Supplementary Material for Combining Planning and Reinforcement Learning for Solving Relational Multiagent Domains

Nikhilesh Prabhakar
The University of Texas at Dallas
Richardson, Texas, USA
nikhilesh.prabhakar@utdallas.edu

Ranveer Singh
The University of Texas at Dallas
Richardson, Texas, USA
ranveer.singh@utdallas.edu

Harsha Kokel
IBM Research
San Jose, California, USA
harsha.kokel@ibm.com

Sriraam Natarajan
The University of Texas at Dallas
Richardson, Texas, USA
sriraam.natarajan@utdallas.edu

Prasad Tadepalli
Oregon State University
Corvallis, Oregon, USA
tadepall@eecs.oregonstate.edu

ACM Reference Format:
Nikhilesh Prabhakar, Ranveer Singh, Harsha Kokel, Sriraam Natarajan, and Prasad Tadepalli . 2025. Supplementary Material for Combining Planning and Reinforcement Learning for Solving Relational Multiagent Domains. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 6 pages.

1 REPREL

RePReL[1] is a hierarchical framework that integrates relational planning and reinforcement learning to solve goal-directed sequential decision-making problems. It leverages the strengths of both approaches to improve convergence and enable effective transfer across multiple tasks. The framework uses a high-level relational planner to decompose goals into subgoals. These subgoals are then passed to a low-level reinforcement learning agent that tries to achieve them with minimal cost. RePReL adapts first-order conditional influence (FOCI) statements to specify bisimilarity conditions of MDPs, justifying safe and effective abstractions for reinforcement learning. The relational representation used in its hierarchical ordered planning facilitates generalization across varying numbers and types of objects without requiring excessive feature engineering. Empirical evaluations demonstrate RePReL's superior performance, efficient learning, and generalization to unseen tasks compared to other planner-RL combinations, such as TaskableRL. RePReL's ability to generalize across a varying number of objects makes it well-suited for goal-directed relational domains.

2 TASK DISTRIBUTOR

The output of the planner is typically the task decomposition and does not bind the tasks to the specific agents. We use a task distributor as part of the relational planner to divide the ordered plan provided into agent-specific sub-plans using agent constraints for the different tasks. The grounded plan (high-level plan) and a set of ordering constraints are used to distribute the tasks to create agent-specific plans (sub-plans). A greedy approach is used to schedule tasks that involve forward chaining [2]. It examines the causal links between operators and prevents tasks from being assigned to agents that cannot execute them.

A **causal link** is defined as an edge between grounded operator O_p and O_q if there exists a literal in the effects of O_p that is also part of the preconditions of O_q. The set of causal links is identified after a plan is derived. The causal links $L = (l_1, l_2, \cdots, l_n)$ define a partial ordering between operations. Our task distributor ensures that the same agent performs the causally linked operations.

The current domains shown in our experiment section assume that a plan can be split into N causally independent sub-plans, where N can be the number of tasks or sub-tasks. As mentioned in Section 3.2, our current implementation of the distributor solely utilizes a greedy approach with causally linked operators to efficiently allocate sub-plans. Sub-tasks are distributed equally, with the current implementation treating each subtask with a cost of 1. The agent-specific task distributor focuses on maintaining an equal task execution cost between agents. Our framework can be easily extended to handle more complex cost functions as there is no inherent assumption about the costs. While we used homogeneous agents for experiments, there is no inherent assumption about the types of agents, and hence heterogeneous agents are easily implementable in the framework. The task distributor for heterogeneous agents would include a cost function that incorporates both task-specific information and the agent's properties. This detail can account for different agent skills for more complex versions of domains such as the dungeon environment, where we can assign the agents different classes that deal different damages based on enemy type.

3 MAREPREL METHODS

We describe the two subroutines used by the MaRePReL algorithm, i.e. **GetAgentActions** and **RePReLStep** in detail

3.1 GetAgentActions

The method 1 is used to get the agent actions given the current state, the current tasks for each agent, the current policy, the D-FOCI statements, and the set of agents.

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowé (eds.), May 19 – 23, 2025, Detroit, Michigan, USA. © 2025 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

We initialize each agent's actions dictionary with *NULL* (no-action). We iterate over the different agents, getting the policy of the agents based don't the current task (which decides the operator) **(lines 2-12)**. Moreover, we use D-FOCI statements F along with our current operator to construct the abstract state $\hat{s}$ from our current state s **(line 6)** by masking out the irrelevant parts of the current state for the task at hand. We evaluate the current state against the terminal condition for the operator to determine whether it is terminal. In case the state is non-terminal, we use the policy to get a new action **(lines 7-10)**. Once we have determined the actions for all the agents, we return a dictionary of their actions **(line 13)**.

Algorithm 1 GetAgentActions

Input: Current state s, current agent tasks ϕ, current operator policies π_o, the D-FOCI statements F, and the set of agent A
Output: The dictionary of actions for the different agents *actions*

1: Initialize the actions dictionary *actions* with *NULL* action for each agent
2: **for** agent $i \in A$ **do**
3: **if** current task for agent i, ϕ_i is not *NULL* **then**
4: Get the current operator o_i based on current task ϕ_i
5: The agent policy, $\pi_i \leftarrow \pi[o_i]$
6: Get the abstract state $\hat{s}$ based on the current state s, operator o_i, and D-FOCI statements F
7: $terminalState \leftarrow \hat{s} \in \beta(o_i)$
8: **if** $\neg terminalState$ **then**
9: The action for the agent $actions[i] \leftarrow \pi_i(\hat{s})$
10: **end if**
11: **end if**
12: **end for**
13: **returns** *actions*

Algorithm 2 RePReLStep

Input: The current state s, the dictionary of agent actions *actions*, the operator buffers $\mathcal{D}$, the terminal reward t_R, the current task for the agents ϕ, the current plan Π, the D-FOCI rules F, and the set of agents A
Output: The next state s', the updated buffers $\mathcal{D}$, the updated tasks for the different agents ϕ, and the plan flag $planValid$

1: Perform the step in the environment with state s using *actions* to get the next state s' and agent rewards *rewards*
2: $planValid \leftarrow True$
3: **for** each agent $i \in A$ **do**
4: **if** the agent task ϕ_i is $\neg NULL$ **then**
5: Get the current operator for the agent i, o_i based on the current task ϕ_i
6: Get the abstract states $\hat{s}, \hat{s'}$ using the operator o_i and D-FOCI statements F for the states s and s' respectively
7: $wasTerminal \leftarrow \hat{s} \in \beta(o_i)$
8: $preConditionMet \leftarrow \hat{s} \in I(o_i)$
9: $isTerminal \leftarrow \hat{s'} \in \beta(o_i)$
10: **if** $isTerminal$ **then**
11: $rewards[i] \leftarrow rewards[i] + t_R$
12: $\phi_i \leftarrow$ Pop the first task from agent plan Π_i
13: **else**
14: **if** $wasTerminal$ or $\neg preConditionMet$ **then**
15: $planValid \leftarrow False$
16: **end if**
17: **end if**
18: $\mathcal{D}_{o_i} \leftarrow \mathcal{D}_{o_i} \cup \{\hat{s}, actions[i], rewards[i], \hat{s'}\}$
19: **end if**
20: **end for**
21: **return** $s', \mathcal{D}, \phi, planValid$

3.2 RePReLStep

The method 2 given the current state s, actions for each agent *actions*, the operator buffers $\mathcal{D}$, the terminal reward t_R, the current tasks for the different agents ϕ, the current plan Π, the set of D-FOCI statements F and the set of agents A, would perform a step in the environment to return the next state $\hat{s}$, the updated trajectory buffers $\mathcal{D}$, agent tasks ϕ as well as the flag indicating whether the current plan is valid $planValid$.

The method begins with taking the joint step for the agents in the environment based on the current state s, and agent actions *actions* to get the updated state s' and agent rewards *rewards* **(line 1)** and we initially assume that the current agent-specific plans are valid **(line 2)**.

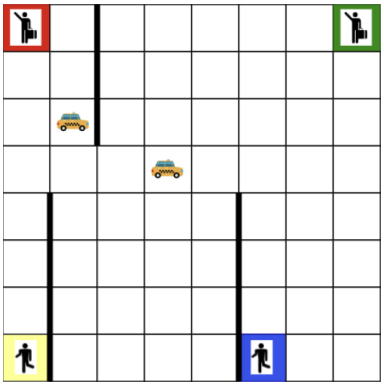
We iterate over each agent, and for agents with pending tasks, we get the current operator for the agent **(line 5)**, the abstract states $\hat{s}, \hat{s'}$ based on the D-FOCI rules F and operator for the agent o_i **(line 6)**. We compute whether the current state or the previous state was terminal **(line 7-8)**. Additionally, we use the multiagent planner (which defines all the operators) to the current state against the terminal condition of the agent's operator. If the current state is terminal, we add the terminal rewards to the agent's reward and update the agent task based on the plan Π **(line 10-12)**. Additionally, we check whether the previous state was terminal and whether the previous state satisfies the precondition of the operator **(line 8-9)**. If the previous state was terminal, or the precondition was not satisfied, it means that our plan is no longer valid, and we would need to compute it again **(line 12-15)**. We add the current transition for the agent to the agent's operator buffer **(line 16)**. Once all the agents have been evaluated, we return the successor state s', the updated buffer $\mathcal{D}$, the updated agent tasks ϕ, and the $planValid$ flag **(line 21)**

4 EXPERIMENT DOMAINS

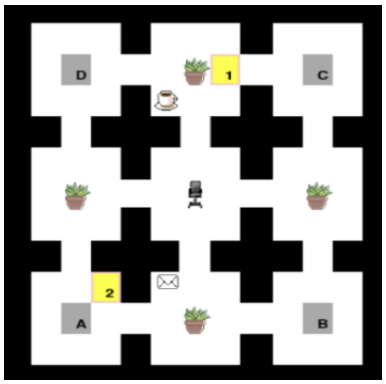
Three relational multiagent domains were considered for the experiments, which are described in the subsequent subsections

4.1 Multiagent Taxi

The agent's goal in the Multiagent Taxi environment is to transport all the passengers to their respective destination locations, which involves picking up the passengers from one location and dropping them off at another. The observation, action, and rewards for the environment are defined as follows



(a) Multiagent Taxi



(b) Multiagent Office

(c) Multiagent Dungeon

Figure 1: Relational Multiagent Domains

(1) **Observation:** For each agent, the observation would contain the agent coordinates and the coordinates of the other agents. For each passenger, we would have its pickup location, drop location, as well as the taxi the passenger may be in. The pickup and drop locations are represented as one hot to indicate one of R, G, B, and Y
(2) **Actions:** The possible actions for an agent in a given state are
    Move Up
    Move Down
    Move Left
    Move Right
    Pickup
    Drop
(3) **Rewards:** There is a -0.1 step cost associated with each step and a -1 cost associated with a no-move action. A reward of +20 will be provided for each pickup and drop. There is a high negative penalty of -100 for crashing into other agents followed by episode termination
(4) **Number of Agents:** While the environment can be extended to four agents, we considered only two agents for the experiments.

4.2 Multiagent Office

The goal in the Multiagent Office environment is to finish all the designated tasks in the environment, which can be either visiting locations – A, B, C, or D –, picking up coffee (C) or mail (M) and potentially dropping off to the office (O). The observations, actions, and rewards for the environment are defined as follows

(1) **Observation:** The observations for each agent would contain each agent's position, inventory for coffee and mail, and the facts of the environment state which is an indicator for each task possible in the environment and the value may indicate the agent which completed the task first
(2) **Actions:** The possible actions for an agent in a given state are
    Move Up
    Move Down
    Move Left
    Move Right
    There are no separate pickup and drop actions, as an item like mail or coffee is automatically added to the inventory when the agent moves to its location in the environment.
(3) **Rewards:** There is a -0.1 step cost associated with each step and a -1 cost for each invalid action. There is a -30 reward for bumping into other agents. There is a terminal reward of 100 provided to all the agents for completing the tasks in the episode
(4) **Number of Agents:** While the environment can be extended to accommodate n agents, we only consider the case with two agents for the experiments.

4.3 Multiagent Dungeon

The goal in the multiagent dungeon environment is to escape by unlocking the door using the keys collected by defeating all the enemies present in the environment. The enemies can be either skeletons with normal attack and hit points or dragons with high attack and normal hit points. The observation, actions, and rewards for the environment are defined as follows

(1) **Observation:** The observation space for each agent would contain each agent's location and orientation, attack, hit points, and defense stance value. It also contains the location, hit points, key status (a binary indicating whether the key is in the door), and another binary value indicating if the agent is holding it. Finally, there is the door location as a value indicating the door-locked status
(2) **Actions:** The possible actions for an agent in the environment are
    Move Up
    Move Down
    Move Left
    Move Right
    Attack
    Defend
    Pickup
    Unlock

When an agent dies, it can no longer act in the environment. Moreover, any keys the agents collect are spawned at the agent's last location.

(3) **Rewards:** There is a -0.1 step cost associated with the environment. Moreover, there is a terminal reward of +100 provided to each agent, when the door is unlocked. There are no rewards provided to any dead agents

(4) **Number of Agents:** While the domain can be expanded to handle n agents, we consider only three agents for our experiments.

5 MAREPREL ABSTRACTIONS

MaRePReL employs an abstract state representation for its underlying RL policies, with the specific abstraction varying for each operator within a given domain. The D-FOCI statements corresponding to each operator are utilized to construct its abstract state and observation. Below, we present the abstract state used for each domain

5.1 Multiagent Taxi

The domain has two operators, **pickup** and **drop**. The abstract state representation considers both the passenger as well as the agent. The abstract representation for the operators is provided below

(1) **pickup**: The abstract state representation contains the locations (x, y) for all the agents, as well as the pickup location for the agent (one of R, G, B, and Y represented as a one-hot encoding in the observation), as well as the taxi_id indicating the taxi the passenger is in (0 if the passenger is not in any taxi).

(2) **drop**: The abstract state representation contains the locations (x, y) for all the agents, as well as the taxi_id for the taxi the passenger is in and the drop location for the agent (one of R, G, B, and Y represented as a one-hot encoding in the observation)

5.2 Multiagent Office

The domain has two operators as well, **visitOrPickup** (picking up something is the same as moving to the place of the object in question), and **deliver**. The abstract state representation considers both the agents and the location (objects) in question

(1) **visitOrPickup**: The abstract state representation contains the location (x,y) of all the agents, the current agent's inventory (mail, coffee), as well as the corresponding fact (value indicating which agent performed the task).

(2) **deliver**: The abstract state representation contains the locations (x, y) of all the agents, the object inventory, and the fact corresponding to visit office

5.3 Multiagent Dungeon

Like the other domains, Multiagent Dungeon has two operators as well **attackEnemy** and **getKeyinDoor**. The operators are parameterized by the agent as well as the enemy. The abstract representation for the operators is provided below

(1) **attackEnemy**: The abstract state consists of all the agent locations (x, y), as well as the orientation, hp, and defense of the current agent. Furthermore, we include the current enemy's location (x, y) and hit points.

(2) **getKeyInDoor**: The abstract state consists of all the agent's locations (x, y), the current enemy location (x,y), and its' corresponding key's status.

6 EXPERIMENT HYPERPARAMETERS

The learning algorithm for DQN (independent learners, parameter sharing, and sub-plan embeddings) and MaRePReL is DQN implemented using the ray rllib [1] package in Python. The last baseline used is Q-Mix, which is also implemented using ray rllib. All the hyperparameters used except the ones described are the default parameters provided in ray for DQN or Q-Mix.

6.1 Hyperparameters Dictionary

The different unique parameters and the values assigned to them are described below

- **model[epsilon_timesteps]**: The total number of time steps over which the epsilon is reduced from its initial value to the final value
- **model[final_epsilon]**: The final epsilon value used after the initial reduction for the remainder of the run
- **model[target_network_update_frequency]**: The number of steps after which we do a backward pass to update the target network parameters
- **model[fcnet_hidden]**: It is a list that determines the number and size of the hidden layer for the model. A value of [256, 256] means that there are two hidden layers of size 256 each.
- **model[fcnet_activation]**: The activation function used in the DQN network
- **model[lstm_cell_size]**: The size of the lstm cell used for the model
- **model[max_seq_len]**: The maximum length used while using lstm in the algorithm
- **replay_buffer[type]**: The type of replay buffer used by the DQN to store the trajectories. The one used throughout the experiments is a MultiAgentPrioritizedReplayBuffer. As the name indicates, it is a combination of Prioritized Replay Buffer and Multiagent Replay Buffer. The buffer stores the state-action transitions for the different agents, with the different transitions relative sample frequency determined by metrics such as the TD-error
- **replay_buffer[capacity]**: The number of transitions that could be stored in the buffer
- **optimizer[alpha]**: The learning rate parameter for the network
- **optimizer[epsilon]**: For the optimizer, it is the value added to the gradients to prevent dividing by 0 errors.
- **double_q**: A boolean flag indicating whether to use double DQN instead of the standard DQN algorithm.
- **n_step**: The number of steps used for bootstrapping in multi-step TD learning

[1] https://docs.ray.io/en/latest/rllib/index.html

Hyperparameter	Value
exploration_config[epsilon_timesteps]	10000
exploration_config[final_epsilon]	0.05
model[fcnet_hidden]	[256, 256]
model[fcnet_activation]	relu
target_network_update_frequency	2000
train batch size	128
replay_buffer_[type]	MultiAgentPrioritized ReplayBuffer
lr	0.001
replay_buffer_config[capacity]	300000
double_q	true
n_step	4

Table 3: DQN (Independent Learners, Parameter Sharing, Plan Embeddings) lower level policies hyperparameters for MultiAgent Dungeon

- **train_batch_size**: The size of the train batch used when training the model
- **lr**: The learning rate for the network
- **mixing_embed_dim**: The size of the hidden layer in the mixing dimension in the Q-mix network. Only applicable in the case of Q-Mix

6.2 Hyperparameters for the different models and experiments

Hyperparameter	Value
exploration_config[epsilon_timesteps]	10000
exploration_config[final_epsilon]	0.01
target_network_update_frequency	2000
train batch size	128
replay_buffer[type]	MultiAgentPrioritized ReplayBuffer
replay_buffer[capacity]	300000
model[fcnet_hidden]	[256, 256]
model[fcnet_activation]	relu
double_q	true
n_step	4

Table 1: DQN (Independent Learners, Parameter Sharing, Plan Embeddings) and MaRePReL lower level policies hyperparameters for MultiAgent Taxi

Hyperparameter	Value
exploration_config[epsilon_timesteps]	10000
exploration_config[final_epsilon]	0.01
model[fcnet_hidden]	[256, 256]
model[fcnet_activation]	relu
target_network_update_frequency	500
train batch size	128
replay_buffer_[type]	MultiAgentPrioritized ReplayBuffer
replay_buffer_config[capacity]	300000
double_q	true
n_step	4

Table 2: DQN (Independent Learners, Parameter Sharing, Plan Embeddings) lower level policies hyperparameters for MultiAgent Office

Hyperparameter	Value
mixing_embed_dim	256
optimizer[alpha]	0.99
optimizer[epsilon]	1e-5
model[lstm_cell_size]	64
model[max_seq_len]	200
model[fcnet_hidden]	[256, 256]
model[fcnet_activation]	relu
exploration_config[epsilon_timesteps]	10000
exploration_config[final_epsilon]	0.01
target_network_update_frequency	2000
train batch size	128
replay_buffer_[type]	ReplayBuffer
replay_buffer_config[capacity]	100000
double_q	true

Table 4: QMIX hyperparameters for Multiagent Taxi and Multiagent Office

REFERENCES

[1] Harsha Kokel, Arjun Manoharan, Sriraam Natarajan, Balaraman Ravindran, and Prasad Tadepalli. 2021. Reprel: Integrating relational planning and reinforcement learning for effective abstraction. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 31. ICAPS, Guangzhou, China, 533–541.

[2] Jonas Kvarnström. 2011. Planning for loosely coupled agents using partial order forward-chaining. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 21. ICAPS, Freiburg, Germany, 138–145.