Eötvös Loránd University – Faculty of Informatics
ELTE Collective Intelligence Course — Assignment 2

Search and Rescue Multi-Agent Simulation

Technical Documentation

2025/26/1

Tutor:

Takács Tamás
Barta Zoltán

Team Members:

Máté Kovács	(u5bky4@inf.elte.hu)
Adorján Nagy-Mohos	(d5vd5e@inf.elte.hu)
Sándor Baranyi	(ct9xfj@inf.elte.hu)

BUDAPEST
December 23, 2025

Contents

1 Introduction

This document provides an in-depth overview of the **Search and Rescue Multi-Agent Simulation**, which utilizes the PettingZoo library for modeling a cooperative multi-agent environment. The simulation is designed to replicate dynamic rescue operations where rescuers navigate an environment containing obstacles, victims, and predefined safe zones to achieve specific objectives.

The primary goal of this project is to develop and evaluate multi-agent reinforcement learning algorithms in a challenging cooperative setting that requires:

- **Coordination**: Multiple agents must work together efficiently without explicit communication.
- **Partial Observability**: Agents have limited vision and must navigate with incomplete information.
- **Credit Assignment**: Individual agents must learn their contribution to team success.
- **Dynamic Interactions**: Victims respond to agent behavior through a commitment mechanism.

The following chapters describe the functionalities, components, mathematical formulations, and implementation details of the project.

2 Functionalities

This project presents a comprehensive simulation of a Search and Rescue scenario, designed within a multi-agent environment using the PettingZoo library. The environment models a dynamic rescue operation where cooperative agents must navigate a space containing both obstacles and targets to achieve predefined goals. The simulation incorporates the following components:

2.1 Environment Setup

The environment is configured through a set of parameters that define the simulation characteristics:

- **Configurable Parameters**: Users can define key elements of the environment, including:
  - Number of rescuers (agents responsible for guiding victims).
  - Number of victims (missing agents to be rescued).
  - Number of trees (obstacles obstructing navigation).
  - Number of safe zones (designated target areas for rescue).
- **Agent Interaction**: Rescuers interact with victims using a commitment-based following mechanism. Upon approaching a victim within the follow radius, the victim commits to following that rescuer. This commitment system implements hysteresis to prevent rapid switching between rescuers, providing stable victim-agent relationships.
- **Safe Zone Placement**: Safe zones are fixed at the four corners of the environment. Each zone is assigned a specific type (e.g., A, B, C, D), and only victims of the matching type are considered successfully rescued when they reach the zone. This type-matching requirement introduces an additional layer of strategic planning for the agents.

2.2 Rescuers (Agents)

Rescuers are intelligent agents that serve as the primary actors in the simulation. They exhibit the following characteristics:

- **Motion Constraints**: Operate under physical constraints such as limited speed, acceleration, and maneuverability. The maximum speed is capped at 0.08 units per timestep, and actions are applied through a velocity-based control scheme.
- **Reactive Behavior**: Demonstrate dynamic behavior by interacting with victims and avoiding obstacles. Agents must adapt their trajectories in real-time based on changing environmental conditions.
- **Commitment Mechanism**: Use a commitment-based system to guide victims. When a rescuer approaches a victim within the follow radius, the victim becomes committed to following that rescuer until either rescued or the rescuer moves too far away.
- **Collision Response**: Agents implement collision detection and response mechanisms for walls, trees, and other agents. Tree collisions result in velocity reflection with damping, while agent-agent proximity triggers soft repulsion forces.

2.3 Victims (Missing Agents)

Victims represent passive entities scattered within the environment that require rescue. Their characteristics include:

- **Type Assignment**: Each victim is assigned a specific type (A, B, C, or D) that corresponds to safe zone types. Types are assigned cyclically: victim i receives type $i \mod 4$.
- **Matching Requirement**: Victims must be guided to their matching safe zones to be considered rescued. A victim of type A can only be saved at the type A safe zone.
- **Following Behavior**: When committed to a rescuer, victims follow the assigned agent with a following force of 0.03 units. The victim velocity is updated as shown in Equation 1.

$$\mathbf{v}_{\text{victim}}^{t+1} = 0.8 \cdot \mathbf{v}_{\text{victim}}^{t} + 0.03 \cdot \hat{\mathbf{d}}_{\text{to\_rescuer}} \tag{1}$$

- **Brownian Motion**: When not committed to any rescuer, victims exhibit simple Brownian motion with noise magnitude 0.0075, simulating random wandering behavior.
- **Commitment Hysteresis**: To prevent rapid switching between rescuers, the commitment system implements hysteresis. A victim releases its current assignment only when the assigned rescuer moves beyond 1.5 times the follow radius, and switches to a new rescuer only if the new rescuer is significantly closer (less than 0.6 times the current distance).

2.4 Safe Zones

Safe zones serve as stationary rescue targets for victims and define the success criteria:

- **Corner Placement**: Safe zones are predefined at the four corners of the environment at positions:
  - Type 0 (A): Top-left $(-0.9, 0.9)$
  - Type 1 (B): Top-right $(0.9, 0.9)$
  - Type 2 (C): Bottom-left $(-0.9, -0.9)$

  - Type 3 (D): Bottom-right $(0.9, -0.9)$

- **Type Identification**: Each safe zone is assigned a unique type, represented by distinct colors (red, green, blue, yellow) for visual identification during rendering.
- **Rescue Condition**: Successful rescues occur only when victims of matching types reach their corresponding safe zones. The rescue is triggered when the Euclidean distance between victim and safe zone falls below the rescue radius of 0.15 units.
- **Optional Randomization**: The environment supports optional randomization of safe zone types while maintaining corner positions, controlled by the `randomize_safe_zones` parameter.

2.5 Obstacles (Trees)

Trees are static obstacles that add complexity to agent navigation:

- **Random Distribution**: Trees are randomly distributed within the environment during initialization and maintain fixed positions throughout an episode. The random placement uses uniform sampling within the range $[-0.8, 0.8]$.
- **Collision Penalty**: Agents are penalized for colliding with trees. Collisions are detected using Euclidean distance between agent and tree centers, with collision occurring when the condition in Equation 2 is satisfied.

$$\|\mathbf{x}_{\text{agent}} - \mathbf{x}_{\text{tree}}\| < r_{\text{agent}} + r_{\text{tree}} \tag{2}$$

  where $r_{\text{agent}} = 0.03$ and $r_{\text{tree}} = 0.05$.

- **Path Obstruction**: Obstacles add complexity by blocking direct paths between rescuers and victims or safe zones. This requires agents to develop navigation strategies that avoid obstacles while pursuing objectives.
- **Line-of-Sight Blocking**: Trees obstruct the line-of-sight between agents and other entities. When an entity is occluded by a tree, it is masked in the observation with zeros, introducing partial observability into the environment.

2.6 Collision Avoidance

The environment implements comprehensive collision avoidance mechanisms:

- **Agent-Agent Avoidance**: Rescuers apply soft repulsion forces when within proximity (0.15 units) of each other. The repulsion force magnitude is given by Equation 3.

$$F_{\text{repulsion}} = 0.005 \cdot \frac{r_{\text{threshold}} - d}{d} \tag{3}$$

  where d is the distance between agents. This prevents clustering while maintaining smooth agent motion.

- **Obstacle Avoidance**: Agents are penalized (-1.0 reward) for colliding with trees. Upon collision, the agent's velocity is reflected about the collision normal with damping factor of approximately 0.5, as shown in Equation 4.

$$\mathbf{v}' = \mathbf{v} - 1.5(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \tag{4}$$

- **Wall Collision Handling**: When agents cross world boundaries $[-1, 1]$, their position is clamped to the boundary and the corresponding velocity component is inverted with 0.5 damping, as shown in Equation 5.

$$v_{\text{axis}} \leftarrow -0.5 \cdot v_{\text{axis}} \tag{5}$$

- **Line-of-Sight Blocking**: Trees obstruct vision between agents. The occlusion check determines whether a tree's bounding region intersects the line segment between observer and target. When visibility is blocked, agents must navigate indirectly.
- **Reactive Behavior**: The environment promotes indirect navigation strategies, such as moving tangentially to obstacles, to simulate realistic path planning behaviors.

2.7 Multi-Agent Interaction and Dynamics

The multi-agent aspects of the environment create complex emergent behaviors:

- **Agent Collaboration**: Although agents operate independently with decentralized policies, their shared objective of rescuing victims promotes emergent collaborative behaviors, such as efficient task distribution and implicit role assignment.
- **Motion Dynamics**: Agents operate under physical constraints including speed limits and acceleration. The velocity update equation is given by Equation 6.

$$\mathbf{v}^{t+1} = 0.8 \cdot \mathbf{v}^t + 0.1 \cdot \mathbf{a}^t \tag{6}$$

  where $\mathbf{a}^t \in [-1, 1]^2$ is the action (acceleration) and velocity is clipped to maximum speed 0.08.
- **Credit Assignment Challenge**: With multiple agents contributing to team success, the environment presents a credit assignment problem. The reward structure is designed to attribute rewards to individual agents based on their specific contributions (e.g., the assigned rescuer receives the rescue bonus).
- **Homogeneous Agents**: All rescuer agents share the same capabilities and policy parameters, promoting the learning of general strategies that work across different agent positions and roles.

2.8 Training and Evaluation Framework

The project provides comprehensive training and evaluation capabilities:

- **Training Framework**: The simulation uses TorchRL's implementation of Multi-Agent PPO (MAPPO) for training. Key components include:
  - SyncDataCollector for parallel experience collection.
  - ClipPPOLoss with configurable clipping threshold.
  - Generalized Advantage Estimation (GAE) for advantage computation.
  - Adam optimizer with configurable learning rate.
- **Curriculum Learning**: Optional curriculum learning support allows for progressive difficulty adjustment:
  - Training starts with fewer obstacles (configurable minimum).

- Difficulty gradually increases through defined stages.
- Each stage maintains consistent observation space size.

- **Configurable Timesteps**: Users can adjust the training duration through the `total_timesteps` parameter to control convergence and computational requirements.
- **Performance Logging**: Comprehensive logging is provided through:
  - **TensorBoard**: Provides real-time performance visualization including loss curves, reward trajectories, and training statistics.
  - **Checkpoint Saving**: Model weights and configuration are saved for later evaluation and fine-tuning.
- **Evaluation Pipeline**: The evaluation system supports:
  - Manual model selection or automatic detection of latest checkpoint.
  - Environment configuration loading from saved checkpoints.
  - Per-episode metrics collection and aggregation.
  - Real-time rendering for visual verification.

2.9 Reward System

The reward system governs agent behavior through a combination of positive rewards and penalties.

2.9.1 Positive Rewards

- **Successful Rescue**: A large reward of +100.0 is awarded to the rescuer who was assigned to (escorting) the victim when it reaches its matching safe zone. This is formalized in Equation 7.

$$R_{\text{success}} = \begin{cases} 100.0 & \text{if victim reaches matching safe zone} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

- **Escort Proximity Reward**: The assigned rescuer receives a bounded exponential reward based on how close the victim is to its target safe zone, as shown in Equation 8.

$$R_{\text{escort\_proximity}} = \exp\left(-\frac{d_{\text{victim-safezone}}}{0.5}\right) \tag{8}$$

  This reward is bounded in $[0, 1]$, providing stable gradients without blow-up near the target.

- **Escort Delta Shaping**: The assigned rescuer receives additional reward for reducing the victim-to-zone distance, as shown in Equation 9.

$$R_{\text{escort\_delta}} = 0.5 \cdot (d_{\text{prev}} - d_{\text{current}}) \tag{9}$$

- **Pickup Shaping**: Unassigned rescuers (not currently escorting any victim) receive shaping rewards for approaching unassigned victims:
  - **Distance Penalty**: $R_{\text{pickup\_dist}} = -0.1 \cdot d_{\text{to\_nearest\_unassigned}}$
  - **Delta Reward**: $R_{\text{pickup\_delta}} = 0.2 \cdot (d_{\text{prev}} - d_{\text{current}})$

2.9.2 Penalties

- **Idle Penalty**: Rescuers with speed below 10^{-3} receive a penalty of -0.01 per step to encourage active exploration. This is formalized in Equation 10.

$$R_{\text{idle}} = \begin{cases} -0.01 & \text{if } \|\mathbf{v}\| < 10^{-3} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

- **Tree Collision**: A penalty of -1.0 is applied when an agent collides with a tree obstacle.
- **Boundary Violation**: A penalty of -0.2 is applied when an agent's position exceeds 0.95 in any dimension (softened to reduce boundary-avoidance bias).
- **Agent Collision**: A penalty of -1.0 is applied to both agents when they are within collision distance (0.15 units) to discourage clustering. This was softened from -5.0 to reduce training instability.
- **Energy Cost** (when energy system is enabled): Movement incurs energy cost which is subtracted as negative reward:

$$R_{\text{energy}} = -(\text{idle\_cost} + \text{movement\_cost\_coeff} \times \|\mathbf{a}\|) \tag{11}$$

  where idle_cost = 0.001 and movement_cost_coeff = 0.01.

2.9.3 Reward Summary Table

Event	Reward	Recipient
Successful rescue	$+100.0$	Assigned rescuer
Escort proximity	$+\exp(-d/0.5)$	Assigned rescuer (per step)
Escort delta	$+0.5 \times \Delta d$	Assigned rescuer (progress)
Pickup delta	$+0.2 \times \Delta d$	Unassigned rescuers
Pickup distance	$-0.1 \times d$	Unassigned rescuers
Idle penalty	-0.01	Stationary rescuer (per step)
Tree collision	-1.0	Colliding rescuer
Boundary violation	-0.2	Violating rescuer
Agent collision	-1.0	Both colliding rescuers
Energy cost	$-(0.001 + 0.01\|\mathbf{a}\|)$	All rescuers (if enabled)

Table 1: Complete reward structure for the Search and Rescue environment.

2.9.4 Combined Reward Function

The total reward for a rescuer agent i at timestep t is given by Equation 12.

$$R_i^t = R_{\text{success}} + R_{\text{escort}} + R_{\text{shaping}} + R_{\text{velocity}} - R_{\text{penalties}} \tag{12}$$

where $R_{\text{penalties}}$ aggregates collision and boundary penalties.

2.10 Observation Space

Each rescuer agent receives a structured observation vector that provides local information about the environment. The observation uses a **partial observability** model where entities beyond the vision radius or occluded by trees are masked.

2.10.1 Observation Components

1. **Self State** (4 values):
   - Velocity: $[v_x, v_y]$
   - Position: $[x, y]$
2. **Agent ID** (N_{rescuers} values):
   - One-hot encoding of agent index for symmetry breaking.
   - Allows shared policy to differentiate between agents.
3. **Energy** (1 value, if energy system enabled):
   - Normalized energy level in $[0, 1]$.
   - Computed as $e_{\text{norm}} = e_{\text{current}}/e_{\text{max}}$.
4. **N Closest Landmarks** ($N_{\text{landmarks}} \times 5$ values):
   - The observation includes the N closest landmarks (safe zones and visible trees combined).
   - Each landmark has 5 values: $[x_{\text{rel}}, y_{\text{rel}}, \text{visible}, \text{is\_safezone}, \text{type}]$
   - Safe zones are always visible; trees are only included if within vision radius and not occluded.
   - Unused slots (when fewer than N landmarks are visible) are zero-padded.
5. **Victims** ($N_{\text{victims}} \times 4$ values):
   - Relative position: $[x_{\text{rel}}, y_{\text{rel}}]$ (normalized by world size)
   - Type index: victim type $\in \{0, 1, 2, 3\}$
   - Visible bit: 1.0 if visible, 0.0 otherwise
   - Masked as $[0, 0, 0, 0]$ if not visible or already saved.
6. **Other Rescuers** ($(N_{\text{rescuers}} - 1) \times 3$ values):
   - Relative position: $[x_{\text{rel}}, y_{\text{rel}}]$ (normalized by world size)
   - Visible bit: 1.0 if visible, 0.0 otherwise
   - Masked as $[0, 0, 0]$ if not visible (beyond vision radius or occluded).

2.10.2 Observation Dimension

The total observation dimension is given by Equation 13.

$$D_{\text{obs}} = 4 + N_{\text{rescuers}} + \mathbb{1}_{\text{energy}} + 5 \times N_{\text{landmarks}} + 4 \times N_{\text{victims}} + 3 \times (N_{\text{rescuers}} - 1) \tag{13}$$

where $\mathbb{1}_{\text{energy}} = 1$ if energy system is enabled, 0 otherwise.

Example (default configuration: 6 rescuers, 12 victims, 9 landmarks, energy enabled):

$$\begin{aligned} D_{\text{obs}} &= 4 + 6 + 1 + (9 \times 5) + (12 \times 4) + (5 \times 3) \\ &= 4 + 6 + 1 + 45 + 48 + 15 = 119 \end{aligned}$$

2.10.3 Visibility and Masking

Observations are partially masked based on visibility constraints:

- Entities beyond the vision radius (default: 0.4) are masked with zeros.
- Entities occluded by trees (line-of-sight blocking via ray-circle intersection) are masked.
- Saved victims are masked to indicate they no longer need rescue.
- Safe zones are always visible regardless of distance (strategic information).

The mathematical observation for agent i is given by Equation 14.

$$\mathbf{o}_i = [\mathbf{v}_i, \mathbf{x}_i, \mathbf{e}_i^{\text{id}}, e_i^{\text{energy}}, \mathbf{L}_i^{\text{landmarks}}, \mathbf{V}_i^{\text{victims}}, \mathbf{R}_i^{\text{rescuers}}] \tag{14}$$

where relative positions are normalized by world size as shown in Equation 15.

$$\mathbf{x}_{\text{rel}} = \frac{\mathbf{x}_{\text{entity}} - \mathbf{x}_i}{\text{world\_size}} \tag{15}$$

3 Structure of the Code

The project is organized into modular components, each responsible for a specific aspect of the simulation pipeline. This section provides detailed documentation of each module.

3.1 Custom Environment: sar_env.py

The `sar_env.py` file defines the multi-agent search-and-rescue environment using the PettingZoo ParallelEnv API. It models agents (rescuers and victims), landmarks (trees and safe zones), and their interactions.

3.1.1 Environment Dynamics

The environment operates in a 2D continuous space where each agent i has a position $\mathbf{x}_i$ and velocity $\mathbf{v}_i$. These are defined in Equation 16.

$$\mathbf{x}_i = [x_i, y_i], \quad \mathbf{v}_i = [v_{x_i}, v_{y_i}] \tag{16}$$

Positions are updated according to velocity as shown in Equation 17.

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t) \tag{17}$$

The world boundaries are defined as shown in Equation 18.

$$x_{\text{min}} \leq x_i \leq x_{\text{max}}, \quad y_{\text{min}} \leq y_i \leq y_{\text{max}} \tag{18}$$

where $x_{\text{min}} = y_{\text{min}} = -1$ and $x_{\text{max}} = y_{\text{max}} = 1$.
Agents are penalized and their velocity is reflected if they move out of bounds.

3.1.2 Collision Detection

Collisions occur when the Euclidean distance between two entities is less than the sum of their radii, as shown in Equation 19.

$$\|\mathbf{x}_1 - \mathbf{x}_2\| \leq r_1 + r_2 \tag{19}$$

where r_1 and r_2 are the radii of the entities.

```python
def is_collision(self, pos1, pos2, r1, r2):
    delta_pos = pos1 - pos2
    dist = np.sqrt(np.sum(np.square(delta_pos)))
    dist_min = r1 + r2
    return dist < dist_min
```

Listing 1: Collision detection implementation.

3.1.3 Reward Mechanism Implementation

The reward computation is implemented in the `_compute_rewards` method. The system uses assignment-aware shaping that dynamically switches between pickup and escort modes.

```python
def _compute_rewards(self, rewards):
    # 1) PICKUP shaping (unassigned agents -> nearest unassigned victim
        )
    for agent in agents:
        if agent_is_escorting[agent]:
            continue  # escort shaping below
        d = nearest_unassigned_victim_dist(agent)
        rewards[agent] -= 0.1 * d  # distance penalty
        rewards[agent] += 0.2 * (prev_dist - d)  # delta shaping

    # 2) SAVE events (sparse success reward)
    for victim in victims:
        if not saved and dist_to_matching_zone < safe_zone_radius:
            saved[victim] = True
            rewards[assigned_rescuer] += 100.0

    # 3) ESCORT shaping (assigned agent -> bring victim to zone)
    for victim in unsaved_victims:
        if has_assignment:
            # Bounded exponential proximity reward
            shaped = exp(-dist_to_zone / 0.5)
            rewards[assigned_agent] += 1.0 * shaped
            # Delta shaping for progress
            rewards[assigned_agent] += 0.5 * (prev_zone_dist -
                dist_to_zone)

    # 4) Boundary penalties
    for agent in agents:
        if abs(pos[0]) > 0.95 or abs(pos[1]) > 0.95:
            rewards[agent] -= 0.2

    # 5) Agent collision penalties
    for pair in agent_pairs:
        if dist < 0.15:
            rewards[agent_i] -= 1.0
            rewards[agent_j] -= 1.0

    # 6) Idle penalty
    for agent in agents:
        if velocity[agent] < 1e-3:
            rewards[agent] -= 0.01
```

Listing 2: Reward computation pseudocode.

3.1.4 Observation Space Construction

The observation for agent i is constructed by concatenating the relevant components. The implementation uses an N-closest-landmarks approach for scalable observation sizes.

```
def _get_obs(self, agent_idx):
    obs = []
    my_pos = rescuer_pos[agent_idx]

    # 1. Self state (velocity, position)
    obs.extend(rescuer_vel[agent_idx])    # 2 values
    obs.extend(my_pos)                    # 2 values

    # 2. Agent ID one-hot
    one_hot = np.zeros(num_rescuers)
    one_hot[agent_idx] = 1.0
    obs.extend(one_hot)                   # num_rescuers values

    # 3. Energy (if enabled)
    if energy_enabled:
        e_norm = energy[agent_idx] / max_energy
        obs.append(clip(e_norm, 0, 1))    # 1 value

    # 4. N closest landmarks (safe zones + visible trees)
    landmarks = []
    for zone in safe_zones:
        rel = (zone.pos - my_pos) / world_size
        landmarks.append((dist, rel_x, rel_y, 1.0, 1.0, zone.type))
    for tree in trees:
        if is_visible(agent_idx, tree):
            rel = (tree.pos - my_pos) / world_size
            landmarks.append((dist, rel_x, rel_y, 1.0, 0.0, 0.0))
    landmarks.sort(by=distance)
    top_n = landmarks[:n_closest_landmarks]
    # Pad with zeros if fewer than N
    for (_, rx, ry, vis, is_sz, type) in top_n:
        obs.extend([rx, ry, vis, is_sz, type])  # 5 values each

    # 5. Victims (rel_x, rel_y, type, visible_bit)
    for victim in victims:
        if not saved[victim] and is_visible(agent_idx, victim):
            rel = (victim.pos - my_pos) / world_size
            obs.extend([rel_x, rel_y, victim.type, 1.0])
        else:
            obs.extend([0, 0, 0, 0])      # masked

    # 6. Other rescuers (rel_x, rel_y, visible_bit)
    for other in rescuers:
        if other != agent_idx:
            if is_visible(agent_idx, other):
                rel = (other.pos - my_pos) / world_size
                obs.extend([rel_x, rel_y, 1.0])
            else:
                obs.extend([0, 0, 0])     # masked

    return np.array(obs, dtype=np.float32)
```

Listing 3: Observation construction.

3.2 Training Pipeline: train.py

The `train.py` file implements the training process using Multi-Agent Proximal Policy Optimization (MAPPO) with TorchRL. The pipeline is optimized for efficient reinforcement learning through parallel simulations, observation preprocessing, performance logging, and model-saving mechanisms.

3.2.1 Reinforcement Learning Setup

The MAPPO algorithm trains rescuer agents to maximize the cumulative reward function. The algorithm optimizes the clipped surrogate objective as shown in Equation 20.

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{20}$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between current and old policies.
- $\hat{A}_t$ is the estimated advantage at time step t.
- $\epsilon = 0.2$ is the clipping threshold that prevents excessively large policy updates.

3.2.2 Hyperparameter Configuration

Key hyperparameters are carefully configured to optimize learning efficiency.

Parameter	**Default**	**Config**
Learning rate α	3×10^{-4}	3×10^{-4}
Batch size	256	256
Frames per batch	2048	2048
Number of epochs	10	20
Total timesteps	100,000	102,400
GAE γ	0.99	0.99
GAE λ	0.95	0.95
Clip epsilon ϵ	0.2	0.2
Entropy coefficient	0.001	0.001
Gradient clip norm	1.0	1.0

Table 2: Training hyperparameters (Default = code default; Config = conf/config.yaml).

3.2.3 Training Process

The training process follows the steps outlined in Algorithm 1.

Algorithm 1 MAPPO Training Loop

```
1: Initialize policy network $\pi_\theta$ and value network $V_\phi$
2: Initialize data collector and replay buffer
3: for iteration = 1 to total_iterations do
4:     Collect frames_per_batch steps using current policy
5:     Compute advantages using GAE: $\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$
6:     for epoch = 1 to num_epochs do
7:         for minibatch in replay buffer do
8:             Compute policy ratio $r_t(\theta)$
9:             Compute clipped objective $L^{\text{PPO}}$
10:            Compute value loss $L^{\text{value}}$
11:            Compute entropy bonus $L^{\text{entropy}}$
12:            Update $\theta, \phi$ using Adam optimizer
13:        end for
14:    end for
15:    Log metrics to TensorBoard
16:    Update policy in collector
17: end for
18: Save final checkpoint
```

3.2.4 Performance Logging

Training progress is logged to facilitate real-time monitoring and offline analysis:

- **TensorBoard Logging**: Metrics such as rewards, losses, and learning rates are visualized in real time.
- **Training Metrics**:
  - `train/loss/objective`: PPO clipped objective loss.
  - `train/loss/critic`: Value function loss.
  - `train/loss/entropy`: Entropy bonus (exploration).
  - `train/loss/total`: Combined loss.
  - `train/episode_reward`: Mean episode reward.
  - `train/total_frames`: Cumulative frames processed.
- **Episode Metrics**: The environment tracks per-episode performance metrics:
  - `rescues_pct`: Percentage of victims successfully rescued.
  - `collisions`: Total collision events (tree + agent collisions).
  - `coverage_cells`: Number of unique grid cells visited (exploration metric).
  - `mean_energy_pct`: Average remaining energy at episode end (if enabled).
  - `recharge_events`: Number of successful recharging events (if enabled).
  - `energy_depleted_steps`: Steps where agents were immobilized due to zero energy.

3.2.5 Model Saving

Trained models are saved with configuration for reproducibility.

```python
checkpoint = {
    "policy_state_dict": policy.state_dict(),
    "critic_state_dict": critic.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    "total_frames": total_frames,
    "iteration": iteration,
    "env_config": env_kwargs
}
torch.save(checkpoint, f"{save_folder}/checkpoint.pt")
```

Listing 4: Checkpoint saving.

3.3 Evaluation Pipeline: eval.py

The `eval.py` script evaluates trained policies by running multiple episodes and computing performance metrics.

3.3.1 Policy Loading

The evaluation pipeline supports both manual and automatic policy loading.

Manual Loading: User provides the file path for the saved model, as shown in Equation 21.

$$\pi_{\theta_{\text{loaded}}} = \text{load}(\text{path\_to\_model}) \tag{21}$$

Automatic Loading: The script identifies the latest saved model using file modification timestamps, as shown in Equation 22.

$$\pi_{\theta_{\text{loaded}}} = \arg\max_{\theta}(t_{\text{saved}}) \tag{22}$$

3.3.2 Evaluation Process

The evaluation runs the environment for N_{episodes} episodes, where agents act based on the loaded policy π_θ. The environment produces a trajectory as shown in Equation 23.

$$\tau = \{(s_t, a_t, r_t, s_{t+1}) \mid t = 0, 1, \dots, T\} \tag{23}$$

The cumulative reward for an episode is given by Equation 24.

$$R_{\text{episode}} = \sum_{t=0}^{T} r_t \tag{24}$$

3.3.3 Performance Metrics

The following metrics are computed over evaluation episodes.

Average Reward is given by Equation 25.

$$\bar{R} = \frac{1}{N_{\text{episodes}}} \sum_{i=1}^{N_{\text{episodes}}} R_{\text{episode},i} \tag{25}$$

Per-Agent Rewards: Individual agent rewards $R_{\text{agent},i}$ are computed for each rescuer.

Success Rate is given by Equation 26.

$$\text{Success Rate} = \frac{N_{\text{successful\_rescues}}}{N_{\text{victims}}} \tag{26}$$

3.3.4 Rendering

To visually verify agent behavior, the evaluation pipeline supports rendering the environment in real time.

The rendering displays:

- Safe zones with type-specific colors and transparency.
- Trees as gray circular obstacles.
- Victims with type-matching colors.
- Rescuers as white circles with vision radius indicators.
- Type labels (A, B, C, D) on zones and victims.

3.4 Workflow Orchestration: main.py

The `main.py` script serves as the central control hub for the project, orchestrating training and evaluation workflows using Hydra for configuration management.

3.4.1 Core Functionality

The script provides three primary execution modes:

1. **Training Mode**: Invokes the `train()` function to train agents using MAPPO.
2. **Evaluation Mode**: Invokes the `evaluate()` function to assess trained policies.
3. **TensorBoard Mode**: Launches TensorBoard server for visualization.

3.4.2 Configuration Management

Hydra enables flexible configuration through YAML files and command-line overrides.

```
# Training with default config
python -m src.main train.active=true

# Training with custom parameters
python -m src.main train.active=true \
    env.victims=12 env.rescuers=6 \
    train.total_timesteps=500000

# Evaluation with rendering
python -m src.main eval.active=true eval.render_mode=human
```

Listing 5: Example usage.

3.4.3 Modular Design

The script is designed to be modular, enabling users to:

- Easily switch between training and evaluation modes.
- Configure environment parameters through configuration files.
- Extend functionality for fine-tuning, testing, or benchmarking policies.

4 Environment Parameters

This section provides a comprehensive reference of all configurable environment parameters.

4.1 Core Parameters

Parameter	Default	Config	Description
`num_rescuers`	2	6	Number of rescuer agents
`num_victims`	2	12	Number of victim entities
`num_trees`	5	8	Number of obstacle trees
`num_safe_zones`	4	4	Number of safe zones (corners)
`max_cycles`	200	300	Maximum steps per episode
`vision_radius`	0.5	0.4	Maximum observation distance
`rescue_radius`	0.15	0.15	Distance for successful rescue
`follow_radius`	0.2	0.2	Distance for victim commitment
`agent_size`	0.03	0.03	Radius of agent entities
`tree_radius`	0.05	0.05	Radius of tree obstacles
`safe_zone_radius`	0.15	0.15	Radius of safe zones
`world_size`	2.0	2.0	World bounds ([-1, 1] range)
`continuous_actions`	True	True	Use continuous action space
`randomize_safe_zones`	False	True	Randomize safe zone types
`n_closest_landmarks`	3	9	Number of landmarks in observation

Table 3: Core environment parameters (Default = code default; Config = conf/config.yaml).

4.2 Energy System Parameters

The energy system adds resource management to the environment, requiring agents to balance exploration with energy conservation.

Parameter	Default	Config	Description
`energy_enabled`	True	True	Enable energy system
`max_energy`	1.0	1.0	Maximum energy capacity
`movement_cost_coeff`	0.01	0.01	Energy cost per unit action
`idle_cost`	0.001	0.001	Base energy cost per step
`energy_depleted_action_scale`	0.0	0.0	Action scale when depleted
`num_chargers`	2	2	Number of recharging stations
`recharge_radius`	0.12	0.12	Distance to activate recharging
`recharge_rate`	0.05	0.05	Energy restored per step
`randomize_chargers`	True	True	Randomize charger positions

Table 4: Energy system parameters.

4.2.1 Energy Mechanics

- **Energy Consumption**: Each step costs $\text{idle\_cost} + \text{movement\_cost\_coeff} \times \|\mathbf{a}\|$ energy.
- **Depletion**: When energy reaches zero, the agent's action is scaled by `energy_depleted_action_scale` (default: 0.0, meaning the agent cannot move).

- **Recharging**: Agents within `recharge_radius` of any charger receive `recharge_rate` energy per step.
- **Observation**: Energy level is included in the observation as a normalized value in $[0, 1]$.

5 Model Architecture

The neural network architecture follows the MAPPO paradigm with decentralized actors and centralized critics, implementing the **Centralized Training, Decentralized Execution (CTDE)** paradigm.

5.1 CTDE Paradigm

The CTDE approach separates training and execution phases:

- **Centralized Training**: During training, the critic network has access to all agents' observations, enabling better credit assignment and value estimation.
- **Decentralized Execution**: During deployment, each agent uses only its local observations to select actions, ensuring scalability and robustness to communication failures.

This is achieved through:

- **Decentralized Actor**: Each agent's policy network (`centralised=False`) processes only local observations.
- **Centralized Critic**: The value network (`centralised=True`) concatenates all agents' observations for global state estimation.
- **Parameter Sharing**: Both networks share parameters across agents (`share_params=True`), reducing sample complexity for homogeneous agents.

5.2 Policy Network (Actor)

The policy network maps local observations to action distributions:

- **Input**: Local observation $[\text{batch}, N_{\text{agents}}, D_{\text{obs}}]$.
- **Architecture**: 2-layer MLP with 64 hidden units per layer.
- **Activation**: Tanh activation function.
- **Output**: Parameters for TanhNormal distribution (location and scale).
- **Parameter Sharing**: Shared across all agents (homogeneous policy).

The policy architecture can be expressed as shown in Equations 27–30.

$$\mathbf{h}_1 = \tanh(W_1\mathbf{o} + b_1) \tag{27}$$
$$\mathbf{h}_2 = \tanh(W_2\mathbf{h}_1 + b_2) \tag{28}$$
$$\boldsymbol{\mu} = W_\mu\mathbf{h}_2 + b_\mu \tag{29}$$
$$\boldsymbol{\sigma} = \text{softplus}(W_\sigma\mathbf{h}_2 + b_\sigma) \tag{30}$$

Actions are sampled from: $\mathbf{a} \sim \text{TanhNormal}(\boldsymbol{\mu}, \boldsymbol{\sigma})$.

5.3 Value Network (Critic)

The value network estimates state values for advantage computation:

- **Input**: All agent observations $[\text{batch}, N_{\text{agents}}, D_{\text{obs}}]$.
- **Architecture**: 2-layer MLP with 128 hidden units per layer.
- **Activation**: Tanh activation function.
- **Output**: State value $[\text{batch}, N_{\text{agents}}, 1]$.
- **Centralized**: True (observes all agent observations for credit assignment).

6 Curriculum Learning

The project supports curriculum learning for progressive difficulty adjustment during training.

6.1 Curriculum Configuration

- **Minimum Trees**: Starting number of obstacles (default: 0).
- **Maximum Trees**: Final number of obstacles (default: 8).
- **Number of Stages**: Progression stages (default: 5).
- **Stage Duration**: Automatically calculated as $\frac{\text{total\_iterations}}{\text{num\_stages}}$.

6.2 Progression Strategy

The number of trees at stage s is computed as shown in Equation 31.

$$N_{\text{trees}}(s) = N_{\min} + \left\lfloor \frac{s}{S-1} \times (N_{\max} - N_{\min}) \right\rfloor \tag{31}$$

where S is the total number of stages.

The curriculum maintains a fixed observation space size by using `max_trees` for observation dimensionality, with unused tree slots masked as zeros.

7 Ablation Studies and Failure Modes

This section documents experimental findings and known failure modes discovered during development.

7.1 Continuous vs. Discrete Control

Metric	**Continuous**	**Discrete**
Rescue %	Higher	Lower
Sample Efficiency	Moderate	Higher
Fine-grained Control	Yes	No
Action Space Size	Infinite ($[-1, 1]^2$)	5

Table 5: Comparison of action space configurations.

Finding: Continuous control enables smoother trajectories and more precise victim escort, leading to higher rescue rates in dense environments.

7.2 Curriculum vs. No Curriculum

Configuration	Dense Map (8 trees) Performance
With Curriculum	Agents successfully navigate and rescue
Without Curriculum	Agents fail to solve — stuck in local optima

Table 6: Curriculum learning ablation results.

Failure Analysis (No Curriculum):

- Agents trained directly on 8-tree environments encounter sparse reward signals.
- Initial random policies rarely achieve rescues, providing no learning signal.
- Policy collapses to boundary-hugging or static behavior.

Curriculum Benefit:

- Early stages (0–2 trees) provide dense reward signal for basic navigation.
- Progressive difficulty allows transfer of learned behaviors.
- Final stage performance matches or exceeds single-stage training on easy environments.

7.3 Known Failure Modes

1. **Hysteresis Loops (Victim Swapping)**:
   - Two agents approach the same victim from opposite sides.
   - Victim commitment oscillates between agents.
   - **Mitigation**: Commitment hysteresis ($1.5\times$ release threshold, $0.6\times$ switch threshold).
2. **Sparse Reward Plateaus**:
   - In high-tree environments, random policies rarely achieve rescues.
   - No gradient signal for policy improvement.
   - **Mitigation**: Dense shaping rewards (pickup/escort deltas).
3. **Energy Starvation**:
   - Agents deplete energy far from chargers.
   - Become immobile, unable to complete rescues.
   - **Mitigation**: Energy cost as negative reward encourages efficient pathing.
4. **Boundary Clustering**:
   - Agents learn to minimize boundary penalties by staying near center.
   - Victims near edges are neglected.
   - **Mitigation**: Softened boundary penalty (-0.2 instead of -1.0).

8 Usage Guide

8.1 Docker Execution

```
# Build image
docker build -t student-search:latest -f docker/Dockerfile .

# Training
docker run --rm \
    -v "${PWD}/search_rescue_logs:/app/search_rescue_logs" \
    student-search:latest

# Evaluation
docker run --rm \
    -v "${PWD}/search_rescue_logs:/app/search_rescue_logs" \
    student-search:latest eval.active=true
```

Listing 6: Docker execution commands.

8.2 Local Execution

```
# Install dependencies
pip install -e .
pip install -e ".[dev]"

# Training
python -m src.main train.active=true

# Evaluation with rendering
python -m src.main eval.active=true eval.render_mode=human

# Custom configuration
python -m src.main train.active=true \
    env.victims=12 env.rescuers=6 \
    train.total_timesteps=500000 \
    train.learning_rate=0.001
```

Listing 7: Local execution commands.

9 Conclusion

This documentation has provided a comprehensive overview of the Search and Rescue Multi-Agent Simulation project. The key contributions include:

1. A flexible multi-agent environment implementing realistic search and rescue dynamics with partial observability.

2. A commitment-based victim following system with hysteresis for stable agent-victim relationships.

3. A carefully designed reward structure with assignment-aware shaping that enables effective credit assignment.

4. Integration with TorchRL for efficient MAPPO training using the CTDE paradigm.

5. Optional curriculum learning for progressive difficulty adjustment (0 to 8 trees across 5 stages).
6. An energy management system with recharging stations for strategic resource planning.
7. Comprehensive episode metrics tracking (rescue percentage, collisions, coverage).
8. Full Docker support and Hydra-based configuration management.

The modular design allows for easy extension and experimentation with different algorithms, environment configurations, and training strategies.