A Illustration for GNN Architecture

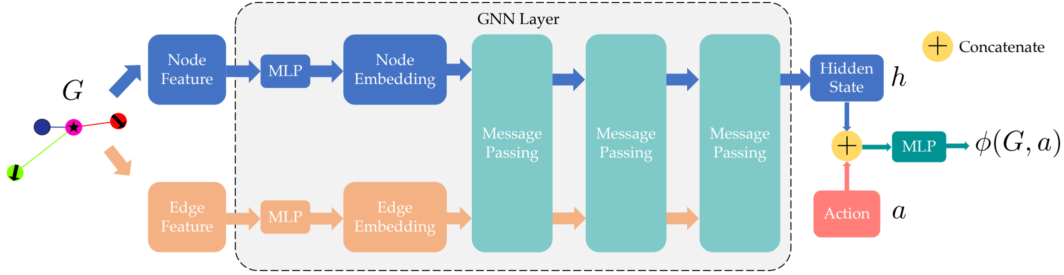

Figure 5: The GNN architecture of the proposed CAM. A state for each agent is an egocentric graph G, which includes the node features and edge features. The CAM first encodes the graph using the GNN layer. The GNN layer then outputs the hidden state h for the current agent based on the updated embedding of the corresponding node. Given action a and another MLP f, the CAM generates the admissibility score using $\phi(G, a) = f(h, a)$.

B Details for the UR5 Environment

Dynamics. Each UR5 agent controls a 5D robot arm. The state $x = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5] \in [-2\pi, 2\pi]^5$, representing the angles on the joints of base to shoulder, shoulder to lift, elbow, wrist 1, and wrist 2. The action is the angular velocity $a = [\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5] \in [-1, 1]^5$, and the dynamics to compute the next state x' are

$$x' = f(x, a) = x + a \cdot dt \tag{3}$$

We choose $dt = 0.1$ in our experiment.

Graph Representation. Each node $v \in V$ is simply a zero scalar. For each agent. Each edge $e : (v_i, v_j) \in E$ is a 18D vector, which concatenates 4 vectors. The first 4D vector is a one-hot vector representing the arm ID of v_i. The second 5D vector represents the state of the arm v_i. The third and fourth vectors follow the same representation as to the first and the second vectors, but those of arm v_j replace the values. The graph is complete for this environment, i.e., each arm can affect any other arm in the environment.

Training Distribution. The training distribution is that only two arms should be in the graph.

Collision and Goal-reaching Criteria. We define collision as the contact or penetration between one arm and another arm or static obstacle. If the state of the arm is within 0.3 distance of the goal, then the goal is reached.

Preference Function ω. The preference function calculates the negative of distance to the goal based on the next state: $\omega(G, \mathcal{T}, a) = -||f(x_G, a) - \mathcal{T}||_2^2$.

Hyperparameters. For each time step, we sample $N = 2000$ candidate actions for every agent. The initial learning rate of the ADAM optimizer is 3e-4, and we decrease the learning rate with a factor of 0.5 when the performance in the validation environment meets a plateau until we reach a minimum learning rate of 1e-4. We set $\gamma_1, \gamma_2, \gamma_3$ as 0, 1e-1, 1e-2. We use a batch size of 256 and update the network every 20 trajectories.

C Details for the Car Environment

Dynamics. Each Car agent follows the Dubins' Car dynamics [51]. The state $x = [p_x, p_y, \theta], p_x, p_y \in \mathbb{R}, \theta \in [0, 2\pi)$, representing the positions on 2D plane and the heading angle. We apply a constant $v = 0.05$ as the velocity for each agent. The action is the angular velocity $a = [\dot{\theta}] \in [-\frac{2}{3}\pi, \frac{2}{3}\pi]$, and the dynamics to compute the next state x' are

$$x' = f(x, a) = x + \begin{bmatrix} v \cdot \sin(\theta + \dot{\theta} \cdot dt) \\ v \cdot \cos(\theta + \dot{\theta} \cdot dt) \\ \dot{\theta} \cdot dt \end{bmatrix} \tag{4}$$

We choose $dt = 1$ in our experiment.

Graph Representation. Each node $v \in V$ is a one-hot vector indicating whether the current node is an agent or a static obstacle. Given each edge $e : (v_i, v_j) \in E$, the feature vector of the edge is $[I, \sin(\theta_i), \cos(\theta_i), \sin(\theta_j), \cos(\theta_j), p_{x,i} - p_{x,j}, p_{y,i} - p_{y,j}]$, where I is a one-hot vector, representing whether the current edge is obstacle-to-agent or agent-to-agent. For the obstacle-to-agent edges, $\sin(\theta_i), \cos(\theta_i)$ are padded as 0.

We define each node's neighbors, including agents and obstacles, as those nodes having a distance within 1.5 to the current node. We only connect edges from these neighbor nodes to each node.

Training Distribution. The training distribution is that only 0-2 neighbor agents and 0-9 neighbor obstacles should be in the graph.

Collision and Goal-reaching Criteria. We define the agent and the obstacle as circles with a radius of 0.15. The goals are circles with a radius of 0.3. The agent is in a collision if it is within a distance of 0.3 to other agents or obstacles. If the agent's position is within a distance of 0.45 to the 2D goal, then the goal is reached.

Preference Function ω**.** The preference function calculates the negative of distance to the goal based on the next state: $\omega(G, \mathcal{T}, a) = -||(p'_x, p'_y) - \mathcal{T}||_2^2, p'_x, p'_y \in f(x_G, a)$.

Hyperparameters. For each time step, we sample $N = 2000$ candidate actions for every agent. The initial learning rate of the ADAM optimizer is 1e-3, and we decrease the learning rate with a factor of 0.5 when the performance in the validation environment meets a plateau until we reach a minimum learning rate of 1e-5. We set $\gamma_1, \gamma_2, \gamma_3$ as 0, 2e-2, 1e-2. We use a batch size of 256 and update the network every 10 trajectories.

D Details for the Dynamic Dubins Environment

Dynamics. The agent follows dynamic Dubins' car and controls a 2D action. The state $x = [p_x, p_y, v, \theta], p_x, p_y \in \mathbb{R}, v \in [0, 1], \theta \in [0, 2\pi)$, representing the position, velocity, and heading angle in the 2D space. The 2D action $a \in \mathcal{A} = [-1, 1]^2$ is composed of the acceleration rate q and the angular velocity $\dot{\theta}$, and the dynamics to compute the next state x' is

$$x' = f(x, a) = x + \begin{bmatrix} v \cdot \sin(\theta + \dot{\theta} \cdot dt) \\ v \cdot \cos(\theta + \dot{\theta} \cdot dt) \\ q \cdot dt \\ \dot{\theta} \cdot dt \end{bmatrix} \tag{5}$$

We choose $dt = 0.05$ in our experiment.

Graph Representation. Each node is a one-hot vector indicating whether the current node is an agent or a static obstacle. Given each edge $e : (i, j) \in E$, the feature vector of the edge is $[I, v_i, v_j, \sin(\theta_i), \cos(\theta_i), \sin(\theta_j), \cos(\theta_j), p_{x,i} - p_{x,j}, p_{y,i} - p_{y,j}]$, where I is a one-hot vector, representing whether the current edge is obstacle-to-agent or agent-to-agent. For the obstacle-to-agent edges, $v_i, \sin(\theta_i), \cos(\theta_i)$ are padded as 0.

We define each node's neighbors, including agents and obstacles, as those nodes having a distance within 1.5 to the current node. We only connect edges from these neighbor nodes to each node.

Training Distribution. The training distribution is that only 0-2 neighbor agents and 0-9 neighbor obstacles should be in the graph.

Collision and Goal-reaching Criteria. We define the agent and the obstacle as circles with a radius of 0.15. The goals are circles with a radius of 0.3. The agent is in a collision if it is within a distance of 0.3 to other agents or obstacles. If the agent's position is within a distance of 0.45 to the 2D goal, then the goal is reached.

Preference Function ω**.** The preference function calculates the negative of distance to the goal based on the next state: $\omega(G, \mathcal{T}, a) = -||(p'_x, p'_y) - \mathcal{T}||_2^2, p'_x, p'_y \in f(x_G, a)$.

Hyperparameters. For each time step, we sample $N = 2000$ candidate actions for every agent. The initial learning rate of the ADAM optimizer is 1e-3, and we decrease the learning rate with a factor of

0.5 when the performance in the validation environment meets a plateau until we reach a minimum learning rate of 1e-5. We set $\gamma_1, \gamma_2, \gamma_3$ as 0, 2e-2, 1e-2. We use a batch size of 256 and update the network every 10 trajectories.

E Details for the Drone Environment

Dynamics. Each Drone agent follows a 3D quadrotor model adopted from [52] and controls a 4D action. The state $x = [p_x, p_y, p_z, v_x, v_y, v_z, \alpha, \beta, \gamma], p_x, p_y, p_z \in \mathbb{R}, v_x, v_y, v_z \in [-1, 1], \alpha, \beta, \gamma \in [-\pi/2, \pi/2]$, representing the position and velocity in the 3D space, and the angles in pitch, roll, and yaw. The action is the instant acceleration q and the angular velocities, i.e., $a = [q, \dot{\alpha}, \dot{\beta}, \dot{\gamma}] \in [-1, 1]^4$, and the dynamics to compute the next state x' is

$$\begin{aligned} f(x, t+dt) &= f(x,t) + \begin{bmatrix} v_x \\ v_y \\ v_z \\ -\sin(\beta) \cdot q \\ \cos(\beta)\sin(\alpha) \cdot q \\ \cos(\beta)\cos(\alpha) \cdot q - g \\ \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix} \cdot dt \\ f(x, 0) &= x \\ x' &= f(x, k \cdot dt) \end{aligned} \tag{6}$$

g is the gravity acceleration of the Earth as 9.8. We choose $dt = 0.01$ and $k = 10$ in our experiment.

Graph Representation. Each node $v \in V$ is a one-hot vector indicating whether the current node is an agent or a static obstacle. Given each edge $e : (v_i, v_j) \in E$, the feature vector of the edge is $[I, v_{x,i}, v_{y,i}, v_{z,i}, \sin(\alpha_i), \cos(\alpha_i), \sin(\beta_i), \cos(\beta_i), v_{x,j}, v_{y,j}, v_{z,j}, \sin(\alpha_j), \cos(\alpha_j), \sin(\beta_j), \cos(\beta_j), p_{x,i} - p_{x,j}, p_{y,i} - p_{y,j}, p_{z,i} - p_{z,j}]$, where I is a one-hot vector, representing whether the current edge is obstacle-to-agent or agent-to-agent. For the obstacle-to-agent edges, $v_{x,i}, v_{y,i}, v_{z,i}, \sin(\alpha_i), \cos(\alpha_i), \sin(\beta_i), \cos(\beta_i), p_{z,i} - p_{z,j}$ are padded as 0.

We define each node's neighbors, including agents and obstacles, as those nodes having a distance within 1.5 to the current node. We only connect edges from these neighbor nodes to each node.

Training Distribution. The training distribution is that only 0-2 neighbor agents and 0-9 neighbor obstacles should be in the graph.

Collision and Goal-reaching Criteria. We define the agent as spheres with a radius of 0.15. The obstacles are vertical cylinders with a radius of 0.15 and infinite height. The goals are spheres with a radius of 0.3. The agent is in a collision if it has a distance within 0.3 to other agents or obstacles in the 3D space. An agent reaches its goal if its 3D position has a distance of 0.45 to its goal.

Preference Function ω. The preference function calculates the negative of an error between the action a and a linear–quadratic regulator (LQR) controller π. The LQR controller π takes the goal and the current state, and gives an action which only considers the goal-reaching: $\omega(G, \mathcal{T}, a) = -||\pi(x_G, \mathcal{T}) - a||_2^2$.

Hyperparameters. For each time step, we sample $N = 2000$ candidate actions for every agent. The initial learning rate of the ADAM optimizer is 1e-3, and we decrease the learning rate with a factor of 0.5 when the performance in the validation environment meets a plateau until we reach a minimum learning rate of 1e-5. We set $\gamma_1, \gamma_2, \gamma_3$ as 0, 1e-1, 1e-2. We use a batch size of 256 and update the network every 20 trajectories.

F Details for the Chasing Game

We provide more details for the chasing game in this section. We define the safety rates of the chasing game in the same way as those of the navigation task. The reward is defined in another way, to reflect on the situation that one Drone agent could drift away from its target agent though it executes

the most preferred action. We want to avoid penalizing such behavior because the most preferred action should not be discouraged if it is also safe at the same time. We define reward as follows: $R = \sum_{t=1}^{T} \text{clip}(d_{t-1} - d_t, 0, 2)$, where d_{t-1} and d_t indicate the distances between the agent and the target agent in 2D or 3D space at $t-1$-th and t-th step. Such a reward keeps track of the agent's improvement at every step while not penalizing on no progress. The preference for the agent follows the same way we defined in the navigation tasks of the Car and Drone. We update the goal for each agent to be the position of the target agent at every step.

G Details for the Experiments

Reproducing the Baselines. The inputs to the networks of DDPG and MACBF are the same as those to the CAM. For DDPG, we use the GNN architectures as the actor and Q-critic network. The GNN actor takes an egocentric graph for each agent and outputs the action. The GNN critic takes the graph and action and outputs the Q value. For MACBF, since it requires differentiable dynamics, we implement an additional dynamic function using PyTorch to guarantee the differentiability. When computing the derivative of the CBF value over time, we assume the graph structure remains the same, i.e., no adding or removing edges. We also find that replacing the temporary dataset of MACBF with the replay buffer, typically used for the RL methods, improves the stability of MACBF.

Notes on why MA-DDPG is not deployed here [12]: Our GNN implementation deviates from the MA-DDPG algorithm because of scalability concerns. In MA-DDPG, the actor network for each agent is independent and does not share the weight. As a result, MA-DDPG requires the same number of agents for the training and inference tasks. However, in our experiment, since we focus on the generalization for scalability, the number of agents during inference varies, while our training tasks are fixed to have three agents. Thus, it is impossible to deploy the MA-DDPG algorithm.

Running the Experiments. All the methods are trained on CPU as Intel Core i7-11700F. The GPU is NVIDIA RTX 3080. We train each method until we reach convergence, which typically requires fewer than two days.

Accelerating the Computation. Though the GNN can be deployed on decentralized devices at inference time, we conduct our experiment on one single desktop, which requires computation optimization. Assume at a time step of the inference time that there are 512 agents in the environment. We need to test 2000 actions for each agent, and each agent samples at least 100 subgraphs. We can infer that there are at least $100 \times 2000 \times 512 \approx 1.02 \times 10^8$ forward operations for the GNN to compute. We need to accelerate such computation; otherwise, the computational cost will be intractable.

We conduct three improvements for the acceleration. The first improvement is to merge the computation of the hidden state for all state-action pairs which share the same input graph into one forward passing in the GNN layer, as mentioned in Section 2.2. The second improvement is to parallelize the computation of the CAM value by stacking the pairs of hidden states and actions as a matrix and then feeding it to the GPU. Such stacking enables the computation with batches of state-action pairs. The third improvement is to choose which agents to compute for each batch adaptively. By inspecting the CAM values of the computed batches, if we have already found admissible actions for one specific agent, then the rest of the state-action pairs for this agent do not need to be computed, and the agent chooses the action among those admissible actions that have been computed. By filtering out the determined agents, we save the unnecessary computation of the state-action pairs of these agents.

H Ablation Study: Varying Obstacle Density

In the main paper, we test the scenarios where the densities of both the agents and the obstacles vary. In this section, we focus on how the obstacle density affects the performance of each method.

We fixed the map size to be 4x4 and increased the obstacle number from 0 to 20. We average the results over 100 randomly generated environments. We do not include the UR5 environment since the obstacle number for UR5 is fixed to be 4. As shown, the CAM significantly outperforms other methods and maintains the safety rate near 100%. On the other hand, MACBF and DDPG show a lack of robustness when the obstacle density increases. We also observe a remarkable degradation of the performance of MACBF for the Drone environment. When there are 0 obstacles, the safety rate of MACBF is close to 99%, but then the safety rate jumps to around 90% when there are 5

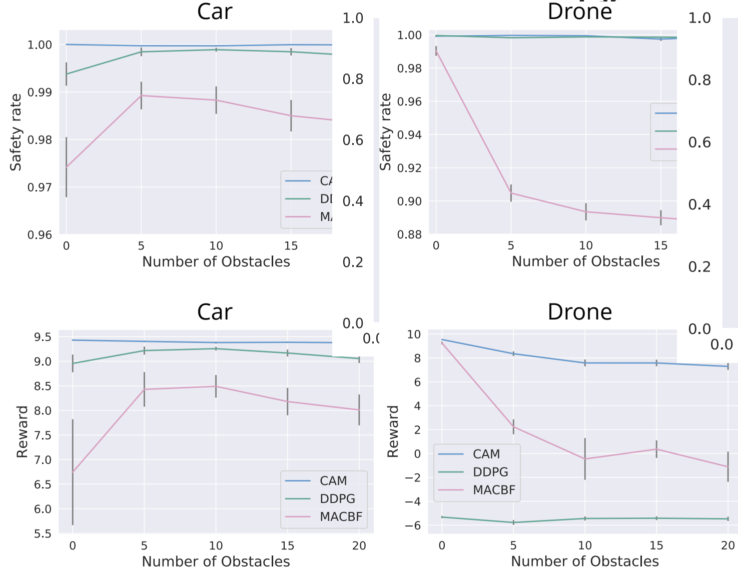

Figure 6: The performance of all methods on Car and Drone with regard to varying obstacle numbers.

obstacles. Such a result shows that MACBF works better to avoid inter-agent collisions and cannot deal well with obstacle-agent collision avoidance. This result explains why the performance of MACBF improves while the density of obstacles decreases in the Drone environment in Figure 4.

I Inference Time

Figure 7 illustrates the running time during inference. We average the running time over all the agents and all the time steps in the test cases. With the three acceleration improvements mentioned in Section G, we can now reduce the running time to around 4.3 milliseconds at most. The computation time can further be improved when deployed in real-world experiments. In that case, each agent can compute the admissible set independently, since our GNN architecture is fully decentralized. It only requires observations of nearby agents and obstacles and does not need inter-agent communication.

We find that in Figure 7, the trends of the running time for Car and Drone are similar - the running time decreases and then increases as the density of agents grows. The running time first decreases, since our method utilizes the GPU and computes the admissible actions of all the agents in parallel. However, when the number of agents grows to a certain threshold between 32 and 128, the memory of the GPU will reach its limitation and cannot take more agents beyond this threshold. If the number of agents is higher than this threshold, we need multiple GPU forward passes to cover all the agents. As a result, the running time increases afterward. More GPU memory should make the computation even faster in such a centralized computation scenario.

J Details for the Single Agent Example

In this section, we provide an additional single-agent experiment as a proof of concept. This example has more complex dynamics than the one in the main paper. We provide the details as follows.

Dynamics. The agent here follows the single integrator and controls a 2D action. The state $x = [p_x, p_y]$, representing the position in the 2D space. The 2D action $a = [v_x, v_y] \in \mathcal{A} = [-1, 1]^2$ is composed of the velocities along x and y axis, and the dynamics to compute the next state x' is

$$x' = f(x, a) = x + \begin{bmatrix} v_x \\ v_y \end{bmatrix} \cdot dt \tag{7}$$

We choose $dt = 0.05$ in our experiment. In this example, we use an MLP to represent CAM, since the state here is solely a vector.

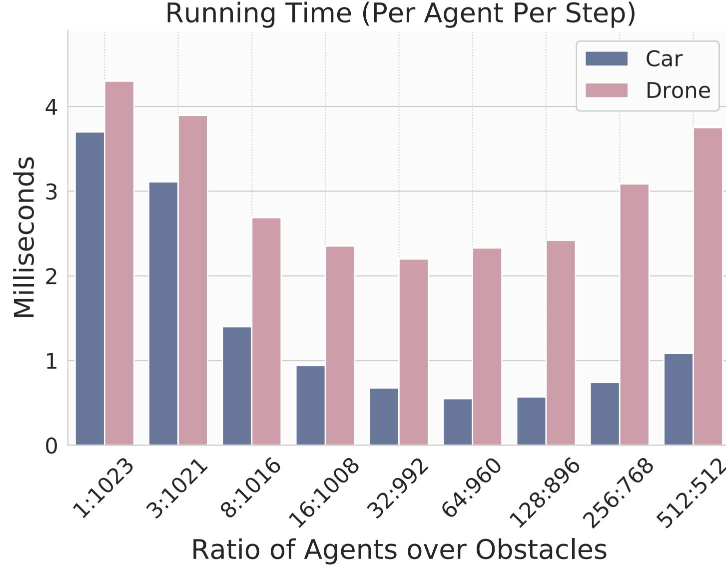

Figure 7: The average running time of our method on Car and Drone during inference for each agent at each time step (in milliseconds). The running time is averaged over the 100 test cases.

Task Distribution. The goal state $[p_x = 0, p_y = 1.7]$, and the obstacles are all fixed during training and test. The starting state of the agent is fixed as $[p_x = 0, p_y = -1.7]$.

Collision and Goal-reaching Criteria. We define the agent as a circle with a radius of 0.15. The goals are circles with a radius of 0.15. The agent is in a collision if it intersects with the obstacles. If the agent intersects with the 2D goal, then the goal is reached.

Preference Function ω. The preference function calculates the negative of distance to the goal based on the next state: $\omega(x, \mathcal{T}, a) = -||(p'_x, p'_y) - \mathcal{T}||_2^2, [p'_x, p'_y] = x' = f(x, a)$.

J.1 Forward-Invariance Property

To further investigate whether the learned CAM holds the forward-invariance property, we illustrate the boundary of the CAM $\{x \mid \exists a_1, a_2 \in \mathcal{A}, \phi(x, a_1) \geq 0, \phi(x, a_2) < 0\}$ in Figure 8.

We show that the trajectory never leaves the admissible region of CAM, i.e., the trajectory holds the forward-invariance property. Once the trajectory meets the boundary, it starts to follow the most preferred action among admissible actions, instead of the most preferred action among all sampled actions. Meanwhile, we find that around 0.9% states in the boundary states violate the forward-invariance property, i.e., they enter the inadmissible region after executing the most admissible action.

We emphasize that since we only train on transitions from collected trajectories, CAM can be asymmetric and violate the forward-invariance property at some unseen states. We design the CAM to maintain the forward invariance and focus on states that could exist along possible trajectories. If the agent does not meet some states at any time step during inference, then we do not require these states to obey the forward invariance. Empirically, we have shown that as long as the training tasks and inference tasks are under the same distribution, the forward invariance often holds for inference trajectories once the CAM learns from enough training data.

K Additional Single Agent Environment: Dynamic Dubins

In this section, we provide an additional single-agent experiment as a proof of concept. This example has more complex dynamics than the one in the main paper.

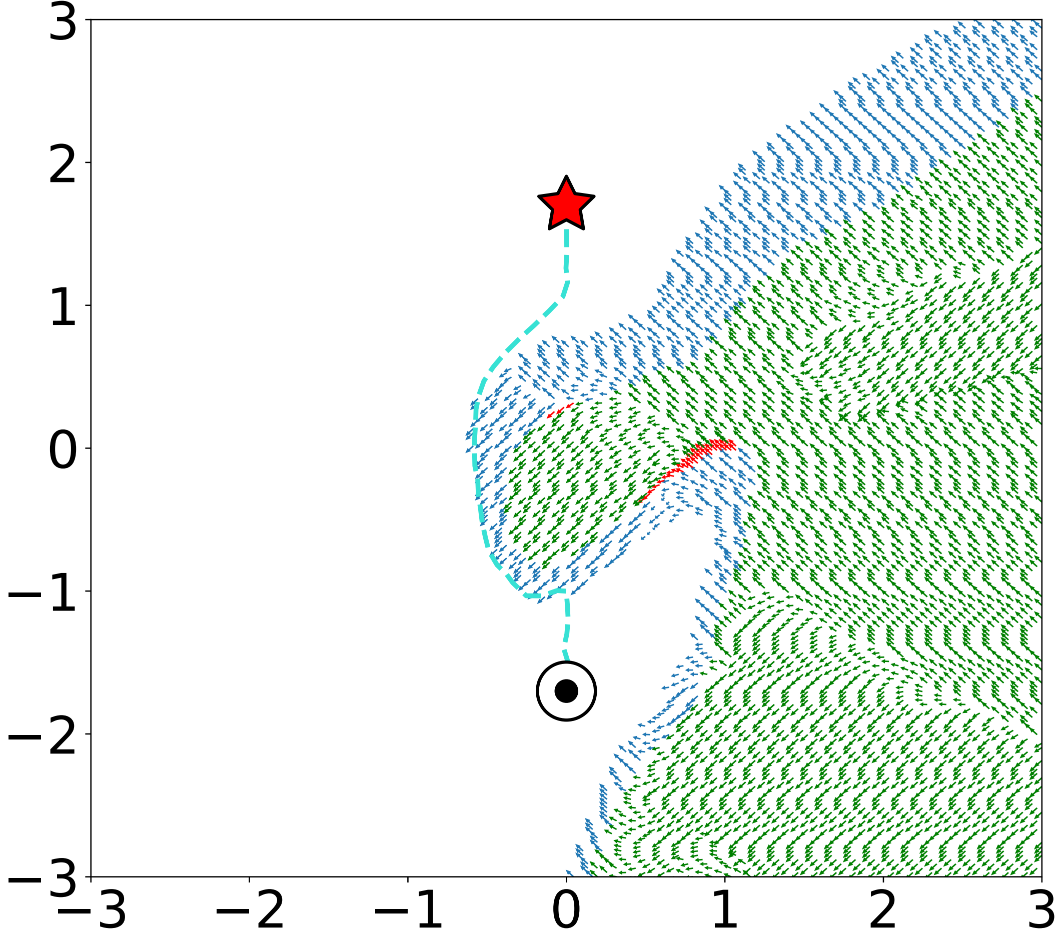

Figure 8: We show that the forward-invariance property holds along the trajectory. The direction of each arrow indicates the most admissible action to take on each state x: $\arg\max_{a\in\mathcal{A}} \phi(x,a)$. The green arrows represent the states in the inadmissible region $\{x \mid \forall a \in \mathcal{A}, \phi(x,a) < 0\}$. Both the blue arrows and the red arrows represent the states on the boundary: $\{x \mid \exists a_1, a_2 \in \mathcal{A}, \phi(x,a_1) \geq 0, \phi(x,a_2) < 0\}$. The blue arrows are those boundary states that hold the forward-invariance property. The red arrows are those boundary states that enter the inadmissible region after executing the most admissible action. Note: Since we only train on transitions from collected trajectories, CAM can be asymmetric and violate the forward-invariance property at some unseen states.

K.1 Forward-Invariance Property

We sample around 40 trajectories starting from the initial states, and illustrate the collected transitions in Figure 9. The direction of each arrow indicates the action that CAM agent takes on each state x. The blue arrows represent the admissible region $\{x \mid \forall a \in \mathcal{A}, \phi(x,a) \geq 0\}$. Both the red and the green arrows represent the states on the boundary: $\{x \mid \exists a_1, a_2 \in \mathcal{A}, \phi(x,a_1) \geq 0, \phi(x,a_2) < 0\}$. Only the red arrows represent those boundary transitions that violate the forward-invariance property. The black arrows represent the states in the inadmissible region $\{x \mid \forall a \in \mathcal{A}, \phi(x,a) < 0\}$.

We show that around 98.6% of transitions along the trajectories are admissible and forward invariant. There are around 27.7% of states on the boundary. Around 0.4% of states violate the forward-invariance property, i.e., the CAM agent crosses the boundary and enters the inadmissible region after executing the action. There are 1% of transitions that are inadmissible, but all of them return back to the admissible regions after executing sequences of most admissible actions.

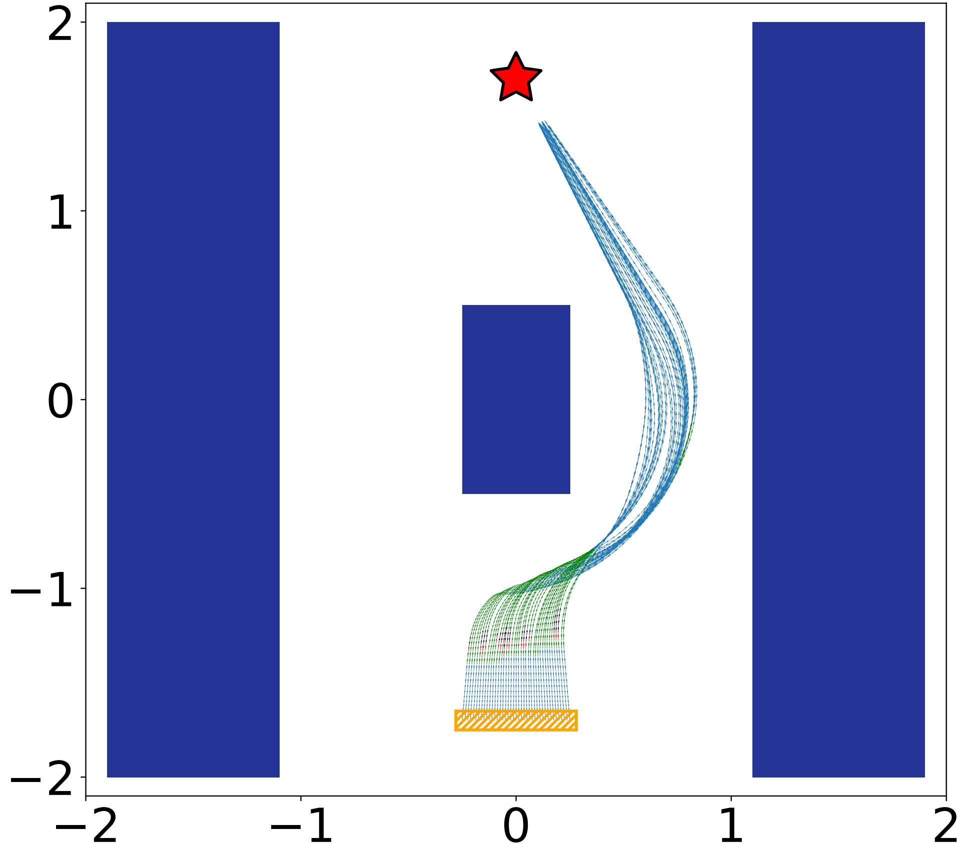

Figure 9: An additional single-agent example for the proposed CAM dealing with more complex dynamics, where the agent needs to catch failure early on. The red star represents the goal state, and three blue boxes represent the obstacles. The orange region denotes the set of initial states. We illustrate state-action pairs (x, a) on sampled trajectories starting from initial states. All trajectories reach the goal successfully with no collisions. The direction of each arrow indicates the action that CAM agent takes on each state x. The blue arrows represent the admissible region $\{x \mid \forall a \in \mathcal{A}, \phi(x, a) \geq 0\}$. Both the red and the green arrows represent the states on the boundary: $\{x \mid \exists a_1, a_2 \in \mathcal{A}, \phi(x, a_1) \geq 0, \phi(x, a_2) < 0\}$. Only the red arrows represent those boundary transitions that violate the forward-invariance property. The black arrows represent the states in the inadmissible region $\{x \mid \forall a \in \mathcal{A}, \phi(x, a) < 0\}$.

K.2 Details for the Additional Single Agent Environment

Dynamics. The agent here is one dynamic Dubins' car and controls a 2D action. The state $x = [p_x, p_y, v, \theta], p_x, p_y \in \mathbb{R}, v \in [0, 1], \theta \in [0, 2\pi)$, representing the position, velocity, and heading angle in the 2D space. The 2D action $a \in \mathcal{A} = [-1, 1]^2$ is composed of the acceleration rate q and the angular velocity $\dot{\theta}$, and the dynamics to compute the next state x' is

$$x' = f(x, a) = x + \begin{bmatrix} v \cdot \sin(\theta + \dot{\theta} \cdot dt) \\ v \cdot \cos(\theta + \dot{\theta} \cdot dt) \\ q \cdot dt \\ \dot{\theta} \cdot dt \end{bmatrix} \tag{8}$$

We choose $dt = 0.05$ in our experiment. In this example, we use an MLP to represent CAM, since the state here is solely a vector.

Note that the maximum acceleration rate is saturated here. At each time step, the agent can change the velocity by increasing or decreasing 0.05 at most. As a result, the agent needs to catch failure early on.

Task Distribution. The goal state ($p_x = 0, p_y = 1.7$), and the obstacles are all fixed during training and test. The starting state of the agent is randomized within the ditribution of $p_x \in [-0.25, 0.25], p_y = -1.7, v = 0, \theta = \frac{\pi}{2}$.

Collision and Goal-reaching Criteria. We define the agent as a circle with a radius of 0.15. The goals are circles with a radius of 0.15. The agent is in a collision if it intersects with the obstacles. If the agent intersects with the 2D goal, then the goal is reached.

Preference Function ω. The preference function calculates the negative of distance to the goal based on the next state: $\omega(x, \mathcal{T}, a) = -||(p'_x, p'_y) - \mathcal{T}||_2^2, p'_x, p'_y \in f(x, a)$.

K.3 Performance Progress in the Additional Single Agent Environment

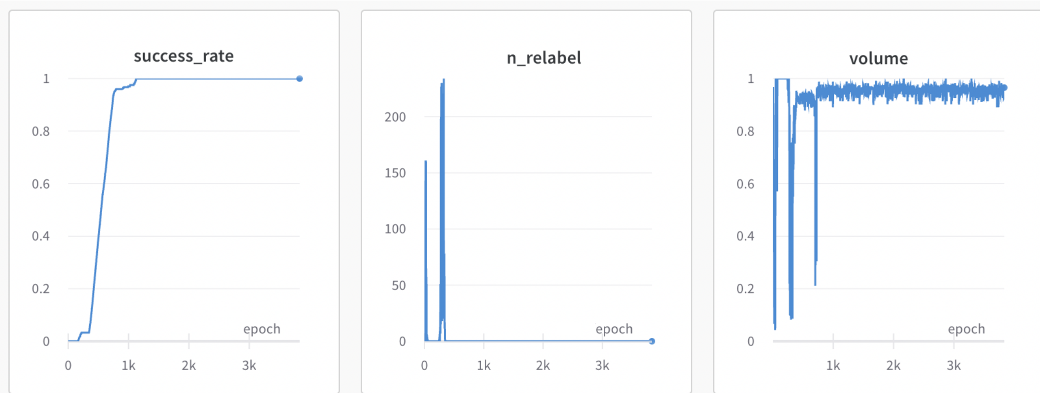

Figure 10: The progress of the performance during training. The x-axis for each figure is the number of sampled trajectories during training. From left to right: **(i)** Averaged success rate over a window of the nearest 20 trajectories. A trajectory succeeds if and only if the agent reaches the goal with no collision. **(ii)** The number of relabelled transitions along each trajectory. **(iii)** The ratio of admissible actions $p(\phi(x, a) \geq 0), a \sim \mathcal{A}$, averaged over all states $\{x\}$ along each trajectory.

After the training converges, we sample the initial state from $p_x = -0.25$ to $p_x = 0.25$ with an interval equal to 0.05. As shown in Figure 9, all the sampled trajectories reach the goal successfully with no collisions. We provide the video in the supplementary material with more details.

Additionally, we show the progress during training in Figure 10. We observe that a great improvement in the success rate happens right after our algorithm relabels a relatively high number of transitions. This observation is consistent with our relabelling mechanism design, which aims to catch the failure early using backtracing.

We also observe a slight improvement starting around the 700-th trajectory, where the success rate increases gradually from 95% to 100%. This happens at the same time when the ratio of admissible actions drops. We infer that at that specific time, the agent has explored some new states, which are critical for leading the agent to success. Though the agent is not familiar with these states initially, by collecting these new experiences and learning from them, the ratio becomes stable again and the success rate grows afterward.