dashed lines show out-of-distribution accuracy. Belief accuracy is significantly worse out-of-distribution throughout substantial portions of episodes in most environments especially at higher search budgets (darker lines). The exception being LevelBasedForaging-v3 where poor accuracy occurs in both settings due to its large initial belief space ($|S_0| = 5.0 \times 10^6$). Y-axis scales differ between plots

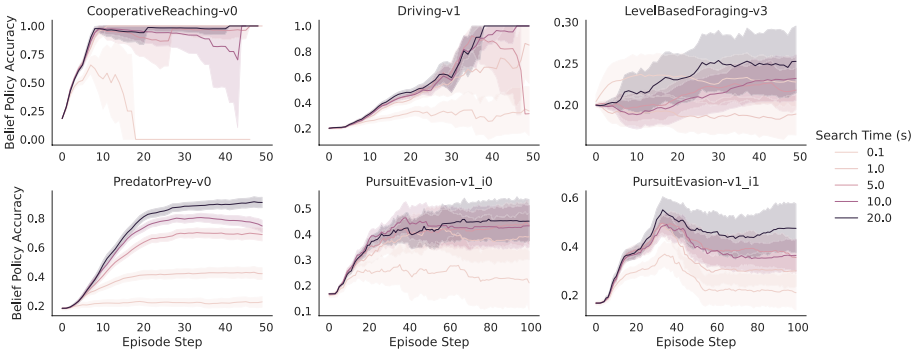


Fig. 13 Probability assigned by the combined method's belief to the true policy of the other agent throughout an episode in the in-distribution setting. Each plot shows a different environment, with each line representing a different search budget (shaded areas show 95% confidence intervals). Across all environments, belief accuracy either improves or remains stable with increased search budget (never degrading), and with adequate search time, shows improvement during early episode stages as agents interact and reduce uncertainty about each other's policies

denced in Figs. 13, 14 and 15. Improved belief accuracy directly translates to more accurate value estimates for actions and thus a better policy for the planning agent.

Another explanation is the ability of planning to generalize to novel situations. With an accurate belief and model of the environment, planning allows the agent to improve its action value estimates online for any encountered state, regardless of the state's rarity. In contrast, RL-BR performs all policy improvement offline, making it brittle when faced with situations rarely encountered during training. Evidence of this can be seen in the success rate of each method in the Driving-v1 environment shown in Table 5. Specifically, all methods succeed the vast majority of the time ($> 93\%$), however in the minority of failures RL-BR crashes significantly more often than the planning methods (4.14% vs 0.98%), suggesting that planning methods are able to improve on the robustness of RL-BR in a minority situations, likely those rare in RL-BR's training distribution. Given the large negative reward for crashing, this small difference in crash probability has a big impact of final expected return. We believe this observation most likely holds beyond the environments used in our experiments and propose further investigation of this as an interesting direction for future research.

In two environments, LevelBasedForaging-v3 and PursuitEvasion-v1_i1, no planning method reached the performance of RL-BR. In LevelBasedForaging-v3, the failure mode of particle-based planning is attributed to poor belief accuracy due to the large initial belief size as discussed above.

In PursuitEvasion-v1_i1, where the planning agent starts with a perfect belief, the performance gap between planning and learning methods is likely due to the challenge of planning over long horizons. Unlike LevelBasedForaging-v3, belief inaccuracy is not the main cause, as indicated by the increasing performance of the Combined method with search time[6]. Rather, the environment's structure presents a fundamental long horizon planning challenge. Specifically, reaching the goal requires the agent to plan over many time steps

[6] In Fig. 12 we see belief state accuracy decreases over time in the PursuitEvasion-v1_i1 environment, this is expected due to the naturally growing uncertainty over time, this is in contrast the other environments where we expect more certainty over time.

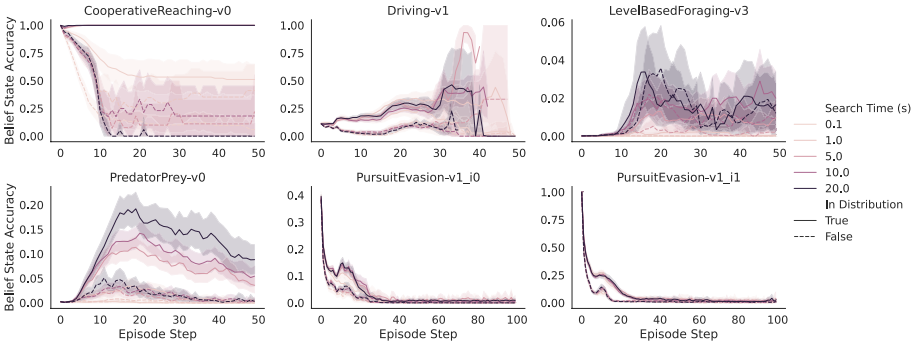


Fig. 14 Probability assigned by POTMMCP's belief to the true environment state throughout an episode. Each plot shows a different environment, with each line representing a different search budget (shaded areas show 95% confidence intervals). Solid lines show in-distribution accuracy; dashed lines show out-of-distribution accuracy. Like the combined method in Fig. 12, belief accuracy is significantly worse out-of-distribution throughout substantial portions of episodes in most environments, especially at higher search budgets (darker lines) LevelBasedForaging-v3 again shows poor accuracy in both settings due to its large initial belief space. Y-axis scales differ between plots

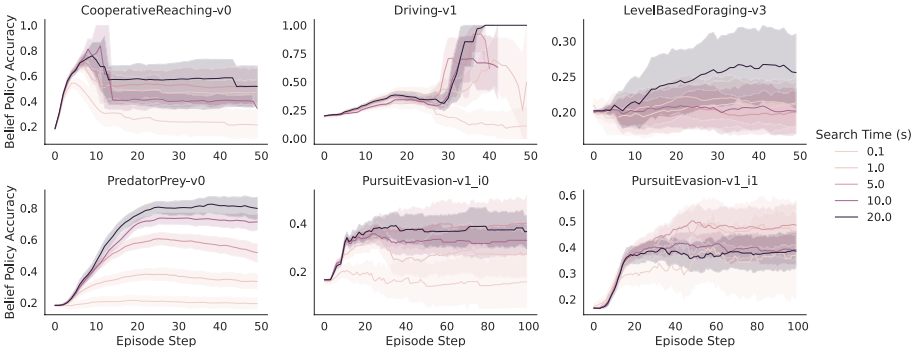


Fig. 15 Probability assigned by POTMMCP's belief to the true policy of the other agent throughout an episode in the in-distribution setting. Each plot shows a different environment, with each line representing a different search budget (shaded areas show 95% confidence intervals). Like the combined method in Fig. 13, belief accuracy either improves or remains stable with increased search budget (never degrading), and with adequate search time, shows improvement during early episode stages as agents interact and reduce uncertainty about each other's policies

Table 5 Percentage of in-distribution episodes in the Driving-v1 environment that ended with a crash (*Crashed*), the agent reaching the goal (*Success*), or the step limit being reached (*Timedout*)

Algorithm	Crashed	Success	Timedout
IPOMCP	0.98%	97.46%	1.56%
POTMMCP	0.12%	99.88%	0.00%
RL-BR	4.14%	93.12%	2.74%
COMBINED	0.46%	99.51%	0.03%

For the planning based methods (IPOMCP, POTMMCP, COMBINED) results using the maximum search time (20 s) are shown

while avoiding large negative rewards (in this case, being spotted by the other agent). If the agent is unable to plan deep enough to find trajectories where it reaches the goal, it instead gets stuck in the sub-optimal strategy of ignoring the goal and instead focusing on just avoiding the other agent. RL-BR benefits from significantly more compute to move beyond the sub-optimal strategy. The pure planning methods on the other hand are less capable of doing this, given their limited search budget. We see evidence of this in the longer average episode times for the planning methods in this environment, with the gap in episode length much greater in PursuitEvasion-v1_i1 than for any other environment (Fig. 16). Additionally, we found that $10\times$ more training steps were required to train RL-BR till convergence in PursuitEvasion-v1_i1 compared to PursuitEvasion-v1_i0 (Fig. 21). The gap in performance between RL-BR and the planning methods highlights the challenge of long-horizons for planning methods. Fortunately, improved search policies, such as those used in the Combined and POTMMCP methods as well as in other domains [16], offer a way to overcome the challenge of planning over long-horizons.

5.6.3 Out-of-distribution performance

In this section, we explore how well each planning and learning method generalizes when the planning and test populations differ ($P_{\text{plan}} \neq P_{\text{test}}$) within each environment. We include results for INTMCP and POMCP, which do not integrate a planning population into the planning process. Instead, INTMCP models other agents using a level two I-POMDP, while POMCP models other agents as uniform random, serving as baselines for common agent modeling frameworks. The main results are shown in Fig. 17.

Combining learning and planning leads to worse performance than learning alone in four of six environments This outcome is largely attributed to poor belief accuracy similar to the LevelBasedForaging-v3 environment in the in-distribution setting. As shown in Fig. 12, in all environments except LevelBasedForaging-v3, incorrect models of the other agent in the out-of-distribution setting results in worse beliefs about the environment state compared to the in-distribution setting, despite having a perfect environment model in both cases. This highlights a critical challenge in integrating planning methodologies in the multi-agent setting: the quality of the model predicting other agents' behaviors profoundly impacts belief accuracy and performance. Finding more robust methods for modelling the other agents, or developing planning techniques better able to handle model inaccuracy in multi-agent settings are both promising directions for future work.

Planning methods that modeled the other agent naively (POMCP) or recursively (INTMCP) generally performed worse than other approaches While frameworks based on recursive reasoning hold promise for domains like human-robot interaction (HRI) [8], they showed limited utility in our experiments, where the other agent policies did not behave randomly or employ explicit recursive reasoning. Rather our experiments show the benefits that can be gained by explicitly considering the possible types of other agent behaviour. Furthermore, a drawback of INTMCP is its reliance on nested MCTS which disperses simulations across multiple decision trees, thus requiring a higher number of simulations for effective decision-making. Improving the accuracy of recursive models remains a practical challenge,

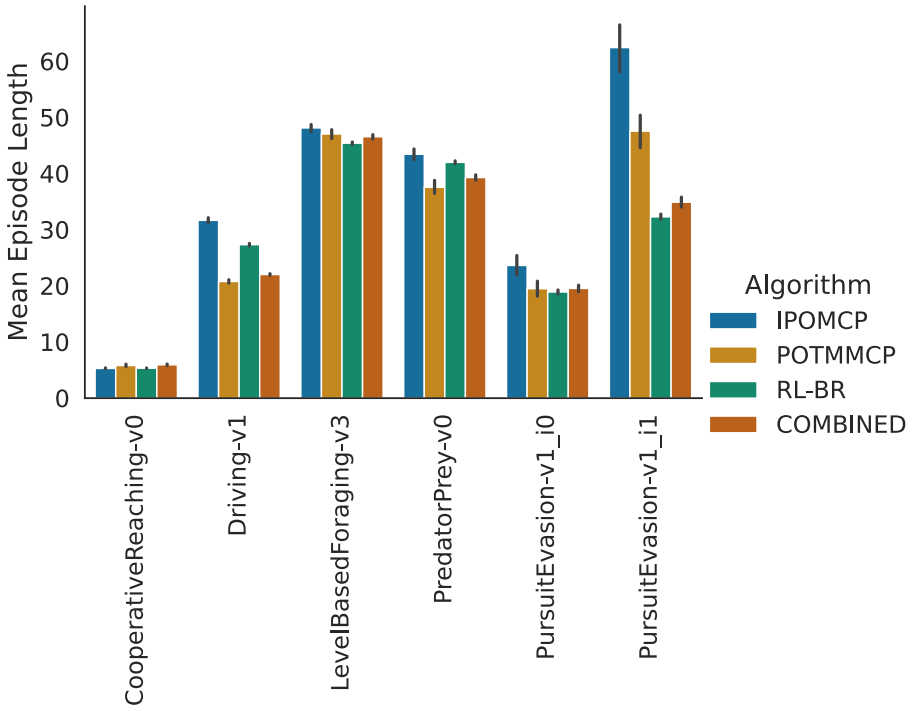


Fig. 16 Mean episode length for each algorithm in each environment for the in-distribution setting. Results for planning (POTMMCP, IPOMCP) and combined methods are using the maximum search time (20 s). Maximum episode length is 50 for all environments except PursuitEvasion-v1, where it is 100. PursuitEvasion-v1_i1 had the longest effective planning horizon. All other environments had shorter effective planning horizons due to the availability of more frequent positive rewards for the agent

suggesting that adopting diverse policies might offer a more universally resilient solution, especially considering the burgeoning field of population-based training [53, 101, 124, 125].

Planning methods performed no better or even worse than pure learning in five out of six environments Again, this trend is likely due to the formation of inaccurate beliefs about the environment state (as shown in Fig. 12). This result again highlights the reliance on accurate other agent models for planning in multi-agent settings in current state-of-the-art methods. It also points to a clear need for planning methods better equipped to handle model inaccuracy.

Increasing search budget did not consistently lead to monotonically improving performance for planning methods Unlike the in-distribution setting, higher planning time did not always result in improved performance in the out-of-distribution setting. Performance varied significantly across environments, attributed to the formation of inaccurate beliefs and challenges in predicting other agent's actions (Fig. 18). These findings underscore the complexities and limitations of planning methods in environments where accurate modeling of agents and their interactions is challenging.

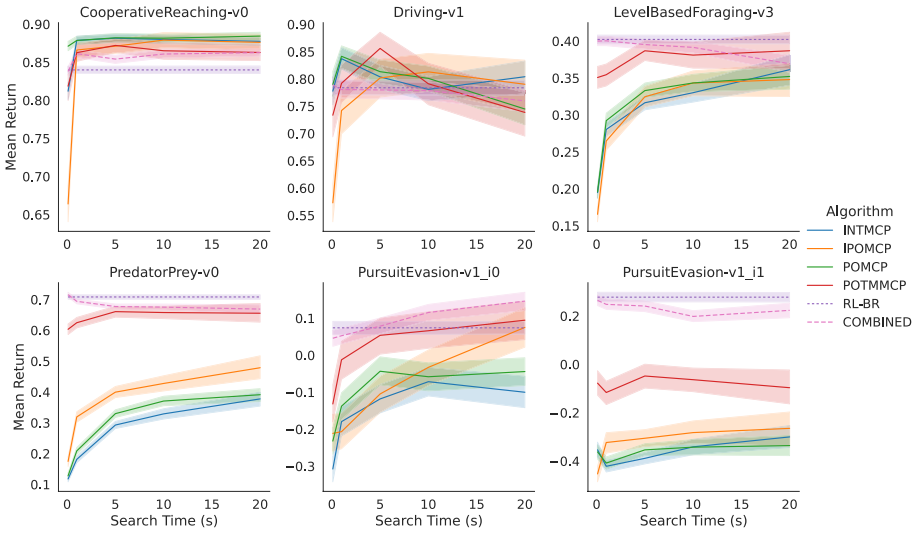


Fig. 17 Out-of-distribution performance of planning, learning, and combined methods in each environment. The plots show the mean return of each method across planning budgets (x-axis) when the planning and test populations are different, with results averaged over the two experiment populations. Shaded areas show 95% confidence intervals

5.6.4 Experimental limitations

While our experimental results provide insights into the relative performance of planning, learning, and combined approaches, there are important limitations to consider. A key consideration is the impact of our chosen reference policy populations on the experimental outcomes. While specific performance metrics would likely differ with alternative populations, we believe our overall conclusions would remain unaffected, based on several factors in our experimental design. First, most of the takeaways we have presented are based on the relative performance between methods and the in- versus out-of-distribution settings using consistent populations. Designing our experiments in this way provides some control for population effects. Second, we deliberately constructed populations with behavioral diversity (as evidenced in Fig. 20) in order to mirror real-world scenarios where agents encounter various skill levels and behaviors. Where possible, we incorporated established strategies from existing literature to maintain consistency with prior work. Additionally, our investigation of underlying mechanisms, particularly the relationship between belief accuracy and performance (Figs. 12, 13, 14 and 15, 18), provides mechanistic explanations for our observations that should generalize beyond specific populations.

Our results also highlight broader challenges in planning under uncertainty in multi-agent settings, particularly regarding belief accuracy and generalization to novel partners. The degradation in performance when encountering out-of-distribution partners, even with combined planning and learning approaches, suggests a fundamental challenge that may require new methodological advances to address. This limitation is particularly relevant for practical applications where accurate modeling of other agents' behavior patterns is often difficult or impossible to achieve.

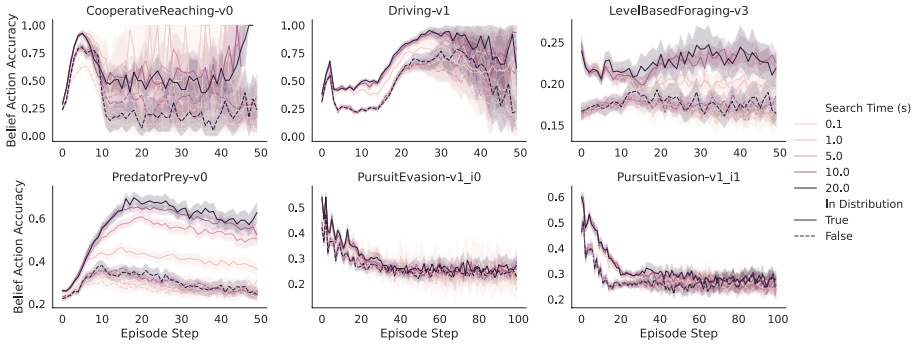


Fig. 18 Probability assigned by the Combined method's belief to the true action of the other agent during an episode in each environment. Each line represents a different search budget, with solid lines show in-distribution accuracy and dashed lines showing out-of-distribution accuracy. In general belief accuracy is significantly worse in the out-of-distribution setting, where the planning agent's model of the other agent is inaccurate

6 Conclusion and future directions

In this paper we presented POSGGym, a library focused on facilitating research at the intersection of decision-theoretic planning and RL. The key contributions of POSGGym are twofold: its provision of a diverse set of discrete and continuous environments complete with their dynamics models suitable for planning, and its collection of diverse reference policies. By designing the API to be similar to popular MARL libraries, users of POSGGym are more easily able to utilize the growing ecosystem of tools for deep RL and apply them to planning problems. POSGGym is fully open-source and accepting contributions of new multi-agent, planning benchmarks.

Using POSGGym we conducted an extensive empirical evaluation of existing state-of-the-art planning under uncertainty methods and combined planning and RL. Our investigation was the first comprehensive analysis comparing state-of-the-art online planners for large, partially observable multi-agent environments. Furthermore, it is the first to explore combining learning with planners based on particle beliefs. Our results demonstrate that combining RL with particle based planning can be an effective method for improving the robustness of agents in the multi-agent setting, given an accurate model of the environment and other agents. We also discover some key limitations of combining existing methods of particle based planning with learning, namely performance can suffer if there is inaccuracy in the other agent model, or due to poor belief approximation in environments with dense belief spaces. This highlights the critical need for developing planning methods that are robust to model uncertainty and a better quantified understanding of the relationship between model accuracy and planning performance - a direction we hope POSGGym will help facilitate.

We hope that POSGGym along with our empirical results will spur further research on the integration of decision-theoretic planning and RL in partially observable multi-agent

domains, so as to gain the best of model-driven and data-driven techniques. This interface between planning and learning is vast with many avenues for further research. In particular, improving the robustness to novel partners, and finding more scalable representations of complex beliefs.

Appendix A Agent populations

POSGGym comes with a diverse set of policies for the majority of the supported environments. Depending on the environment the available policies are a mix of heuristic and deep RL policies. In this section we provide some details on the general training procedures used for the deep RL policies in POSGGym. We also go into greater detail about the policy populations used in our experiments. Over time we expect to update the set of policies included in POSGGym, for the full up-to-date list please check out the documentation at https://posggym.readthedocs.io/. All code used for training the RL policies is available at https://github.com/RDLLab/posggym-baselines.

A.1 Reinforcement learning policy training

Every RL policy included with POSGGym to-date uses a LSTM actor-critic neural network architecture and was trained using Proximal Policy Optimization (PPO) [120]. LSTM's allow each policy to be conditioned on histories of action and observations, which is important for the majority of partially observable environments. The specific architecture used consisted of a fully-connected network (FCN) trunk, followed by a single layer LSTM, and then separate FCN actor and critic heads. The specific neural network architecture and training hyperparameters for each environment are shown in Table 6. For the grid-world problems, hyperparameters values were chosen based on commonly used values in the literature, since we found these worked well in general. For the continuous environments, some hyperparameter tuning was conducted to select appropriate values. The number of training steps was chosen such that policies could train until convergence, as indicated by their learning curves.

We used different multi-agent training schemes depending on the environment, while the same RL algorithm and neural network architecture was used for each individual policy. The two schemes we used were K-Level Reasoning (KLR) [57] and independent self-play with best response (SP-BR, commonly refered to as Independent PPO when using PPO as the RL algorithm) [126]. We used these two methods as they have been extensively studied [51, 53, 57, 126], are simple to implement (and thus replicate), and produced diverse populations of policies when combined with policy pruning to remove similar policies. Figure 19 provides a visualization of the training schema used. The following sections contain a high level overview of each scheme.

Table 6 Training hyperparameters for POSGGym RL policies in grid-world and continuous environments

	Grid-world				
Hyperparameter	LBF	Driving	Predator Prey	Pursuit Evasion	Continuous
Training steps	100M	32M	100M	10M	100M
Trunk layer sizes	[64, 64]	[64, 64]	[64, 64]	[64, 32]	[256, 256]
LSTM size	64	64	64	256	256
Head layer sizes	[64]	[64]	[64]	-	-
γ	0.99	0.99	0.99	0.99	0.99
Learning rate	0.0003	0.0003	0.0003	0.0003	0.0003
GAE λ	0.95	0.95	0.95	0.95	0.95
Batch size	6144	6144	6144	2048	65,536
Mini-batch size	2048	2048	2048	256	2048
Rollout horizon	64	64	64	100	100
Update epochs	2	2	2	2	2
BPTT sequence length	10	10	10	20	20
Entropy bonus	0.01	0.01	0.01	0.001	0.001
Value function coeff.	0.5	0.5	0.5	1.0	1.0
Clip parameter	0.2	0.2	0.2	0.3	0.5
Global gradient clipping	10	10	10	10	10

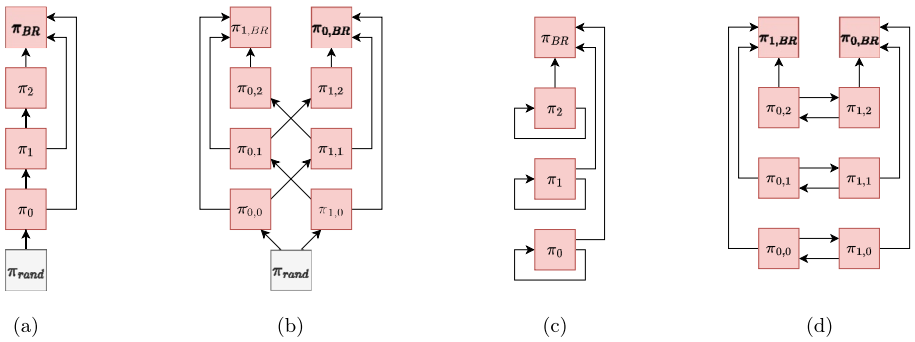


Fig. 19 Multi-agent training schemas used for generating RL policies for POSGGym environments. (a) KLR in symmetric environment, (b) KLR in asymmetric environment with two agents, (c) self-play in symmetric environment, (d) self-play in asymmetric environment with two agents. Each box is an independent policy and arrows indicate which policy a given policy was trained against. Figure adapted from [57]

A.1.1 K-level reasoning

In K-Level Reasoning (KLR) policies are trained in a hierarchy, the level $K = 0$ policy is trained against a uniform random policy, level $K = 1$ is trained against the level $K = 0$, and so on with the level K policy trained as a best response to the level $K - 1$ policy for $K > 0$.

Finally, the best-response policy K_{BR} is trained against all K level policies, excluding the random policy and the K_{BR} policy itself. In our implementation we used the Synchronous KLR Best-Response (SyKLRBR) training method [57] which trains all policies synchronously and was shown to converge in less total wall time and lead to generally more robust policies.

A.1.2 Self-play

Self-play training involves training independent policies against themselves [51, 126]. For asymmetric environments this meant training a set of policies; one for each agent in the environment for each training seed. While for symmetric environments a single policy is used by all agents in the environment. In self-play Best-Response (SP-BR) an additional best-response policy π_{BR} is trained against a uniform distribution over all the independent policies trained.

A.2 Experiment populations

For our experiments we used a set P of 10 to 12 policies for each environment we tested in.

CooperativeReaching-v1 P consisted of 11 heuristic policies `H[1-11]`. With $P_0 = \{$`H1, H2, H3, H4, H5`$\}$ and $P_1 = \{$`H6, H7, H8, H9, H10, H11`$\}$. These policies were based on prior work [101] with some adjustments made to ensure the population had diverse returns (Fig. 20a).

Driving-v1 P consisted of 10 policies: five heuristic and five RL trained policies. $P_0 = \{$ `A0, A40, A60, A80, A100`$\}$ was made up of the heuristic policies, while $P_1 = \{$ `RL1, RL2, RL3, RL4, RL5`$\}$ contained all the RL policies. Each heuristic policy followed the shortest path from the agent's start position to the goal but differed on how aggressive they were, from least aggressive `A0` to most aggressive `A100`. The aggressiveness of a policy controlled how far away another agent had to be within the agent's field of vision before the policy would stop the agent's vehicle from moving. `A0` would stop if another agent was observed anywhere and would only continue once that agent was out of view. Conversely, `A100` would continue along the shortest path irrespective of how close another observed agent was. `A[40-80]` followed policies between the two extremes. The RL policies `RL[1-5]` were produced by first training six policies using SP-BR and then pruning away any similar policies based on pairwise returns to give the final set of five policies. The pairwise returns for each policy in the population P for this environment are shown in Fig. 20b.

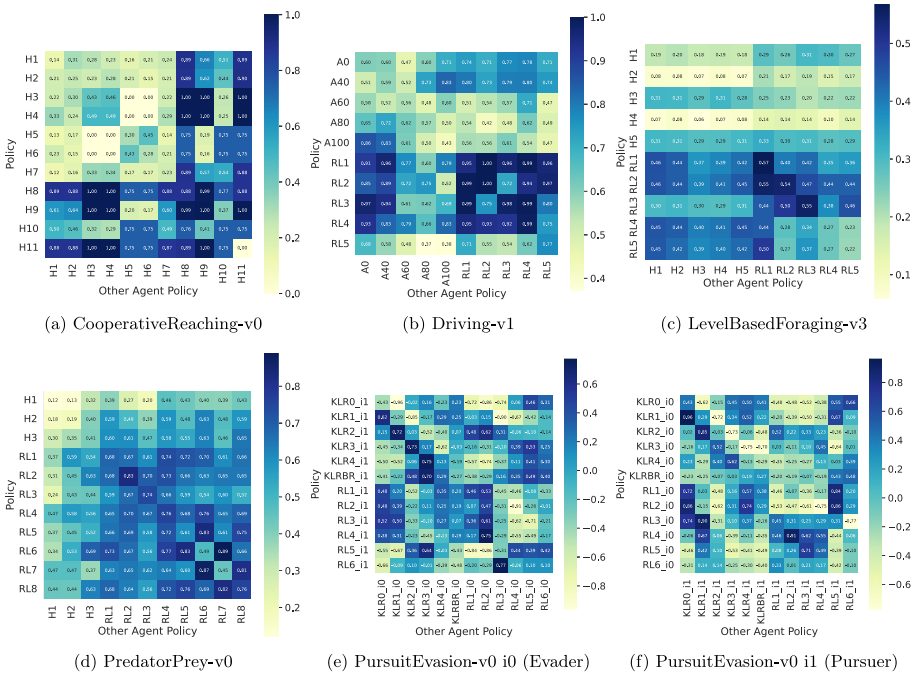
- LevelBasedForaging-v3 P consisted of 10 policies: five heuristic and five RL trained policies. $P_0 = \{$`H1, H2, H3, H4, H5`$\}$ contained the heuristic policies, while $P_1 = \{$`RL1, RL2, RL3, RL4, RL5`$\}$ contained all the RL policies. The heuristic policies were based on prior work [101], adapted to deal with partial observability.

We pruned many of the heuristic policies used in the prior work as we found that they resulted in similar behaviours based on their returns. The five heuristic policies used were:`H1` always goes to the closest observed food, irrespective of the foods level.

- `H2` goes towards the visible food closest to the centre of visible players, irrespective of food level.
- `H3` goes towards the closest visible food with a compatible level.
- `H4` selects and goes towards the visible food that is furthest from the center of visible players and that is compatible with the agents level.
- `H5` targets a random visible food whose level is compatible with all visible agents.For the RL policies we trained a population of 13 RL policies including six using SB-BR and seven using SyKLRBR (up to $K = 5$). The resulting five RL policies `RL[1-5]` were found by pruning away similar policies from the full set of 13 policies. To do this the policies were clustered based on their pairwise returns then a single policy from each cluster was chosen. The pairwise returns for each policy in the population P for this environment are shown in Fig. 20c.

- PredatorPrey-v0 P consisted of 11 policies: three heuristic and eight RL trained policies. $P_0 = \{$`H1, H2, H3, RL1, RL2`$\}$ was made up of a mix of heuristic and RL policies, while $P_1 = \{$`RL3, RL4, RL5, RL6, RL7, RL8`$\}$ contained the remaining RL policies. The heuristic policies were chosen based on trying various heuristics and selecting those that had diverse pairwise returns. The three heuristic policies used were:`H1` moves towards closest observed prey, closest observed predator, or explores randomly, in that order.
- `H2` moves towards closest observed prey, closest observed predator, or explores in a clockwise spiral around arena, in that order.
- `H3` moves towards closest observed prey to the closest observed predator or explores in a clockwise spiral around arena, in that order.For the RL policies we followed an identical protocol to the LevelBasedForaging environment; first training 13 policies using SP-BR and SyKLRBR and then pruning similar policies to produce the final population of RL policies. The pairwise returns for each policy in the population P for this environment are shown in Fig. 20d.

PursuitEvasion-v0 (evader "0"and pursuer "1") For both agents $X \in \{0, 1\}$, P consisted of 12 RL policies. $P_0 = \{$`KLR0_i`X`, KLR1_i`X`, KLR2_i`X`, KLR3_i`X`, KLR4_i`X`, KLRBR_i`$X\}$ was a population of KLR policies trained using SyKLRBR, while $P_1 = \{$`RL1_i`X`, RL2_i`X`, RL3_i`X`, RL4_i`X`, RL5_i`X`, RL6_i`$X\}$ contained RL policies trained using a mix of SP-BR and SyKLRBR. For the RL policies a population of 30 SyKLRBR policies (five separate populations of six policies with up to $K = 4$) and five self-play (no best-response) policies were trained. The most diverse (based on pairwise returns) SyKLRBR population of six policies was then selected for P_0. P_1 was then chosen by pruning away similar policies from the remaining 29 SyKLRBR and self-play policies, based on pairwise returns. The pairwise returns for each policy in the population P for this environment are shown in Fig. 20e and f.



(a) CooperativeReaching-v0 (b) Driving-v1 (c) LevelBasedForaging-v3

(d) PredatorPrey-v0 (e) PursuitEvasion-v0 i0 (Evader) (f) PursuitEvasion-v0 i1 (Pursuer)

Fig. 20 Payoff tables for POSGGym agent policies for the environments used in the experiments. Each table shows the mean returns for the row policy when paired with the column policy after 1000 episodes

Appendix B Planning experiment details

The implementation of the planning methods used in our experiments were based on those used in prior work [78, 86]. The hyperparameters used are shown in Table 7. For all methods we used normalized Q-values during planning as per [87]. For methods that used UCB (INTMCP, IPOMCP, POMCP) we used rollouts with a random policy for leaf node evaluations. For INTMCP and POMCP actions for the other agent during rollouts were chosen using a random policy, while for IPOMCP they were chosen using the other agent policy sampled from the root belief. POTMMCP used the value function from its meta-search policy for leaf node evaluation where available, otherwise used rollouts using the meta-policy for action selection. After each update we used rejection sampling for belief reinvigoration for all methods. This was used over other methods such as weighted particle filtering [127] as it did not require access to an explicit observation function and so is applicable to a wider range of environments where only a generative model is available.

Table 7 Planning hyperparameters used in our experiments

Hyperparameter	Value
Per step search time (S)	[0.1, 1, 5, 10, 20]
Discount (γ)	0.99
Discount horizon (ϵ)	0.01
Belief particles	$\lceil 100 \times S \times 1.16 \rceil$
C_{PUCB}	1.25
PUCB exploration (λ)	0.25
C_{UCB}	$\sqrt{2}$

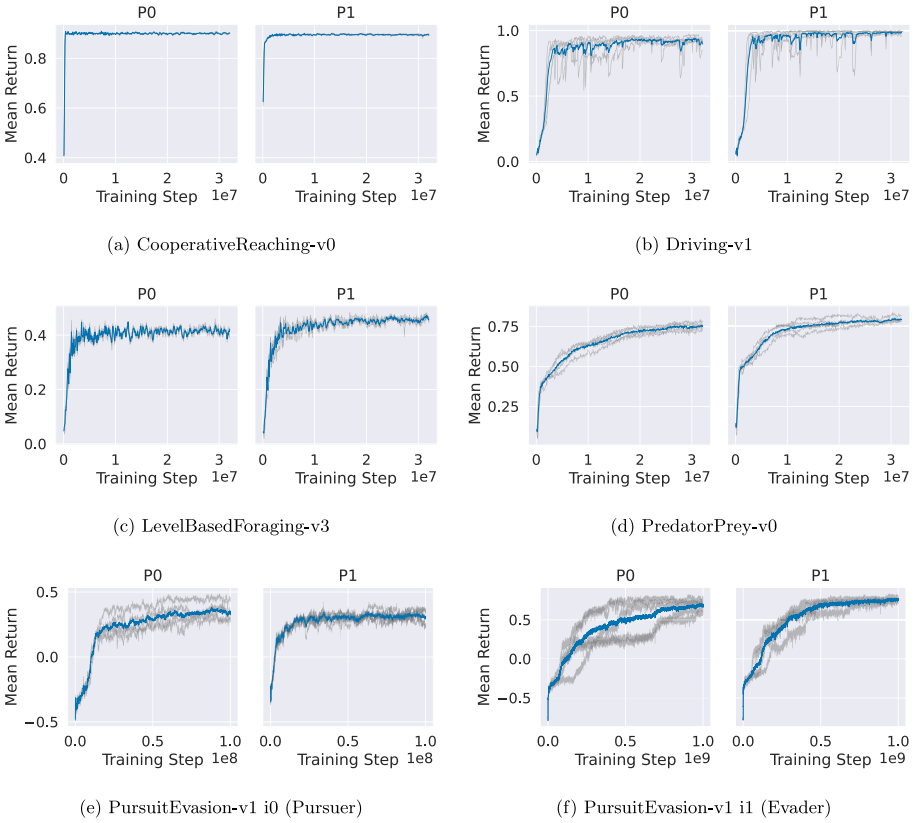
Appendix C Learning experiment details

For the learning based method (RL-BR) used in our experiments we trained a single deep RL policy $\pi_{BR,k}$ as a BR against each population $P_k \in [P_0, P_1]$ of other agents in each environment. PPO [120] was used as the RL algorithm. During training at the start of each episode a policy for the other agent π_{-i} was sampled from a uniform distribution over the planning population being trained against and this policy was used to select actions for the other agent $-i$ while actions for the ego agent i were sampled from the BR policy $\pi_{BR,k}$. In this way each policy $\pi_{BR,k}$ was trained to maximize its expected return against the uniform mixture over the planning population P_k.

The BR policy used the same neural network architecture used by the population policies (see Section 7.1) with a FCN trunk, followed by an LSTM layer, then finally separate FCN policy and value function heads. The hyperparameters are shown in Table 8. We trained five separate policies using different seeds for each combination of population and environment. Figure 21 shows the learning curve for each policy as well as the average learning curve across seeds for each population and environment.

Table 8 Training hyperparameters for RL-BR policies used in our experiments

Hyperparameter	Value
Training steps	100M (PursuitEvasion-v1 i0) 1B (PursuitEvasion-v1 i1) 32M (all other environments)
Parallel workers	32
Trunk layer sizes	[64, 64]
LSTM size	64
Head layer sizes	[64]
Discount (γ)	0.99
Learning rate	3×10^{-4}
GAE λ	0.95
Batch size	65536
Mini-batch size	2048
Rollout horizon	64
Update epochs	2
BPTT sequence length	10
Entropy bonus	0.01
Value function coeff.	0.5
Clip parameter	0.2
Global gradient clipping	10



(a) CooperativeReaching-v0 (b) Driving-v1

(c) LevelBasedForaging-v3 (d) PredatorPrey-v0

(e) PursuitEvasion-v1 i0 (Pursuer) (f) PursuitEvasion-v1 i1 (Evader)

Fig. 21 Learning curves for the RL-BR policy in each environment against each policy population P_0, P_1. Grey lines show the mean episode return throughout training for each of the five different seeds. The blue line shows the average across seeds

Appendix D Generalization results

Figure 22 shows in- vs out-of-distribution performance for each method, except INTMCP and POMCP which only have out-of-distribution results.

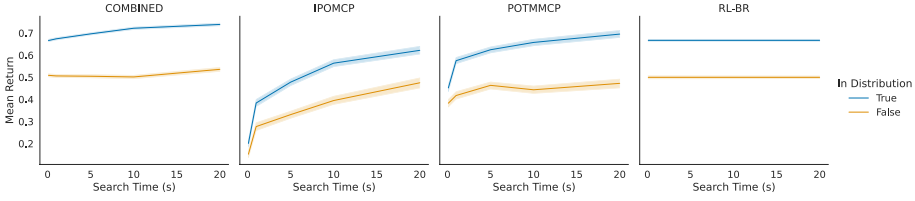


Fig. 22 In-distribution ($P_{\text{plan}} = P_{\text{test}}$) versus out-of-distribution ($P_{\text{plan}} \neq P_{\text{test}}$) performance for each environment for planning (IPOMCP, POTMMCP), learning (RL-BR), and combined methods across planning budgets (x-axis)

Author Contributions J.S and R.N contributed to the implementation of the main library and the experiments. All authors contributed to the write-up and review of the paper.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

Data Availability Code for POSGGym is available at https://github.com/RDLLab/posggym, while experim ent code is available at https://github.com/RDLLab/posggym-baselines. Links to all code and data are also provided within the manuscript.

Declarations

Competing interests The authors declare no competing interests.

References

1. Blythe, J. (1999). Decision-theoretic planning. *AI Magazine, 20*(2), 37–37.
2. Boutilier, C., Dean, T., & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research, 11*, 1–94.
3. Smallwood, R. D., & Sondik, E. J. (1973). The optimal control of partially observable Markov processes over a finite horizon. *Operations Research, 21*(5), 1071–1088.
4. Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence, 101*(1–2), 99–134.
5. Seuken, S., & Zilberstein, S. (2008). Formal models and algorithms for decentralized decision making under uncertainty. *Autonomous Agents and Multi-Agent Systems, 17*, 190–250.
6. Hansen, E., Bernstein, D., & Zilberstein, S. (2004). Dynamic programming for partially observable stochastic games. *AAAI, 4*, 709–715.
7. Kurniawati, H. (2022). Partially Observable Markov Decision Processes and Robotics. *Annual Review of Control, Robotics, and Autonomous Systems, 5*(1), 253–277.
8. Woodward, M.P., Wood, R.J. (2012). Learning from humans as an I-POMDP. *arXiv preprint arXiv:1204.0274*.
9. Carr, S., Jansen, N., Bharadwaj, S., Spaan, M. T., Topcu, U. (2021). Safe policies for factored partially observable stochastic games. In *Robotics: Science and Systems*.
10. Seymour, R., & Peterson, G. L. (2009). A trust-based multiagent system. *International Conference on Computational Science and Engineering, 3*, 109–116.
11. Ng, B., Meyers, C., Boakye, K., Nitao, J. (2010). Towards applying interactive POMDPs to real-world adversary modeling. In *Innovative Applications of Artificial Intelligence.*
12. Bernstein, D., Givan, R., Immerman, N., Zilberstein, S. (2002). The complexity of decentralized control of markov decision processes. *Mathematics of Operations Research, 27*(4).
13. LeCun, Y., Bengio, Y., Hinton, G. (2015). Deep learning. *Nature, 521*(7553).
14. Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D. (2020). Scaling laws for neural language models. arXiv preprint arXiv:2001.08361.
15. Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature, 529*(7587).
16. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., & Graepel, T. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science, 362*(6419), 1140–1144.

17. Brown, N., Sandholm, T. (2018). Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science, 359*(6374).
18. Brown, N., Bakhtin, A., Lerer, A., & Gong, Q. (2020). Combining deep reinforcement learning and search for imperfect-information games. *Advances in Neural Information Processing Systems, 33*, 17057–17069.
19. Lerer, A., Hu, H., Foerster, J., Brown, N. (2020). Improving policies via search in cooperative partially observable games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, (vol. 34, pp. 7187–7194).
20. Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C. (2019). Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680.
21. Vinyals, O., Babuschkin, I., Czarnecki, W., Mathieu, M., Dudzik, A., Chung, J., Choi, D., Powell, R., Ewalds, T., Georgiev, P. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature, 575*(7782).
22. Perolat, J., De Vylder, B., Hennes, D., Tarassov, E., Strub, F., de Boer, V., Muller, P., Connor, J., Burch, N., Anthony, T. (2022). Mastering the game of Stratego with model-free multiagent reinforcement learning. *Science, 378*(6623).
23. Ye, D., Liu, Z., Sun, M., Shi, B., Zhao, P., Wu, H., Yu, H., Yang, S., Wu, X., Guo, Q., et al. (2020). Mastering complex control in moba games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, (vol. 34, pp. 6672–6679).
24. Terry, J., Black, B., Grammel, N., Jayakumar, M., Hari, A., Sullivan, R., Santos, L. S., Dieffendahl, C., Horsch, C., Perez-Vicente, R., et al. (2021). Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems., 34*, 15032–15043.
25. Leibo, J. Z., Dueñez-Guzman, E. A., Vezhnevets, A., Agapiou, J. P., Sunehag, P., Koster, R., Matyas, J., Beattie, C., Mordatch, I., Graepel, T. (2021). Scalable evaluation of multi-agent reinforcement learning with melting pot. In *International Conference on Machine Learning*, (pp. 6187–6199). PMLR.
26. Samvelyan, M., Rashid, T., De Witt, C. S., Farquhar, G., Nardelli, N., Rudner, T. G., Hung, C.-M., Torr, P. H., Foerster, J., Whiteson, S. (2019). The starcraft multi-agent challenge. arXiv preprint arXiv:1902.04043.
27. Ellis, B., Cook, J., Moalla, S., Samvelyan, M., Sun, M., Mahajan, A., Foerster, J., & Whiteson, S. (2023). Smacv2: An improved benchmark for cooperative multi-agent reinforcement learning. *Advances in Neural Information Processing Systems, 36*, 37567–37593.
28. Lanctot, M., Lockhart, E., Lespiau, J.-B., Zambaldi, V., Upadhyay, S., Pérolat, J., Srinivasan, S., Timbers, F., Tuyls, K., Omidshafiei, S., et al. (2019). Openspiel: A framework for reinforcement learning in games. arXiv preprint arXiv:1908.09453.
29. Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., Stoica, I. (2018). Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, (pp. 3053–3062). PMLR.
30. Huang, S., Dossa, R. F. J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., & AraÃšjo, J. G. (2022). Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research, 23*(274), 1–18.
31. Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research, 22*(268), 1–8.
32. Petrenko, A., Huang, Z., Kumar, T., Sukhatme, G., Koltun, V. (2020). Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *International Conference on Machine Learning*, (pp. 7652–7662). PMLR.
33. Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. *Conference on Theoretical Aspects of Rationality and Knowledge*.
34. Shapley, L. (1953). Stochastic games. *Proceedings of the National Academy of Sciences, 39*(10).
35. Albrecht, S. V., & Stone, P. (2018). Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence, 258*, 66–95.
36. Oliehoek, F. A., Amato, C., et al. (2016). *A Concise Introduction to Decentralized POMDPs* (Vol. 1). Springer International Publishing.
37. Gmytrasiewicz, P. J., & Doshi, P. (2005). A framework for sequential planning in multi-agent settings. *Journal of Artificial Intelligence Research, 24*, 49–79.
38. Bowling, M., McCracken, P. (2005). Coordination and adaptation in impromptu teams. In *AAAI*, (vol. 5, pp. 53–58).
39. Stone, P., Kaminka, G., Kraus, S., Rosenschein, J. (2010). Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *Proceedings of the AAAI Conference on Artificial Intelligence*, (vol. 24, pp. 1504–1509).

40. Albrecht, S. V., Stone, P. (2017). Reasoning about hypothetical agent behaviours and their parameters. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, (pp. 547–555).
41. Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, (pp. 72–83). Springer.
42. Silver, D., Veness, J. (2010). Monte-Carlo planning in large POMDPs. *Advances in Neural Information Processing Systems, 23*.
43. Roy, N., Gordon, G., & Thrun, S. (2005). Finding approximate POMDP solutions through belief compression. *Journal of Artificial Intelligence Research, 23*, 1–40.
44. Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning, 47*(2), 235–256.
45. Rosin, C. D. (2011). Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence., 61*(3), 203–230.
46. Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement Learning: An Introduction* (Vol. 1). MIT press.
47. Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks, 5*(2), 157–166.
48. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780.
49. Hausknecht, M., Stone, P. (2015). Deep recurrent Q-learning for partially observable MDPs. In *2015 AAAI Fall Symposium Series*.
50. Heinrich, J., Lanctot, M., Silver, D. (2015). Fictitious self-play in extensive-form games. In *International Conference on Machine Learning*, (pp. 805–813). PMLR.
51. Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation, 6*(2).
52. Brown, N., & Sandholm, T. (2019). Superhuman AI for multiplayer poker. *Science, 365*(6456), 885–890.
53. Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Pérolat, J., Silver, D., Graepel, T. (2017). A unified game-theoretic approach to multiagent reinforcement learning. *Advances in Neural Information Processing Systems, 30*.
54. Gleave, A., Dennis, M., Wild, C., Kant, N., Levine, S., Russell, S. (2019). Adversarial policies: Attacking deep reinforcement learning. In *International Conference on Learning Representations*.
55. Jaderberg, M., Czarnecki, W., Dunning, I., Marris, L., Lever, G., Castaneda, A. G., Beattie, C., Rabinowitz, N., Morcos, A., Ruderman, A. (2019). Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science, 364*(6443).
56. Team, O. E. L., Stooke, A., Mahajan, A., Barros, C., Deck, C., Bauer, J., Sygnowski, J., Trebacz, M., Jaderberg, M., Mathieu, M. (2021). Open-ended learning leads to generally capable agents. arXiv preprint arXiv:2107.12808.
57. Cui, B., Hu, H., Pineda, L., & Foerster, J. (2021). K-level reasoning for zero-shot coordination in hanabi. *Advances in Neural Information Processing Systems, 34*, 8215–8228.
58. He, H., Boyd-Graber, J., Kwok, K., Hal Daumé, I. I. I. (2016). Opponent modeling in deep reinforcement learning. *International Conference on Machine Learning*, 1804–1813.
59. Bakhtin, A., Wu, D. J., Lerer, A., Gray, J., Jacob, A. P., Farina, G., Miller, A. H., Brown, N. (2022). Mastering the game of no-press diplomacy via human-regularized reinforcement learning and planning. In *International Conference on Learning Representations*.
60. Rutherford, A., Ellis, B., Gallici, M., Cook, J., Lupu, A., Ingvarsson, G., Willi, T., Khan, A., Witt, C. S., Souly, A., Bandyopadhyay, S., Samvelyan, M., Jiang, M., Lange, R. T., Whiteson, S., Lacerda, B., Hawes, N., Rocktäschel, T., Lu, C., Foerster, J. N. (2023). JaxMARL: Multi-agent RL environments in JAX. In *Second Agent Learning in Open-Endedness Workshop*. https://openreview.net/forum?id=BlhQN9Jfpf
61. Suarez, J., Du, Y., Isola, P., Mordatch, I. (2019). Neural MMO: A massively multiagent game environment for training and evaluating intelligent agents. arXiv preprint arXiv:1903.00784.
62. Zheng, L., Yang, J., Cai, H., Zhou, M., Zhang, W., Wang, J., Yu, Y. (2018). Magent: A many-agent reinforcement learning platform for artificial collective intelligence. *AAAI, 32*.
63. Lechner, M., Yin, L., Seyde, T., Wang, T.-H., Xiao, W., Hasani, R., Rountree, J., Rus, D. (2023). Gigastep - one billion steps per second multi-agent reinforcement learning. In *37th Conferences on Neural Information Processing Systems Datasets and Benchmarks Track*.
64. Lopez, P. A., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flötteröd, Y.-P., Hilbrich, R., Lücken, L., Rummel, J., Wagner, P., Wießner, E. (2018). Microscopic traffic simulation using sumo. *International conference on intelligent transportation systems*, 2575–2582.
65. Zhang, H., Feng, S., Liu, C., Ding, Y., Zhu, Y., Zhou, Z., Zhang, W., Yu, Y., Jin, H., Li, Z. (2019). Cityflow: A multi-agent reinforcement learning environment for large scale city traffic scenario. *The World Wide Web Conference*.

66. Tunyasuvunakool, S., Muldal, A., Doron, Y., Liu, S., Bohez, S., Merel, J., Erez, T., Lillicrap, T., Heess, N., Tassa, Y. (2020). Dm_control: Software and tasks for continuous control. *Software Impacts, 6*.
67. Peng, B., Rashid, T., Witt, C., Kamienny, P.-A., Torr, P., Böhmer, W., & Whiteson, S. (2021). Facmac: Factored multi-agent centralised policy gradients. *Advances in Neural Information Processing Systems., 34*, 12208–12221.
68. Papoudakis, G., Christianos, F., Schäfer, S. (2021). Lukas andAlbrecht: Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks. *Advances in Neural Information Processing Systems Track on Datasets and Benchmarks*.
69. Zha, D., Lai, K.-H., Cao, Y., Huang, S., Wei, R., Guo, J., Hu, X. (2019). RLCard: A toolkit for reinforcement learning in card games. arxiv preprint arXiv:1910.04376.
70. Koyamada, S., Okano, S., Nishimori, S., Murata, Y., Habara, K., Kita, H., & Ishii, S. (2023). Pgx: Hardware-accelerated parallel game simulators for reinforcement learning. *Advances in Neural Information Processing Systems, 36*, 45716–45743.
71. Osborne, M., & Rubinstein, A. (1994). *A Course in Game Theory*. MIT press.
72. Spaan, M., Oliehoek, F. (2008). The multiagent decision process toolbox: Software for decision-theoretic planning in multiagent systems. *Proceedings of the AAMAS Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains (MSDM)*.
73. Carmo Alves, M. A., Varma, A., Elkhatib, Y., Soriano Marcolino, L. (2022). Adleap-mas: An open-source multi-agent simulator for ad-hoc reasoning. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, (pp. 1893–1895).
74. Wu, F., Zilberstein, S., Chen, X. (2011). Online planning for ad hoc autonomous agent teams. In *International Joint Conference on Artificial Intelligence*.
75. Barrett, S., Stone, P., Kraus, S. (2011). Empirical evaluation of ad hoc teamwork in the pursuit domain. In *Autonomous Agents and Multiagent Systems*, (pp. 567–574).
76. Barrett, S., Agmon, N., Hazon, N., Kraus, S., Stone, P. (2014). Communicating with unknown teammates. In *Proceedings of the International Conference on Autonomous Agents and Multi-agent Systems*, (pp. 1433–1434).
77. Yourdshahi, E. S., Pinder, T., Dhawan, G., Marcolino, L. S., Angelov, P. (2018). Towards large scale ad-hoc teamwork. In *International Conference on Agents*, (pp. 44–49).
78. Schwartz, J., Kurniawati, H., Hutter, M. (2023). Combining a meta-policy and Monte-Carlo planning for scalable type-based reasoning in partially observable environments. arXiv preprint arXiv:2306.06067.
79. Czechowski, A., Oliehoek, F. A. (2021). Decentralized MCTS via learned teammate models. In *International Joint Conferences on Artificial Intelligence*, (pp. 81–88).
80. Choudhury, S., Gupta, J. K., Morales, P., & Kochenderfer, M. J. (2022). Scalable Online Planning for Multi-Agent MDPs. *Journal of Artificial Intelligence Research, 73*, 821–846.
81. Cowling, P. I., Powley, E. J., & Whitehouse, D. (2012). Information set monte carlo tree search. *Transactions on Computational Intelligence and AI in Games, 4*(2), 120–143.
82. Amato, C., Oliehoek, F. (2015). Scalable planning and learning for multiagent POMDPs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, (vol. 29).
83. Panella, A., & Gmytrasiewicz, P. (2017). Interactive POMDPs with finite-state models of other agents. *Autonomous Agents and Multi-Agent Systems, 31*, 861–904.
84. Kakarlapudi, A., Anil, G., Eck, A., Doshi, P., Soh, L.-K. (2022). Decision-theoretic planning with communication in open multiagent systems. In *Uncertainty in Artificial Intelligence*, (pp. 938–948). PMLR.
85. Eck, A., Shah, M., Doshi, P., Soh, L.-K. (2020). Scalable decision-theoretic planning in open and typed multiagent systems. *AAAI*.
86. Schwartz, J., Zhou, R., Kurniawati, H. (2022). Online planning for interactive-POMDPs using nested monte carlo tree search. In *International Conference on Intelligent Robots and Systems*, (pp. 8770–8777). IEEE.
87. Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., & Graepel, T. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature, 588*(7839), 604–609.
88. Timbers, F., Bard, N., Lockhart, E., Lanctot, M., Schmid, M., Burch, N., Schrittwieser, J., Hubert, T., Bowling, M. (2022). Approximate exploitability: Learning a best response. In *Proceedings of the International Joint Conference on Artificial Intelligence*, (pp. 3487–3493).
89. Hu, H., Lerer, A., Brown, N., Foerster, J. (2021). Learned belief search: Efficiently improving policies in partially observable settings. arxiv preprint arXiv:2106.09086.
90. Li, Z., Lanctot, M., McKee, K. R., Marris, L., Gemp, I., Hennes, D., Muller, P., Larson, K., Bachrach, Y., Wellman, M. P. (2023). Combining tree-search, generative models, and nash bargaining concepts in game-theoretic reinforcement learning. arxiv preprint arXiv:2302.00797.
91. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W. (2016). OpenAI Gym. arXiv preprint arXiv:1606.01540.

92. Foundation, F. (2022). Gymnasium. *GitHub*.
93. Ross, S., Pineau, J., Paquet, S., & Chaib-Draa, B. (2008). Online Planning Algorithms for POMDPs. *Journal of Artificial Intelligence Research, 32*, 663–704.
94. Tabrez, A., Luebbers, M. B., & Hayes, B. (2020). A survey of mental modeling techniques in human-robot teaming. *Current Robotics Reports, 1*, 259–267.
95. Ooi, J., Wornell, G. (1996). Decentralized control of a multiple access broadcast channel: Performance bounds. *Conference on Decision and Control, 1*.
96. Dibangoye, J. S., Amato, C., Buffet, O., & Charpillet, F. (2016). Optimally solving Dec-POMDPs as continuous-state mdps. *Journal of Artificial Intelligence Research, 55*, 443–497.
97. Peralez, J., Delage, A., Buffet, O., Dibangoye, J. S. (2024). Solving hierarchical information-sharing Dec-POMDPs: an extensive-form game approach. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 40414–40438.
98. Doshi, P., Gmytrasiewicz, P.J. (2005). A particle filtering based approach to approximating interactive POMDPs. In *AAAI,* (pp. 969–974).
99. Doshi, P., Perez, D. (2008). Generalized point based value iteration for interactive POMDPs. In *AAAI,* (pp. 63–68).
100. Sonu, E., & Doshi, P. (2015). Scalable solutions of interactive POMDPs using generalized and bounded policy iteration. *Autonomous Agents and Multi-Agent Systems, 29*, 455–494.
101. Rahman, A., Cui, J., Stone, P. (2024a). Generating teammates for training robust ad hoc teamwork agents via best-response diversity. In *AAAI*.
102. Rahman, M., Cui, J., Stone, P. (2024b). Minimum coverage sets for training robust ad hoc teamwork agents. In *AAAI,* (vol. 38, pp. 17523–17530).
103. Christianos, F., Schäfer, L., & Albrecht, S. (2020). Shared experience actor-critic for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems, 33*, 10707–10717.
104. Papoudakis, G., Christianos, F., & Albrecht, S. (2021). Agent modelling under partial observability for deep reinforcement learning. *Advances in Neural Information Processing Systems, 34*, 19210–19222.
105. Rahman, A., Carlucho, I., Höpner, N., & Albrecht, S. V. (2023). A general learning framework for open ad hoc teamwork using graph-based policy learning. *Journal of Machine Learning Research, 24*(298), 1–74.
106. McKee, K. R., Leibo, J. Z., Beattie, C., & Everett, R. (2022). Quantifying the effects of environment and population diversity in multi-agent reinforcement learning. *Autonomous Agents and Multi-Agent Systems, 36*(1), 21.
107. Lerer, A., Peysakhovich, A. (2019). Learning existing social conventions via observationally augmented self-play. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, (pp. 107–114).
108. Tan, M. (1993). Multi-agent reinforcement learning: Independent vs cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, (pp. 330–337).
109. Leibo, J., Zambaldi, V., Lanctot, M., Marecki, J., & Graepel, T. (2017). Multi-agent Reinforcement Learning in Sequential Social Dilemmas. *Autonomous Agents and Multi-Agent Systems, 16*, 464–473.
110. Xing, D., Liu, Q., Zheng, Q., Pan, G., Zhou, Z.: Learning with generated teammates to achieve type-free ad-hoc teamwork. In International Joint Conference on Artificial Intelligence, pp. 472–478 (2021)
111. O'Callaghan, D., Mannion, P. (2021). Tunable behaviours in sequential social dilemmas using multi-objective reinforcement learning. In *Autonomous Agents and Multi-Agent Systems*, (pp. 1610–1612).
112. Seaman, I. R., van de Meent, J.-W., Wingate, D. (2018). Nested reasoning about autonomous agents using probabilistic programs. arXiv preprint arXiv:1812.01569.
113. Blomqvist, V. (2023). Pymunk . https://pymunk.org
114. Gupta, J., Egorov, M., Kochenderfer, M. (2017). Cooperative multi-agent control using deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 66–83.
115. De Souza, C., Newbury, R., Cosgun, A., Castillo, P., Vidolov, B., Kulić, D. (2021). Decentralized multi-agent pursuit using deep reinforcement learning. *Robotics and Automation Letters, 6*(3).
116. de Souza, C., Castillo, P., Vidolov, B. (2022). Local interaction and navigation guidance for hunters drones: A chase behavior approach with real-time tests. *Robotica, 40*(8).
117. Angelani, L. (2012). Collective predation and escape strategies. *Physical Review Letters, 109*(11).
118. Janosov, M., Virágh, C., Vásárhelyi, G., Vicsek, T. (2017) Group chasing tactics: How to catch a faster prey. *New Journal of Physics*.
119. Albrecht, S., Crandall, J., & Ramamoorthy, S. (2016). Belief and truth in hypothesised behaviours. *Artificial Intelligence, 235*, 63–94.
120. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.
121. Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., Mordatch, I. (2019). Emergent tool use from multi-agent autocurricula. *International Conference on Learning Representations*.

122. Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., & Wu, Y. (2022). The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems, 35*, 24611–24624.
123. Fickinger, A., Hu, H., Amos, B., Russell, S., & Brown, N. (2021). Scalable online planning via reinforcement learning fine-tuning. *Advances in Neural Information Processing Systems, 34*, 16951–16963.
124. Lupu, A., Cui, B., Hu, H., Foerster, J. Trajectory diversity for zero-shot coordination. In *International Conference on Machine Learning*, pp. 7204–7213. PMLR.
125. Xing, D., Liu, Q., Zheng, Q., Pan, G., Zhou, Z.H. (2021) Learning with generated teammates to achieve type-free ad-hoc teamwork. In *International Joint Conference on Artificial Intelligence*, pp. 472–478.
126. Witt, C., Gupta, T., Makoviichuk, D., Makoviychuk, V., Torr, P., Sun, M., Whiteson, S. (2020). Is independent learning all you need in the starcraft multi-agent challenge? arxiv preprint arXiv:2011.09533.
127. Sunberg, Z., & Kochenderfer, M. (2018). Online Algorithms for POMDPs with Continuous State, Action, and Observation Spaces. *International Conference on Automated Planning and Scheduling, 28*, 259–263.

Authors and Affiliations

Jonathon Schwartz[1] · Rhys Newbury[2] · Dana Kulić[2] · Hanna Kurniawati[1]

✉ Jonathon Schwartz
jonathon.schwartz@anu.edu.au

Rhys Newbury
rhys.newbury@monash.edu.au

Dana Kulić
dana.kulic@monash.edu

Hanna Kurniawati
hanna.kurniawati@anu.edu.au

[1] School of Computing, Australian National University, Canberra, ACT, Australia

[2] Department of Electrical and Computer Systems Engineering, Monash University, Clayton, VIC, Australia