Abmarl

Release 0.2.2

Edward Rusu

Aug 21, 2023

CONTENTS

Abmarl is a package for developing Agent-Based Simulations and training them with MultiAgent Reinforcement Learning (MARL). We provide an intuitive command line interface for engaging with the full workflow of MARL experimentation: training, visualizing, and analyzing agent behavior. We define an *Agent-Based Simulation Interface* and *Simulation Manager*, which control which agents interact with the simulation at each step. We support *integration* with popular reinforcement learning simulation interfaces, including *gym.Env* and *MultiAgentEnv*. We define our own *GridWorld Simulation Framework* for creating custom grid-based Agent Based Simulations.

Abmarl leverages RLlib's framework for reinforcement learning and extends it to more easily support custom simulations, algorithms, and policies. We enable researchers to rapidly prototype MARL experiments and simulation design and lower the barrier for pre-existing projects to prototype RL as a potential solution.

CHAPTER

ONE

DESIGN

A reinforcement learning experiment in Abmarl contains two interacting components: a Simulation and a Trainer.

The Simulation contains agent(s) who can observe the state (or a substate) of the Simulation and whose actions affect the state of the simulation. The simulation is discrete in time, and at each time step agents can provide actions. The simulation also produces rewards for each agent that the Trainer can use to train optimal behaviors. The Agent-Simulation interaction produces state-action-reward tuples (SARs), which can be collected in *rollout fragments* and used to optimize agent behaviors.

The Trainer contains policies that map agents' observations to actions. Policies are one-to-many with agents, meaning that there can be multiple agents using the same policy. Policies may be heuristic (i.e. coded by the researcher) or trainable by the RL algorithm.

In Abmarl, the Simulation and Trainer are specified in a single Python configuration file. Once these components are set up, they are passed as parameters to RLlib's tune command, which will launch the RLlib application and begin the training process. The training process will save checkpoints to an output directory, from which the user can visualize and analyze results. The following diagram demonstrates this workflow.

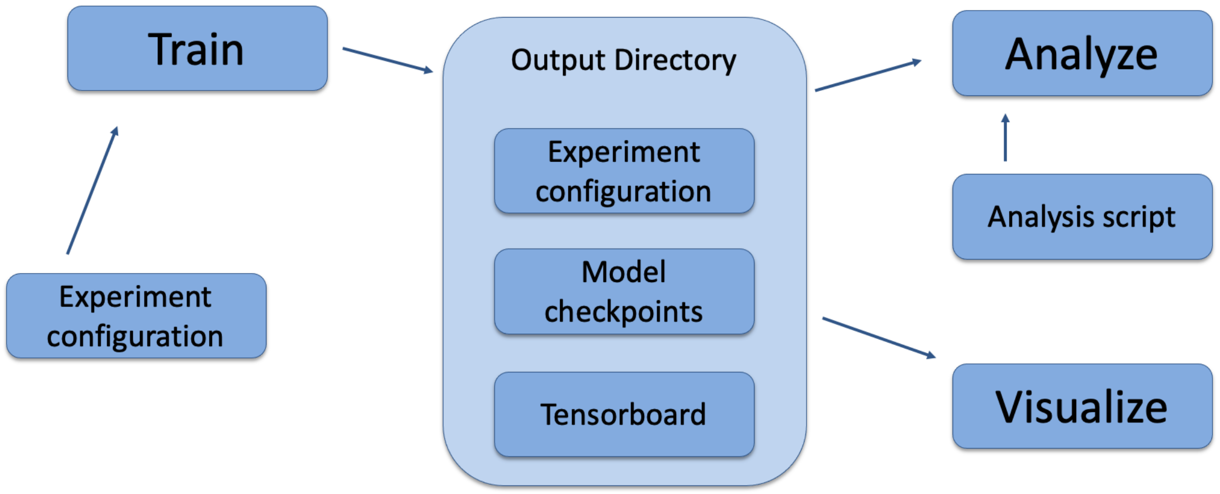


Fig. 1: Abmarl's usage workflow. An experiment configuration is used to train agents' behaviors. The policies and simulation are saved to an output directory. Behaviors can then be analyzed or visualized from the output directory.

1.1 Creating Agents and Simulations

Abmarl provides three interfaces for setting up an agent-based simulations.

1.1.1 Agent

First, we have *Agents*. An agent is an object with an observation and action space. Many practitioners may be accustomed to gym.Env's interface, which defines the observation and action space for the *simulation*. However, in heterogeneous multiagent settings, each *agent* can have different spaces; thus we assign these spaces to the agents and not the simulation.

An agent can be created like so:

```
from gym.spaces import Discrete, Box
from abmarl.sim import Agent
agent = Agent(
    id='agent0',
    observation_space=Box(-1, 1, (2,)),
    action_space=Discrete(3)
)
```

At this level, the Agent is basically a dataclass. We have left it open for our users to extend its features as they see fit.

1.1.2 Agent Based Simulation

Next, we define an *Agent Based Simulation*, or ABS for short, with the ususal `reset` and `step` functions that we are used to seeing in RL simulations. These functions, however, do not return anything; the state information must be obtained from the getters: `get_obs`, `get_reward`, `get_done`, `get_all_done`, and `get_info`. The getters take an agent's id as input and return the respective information from the simulation's state. The ABS also contains a dictionary of agents that "live" in the simulation.

An Agent Based Simulation can be created and used like so:

```
from abmarl.sim import Agent, AgentBasedSimulation
class MySim(AgentBasedSimulation):
    def __init__(self, agents=None, **kwargs):
        self.agents = agents
     ... # Implement the ABS interface

# Create a dictionary of agents
agents = {f'agent{i}': Agent(id=f'agent{i}', ...) for i in range(10)}
# Create the ABS with the agents
sim = MySim(agents=agents)
sim.reset()
# Get the observations
obs = {agent.id: sim.get_obs(agent.id) for agent in agents.values()}
# Take some random actions
sim.step({agent.id: agent.action_space.sample() for agent in agents.values()})
# See the reward for agent3
print(sim.get_reward('agent3'))
```

Warning: Implementations of AgentBasedSimulation should call `finalize` at the end of its `__init__`. Finalize ensures that all agents are configured and ready to be used for training.

Note: Instead of treating agents as dataclasses, we could have included the relevant information in the Agent Based Simulation with various dictionaries. For example, we could have `action_spaces` and `observation_spaces` that maps agents' ids to their action spaces and observation spaces, respectively. In Abmarl, we favor the dataclass approach and use it throughout the package and documentation.

1.1.3 Simulation Managers

The Agent Based Simulation interface does not specify an ordering for agents' interactions with the simulation. This is left open to give our users maximal flexibility. However, in order to interace with RLlib's learning library, we provide a *Simulation Manager* which specifies the output from `reset` and `step` as RLlib expects it. Specifically,

1. Agents that appear in the output dictionary will provide actions at the next step.
2. Agents that are done on this step will not provide actions on the next step.

Simulation managers are open-ended requiring only `reset` and `step` with output described above. For convenience, we have provided two managers: *Turn Based*, which implements turn-based games; and *All Step*, which has every non-done agent provide actions at each step.

Simluation Managers "wrap" simulations, and they can be used like so:

```python
from abmarl.managers import AllStepManager
from abmarl.sim import AgentBasedSimulation, Agent
class MySim(AgentBasedSimulation):
    ... # Define some simulation

# Instatiate the simulation
sim = MySim(agents=...)
# Wrap the simulation with the simulation manager
sim = AllStepManager(sim)
# Get the observations for all agents
obs = sim.reset()
# Get simulation state for all non-done agents, regardless of which agents
# actually contribute an action.
obs, rewards, dones, infos = sim.step({'agent0': 4, 'agent2': [-1, 1]})
```

1.1.4 External Integration

In order to train agents in a Simulation Manager using RLlib, we must wrap the simulation with either a *GymWrapper* for single-agent simulations (i.e. only a single entry in the *agents* dict) or a *MultiAgentWrapper* for multiagent simulations.

1.2 Training with an Experiment Configuration

In order to run experiments, we must define a configuration file that specifies Simulation and Trainer parameters. Here is the configuration file from the *Corridor tutorial* that demonstrates a simple corridor simulation with multiple agents.

```
# Import the MultiCorridor ABS, a simulation manager, and the multiagent
# wrapper needed to connect to RLlib's trainers
from abmarl.sim.corridor import MultiCorridor
from abmarl.managers import TurnBasedManager
from abmarl.external import MultiAgentWrapper

# Create and wrap the simulation
# NOTE: The agents in `MultiCorridor` are all homogeneous, so this simulation
# just creates and stores the agents itself.
sim = MultiAgentWrapper(TurnBasedManager(MultiCorridor()))

# Register the simulation with RLlib
sim_name = "MultiCorridor"
from ray.tune.registry import register_env
register_env(sim_name, lambda sim_config: sim)

# Set up the policies. In this experiment, all agents are homogeneous,
# so we just use a single shared policy.
ref_agent = sim.unwrapped.agents['agent0']
policies = {
    'corridor': (None, ref_agent.observation_space, ref_agent.action_space, {})
}
def policy_mapping_fn(agent_id):
    return 'corridor'

# Experiment parameters
params = {
    'experiment': {
        'title': f'{sim_name}',
        'sim_creator': lambda config=None: sim,
    },
    'ray_tune': {
        'run_or_experiment': 'PG',
        'checkpoint_freq': 50,
        'checkpoint_at_end': True,
        'stop': {
            'episodes_total': 2000,
        },
        'verbose': 2,
        'config': {
            # --- simulation ---
            'env': sim_name,
            'horizon': 200,
            'env_config': {},
            # --- Multiagent ---
            'multiagent': {
                'policies': policies,
                'policy_mapping_fn': policy_mapping_fn,
```

(continues on next page)

(continued from previous page)

```
            },
            # --- Parallelism ---
            "num_workers": 7,
            "num_envs_per_worker": 1,
        },
    }
}
```

Warning: The simulation must be a *Simulation Manager* or an *External Wrapper* as described above.

Note: This example has `num_workers` set to 7 for a computer with 8 CPU's. You may need to adjust this for your computer to be *<cpu count> - 1*.

1.2.1 Experiment Parameters

The strucutre of the parameters dictionary is very important. It *must* have an *experiment* key which contains both the *title* of the experiment and the *sim_creator* function. This function should receive a config and, if appropriate, pass it to the simulation constructor. In the example configuration above, we just retrun the already-configured simulation. Without the title and simulation creator, Abmarl may not behave as expected.

The experiment parameters also contains information that will be passed directly to RLlib via the *ray_tune* parameter. See RLlib's documentation for a list of common configuration parameters.

1.2.2 Command Line

With the configuration file complete, we can utilize the command line interface to train our agents. We simply type `abmarl train multi_corridor_example.py`, where *multi_corridor_example.py* is the name of our configuration file. This will launch Abmarl, which will process the file and launch RLlib according to the specified parameters. This particular example should take 1-10 minutes to train, depending on your compute capabilities. You can view the performance in real time in tensorboard with `tensorboard --logdir ~/abmarl_results`.

Note: By default, the "base" of the output directory is the home directory, and Abmarl will create the *abmarl_results* directory there. The base directory can by configured in the *params* under *ray_tune* using the *local_dir* parameter. This value should be a full path. For example, `'local_dir':  '/usr/local/scratch'`.

1.3 Debugging

It may be useful to trial run a simulation after setting up a configuration file to ensure that the simulation mechanics work as expected. Abmarl's `debug` command will run the simulation with random actions and create an output directory, wherein it will copy the configuration file and output the observations, actions, rewards, and done conditions for each step. The data from each episode will be logged to its own file in the output directory. For example, the command

```
abmarl debug multi_corridor_example.py -n 2 -s 20 --render
```

will run the *MultiCorridor* simulation with random actions and output log files to the directory it creates for 2 episodes and a horizon of 20, as well as render each step in each episode.

1.4 Visualizing

We can visualize the agents' learned behavior with the `visualize` command, which takes as argument the output directory from the training session stored in `~/abmarl_results`. For example, the command

```
abmarl visualize ~/abmarl_results/MultiCorridor-2020-08-25_09-30/ -n 5 --record
```

will load the experiment (notice that the directory name is the experiment title from the configuration file appended with a timestamp) and display an animation of 5 episodes. The `--record` flag will save the animations as *.mp4* videos in the training directory.

By default, each episode has a *horizon* of 200 steps (i.e. it will run for up to 200 steps). It may end earlier depending on the *done* condition from the simulation. You can control the *horizon* with `-s` or `--steps-per-episode` when running the visualize command.

1.5 Analyzing

The simulation and trainer can also be loaded into an analysis script for post-processing via the `analyze` command. The analysis script must implement the following *run* function. Below is an example that can serve as a starting point.

```python
# Load the simulation and the trainer from the experiment as objects
def run(sim, trainer):
    """
    Analyze the behavior of your trained policies using the simulation and trainer
    from your RL experiment.

    Args:
        sim:
            Simulation Manager object from the experiment.
        trainer:
            Trainer that computes actions using the trained policies.
    """
    # Run the simulation with actions chosen from the trained policies
    policy_agent_mapping = trainer.config['multiagent']['policy_mapping_fn']
    for episode in range(100):
        print('Episode: {}'.format(episode))
        obs = sim.reset()
        done = {agent: False for agent in obs}
        while True: # Run until the episode ends
            # Get actions from policies
            joint_action = {}
            for agent_id, agent_obs in obs.items():
                if done[agent_id]: continue # Don't get actions for done agents
                policy_id = policy_agent_mapping(agent_id)
                action = trainer.compute_action(agent_obs, policy_id=policy_id)
                joint_action[agent_id] = action
            # Step the simulation
            obs, reward, done, info = sim.step(joint_action)
```

(continues on next page)

(continued from previous page)

```
            if done['__all__']:
                break
```

Analysis can then be performed using the command line interface:

```
abmarl analyze ~/abmarl_results/MultiCorridor-2020-08-25_09-30/ my_analysis_script.py
```

CHAPTER

TWO

GRIDWORLD SIMULATION FRAMEWORK

Abmarl provides a GridWorld Simulation Framework for setting up grid-based Agent Based Simulations, which can be connected to Reinforcement Learning algorithms through Abmarl's *AgentBasedSimulation* interface. The GridWorld Simulation Framework is a *grey box*: we assume users have working knowledge of Python and object-oriented programming. Using the *built in features* requires minimal knowledge, but extending them and creating new features requires more knowledge. In addition to the design documentation below, see the *GridWorld tutorials* for in-depth examples on using and extending the GridWorld Simulation Framework.

2.1 Framework Design

The GridWorld Simulation Framework utilizes a modular design that allows users to create new features and plug them in as components of the simulation. Every component inherits from the *GridWorldBaseComponent* class and has a reference to a *Grid* and a dictionary of *Agents*. These components make up a *GridWorldSimulation*, which extends the *AgentBasedSimulation* interface. For example, a simulation might look something like this:

```python
from abmarl.sim.gridworld.base import GridWorldSimulation
from abmarl.sim.gridworld.state import PositionState
from abmarl.sim.gridworld.actor import MoveActor
from abmarl.sim.gridworld.observer import SingleGridObserver

class MyGridSim(GridWorldSimulation):
    def __init__(self, **kwargs):
        self.agents = kwargs['agents']
        self.position_state = PositionState(**kwargs)
        self.move_actor = MoveActor(**kwargs)
        self.observer = SingleGridObserver(**kwargs)

    def reset(self, **kwargs):
        self.position_state.reset(**kwargs)

    def step(self, action_dict):
        for agent_id, action in action_dict.items():
            self.move_actor.process_action(self.agents[agent_id], action)

    def get_obs(self, agent_id, **kwargs):
        return self.observer.get_obs(self.agents[agent_id])
    ...
```

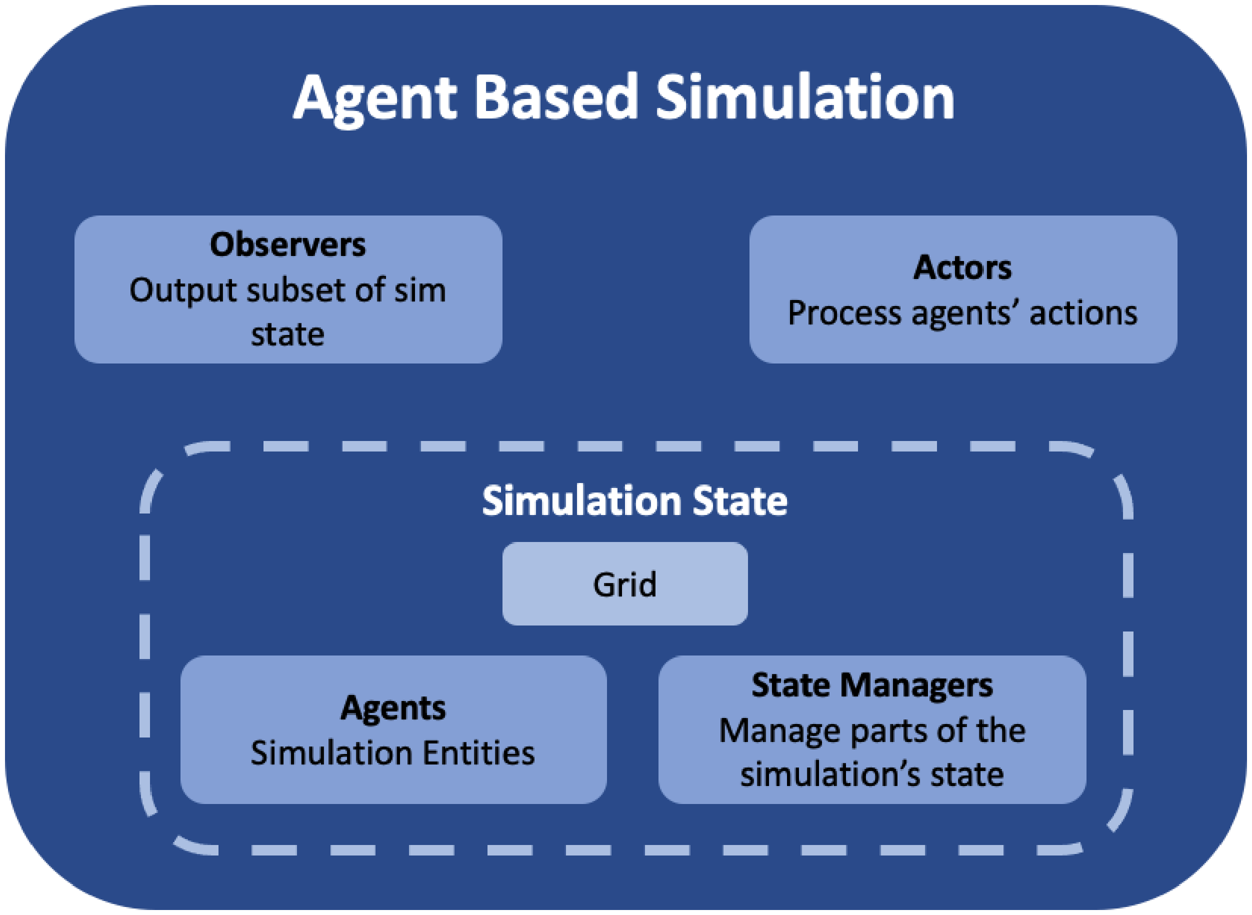


Fig. 1: Abmarl's GridWorld Simulation Framework. A simulation has a Grid, a dictionary of agents, and various components that manage the various features of the simulation. The componets shown in medium-blue are user-configurable and -creatable.

2.1.1 Agent

Every entity in the simulation is a *GridWorldAgent* (e.g. walls, foragers, resources, fighters, etc.). GridWorldAgents are *PrincipleAgents* with specific parameters that work with their respective components. Agents must be given an *encoding*, which is a positive integer that correlates to the type of agent and simplifies the logic for many components of the framework. GridWorldAgents can also be configured with an *initial position*, the ability to *block* other agents' abilities, and visualization parameters such as *shape* and *color*.

Following the dataclass model, additional agent classes can be defined that allow them to work with various components. For example, *GridObservingAgents* can work with *Observers*, and *MovingAgents* can work with the *MoveActor*. Any new agent class should inhert from *GridWorldAgent* and possibly from *ActingAgent* or *ObservingAgent* as needed. For example, one can define a new type of agent like so:

```
from abmarl.sim.gridworld.agent import GridWorldAgent
from abmarl.sim import ActingAgent

class CommunicatingAgent(GridWorldAgent, ActingAgent):
    def __init__(self, broadcast_range=None, **kwargs):
        super().__init__(**kwargs)
        self.broadcast_range = broadcast_range
        ...
```

Warning: Agents should follow the dataclass model, meaning that they should only be given parameters. All functionality should be written in the simulation components.

2.1.2 Grid

The *Grid* stores *Agents* in a two-dimensional numpy array. The Grid is configured to be a certain size (rows and columns) and to allow types of Agents to overlap (occupy the same cell). For example, you may want a ForagingAgent to be able to overlap with a ResourceAgent but not a WallAgent. The *overlapping* parameter is a dictionary that maps the Agent's *encoding* to a list of other Agents' *encodings* with which it can overlap. For example,

```
from abmarl.sim.gridworld.grid import Grid

overlapping = {
    1: [2],
    2: [1, 3],
    3: [2, 3]
}
grid = Grid(5, 6, overlapping=overlapping)
```

means that agents whose *encoding* is 1 can overlap with other agents whose *encoding* is 2; agents whose *encoding* is 2 can overlap with other agents whose *encoding* is 1 or 3; and agents whose *encoding* is 3 can overlap with other agents whose *encoding* is 2 or 3.

Warning: To avoid undefined behavior, the *overlapping* should be symmetric, so that if 2 can overlap with 3, then 3 can also overlap with 2.

Note: If *overlapping* is not specified, then no agents will be able to occupy the same cell in the Grid.

Interaction between simulation components and the *Grid* is *data open*, which means that we allow components to access the internals of the Grid. Although this is possible and sometimes necessary, the Grid also provides an interface for safer interactions with components. Components can *query* the Grid to see if an agent can be placed at a specific position. Components can *place* agents at a specific position in the Grid, which will succeed if that cell is available to the agent as per the *overlapping* configuration. And Components can *remove* agents from specific positions in the Grid.

2.1.3 State

State Components manage the state of the simulation alongside the *Grid*. At the bare minimum, each State resets the part of the simulation that it manages at the the start of each episode.

2.1.4 Actor

Actor Components are responsible for processing agent actions and producing changes to the state of the simulation. Actors assign supported agents with an appropriate action space and process agents' actions based on the Actor's key. The result of the action is a change in the simulation's state, and Actors should return that change in a reasonable form. For example, the *MoveActor* appends *MovingAgents'* action spaces with a 'move' channel and looks for the 'move' key in the agent's incoming action. After a move is processed, the MoveActor returns if the move was successful.

2.1.5 Observer

Observer Components are responsible for creating an agent's observation of the state of the simulation. Observers assign supported agents with an appropriate observation space and generate observations based on the Observer's key. For example, the *SingleGridObserver* generates an observation of the nearby grid and stores it in the 'grid' channel of the *ObservingAgent's* observation.

2.1.6 Done

Done Components manage the "done state" of each agent and of the simulation as a whole. Agents that are reported as done will cease sending actions to the simulation, and the episode will end when all the agents are done or when the simulation is done.

2.1.7 Component Wrappers

The GridWorld Simulation Framework also supports *Component Wrappers*. Wrapping a component can be useful when you don't want to add a completely new component and only need to make a modification to the way a component already works. A component wrapper is itself a component, and so it must implement the same interface as the wrapped component to ensure that it works within the framework. A component wrapper also defines additional functions for wrapping spaces and data to and from those spaces: `check_space` for ensuring the space can be transformed, `wrap_space` to perform the transformation, and `wrap_point` to map data to the transformed space.

As its name suggests, a *Component Wrapper* stands between the underlying component and other objects with which it exchanges data. As such, a wrapper typically modifies the incoming/outgoing data before leveraging the underlying component for the actual datda processing. The main difference among wrapper types is in the direction of data flow, which we detail below.

Actor Wrappers

An *Actor Wrappers* receives actions in the *wrapped_space* through the `process_action` function. It can modify the data before sending it to the underlying Actor to process. An Actor Wrapper may need to modify the action spaces of corresponding agents to ensure that the action arrives in the correct format.

2.2 Built-in Features

Below is a list of some features that are available to use out of the box. Rememeber, you can create your own features in the GridWorld Simulation Framework and use many combinations of components together to make up a simulation.

2.2.1 Position

Agents have *positions* in the *Grid* that are managed by the *PositionState*. Agents can be configured with an *initial position*, which is where they will start at the beginning of each episode. If they are not given an *initial position*, then they will start at a random cell in the grid. Agents can overlap according to the *Grid's overlapping* configuration. For example, consider the following setup:

```
import numpy as np
from abmarl.sim.gridworld.agent import GridWorldAgent
from abmarl.sim.gridworld.grid import Grid
from abmarl.sim.gridworld.state import PositionState

agent0 = GridWorldAgent(
    id='agent0',
    encoding=1,
    initial_position=np.array([2, 4])
)
agent1 = GridWorldAgent(
    id='agent1',
    encoding=1
)
position_state = PositionState(
    agents={'agent0': agent0, 'agent1': agent1},
    grid=Grid(4, 5)
)
position_state.reset()
```

agent0 is configured with an *initial position* and *agent1* is not. At the start of each episode, *agent0* will be placed at (2, 4) and *agent1* will be placed anywhere in the grid (except for (2,4) because they cannot overlap).

2.2.2 Movement

MovingAgents can move around the *Grid* in conjunction with the *MoveActor*. MovingAgents require a *move range* parameter, indicating how many spaces away they can move in a single step. Agents cannot move out of bounds and can only move to the same cell as another agent if they are allowed to overlap. For example, in this setup

```
import numpy as np
from abmarl.sim.gridworld.agent import MovingAgent
from abmarl.sim.gridworld.grid import Grid
```

(continues on next page)

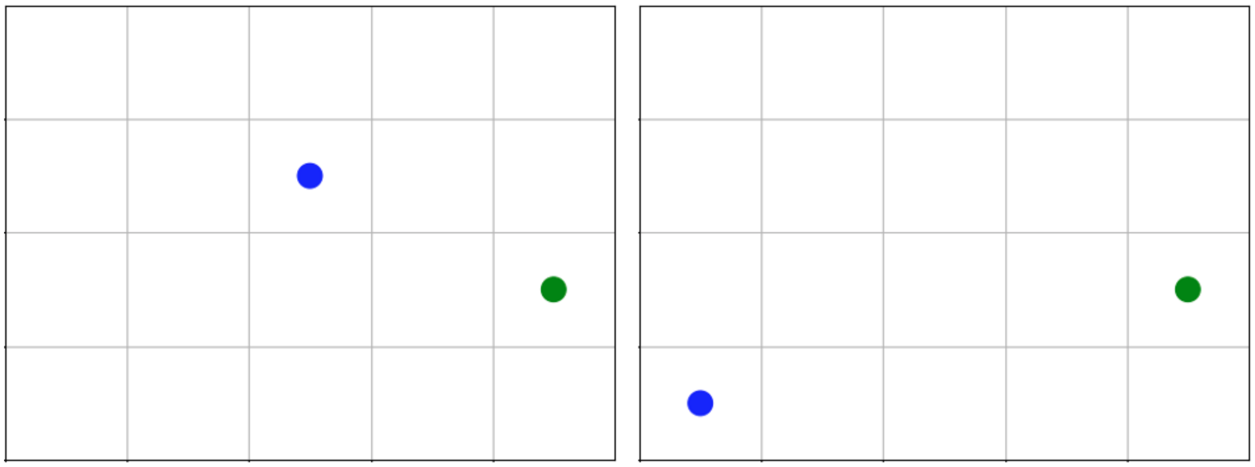


Fig. 2: agent0 in green starts at the same cell in every episode, and agent1 in blue starts at a random cell each time.

(continued from previous page)

```python
from abmarl.sim.gridworld.state import PositionState
from abmarl.sim.gridworld.actor import MoveActor

agents = {
    'agent0': MovingAgent(
        id='agent0', encoding=1, move_range=1, initial_position=np.array([2, 2])
    ),
    'agent1': MovingAgent(
        id='agent1', encoding=1, move_range=2, initial_position=np.array([0, 2])
    )
}
grid = Grid(5, 5, overlapping={1: [1]})
position_state = PositionState(agents=agents, grid=grid)
move_actor = MoveActor(agents=agents, grid=grid)

position_state.reset()
move_actor.process_action(agents['agent0'], {'move': np.array([0, 1])})
move_actor.process_action(agents['agent1'], {'move': np.array([2, 1])})
```

agent0 starts at position (2, 2) and can move up to one cell away. *agent1* starts at (0, 2) and can move up to two cells away. The two agents can overlap each other, so when the move actor processes their actions, both agents will be at position (2, 3).

2.2.3 Single Grid Observer

GridObservingAgents can observe the state of the *Grid* around them, namely which other agents are nearby, via the *SingleGridObserver*. The SingleGridObserver generates a two-dimensional array sized by the agent's *view range* with the observing agent located at the center of the array. All other agents within the *view range* will appear in the observation, shown as their *encoding*. For example, the following setup

```python
import numpy as np
from abmarl.sim.gridworld.agent import GridObservingAgent, GridWorldAgent
from abmarl.sim.gridworld.grid import Grid
```

(continues on next page)

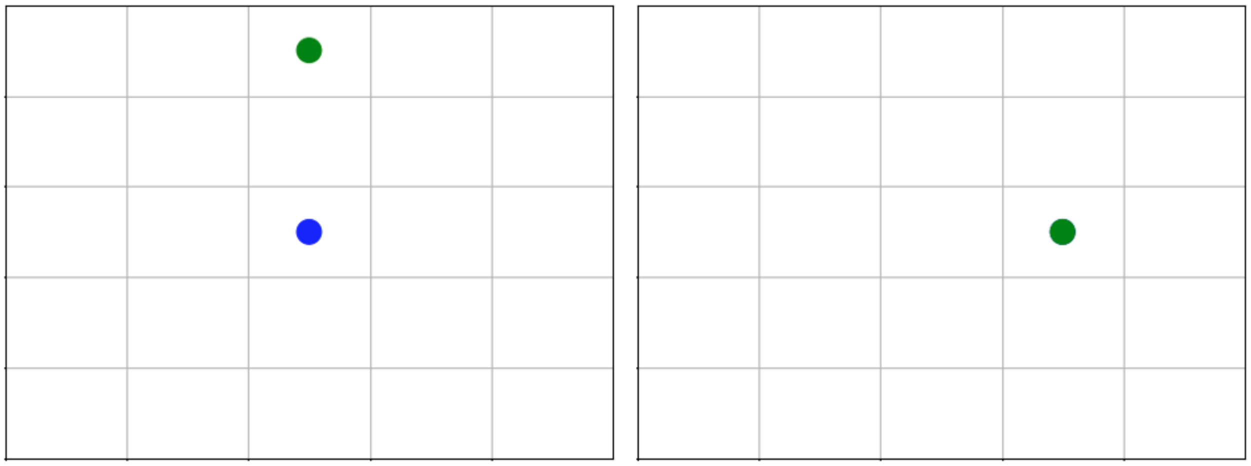


Fig. 3: agent0 and agent1 move to the same cell.

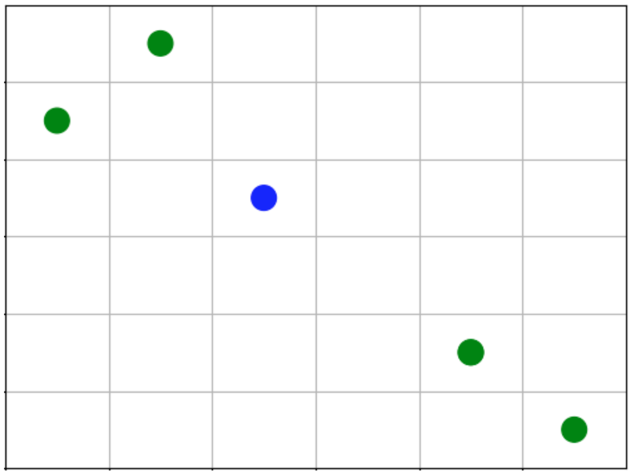
(continued from previous page)

```
from abmarl.sim.gridworld.state import PositionState
from abmarl.sim.gridworld.observer import SingleGridObserver

agents = {
    'agent0': GridObservingAgent(id='agent0', encoding=1, initial_position=np.array([2,␣
↪2]), view_range=3),
    'agent1': GridWorldAgent(id='agent1', encoding=2, initial_position=np.array([0, 1])),
    'agent2': GridWorldAgent(id='agent2', encoding=3, initial_position=np.array([1, 0])),
    'agent3': GridWorldAgent(id='agent3', encoding=4, initial_position=np.array([4, 4])),
    'agent4': GridWorldAgent(id='agent4', encoding=5, initial_position=np.array([4, 4])),
    'agent5': GridWorldAgent(id='agent5', encoding=6, initial_position=np.array([5, 5]))
}
grid = Grid(6, 6, overlapping={4: [5], 5: [4]})
position_state = PositionState(agents=agents, grid=grid)
observer = SingleGridObserver(agents=agents, grid=grid)

position_state.reset()
observer.get_obs(agents['agent0'])
```

will position agents as below and output an observation for *agent0* (blue) like so:

```
[-1, -1, -1, -1, -1, -1, -1],
[-1,  0,  2,  0,  0,  0,  0],
[-1,  3,  0,  0,  0,  0,  0],
[-1,  0,  0,  1,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0, 4*,  0],
[-1,  0,  0,  0,  0,  0,  6]
```

Since *view range* is the number of cells away that can be observed, the observation size is `(2 * view_range + 1) x (2 * view_range + 1)`. *agent0* is centered in the middle of this array, shown by its *encoding*: `1`. All other agents appear in the observation relative to *agent0's* position and shown by their *encodings*. The agent observes some out of bounds cells, which appear as `-1`s. *agent3* and *agent4* occupy the same cell, and the *SingleGridObserver* will randomly select between their *encodings* for the observation.



By setting *observe_self* to False, the *SingleGridObserver* can be configured so that an agent doesn't observe itself and only observes other agents, which may be helpful if overlapping is an important part of the simulation.

Blocking

Agents can block other agents' abilities and characteristics, such as blocking them from view, which masks out parts of the observation. For example, if *agent4* is configured with `blocking=True`, then the observation would like like this:

```
[-1, -1, -1, -1, -1, -1, -1],
[-1,  0,  2,  0,  0,  0,  0],
[-1,  3,  0,  0,  0,  0,  0],
[-1,  0,  0,  1,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0, 4*,  0],
[-1,  0,  0,  0,  0,  0, -2]
```

The -2 indicates that the cell is masked, and the choice of displaying *agent3* over *agent4* is still a random choice. Which cells get masked by blocking agents is determined by drawing two lines from the center of the observing agent's cell to the corners of the blocking agent's cell. Any cell whose center falls between those two lines will be masked, as shown below.

2.2.4 Multi Grid Observer

Similar to the *SingleGridObserver*, the *MultiGridObserver* displays a separate array for every *encoding*. Each array shows the relative positions of the agents and the number of those agents that occupy each cell. Out of bounds indicators (-1) and masked cells (-2) are present in every grid. For example, this setup would show an observation like so:

```
# Encoding 1
[-1, -1, -1, -1, -1, -1, -1],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  1,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0,  0, -2]

```

(continues on next page)

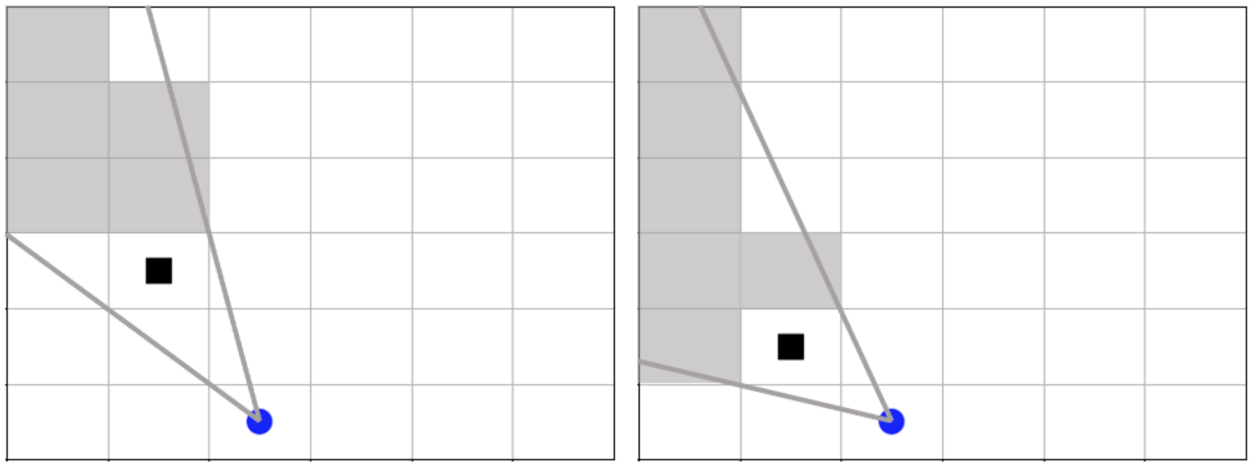
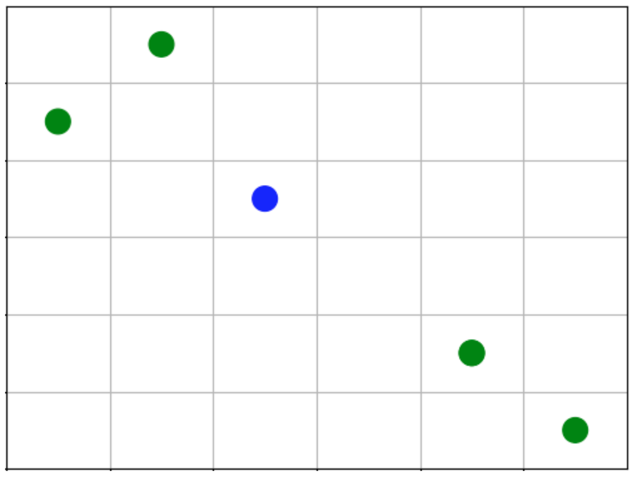


Fig. 4: The black agent is a wall agent that masks part of the grid from the blue agent. Cells whose centers fall betweent the lines are masked. Centers that fall directly on the line or outside of the lines are not masked. Two setups are shown to demonstrate how the masking may change based on the agents' positions.



(continued from previous page)

```
# Encoding 2
[-1, -1, -1, -1, -1, -1, -1],
[-1,  0,  1,  0,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0,  0,  0],
[-1,  0,  0,  0,  0,  0, -2]
...
```

MultiGridObserver may be preferable to *SingleGridObserver* in simulations where there are many overlapping agents.

2.2.5 Health

HealthAgents track their *health* throughout the simulation. *Health* is always bounded between 0 and 1. Agents whose *health* falls to 0 are marked as *inactive*. They can be given an *initial health*, which they start with at the beginning of the episode. Otherwise, their *health* will be a random number between 0 and 1, as managed by the *HealthState*. Consider the following setup:

```python
from abmarl.sim.gridworld.agent import HealthAgent
from abmarl.sim.gridworld.grid import Grid
from abmarl.sim.gridworld.state import HealthState

agent0 = HealthAgent(id='agent0', encoding=1)
grid = Grid(3, 3)
agents = {'agent0': agent0}
health_state = HealthState(agents=agents, grid=grid)
health_state.reset()
```

agent0 will be assigned a random *health* value between 0 and 1.

2.2.6 Attacking

Health becomes more interesting when we let agents attack one another. *AttackingAgents* work in conjunction with the *AttackActor*. They have an *attack range*, which dictates the range of their attack; an *attack accuracy*, which dictates the chances of the attack being successful; and an *attack strength*, which dictates how much *health* is depleted from the attacked agent. An agent's choice to attack is a boolean–either attack or don't attack–and then the AttackActor determines the successfulness based on the state of the simulation and the attributes of the AttackingAgent. The AttackActor requires an *attack mapping* dictionary which determines which *encodings* can attack other *encodings*, similar to the *overlapping* parameter for the *Grid*. Consider the following setup:

```python
import numpy as np
from abmarl.sim.gridworld.agent import AttackingAgent, HealthAgent
from abmarl.sim.gridworld.grid import Grid
from abmarl.sim.gridworld.state import PositionState, HealthState
from abmarl.sim.gridworld.actor import AttackActor

agents = {
    'agent0': AttackingAgent(
        id='agent0',
        encoding=1,
```

(continues on next page)

(continued from previous page)

```
        initial_position=np.array([0, 0]),
        attack_range=1,
        attack_strength=1,
        attack_accuracy=1
    ),
    'agent1': HealthAgent(id='agent1', encoding=2, initial_position=np.array([1, 0])),
    'agent2': HealthAgent(id='agent2', encoding=3, initial_position=np.array([0, 1]))
}
grid = Grid(2, 2)
position_state = PositionState(agents=agents, grid=grid)
health_state = HealthState(agents=agents, grid=grid)
attack_actor = AttackActor(agents=agents, grid=grid, attack_mapping={1: [2]})

position_state.reset()
health_state.reset()
attack_actor.process_action(agents['agent0'], {'attack': True})
attack_actor.process_action(agents['agent0'], {'attack': True})
```

Here, *agent0* attempts to make two attack actions. The first one is successful because *agent1* is within its *attack range* and is attackable according to the *attack mapping*. *agent1*'s *health* will be depleted by 1, and as a result its *health* will fall to 0 and it will be marked as *inactive*. The second attack fails because, although *agent2* is within range, it is not a type that *agent0* can attack.

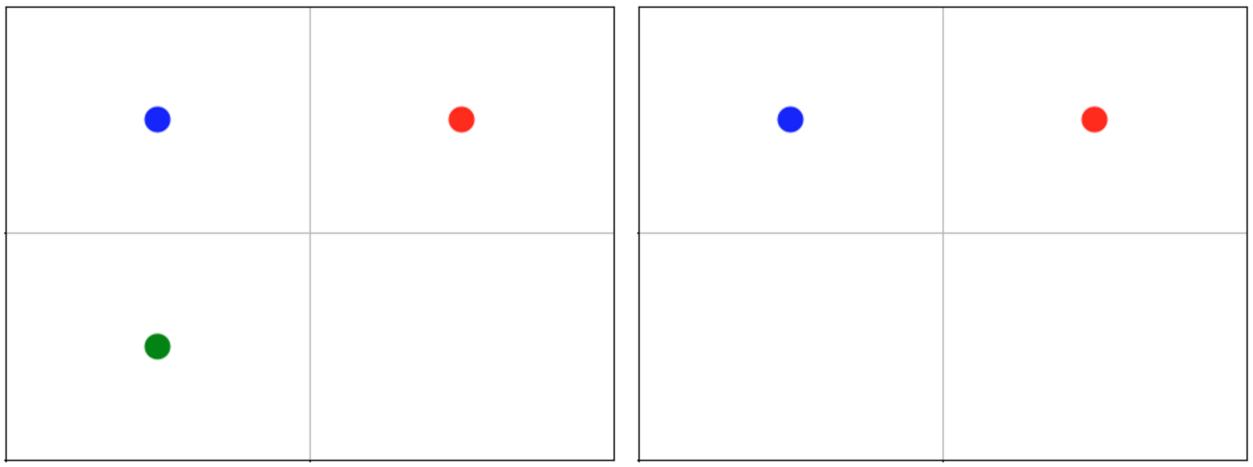


Fig. 5: agent0 in blue performs two attacks. The first is successful, but the second is not. agent1 in green is killed, but agent2 in red is still active.

Note: Attacks can be blocked by *blocking* agents. If an attackable agent is masked from an attacking agent, then it cannot be attacked by that agent. The masking is determined the same way as view blocking described above.

2.2.7 RavelActionWrapper

The *RavelActionWrapper* transforms Discrete, MultiBinary, MultiDiscrete, bounded integer Box, and any nesting of those spaces into a Discrete space by "ravelling" their values according to numpy's `ravel_multi_index` function. Thus, actions that are represented by arrays are converted into unique Discrete numbers. For example, we can apply the RavelActionWrapper to the MoveActor, like so:

```
from abmarl.sim.gridworld.agent import MovingAgent
from abmarl.sim.gridworld.grid import Grid
from abmarl.sim.gridworld.state import PositionState
from abmarl.sim.gridworld.actor import MoveActor
from abmarl.sim.gridworld.wrapper import RavelActionWrapper

agents = {
    'agent0': MovingAgent(id='agent0', encoding=1, move_range=1),
    'agent1': MovingAgent(id='agent1', encoding=1, move_range=2)
}
grid = Grid(5, 5)
position_state = PositionState(agents=agents, grid=grid)
move_actor = MoveActor(agents=agents, grid=grid)
for agent in agents.values():
    agent.finalize()
position_state.reset()

# Move actor without wrapper
actions = {
    agent.id: agent.action_space.sample() for agent in agents.values()
}
print(actions)
# >>> {'agent0': OrderedDict([('move', array([1, 1]))]), 'agent1': OrderedDict([('move',␣
↪array([ 2, -1]))])}

# Wrapped move actor
move_actor = RavelActionWrapper(move_actor)
actions = {
    agent.id: agent.action_space.sample() for agent in agents.values()
}
print(actions)
# >>> {'agent0': OrderedDict([('move', 1)]), 'agent1': OrderedDict([('move', 22)])}
```

The actions from the unwrapped actor are in the original *Box* space, whereas after we apply the wrapper, the actions from the wrapped actor are in the transformed *Discrete* space. The actor will receive move actions in the *Discrete* space and convert them to the *Box* space before passing them to the MoveActor.

CHAPTER

THREE

FEATURED USE CASES

3.1 Emergent Collaborative and Competitive Behavior

In this experiment, we study how collaborative and competitive behaviors emerge among agents in a partially observable stochastic game. In our simulation, each agent occupies a square and can move around the map. Each agent can "attack" agents that are on a different "team"; the attacked agent loses its life and is removed from the simulation. Each agent can observe the state of the map in a region surrounding its location. It can see other agents and what team they're on as well as the edges of the map. The diagram below visually depicts the agents' observation and action spaces.

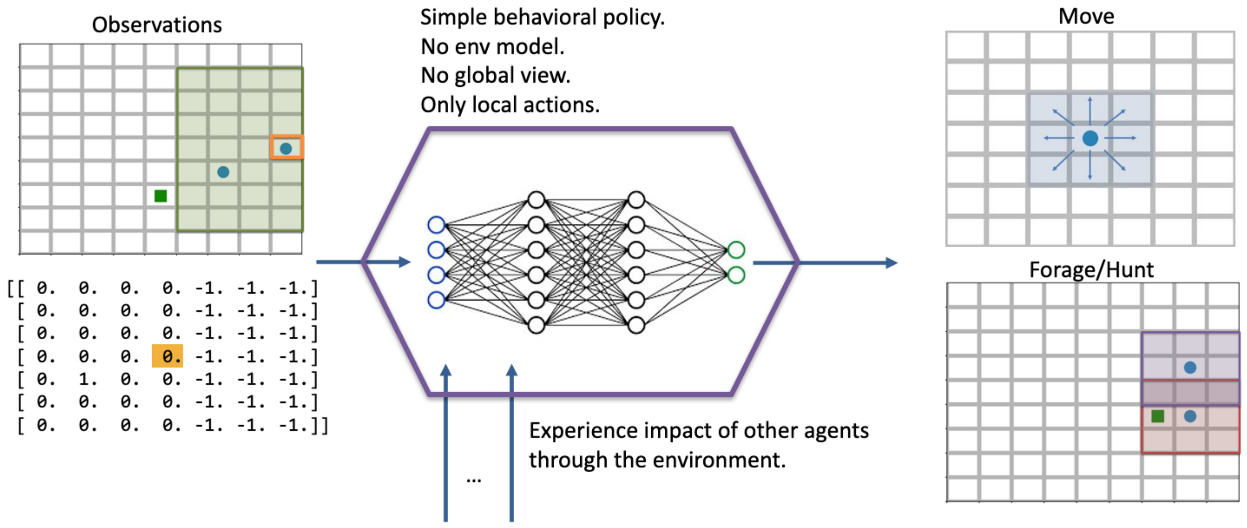


Fig. 1: Each agent has a partial observation of the map centered around its location. The green box shows the orange agent's observation of the map, and the matrix below it shows the actual observation. Each agent can choose to move or to "attack" another agent in one of the nearby squares. The policy is just a simple 2-layer MLP, each layer having 64 units. We don't apply any kind of specialized architecture that encourages collaboration or competition. Each agent is simple: they do not have a model of the simulation; they do not have a global view of the simulation; their actions are only local in both space and in agent interaction (they can only interact with one agent at a time). Yet, we will see efficient and complex strategies emerge, collaboration and competition from the common or conflicting interest among agents.

In the various examples below, each policy is a two-layer MLP, with 64 units in each layer. We use RLlib's A2C Trainer with default parameters and train for two million episodes on a compute node with 72 CPUs.

Attention: This page makes heavy use of animated graphics. It is best to read this content on our html site instead of our pdf manual.

3.1.1 Single Agent Foraging

We start by considering a single foraging agent whose objective is to move around the map collecting resource agents. The single forager can see up to three squares away, move up to one square away, and forage ("attack") resources up to one square away. The forager is rewarded for every resource it collects and given a small penalty for attempting to move off the map and an even smaller "entropy" penalty every time-step to encourage it to act quickly. At the beginning of every episode, the agents spawn at random locations in the map. Below is a video showing a typical full episode of the learned behavior and a brief analysis.

Note: From an Agent Based Modeling perspective, the resources are technically agents themselves. However, since they don't do or see anything, we tend not to call them agents in the text that follows.

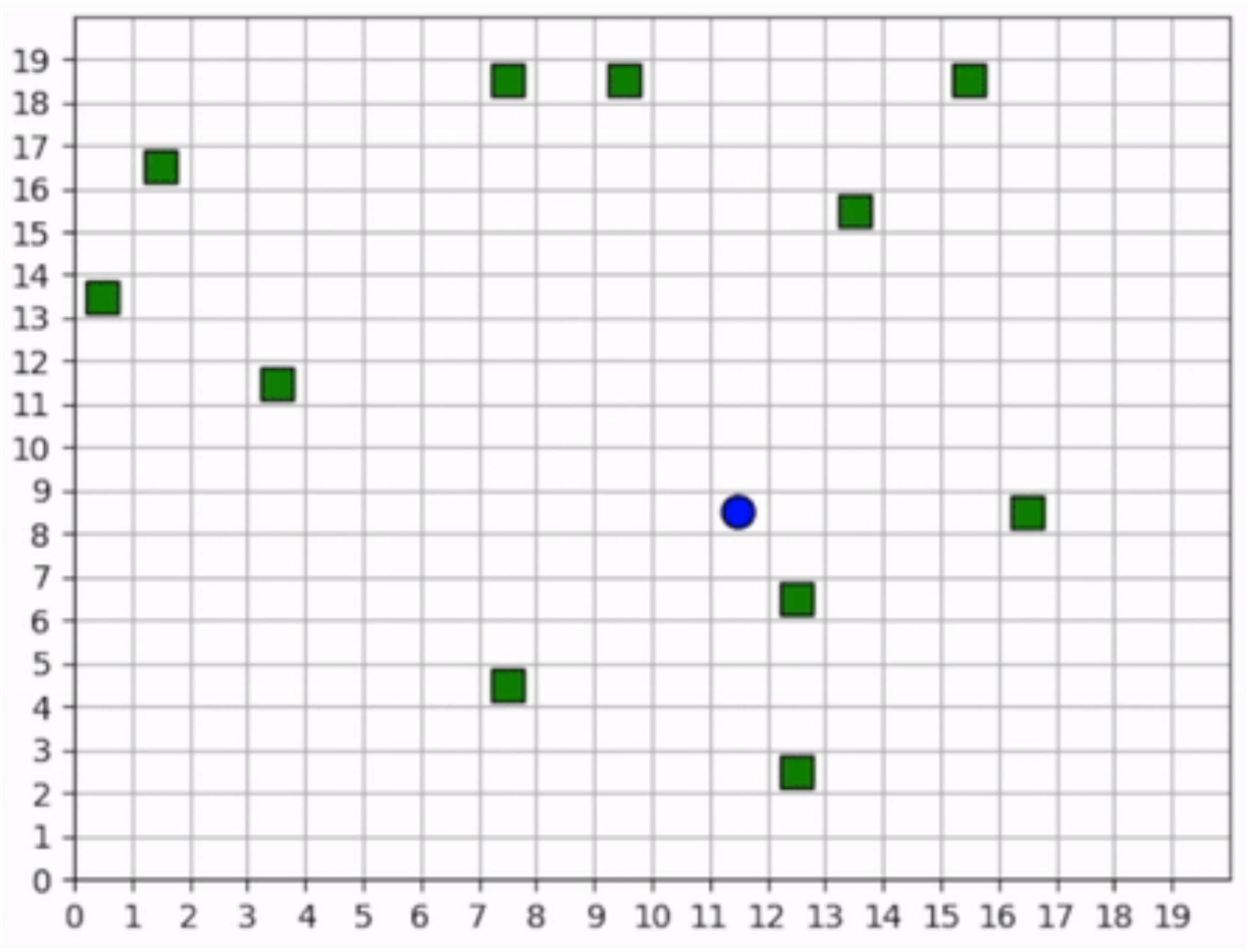


Fig. 2: A full episode showing the forager's learned strategy. The forager is the blue circle and the resources are the green squares. Notice how the forager bounces among resource clusters, greedily collecting all local resources before exploring the map for more.

When it can see resources

The forager moves toward the closest resource that it observes and collects it. Note that the foraging range is 1 square: the forager rarely waits until it is directly over a resource; it usually forages as soon as it is within range. In some cases, the forager intelligently places itself in the middle of 2-3 resources in order to forage within the least number of moves. When the resources are near the edge of the map, it behaves with some inefficiency, likely due to the small penalty we give it for moving off the map, which results in an aversion towards the map edges. Below is a series of short video clips showing the foraging strategy.

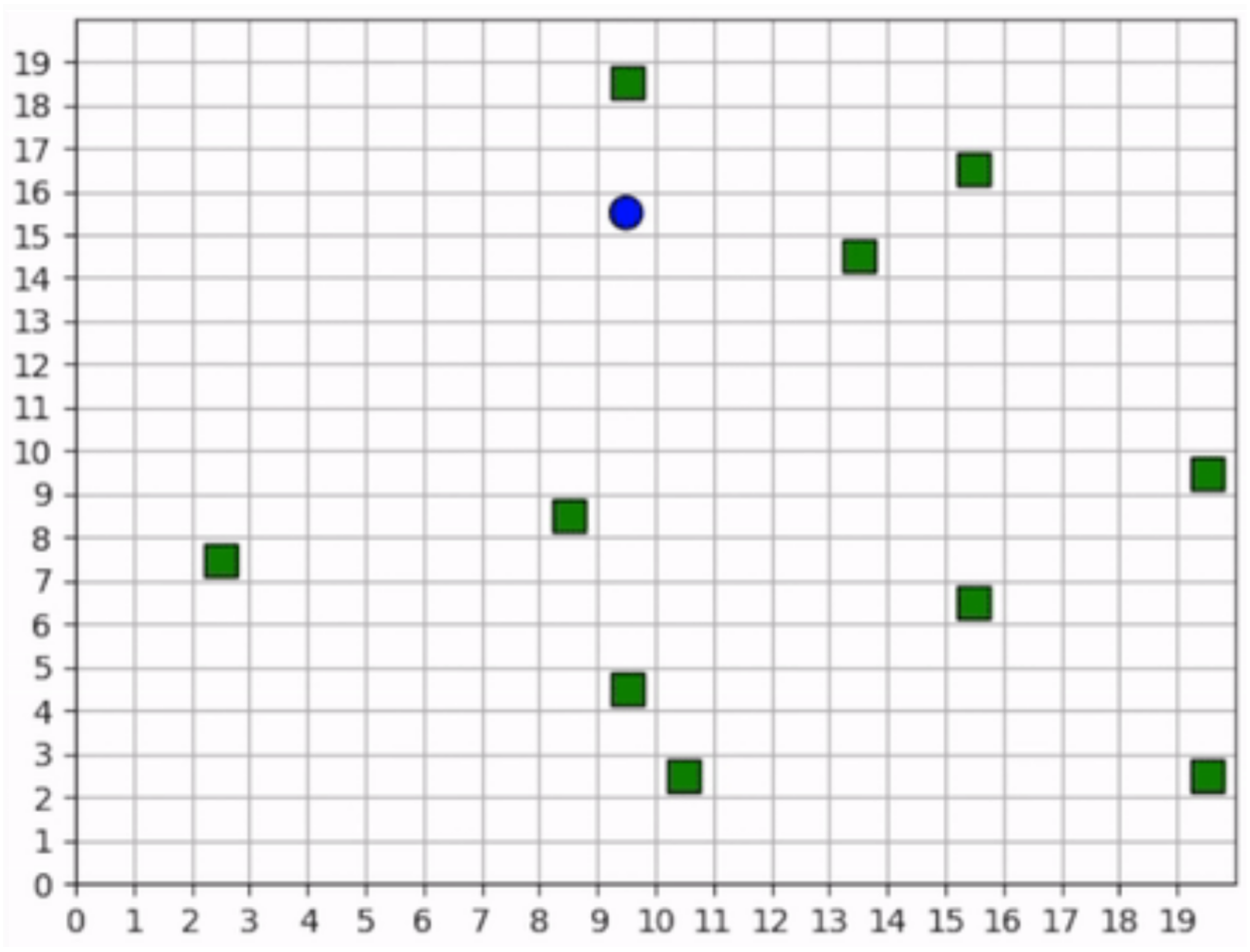


Fig. 3: The forager learns an effective foraging strategy, moving towards and collecting the nearest resources that it observes.

When it cannot see resources

The forager's behavior when it is near resources is not surprising. But how does it behave when it cannot see any resources? The forager only sees that which is near it and does not have any information distinguishing one "deserted" area of the map from another. Recall, however, that it observes the edges of the map, and it uses this information to learn an effecive exploration strategy. In the video below, we can see that the forager learns to explore the map by moving along its edges in a clockwise direction, occasionally making random moves towards the middle of the map.

Important: We do not use any kind of heuristic or mixed policy. The exporation strategy *emerges* entirely from

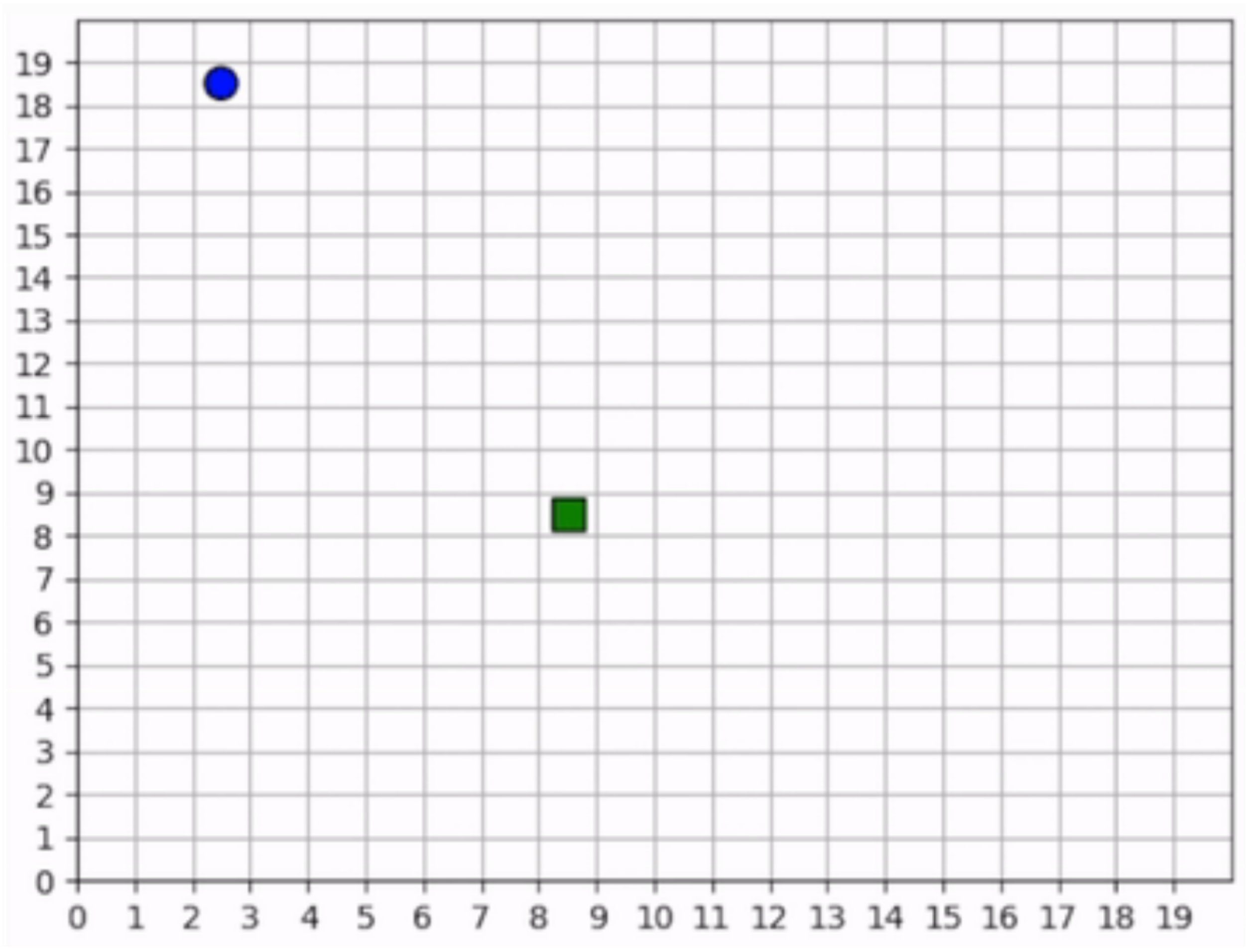


Fig. 4: The forager learns an effective exploration strategy, moving along the edge of the map in a clockwise direction.

reinforcement learning.

3.1.2 Multiple Agents Foraging

Having experimented with a single forager, let us now turn our attention to the strategies learned by multiple foragers interacting in the map at the same time. Each forager is homogeneous with each other as described above: they can all move up to one square away, observe up to three squares away, and are rewarded the same way. The observations include other foragers in addition to the resources and map edges. All agents share a single policy. Below is a brief analysis of the learned behaviors.

Cover and explore

Our reward schema implicitly encourages the foragers to collaborate because we give a small penalty to each one for taking too long. Thus, the faster they can collect all the resources, the less they are penalized. Furthermore, because each agent trains the same policy, there is no incentive for competitive behavior. An agent can afford to say, "I don't need to get the resource first. As long as one of us gets it quickly, then we all benefit". Therefore, the foragers learn to spread out to *cover* the map, maximizing the amount of squares that are observed.

In the video clips below, we see that the foragers avoid being within observation distance of one another. Typically, when two foragers get too close, they repel each other, each moving in opposite directions, ensuring that the space is *covered*. Furthermore, notice the dance-like exploration strategy. Similar to the single-agent case above, they learn to *explore* along the edges of the map in a clockwise direction. However, they're not as efficient as the single agent because they "repel" each other.

Important: We do not directly incentivize agents to keep their distance. No part of the reward schema directly deals with the agents' distances from each other. These strategies are *emergent*.

Breaking the pattern

When a forager observes a resource, it breaks its "cover and explore" strategy and moves directly for the resource. Even multiple foragers move towards the same resource. They have no reason to coordinate who will get it because, as we stated above, there is no incentive for competition, so no need to negotiate. If another forager gets there first, everyone benefits. The foragers learn to prioritize collecting the resources over keeping their distance from each other.

Tip: We should expect to see both of these strategies occuring at the same time within a simulation because while some agents are "covering and exploring", others are moving towards resources.

3.1.3 Introducing Hunters

So far, we have seen intelligent behaviors emerge in both single- and multi-forager scenarios; we even saw the emergence of collaborative behavior. In the following experiments, we explore competitive emergence by introducing hunters into the simulation. Like foragers, hunters can move up to one square away and observe other agents and map edges up to three squares away. Hunters, however, are more effective killers and can attack a forager up to two squares away. They are rewarded for successful kills, they are and penalized for bad moves and for taking too long, exactly the same way as foragers.

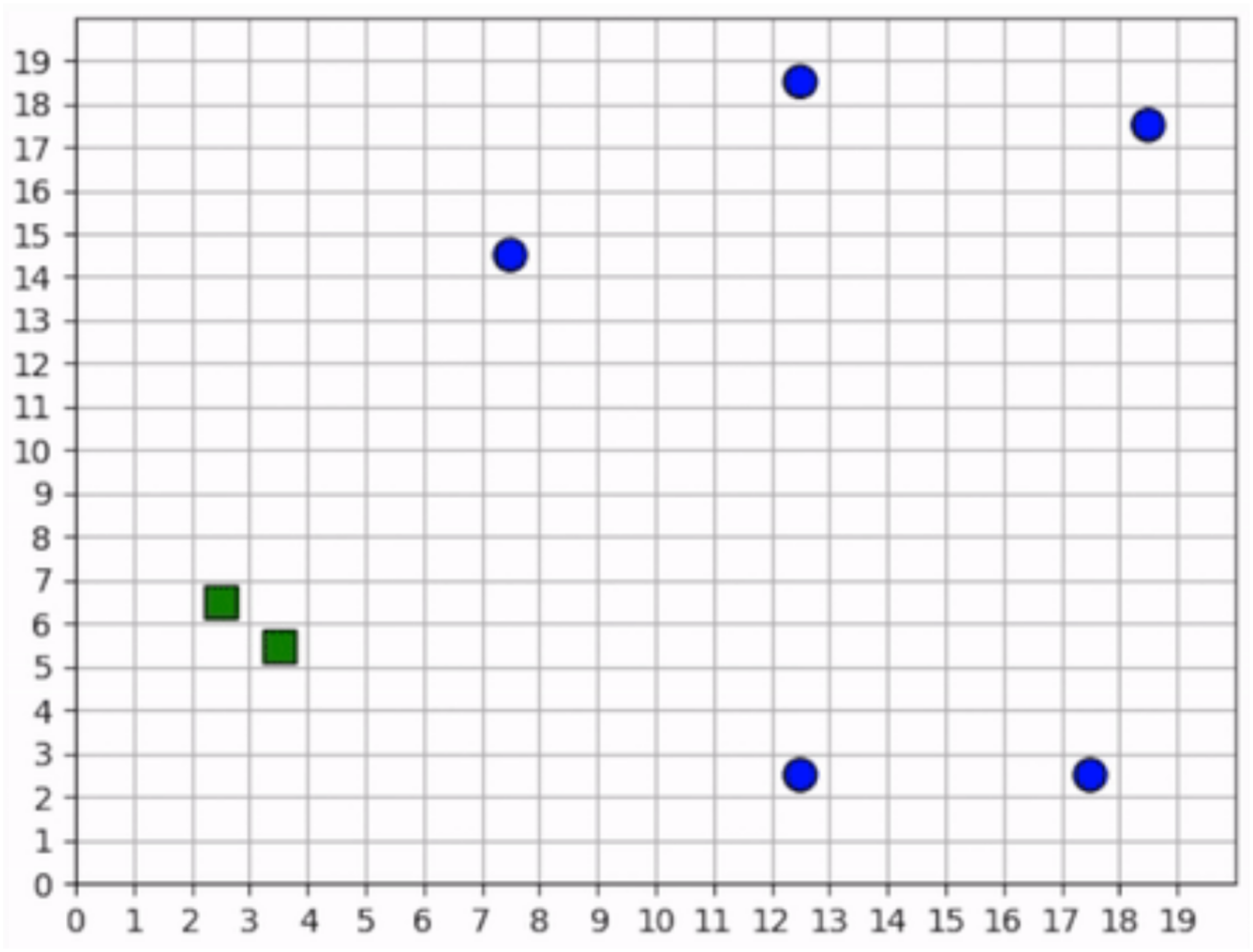


Fig. 5: The foragers cover the map by spreading out and explore it by traveling in a clockwise direction.

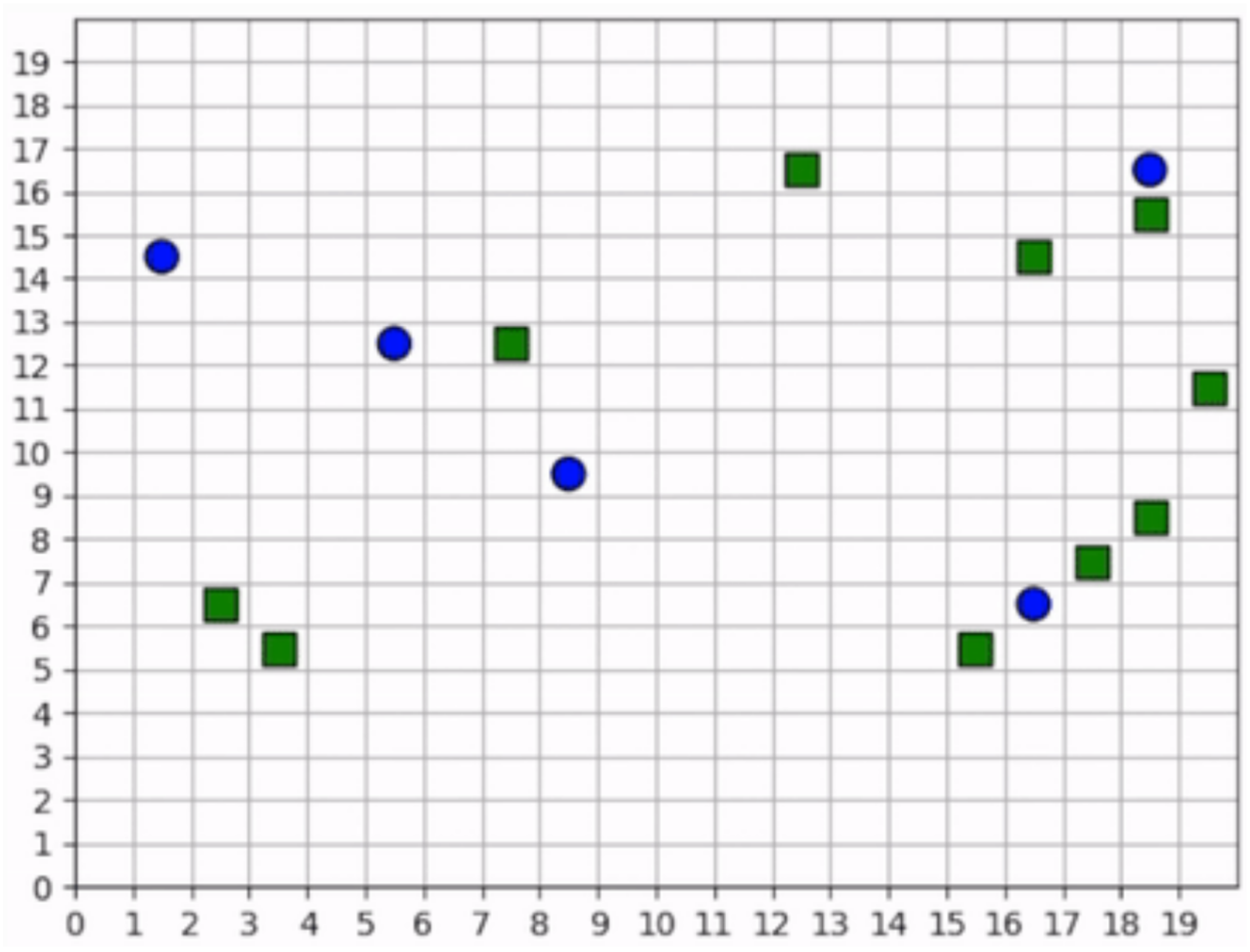
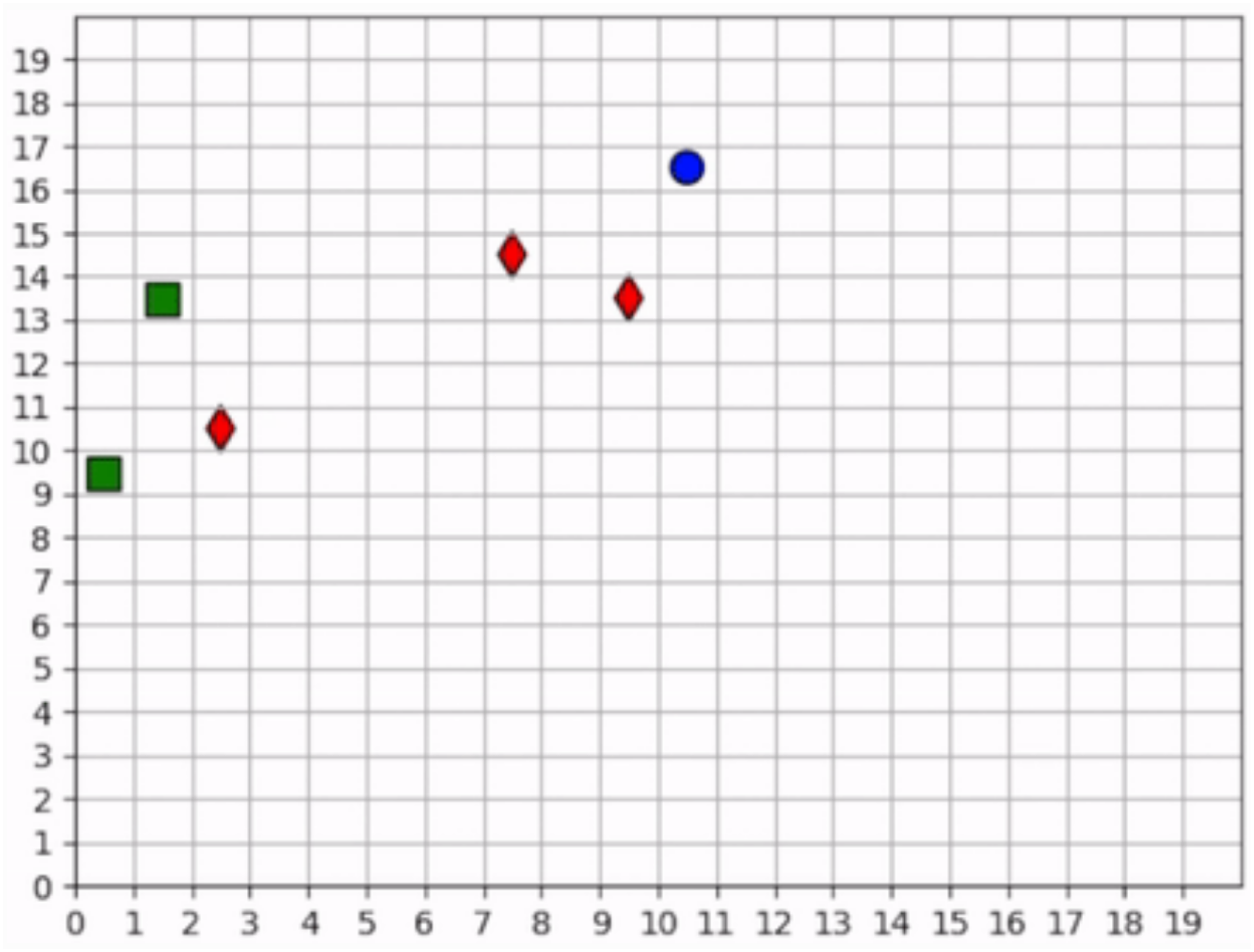


Fig. 6: The foragers move towards resources to forage, even when there are other foragers nearby.

However, the hunters and foragers have completely different objectives: a forager tries to clear the map of all *resources*, but a hunter tries to clear the map of all *foragers*. Therefore, we set up two policies. All the hunters will train the same policy, and all the foragers will train the same policy, and these policies will be distinct.

The learned behaviors among the two groups in this mixed collaborate-competitive simulation are tightly integrated, with multiple strategies appearing at the same time within a simulation. Therefore, in contrast to above, we will not show video clips that capture a single strategy; instead, we will show video clips that capture multiple strategies and attempt to describe them in detail.

First Scenario



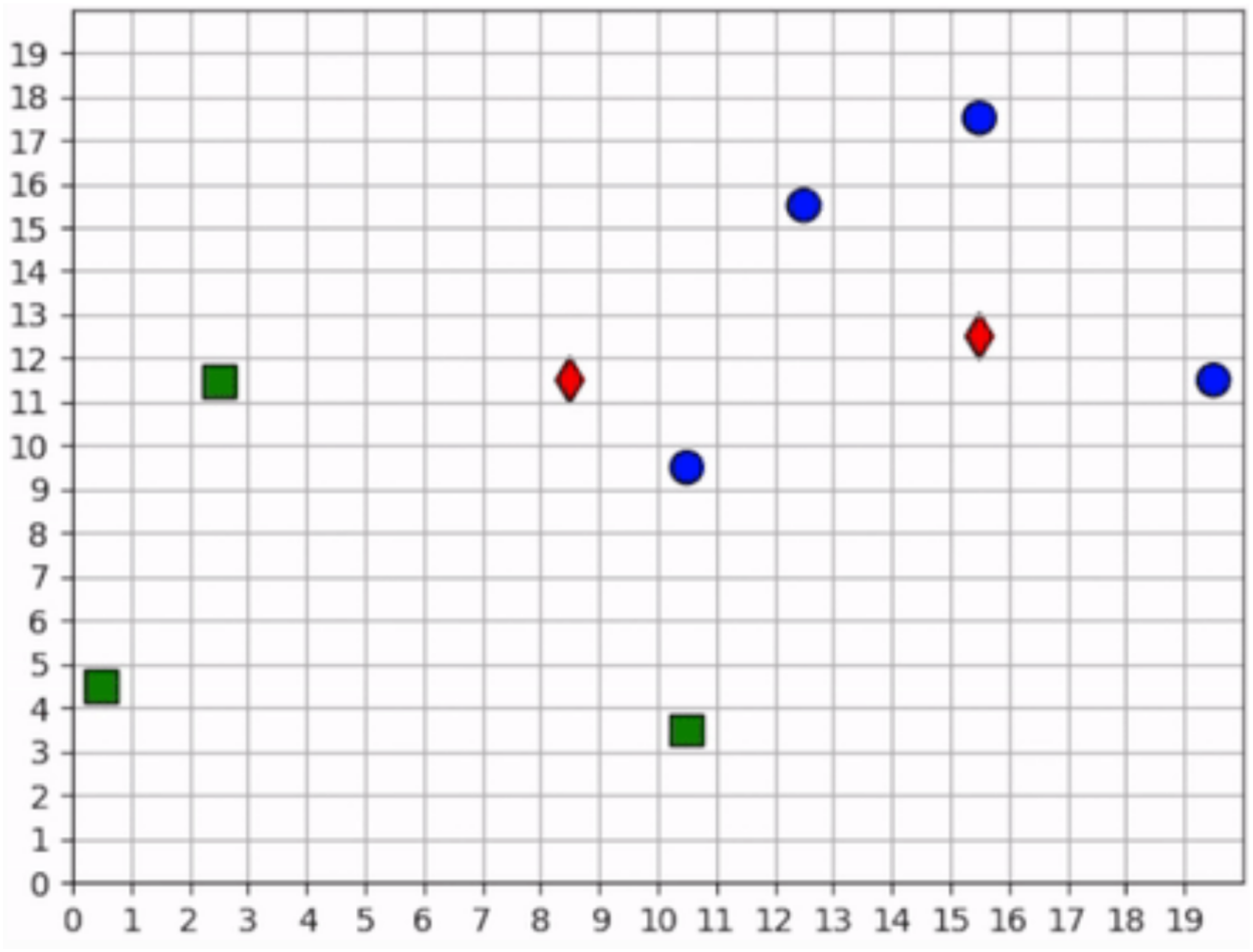
Two of the foragers spawn next to hunters and are killed immediately. Afterwards, the two hunters on the left do not observe any foragers for some time. They seem to have learned the *cover* strategy by spreading out, but they don't seem to have learned an effecient *explore* strategy since they mostly occupy the same region of the map for the duration of the simulation.

Three foragers remain at the bottom of the map. These foragers work together to collect all nearby resources. Just as they finish the resource cluster, a hunter moves within range and begins to chase them towards the bottom of the map. When they hit the edge, they split in two directions. The hunter kills one of them and then waits for one step, unsure about which forager to persue next. After one step, we see that it decides to persue the forager to the right.

Meanwhile, the forager to the left continues to run away, straight into the path of another hunter but also another resource. The forager could get away by running to the right, but it decides to collect the resource at the cost of its own life.

The last remaining forager has escaped the hunter and has conveniently found another cluster of resources, which it collects. A few frames later, it encounters the same hunter, and this time it is chased all the way across the map. It manages to evade the hunter and collect one final resource before encountering yet another hunter. At the end, we see both hunters chasing the forager to the top of the map, boxing it in and killing it.

Second scenario



None of the foragers are under threat at the beginning of this scenario. They clear a cluster of resources before one of them wanders into the path of a hunter. The hunter gives chase, and the forager actually leads the hunter back to the group. This works to its benefit, however, as the hunter is repeatedly confused by the foragers exercising the *splitting* strategy. Meanwhile the second hunter has spotted a forager and joins the hunt. The two hunters together are able to split up the pack of foragers and systematically hunt them down. The last forager is chased into the corner and killed.

Note: Humorously, the first forager that was spotted is the one who manages to stay alive the longest.

CHAPTER

FOUR

INSTALLATION

4.1 User Installation

You can install abmarl via *pip*:

```
pip install abmarl
```

4.2 Developer Installation

To install Abmarl for development, first clone the repository and then install via pip's development mode.

```
git clone git@github.com:LLNL/Abmarl.git
cd abmarl
pip install -r requirements.txt
pip install -e . --no-deps
```

Warning: If you are using *conda* to manage your virtual environment, then you must also install ffmpeg.

CHAPTER

FIVE

FULL TUTORIALS

We provide tutorials that demonstrate how to train, visualize, and analyze MARL policies. We also provide tutorials on the GridWorldSimulation framework.

5.1 MultiCorridor

MultiCorridor extends RLlib's simple corridor, wherein agents must learn to move to the right in a one-dimensonal corridor to reach the end. Our implementation provides the ability to instantiate multiple agents in the simulation and restricts agents from occupying the same square. Every agent is homogeneous: they all have the same action space, observation space, and objective function.

Fig. 1: Animation of agents moving left and right in a corridor until they reach the end.

This tutorial uses the MultiCorridor simulation and the MultiCorridor configuration.

5.1.1 Creating the MultiCorridor Simulation

The Agents in the Simulation

It's helpful to start by thinking about what we want the agents to learn and what information they will need in order to learn it. In this tutorial, we want to train agents that can reach the end of a one-dimensional corridor without bumping into each other. Therefore, agents should be able to move left, move right, and stay still. In order to move to the end of the corridor without bumping into each other, they will need to see their own position and if the squares near them are occupied. Finally, we need to decide how to reward the agents. There are many ways we can do this, and we should at least capture the following:

- The agent should be rewarded for reaching the end of the corridor.
- The agent should be penalized for bumping into other agents.
- The agent should be penalized for taking too long.

Since all our agents are homogeneous, we can create them in the Agent Based Simulation itself, like so:

```
from enum import IntEnum

from gym.spaces import Box, Discrete, MultiBinary
import numpy as np

from abmarl.sim import Agent, AgentBasedSimulation

class MultiCorridor(AgentBasedSimulation):

    class Actions(IntEnum): # The three actions each agent can take
        LEFT = 0
        STAY = 1
        RIGHT = 2

    def __init__(self, end=10, num_agents=5):
        self.end = end
        agents = {}
        for i in range(num_agents):
            agents[f'agent{i}'] = Agent(
                id=f'agent{i}',
                action_space=Discrete(3), # Move left, stay still, or move right
                observation_space={
                    'position': Box(0, self.end-1, (1,), int), # Observe your own␣
↪position
                    'left': MultiBinary(1), # Observe if the left square is occupied
                    'right': MultiBinary(1) # Observe if the right square is occupied
                }
            )
        self.agents = agents

        self.finalize()
```

Here, notice how the agents' *observation_space* is a *dict* rather than a *gym.space.Dict*. That's okay because our *Agent* class can convert a *dict of gym spaces* into a *Dict* when `finalize` is called at the end of `__init__`.

Resetting the Simulation

At the beginning of each episode, we want the agents to be randomly positioned throughout the corridor without occupying the same squares. We must give each agent a position attribute at reset. We will also create a data structure that captures which agent is in which cell so that we don't have to do a search for nearby agents but can directly index the space. Finally, we must track the agents' rewards.

```
def reset(self, **kwargs):
    location_sample = np.random.choice(self.end-1, len(self.agents), False)
    # Track the squares themselves
    self.corridor = np.empty(self.end, dtype=object)
    # Track the position of the agents
    for i, agent in enumerate(self.agents.values()):
        agent.position = location_sample[i]
        self.corridor[location_sample[i]] = agent

    # Track the agents' rewards over multiple steps.
```

(continues on next page)

(continued from previous page)

```
    self.reward = {agent_id: 0 for agent_id in self.agents}
```

Stepping the Simulation

The simulation is driven by the agents' actions because there are no other dynamics. Thus, the MultiCorridor Simulation only concerns itself with processing the agents' actions at each step. For each agent, we'll capture the following cases:

- An agent attempts to move to a space that is unoccupied.
- An agent attempts to move to a space that is already occupied.
- An agent attempts to move to the right-most space (the end) of the corridor.

```
def step(self, action_dict, **kwargs):
    for agent_id, action in action_dict.items():
        agent = self.agents[agent_id]
        if action == self.Actions.LEFT:
            if agent.position != 0 and self.corridor[agent.position-1] is None:
                # Good move, no extra penalty
                self.corridor[agent.position] = None
                agent.position -= 1
                self.corridor[agent.position] = agent
                self.reward[agent_id] -= 1 # Entropy penalty
            elif agent.position == 0: # Tried to move left from left-most square
                # Bad move, only acting agent is involved and should be penalized.
                self.reward[agent_id] -= 5 # Bad move
            else: # There was another agent to the left of me that I bumped into
                # Bad move involving two agents. Both are penalized
                self.reward[agent_id] -= 5 # Penalty for offending agent
                # Penalty for offended agent
                self.reward[self.corridor[agent.position-1].id] -= 2
        elif action == self.Actions.RIGHT:
            if self.corridor[agent.position + 1] is None:
                # Good move, but is the agent done?
                self.corridor[agent.position] = None
                agent.position += 1
                if agent.position == self.end-1:
                    # Agent has reached the end of the corridor!
                    self.reward[agent_id] += self.end ** 2
                else:
                # Good move, no extra penalty
                    self.corridor[agent.position] = agent
                    self.reward[agent_id] -= 1 # Entropy penalty
            else: # There was another agent to the right of me that I bumped into
                # Bad move involving two agents. Both are penalized
                self.reward[agent_id] -= 5 # Penalty for offending agent
                # Penalty for offended agent
                self.reward[self.corridor[agent.position+1].id] -= 2
        elif action == self.Actions.STAY:
            self.reward[agent_id] -= 1 # Entropy penalty
```

Attention: Our reward schema reveals a training dynamic that is not present in single-agent simulations: an agent's reward does not entirely depend on its own interaction with the simulation but can be affected by other agents' actions. In this case, agents are slightly penalized for being "bumped into" when other agents attempt to move onto their square, even though the "offended" agent did not directly cause the collision. This is discussed in MARL literature and captured in the way we have designed our Simulation Managers. In Abmarl, we favor capturing the rewards as part of the simulation's state and only "flushing" them once they rewards are asked for in `get_reward`.

Note: We have not needed to consider the order in which the simulation processes actions. Our simulation simply provides the capabilities to process *any* agent's action, and we can use *Simulation Managers* to impose an order. This shows the flexibility of our design. In this tutorial, we will use the *TurnBasedManager*, but we can use any *Simulation-Manager*.

Querying Simulation State

The trainer needs to see how agents' actions impact the simulation's state. They do so via getters, which we define below.

```
def get_obs(self, agent_id, **kwargs):
    agent_position = self.agents[agent_id].position
    if agent_position == 0 or self.corridor[agent_position-1] is None:
        left = False
    else:
        left = True
    if agent_position == self.end-1 or self.corridor[agent_position+1] is None:
        right = False
    else:
        right = True
    return {
        'position': [agent_position],
        'left': [left],
        'right': [right],
    }

def get_done(self, agent_id, **kwargs):
    return self.agents[agent_id].position == self.end - 1

def get_all_done(self, **kwargs):
    for agent in self.agents.values():
        if agent.position != self.end - 1:
            return False
    return True

def get_reward(self, agent_id, **kwargs):
    agent_reward = self.reward[agent_id]
    self.reward[agent_id] = 0
    return agent_reward

def get_info(self, agent_id, **kwargs):
    return {}
```

Rendering for Visualization

Finally, it's often useful to be able to visualize a simulation as it steps through an episode. We can do this via the render funciton.

```
def render(self, *args, fig=None, **kwargs):
    draw_now = fig is None
    if draw_now:
        from matplotlib import pyplot as plt
        fig = plt.gcf()

    fig.clear()
    ax = fig.gca()
    ax.set(xlim=(-0.5, self.end + 0.5), ylim=(-0.5, 0.5))
    ax.set_xticks(np.arange(-0.5, self.end + 0.5, 1.))
    ax.scatter(np.array(
        [agent.position for agent in self.agents.values()]),
        np.zeros(len(self.agents)),
        marker='s', s=200, c='g'
    )

    if draw_now:
        plt.plot()
        plt.pause(1e-17)
```

5.1.2 Training the MultiCorridor Simulation

Now that we have created the simulation and agents, we can create a configuration file for training.

Simulation Setup

We'll start by setting up the simulation we have just built. Then we'll choose a Simulation Manager. Abmarl comes with two built-In managers: *TurnBasedManager*, where only a single agent takes a turn per step, and *AllStepManager*, where all non-done agents take a turn per step. For this experiment, we'll use the *TurnBasedManager*. Then, we'll wrap the simulation with our *MultiAgentWrapper*, which enables us to connect with RLlib. Finally, we'll register the simulation with RLlib.

```
# MultiCorridor is the simulation we created above
from abmarl.sim.corridor import MultiCorridor
from abmarl.managers import TurnBasedManager
# MultiAgentWrapper needed to connect with RLlib
from abmarl.external import MultiAgentWrapper

# Create an instance of the simulation and register it
sim = MultiAgentWrapper(TurnBasedManager(MultiCorridor()))
sim_name = "MultiCorridor"
from ray.tune.registry import register_env
register_env(sim_name, lambda sim_config: sim)
```

Policy Setup

Now we want to create the policies and the policy mapping function in our multiagent experiment. Each agent in our simulation is homogeneous: they all have the same observation space, action space, and objective function. Thus, we can create a single policy and map all agents to that policy.

```
ref_agent = sim.unwrapped.agents['agent0']
policies = {
    'corridor': (None, ref_agent.observation_space, ref_agent.action_space, {})
}
def policy_mapping_fn(agent_id):
    return 'corridor'
```

Experiment Parameters

Having setup the simulation and policies, we can now bundle all that information into a parameters dictionary that will be read by Abmarl and used to launch RLlib.

```
params = {
    'experiment': {
        'title': f'{sim_name}',
        'sim_creator': lambda config=None: sim,
    },
    'ray_tune': {
        'run_or_experiment': 'PG',
        'checkpoint_freq': 50,
        'checkpoint_at_end': True,
        'stop': {
            'episodes_total': 2000,
        },
        'verbose': 2,
        'config': {
            # --- Simulation ---
            'env': sim_name,
            'horizon': 200,
            'env_config': {},
            # --- Multiagent ---
            'multiagent': {
                'policies': policies,
                'policy_mapping_fn': policy_mapping_fn,
            },
            # --- Parallelism ---
            # Number of workers per experiment: int
            "num_workers": 7,
            # Number of simulations that each worker starts: int
            "num_envs_per_worker": 1, # This must be 1 because we are not "threadsafe"
        },
    }
}
```

Command Line interface

With the configuration file complete, we can utilize the command line interface to train our agents. We simply type `abmarl train multi_corridor_example.py`, where *multi_corridor_example.py* is the name of our configuration file. This will launch Abmarl, which will process the file and launch RLlib according to the specified parameters. This particular example should take 1-10 minutes to train, depending on your compute capabilities. You can view the performance in real time in tensorboard with `tensorboard --logdir ~/abmarl_results`.

Visualizing the Trained Behaviors

We can visualize the agents' learned behavior with the `visualize` command, which takes as argument the output directory from the training session stored in `~/abmarl_results`. For example, the command

```
abmarl visualize ~/abmarl_results/MultiCorridor-2020-08-25_09-30/ -n 5 --record
```

will load the experiment (notice that the directory name is the experiment title from the configuration file appended with a timestamp) and display an animation of 5 episodes. The `--record` flag will save the animations as *.mp4* videos in the training directory.

5.1.3 Extra Challenges

Having successfully trained a MARL experiment, we can further explore the agents' behaviors and the training process. Some ideas are:

- We could enhance the MultiCorridor Simulation so that the "target" cell is a different location in each episode.
- We could introduce heterogeneous agents with the ability to "jump over" other agents. With heterogeneous agents, we can nontrivially train multiple policies.
- We could study how the agents' behaviors differ if they are trained using the *AllStepManager*.
- We could create our own Simulation Manager so that if an agent causes a collision, it skips its next turn.
- We could do a parameter search over both simulation and algorithm parameters to study how the parameters affect the learned behaviors.
- We could analyze how often agents collide with one another and where those collisions most commonly occur.
- And much, much more!

As we attempt these extra challenges, we will experience one of Abmarl's strongest features: the ease with which we can modify our experiment file and launch another training job, going through the pipeline from experiment setup to behavior visualization and analysis!

5.2 GridWorld

The GridWorld Simulation Framework is composed of feature components that fit together to allow users to create a variety of simulations using the same pieces and to easily design their own features. We provide tutorials demonstrating the special features of this framework. First, we create a multi-team battle simulation using built-in features components. We then show how the exact same components can be reconfigured to create a maze-navigation simulation. Finally, we show how easy it is to add custom features as components and plug them into the simulation framework.

5.2.1 Team Battle

The Team Battle scenario involves multiple teams of agents fighting against each other. The goal of each team is to be the last team alive, at which point the simulation will end. Each agent can move around the grid and attack agents from other teams. Each agent can observe the grid around its position. We will reward each agent for successful kills and penalize them for bad moves. This tutorial can be found in full in our repo.

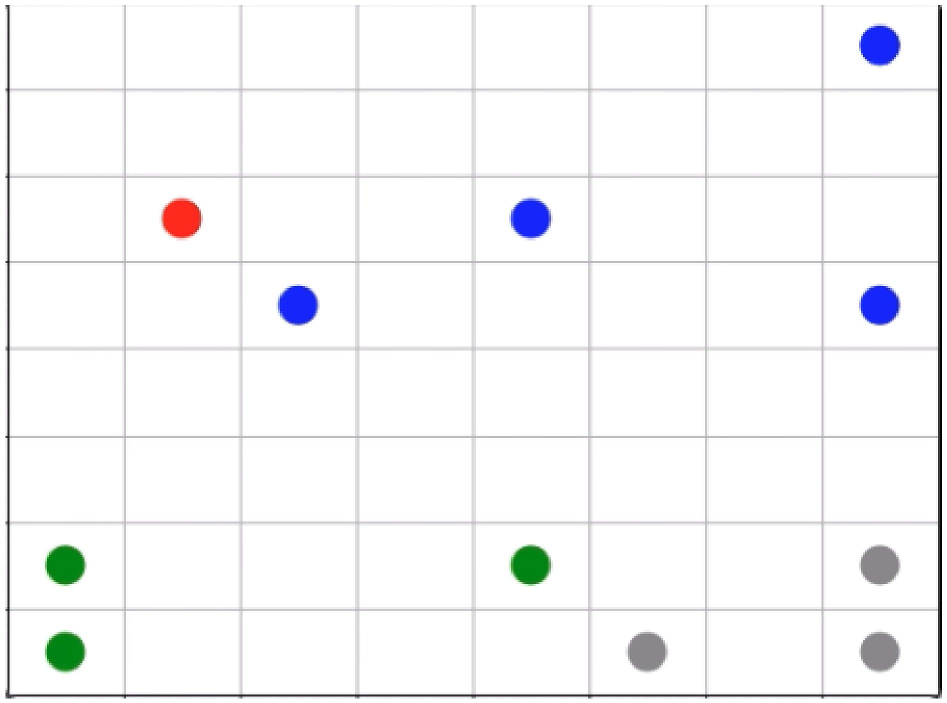


Fig. 2: Agents on four teams battling each other.

First, we import the components that we need. Each component is *already in Abmarl*, so we don't need to create anything new.

```
from matplotlib import pyplot as plt
import numpy as np

from abmarl.sim.gridworld.base import GridWorldSimulation
from abmarl.sim.gridworld.agent import GridObservingAgent, MovingAgent, AttackingAgent,␣
↪HealthAgent
from abmarl.sim.gridworld.state import HealthState, PositionState
from abmarl.sim.gridworld.actor import MoveActor, AttackActor
from abmarl.sim.gridworld.observer import SingleGridObserver
from abmarl.sim.gridworld.done import OneTeamRemainingDone
```

Then, we define our agent types. This simulation will only have a single type: the BattleAgent. Most of the agents' attributes will be the same, and we can preconfigure the class definition to save us time when we create the agents later on.

```
class BattleAgent(GridObservingAgent, MovingAgent, AttackingAgent, HealthAgent):
    def __init__(self, **kwargs):
        super().__init__(
            move_range=1,
```

(continues on next page)

(continued from previous page)

```
            attack_range=1,
            attack_strength=1,
            attack_accuracy=1,
            view_range=3,
            **kwargs
        )
```

Having defined the BattleAgent, we then put all the components together into a single simulation.

```
class TeamBattleSim(GridWorldSimulation):
    def __init__(self, **kwargs):
        self.agents = kwargs['agents']

        # State Components
        self.position_state = PositionState(**kwargs)
        self.health_state = HealthState(**kwargs)

        # Action Components
        self.move_actor = MoveActor(**kwargs)
        self.attack_actor = AttackActor(**kwargs)

        # Observation Components
        self.grid_observer = SingleGridObserver(**kwargs)

        # Done Compoennts
        self.done = OneTeamRemainingDone(**kwargs)

        self.finalize()
```

Next we define the start state of each simulation. We lean on the *State Components* to perform the reset. Note that we must track the rewards explicitly.

```
class TeamBattleSim(GridWorldSimulation):
    ...

    def reset(self, **kwargs):
        self.position_state.reset(**kwargs)
        self.health_state.reset(**kwargs)

        # Track the rewards
        self.rewards = {agent.id: 0 for agent in self.agents.values()}
```

Then we define how the simulation will step forward, leaning on the *Actors* to process their part of the action. The Actors' result determine the agents' rewards.

```
class TeamBattleSim(GridWorldSimulation):
    ...

    def step(self, action_dict, **kwargs):
        # Process attacks:
        for agent_id, action in action_dict.items():
            agent = self.agents[agent_id]
            attacked_agent = self.attack_actor.process_action(agent, action, **kwargs)
```

(continues on next page)

(continued from previous page)

```
            if attacked_agent is not None:
                self.rewards[attacked_agent.id] -= 1
                self.rewards[agent.id] += 1
            else:
                self.rewards[agent.id] -= 0.1

        # Process moves
        for agent_id, action in action_dict.items():
            agent = self.agents[agent_id]
            if agent.active:
                move_result = self.move_actor.process_action(agent, action, **kwargs)
                if not move_result:
                    self.rewards[agent.id] -= 0.1

        # Entropy penalty
        for agent_id in action_dict:
            self.rewards[agent_id] -= 0.01
```

Finally, we define each of the getters using the *Observers* and *Done components*.

```
class TeamBattleSim(GridWorldSimulation):
    ...

    def get_obs(self, agent_id, **kwargs):
        agent = self.agents[agent_id]
        return {
            **self.grid_observer.get_obs(agent, **kwargs)
        }

    def get_reward(self, agent_id, **kwargs):
        reward = self.rewards[agent_id]
        self.rewards[agent_id] = 0
        return reward

    def get_done(self, agent_id, **kwargs):
        return self.done.get_done(self.agents[agent_id])

    def get_all_done(self, **kwargs):
        return self.done.get_all_done(**kwargs)

    def get_info(self, agent_id, **kwargs):
        return {}
```

Now that we've defined our agents and simulation, let's create them and run it. First, we'll create the agents. There will be 4 teams, so we want to color the agent by team and start them at different corners of the grid. Besides that, all agent attributes will be the same, and here we benefit from preconfiguring the attributes in the class definition above.

```
colors = ['red', 'blue', 'green', 'gray'] # Team colors
positions = [np.array([1,1]), np.array([1,6]), np.array([6,1]), np.array([6,6])] # Grid␣
↪corners
agents = {
    f'agent{i}': BattleAgent(
```

(continues on next page)

(continued from previous page)

```
        id=f'agent{i}',
        encoding=i%4+1,
        render_color=colors[i%4],
        initial_position=positions[i%4]
    ) for i in range(24)
}
```

Having created the agents, we can now build the simulation. We will allow agents from the same team to occupy the same cell and allow agents to attack other agents if they are on different teams.

```
overlap_map = {
    1: [1],
    2: [2],
    3: [3],
    4: [4]
}
attack_map = {
    1: [2, 3, 4],
    2: [1, 3, 4],
    3: [1, 2, 4],
    4: [1, 2, 3]
}
sim = TeamBattleSim.build_sim(
    8, 8,
    agents=agents,
    overlapping=overlap_map,
    attack_mapping=attack_map
)
```

Finally, we can run the simulation with random actions and visualize it. The visualization produces an animation like the one at the top of this page.

```
sim.reset()
fig = plt.figure()
sim.render(fig=fig)

done_agents = set()
for i in range(50): # Run for at most 50 steps
    action = {
        agent.id: agent.action_space.sample() for agent in agents.values() if agent.id␣
↪not in done_agents
    }
    sim.step(action)
    sim.render(fig=fig)

    if sim.get_all_done():
        break
    for agent in agents:
        if sim.get_done(agent):
            done_agents.add(agent)
```

Extra Challenges

Having successfully created and run a TeamBattle simulation, we can further explore the GridWorldSimulation framework. Some ideas are:

- Experiment with the number of agents and the impact that has on both the SingleGridObserver and the MultiGridObserver.
- Experiment with the number of agents per team as well as the capabilities of those agents. You might find that a super capable agent is still effective against a team of multiple agents.
- Create a Hunter-Forager simulation, where one team of agents act as immobile resources that can be foraged by another team, which can be hunted by a third team. Try using the same components here, although you may need to use a custom *done condition*.
- Connect this simulation with the Reinforcement Learning capabilities of Abmarl via a *Simulation Manager*. What kind of behaviors do the agents learn?
- And much, much more!

5.2.2 Maze Navigation

Using the same components as we did in the *Team Battle tutorial*, we can create a Maze Navigation Simulation that contains a single moving agent navigating a maze defined by wall agents in the grid. The moving agent's goal is to reach a target agent. We will construct the Grid by *reading a grid file*. This tutorial can be found in full in our repo.

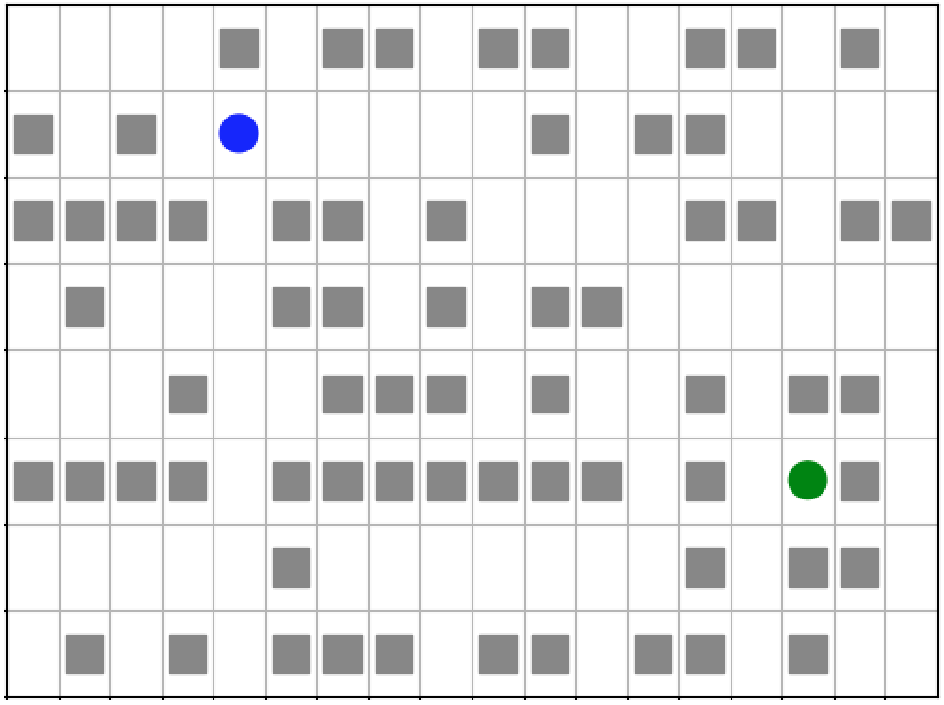


Fig. 3: Agent (blue) navigating a maze to the target (green).

Note: While we have multiple entities like walls and a target agent, the only agent that is actually doing something is the navigation agent. We will use some custom modifications to make this simulation easier, showing that we can easily use our components with custom modifications.

First we import the components that we need. Each feature is already in Abmarl, and they are the same features that we used in the *Team Battle tutorial*.

```
from matplotlib import pyplot as plt
import numpy as np

from abmarl.sim.gridworld.base import GridWorldSimulation
from abmarl.sim.gridworld.agent import GridObservingAgent, MovingAgent, GridWorldAgent
from abmarl.sim.gridworld.state import PositionState
from abmarl.sim.gridworld.actor import MoveActor
from abmarl.sim.gridworld.observer import SingleGridObserver
```

Then, we define our agent types. We need an MazeNavigationAgent, WallAgents to act as the barriers of the maze, and a TargetAgent to indicate the goal. Although we have these three types, we only need to define the MazeNavigationAgent because the WallAgent and the TargetAgent are the same as a generic *GridWorldAgent*.

```
class MazeNavigationAgent(GridObservingAgent, MovingAgent):
    def __init__(self, **kwargs):
        super().__init__(move_range=1, **kwargs)
```

Here we have preconfigured the agent with a *move range* of 1 becuase that makes the most sense for navigating mazes, but we have not preconfigured the *view range* since that is a parameter we may want to adjust, and it is easier to adjust it at the agent's initialization.

Then we define the simulation using the components and define all the necessary functions. We find it convient to explicitly store a reference to the navigation agent and the target agent. Rather than defining a new component for our simple done condition, we just write the condition itself in the function.

```
class MazeNaviationSim(GridWorldSimulation):
    def __init__(self, **kwargs):
        self.agents = kwargs['agents']

        # Store the navigation and target agents
        self.navigator = kwargs['agents']['navigator']
        self.target = kwargs['agents']['target']

        # State Components
        self.position_state = PositionState(**kwargs)

        # Action Components
        self.move_actor = MoveActor(**kwargs)

        # Observation Components
        self.grid_observer = SingleGridObserver(**kwargs)

        self.finalize()

    def reset(self, **kwargs):
        self.position_state.reset(**kwargs)

        # Since there is only one agent that produces actions, there is only one reward.
        self.reward = 0

    def step(self, action_dict, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        # Only the navigation agent will send actions, so we pull that out
        action = action_dict['navigator']
        move_result = self.move_actor.process_action(self.navigator, action, **kwargs)
        if not move_result:
            self.reward -= 0.1

        # Entropy penalty
        self.reward -= 0.01

    def get_obs(self, agent_id, **kwargs):
        # pass the navigation agent itself to the observer becuase it is the only
        # agent that takes observations
        return {
            **self.grid_observer.get_obs(self.navigator, **kwargs)
        }

    def get_reward(self, agent_id, **kwargs):
        # Custom reward function
        if self.get_all_done():
            self.reward = 1
        reward = self.reward
        self.reward = 0
        return reward

    def get_done(self, agent_id, **kwargs):
        return self.get_all_done()

    def get_all_done(self, **kwargs):
        # We define the done condition here directly rather than creating a
        # separate component for it.
        return np.all(self.navigator.position == self.target.position)

    def get_info(self, agent_id, **kwargs):
        return {}
```

With everything defined, we're ready to create and run our simulation. We will create the simulation by reading a simulation file that shows the positions of each agent type in the grid. We will use *maze.txt*, which looks like this:

```
0 0 0 0 W 0 W W 0 W W 0 0 W W 0 W 0
W 0 W 0 N 0 0 0 0 0 W 0 W W 0 0 0 0
W W W W 0 W W 0 W 0 0 0 0 W W 0 W W
0 W 0 0 0 W W 0 W 0 W W 0 0 0 0 0 0
0 0 0 W 0 0 W W W 0 W 0 0 W 0 W W 0
W W W W 0 W W W W W W W 0 W 0 T W 0
0 0 0 0 0 W 0 0 0 0 0 0 0 W 0 W W 0
0 W 0 W 0 W W W 0 W W 0 W W 0 W 0 0
```

In order to assign meaning to the values in the grid file, we must create an *object registry* that maps the values in the files to objects. We will use `W` for WallAgents, `N` for the NavigationAgent, and `T` for the TargetAgent. The values of the *object registry* must be lambda functions that take one argument and produce an agent.

```
object_registry = {
```

(continues on next page)

(continued from previous page)

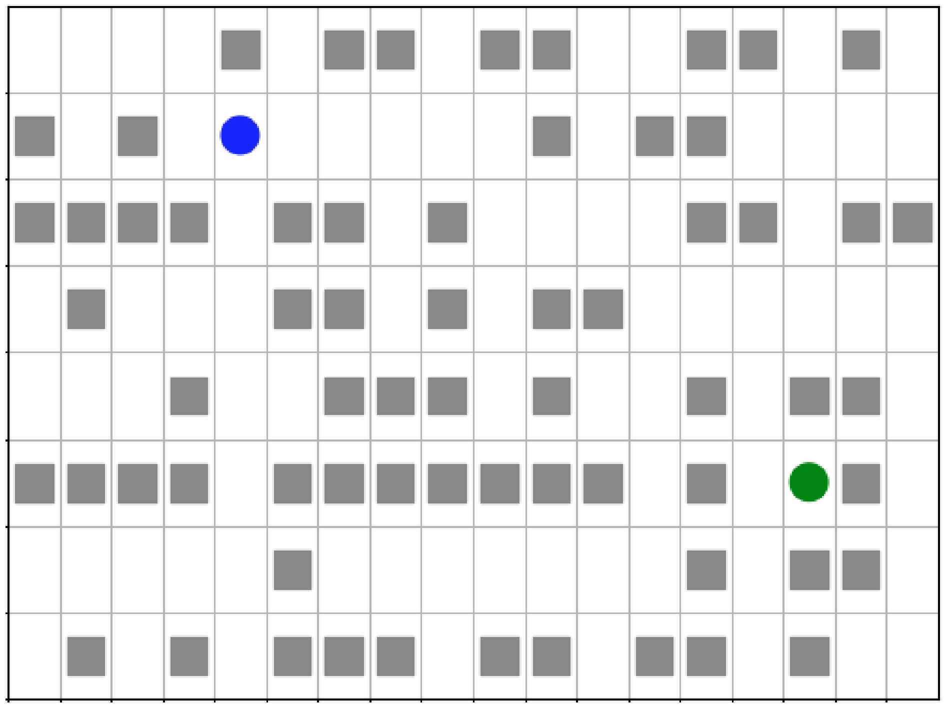
```
    'N': lambda n: MazeNavigationAgent(
        id=f'navigator',
        encoding=1,
        view_range=2, # Observation parameter that we can adjust as desired
        render_color='blue',
    ),
    'T': lambda n: GridWorldAgent(
        id=f'target',
        encoding=3,
        render_color='green'
    ),
    'W': lambda n: GridWorldAgent(
        id=f'wall{n}',
        encoding=2,
        blocking=True,
        render_shape='s'
    )
}
```

Now we can create the simulation from the maze file using the *object registry*. We must allow the navigation agent and the target agent to overlap since that is our done condition, and without it the simulation would never end. The visualization produces an animation like the one at the top of this page.

```
file_name = 'maze.txt'
sim = MazeNaviationSim.build_sim_from_file(
    file_name,
    object_registry,
    overlapping={1: [3], 3: [1]}
)
sim.reset()
fig = plt.figure()
sim.render(fig=fig)

for i in range(100):
    action = {'navigator': sim.navigator.action_space.sample()}
    sim.step(action)
    sim.render(fig=fig)
    done = sim.get_all_done()
    if done:
        plt.pause(1)
        break
```

We can examine the observation to see how the walls effect what the navigation agent can observe. An example state and observation is given below.

```
-1 -2 -2 -2 -1
 0  0  2  0  2
 2  0  1  0  0
-2  2  0  2 -2
-2 -2  0 -2 -2
```



Extra Challenges

We’ve created a starkly different simulation using many of the same components as we did in the *TeamBattle tutorial*. We can further explore the capabilities of the GridWorld Simulation Framework, such as:

- Introduce additional navigating agents and modify the simulation so that the agents race to the target.
- Recreate pacman, frogger, and some of your favorite games from the Arcade Learning Environment. Not all games can be recreated with these components, and some cannot be recreated at all with the GridWorld Simulation Framework (because they are not grid-based).
- Connect this simulation with the Reinforcement Learning capabilities of Abmarl via a *Simulation Manager*. Does the agent learng how to solve mazes quickly?
- And much, much more!

5.2.3 Communication Blocking

Consider a simulation in which some agents send messages to each other in an attempt to reach consensus while another group of agents attempts to block these messages to impede consensus. Abmarl’s GridWorld Simulation Framework already contains the features for the blocking agents; in this tutorial, we show how to create *new* components for the communication feature and connect them with the simulation framework. The tutorial can be found in full in our repo.

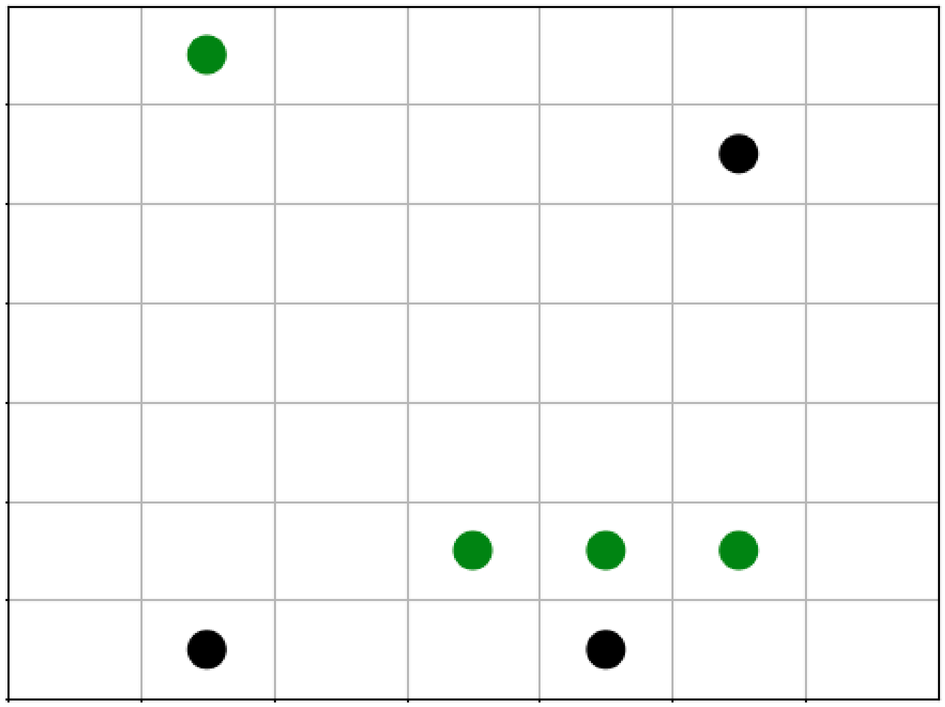


Fig. 4: Blockers (black) move around the maze blocking communications between broadcasters (green). The simulation ends when the broadcasters reach consensus.

Using built-in features

Let's start by laying the groundwork using components already in Abmarl. We create a simulation with *position*, *movement*, and *observations*.

```
from matplotlib import pyplot as plt
import numpy as np

from abmarl.sim.gridworld.agent import MovingAgent, GridObservingAgent
from abmarl.sim.gridworld.base import GridWorldSimulation
from abmarl.sim.gridworld.state import PositionState
from abmarl.sim.gridworld.actor import MoveActor
from abmarl.sim.gridworld.observer import SingleGridObserver

class BlockingAgent(MovingAgent, GridObservingAgent):
    def __init__(self, **kwargs):
        super().__init__(blocking=True, **kwargs)

class BroadcastSim(GridWorldSimulation):
    def __init__(self, **kwargs):
        self.agents = kwargs['agents']
        self.position_state = PositionState(**kwargs)
        self.move_actor = MoveActor(**kwargs)
        self.grid_observer = SingleGridObserver(**kwargs)

        self.finalize()
```

(continues on next page)

(continued from previous page)

```
    def reset(self, **kwargs):
        self.position_state.reset(**kwargs)
        self.rewards = {agent.id: 0 for agent in self.agents.values()}

    def step(self, action_dict, **kwargs):
        # process moves
        for agent_id, action in action_dict.items():
            agent = self.agents[agent_id]
            move_result = self.move_actor.process_action(agent, action, **kwargs)
            if not move_result:
                self.rewards[agent.id] -= 0.1

        # Entropy penalty
        for agent_id in action_dict:
            self.rewards[agent_id] -= 0.01

    def get_obs(self, agent_id, **kwargs):
        agent = self.agents[agent_id]
        return {
            **self.grid_observer.get_obs(agent, **kwargs),
        }

    def get_reward(self, agent_id, **kwargs):
        reward = self.rewards[agent_id]
        self.rewards[agent_id] = 0
        return reward

    def get_done(self, agent_id, **kwargs):
        pass # Define this later

    def get_all_done(self, **kwargs):
        pass # Define this later

    def get_info(self, **kwargs):
        return {}
```

Creating our own communication components

Next we build the communication components ourselves. We know that the GridWorld Simulation Framework is made up of *Agents*, *States*, *Actors*, *Observers*, and *Dones*, so we expect that we'll need to create each of these for our new communication feature. Let's start with the Agent component.

An agent communicates by broadcasting its message to other nearby agents. So we create a new agent with a *broadcast range* and an *initial message*. The *broadcast range* will be used by the BroadcastActor to determine successful broadcasting, and the *initial message*, an optional parameter, will be used by the BroadcastState to set its message.

```
from abmarl.sim import Agent
from abmarl.sim.gridworld.agent import GridWorldAgent

class BroadcastingAgent(Agent, GridWorldAgent):
    def __init__(self, broadcast_range=None, initial_message=None, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(**kwargs)
        self.broadcast_range = broadcast_range
        self.initial_message = initial_message

    @property
    def broadcast_range(self):
        return self._broadcast_range

    @broadcast_range.setter
    def broadcast_range(self, value):
        assert type(value) is int and value >= 0, "Broadcast Range must be a nonnegative␣
↪integer."
        self._broadcast_range = value

    @property
    def initial_message(self):
        return self._initial_message

    @initial_message.setter
    def initial_message(self, value):
        if value is not None:
            assert -1 <= value <= 1, "Initial message must be a number between -1 and 1."
        self._initial_message = value

    @property
    def message(self):
        return self._message

    @message.setter
    def message(self, value):
        self._message = min(max(value, -1), 1)

    @property
    def configured(self):
        return super().configured and self.broadcast_range is not None
```

Note: We could have split the BroadcastingAgent into two agents types: one type of agent that has an internal message and another type that broadcasts. This is usually a better approach because it allows you to separate features and use them in greater combination with other features. We put them together in this tutorial for simplicity.

Next, we create the BroadcastState. This component manages the part of the simulation state that tracks which messages have been sent among the agents. It will be used by the BroadcastObserver to create the agent's observations. It also manages updates to each agent's message.

```
from abmarl.sim.gridworld.state import StateBaseComponent

class BroadcastingState(StateBaseComponent):
    def reset(self, **kwargs):
        for agent in self.agents.values():
            if isinstance(agent, BroadcastingAgent):
                if agent.initial_message is not None:
```

(continues on next page)

(continued from previous page)

```
                    agent.message = agent.initial_message
                else:
                    agent.message = np.random.uniform(-1, 1)

        # Tracks agents receiving messages from other agents
        self.receiving_state = {
            agent.id: [] for agent in self.agents.values() if isinstance(agent,␣
↪BroadcastingAgent)
        }

    def update_receipients(self, from_agent, to_agents):
        """
        Update messages received from other agents.
        """
        for agent in to_agents:
            self.receiving_state[agent.id].append((from_agent.id, from_agent.message))

    def update_message_and_reset_receiving(self, agent):
        """
        Update agent's internal message.

        The agent averages all the messages that it has received from other
        agents in this step.
        """
        receiving_from = self.receiving_state[agent.id]
        self.receiving_state[agent.id] = []

        messages = [message for _, message in receiving_from]
        messages.append(agent.message)
        agent.message = np.average(messages)

        return receiving_from
```

Then we define the BroadcastActor. Similar to attacking, broadcasting will be a boolean action–either broadcast or don't broadcast. We provide a *broadcast mapping* for determining to which encodings each agent can broadcast. The message will be successfully sent to every agent that (1) is within the *broadcast range*, (2) has a compatible encoding, and (3) is not blocked.

```
from gym.spaces import Discrete
from abmarl.sim.gridworld.actor import ActorBaseComponent
import abmarl.sim.gridworld.utils as gu

class BroadcastingActor(ActorBaseComponent):
    """
    Process sending and receiving messages between agents.

    BroadcastingAgents can broadcast to compatible agents within their range
    according to the broadcast mapping and if the agent is not blocked.
    """
    def __init__(self, broadcast_mapping=None, **kwargs):
        super().__init__(**kwargs)
        self.broadcast_mapping = broadcast_mapping
```

(continues on next page)

(continued from previous page)

```
        for agent in self.agents.values():
            if isinstance(agent, self.supported_agent_type):
                agent.action_space[self.key] = Discrete(2)

    @property
    def key(self):
        return 'broadcast'

    @property
    def supported_agent_type(self):
        return BroadcastingAgent

    @property
    def broadcast_mapping(self):
        """
        Dict that dictates to which agents the broadcasting agent can broadcast.

        The dictionary maps the broadcasting agents' encodings to a list of encodings
        to which they can broadcast. For example, the folowing broadcast_mapping:
        {
            1: [3, 4, 5],
            3: [2, 3],
        }
        means that agents whose encoding is 1 can broadcast other agents whose encodings
        are 3, 4, or 5; and agents whose encoding is 3 can broadcast other agents whose
        encodings are 2 or 3.
        """
        return self._broadcast_mapping

    @broadcast_mapping.setter
    def broadcast_mapping(self, value):
        assert type(value) is dict, "Broadcast mapping must be dictionary."
        for k, v in value.items():
            assert type(k) is int, "All keys in broadcast mapping must be integer."
            assert type(v) is list, "All values in broadcast mapping must be list."
            for i in v:
                assert type(i) is int, \
                    "All elements in the broadcast mapping values must be integers."
        self._broadcast_mapping = value

    def process_action(self, broadcasting_agent, action_dict, **kwargs):
        """
        If the agent has chosen to broadcast, then we process their broadcast.

        The processing goes through a series of checks. The broadcast is successful
        if there is a receiving agent such that:
        1. The receiving agent is within range.
        2. The receiving agent is compatible according to the broadcast_mapping.
        3. The receiving agent is observable by the broadcasting agent.

        If the broadcast is successful, then the receiving agent receives the message
        in its observation.
```

(continues on next page)

(continued from previous page)

```
        """
        def determine_broadcast(agent):
            # Generate local grid and a broadcast mask.
            local_grid, mask = gu.create_grid_and_mask(
                agent, self.grid, agent.broadcast_range, self.agents
            )

            # Randomly scan the local grid for receiving agents.
            receiving_agents = []
            for r in range(2 * agent.broadcast_range + 1):
                for c in range(2 * agent.broadcast_range + 1):
                    if mask[r, c]: # We can see this cell
                        candidate_agents = local_grid[r, c]
                        if candidate_agents is not None:
                            for other in candidate_agents.values():
                                if other.id == agent.id: # Cannot broadcast to yourself
                                    continue
                                elif other.encoding not in self.broadcast_mapping[agent.
↪encoding]:
                                    # Cannot broadcast to this type of agent
                                    continue
                                else:
                                    receiving_agents.append(other)
            return receiving_agents

        if isinstance(broadcasting_agent, self.supported_agent_type):
            action = action_dict[self.key]
            if action: # Agent has chosen to attack
                return determine_broadcast(broadcasting_agent)
```

Now we define the BroadcastObserver. The observer enables agents to see all received messages, including their own current message. This observer is unique from all other components we have seen so far because it explicitly relies on the BroadcastingState component, which will have a small impact in how we initialize the simulation.

```
from gym.spaces import Dict, Box
from abmarl.sim.gridworld.observer import ObserverBaseComponent

class BroadcastObserver(ObserverBaseComponent):
    def __init__(self, broadcasting_state=None, **kwargs):
        super().__init__(**kwargs)

        assert isinstance(broadcasting_state, BroadcastingState), \
            "broadcasting_state must be an instance of BroadcastingState"
        self._broadcasting_state = broadcasting_state

        for agent in self.agents.values():
            if isinstance(agent, self.supported_agent_type):
                agent.observation_space[self.key] = Dict({
                    other.id: Box(-1, 1, (1,))
                    for other in self.agents.values() if isinstance(other, self.
↪supported_agent_type)
                })
```

(continues on next page)

(continued from previous page)

```
    @property
    def key(self):
        return 'message'

    @property
    def supported_agent_type(self):
        return BroadcastingAgent

    def get_obs(self, agent, **kwargs):
        if not isinstance(agent, self.supported_agent_type):
            return {}

        obs = {other: 0 for other in agent.observation_space[self.key]}
        receive_from = self._broadcasting_state.update_message_and_reset_receiving(agent)
        for agent_id, message in receive_from:
            obs[agent_id] = message
        obs[agent.id] = agent.message
        return obs
```

Finally, we can create a custom done condition. We want the broadcasting agents to finish when they've reached consensus; that is, when their internal message is within some tolerance of the average message.

```
from abmarl.sim.gridworld.done import DoneBaseComponent

class AverageMessageDone(DoneBaseComponent):
    def __init__(self, done_tolerance=None, **kwargs):
        super().__init__(**kwargs)
        self.done_tolerance = done_tolerance

    @property
    def done_tolerance(self):
        return self._done_tolerance

    @done_tolerance.setter
    def done_tolerance(self, value):
        assert type(value) in [int, float], "Done tolerance must be a number."
        assert value > 0, "Done tolerance must be positive."
        self._done_tolerance = value

    def get_done(self, agent, **kwargs):
        if isinstance(agent, BroadcastingAgent):
            average = np.average([
                other.message for other in self.agents.values()
                if isinstance(other, BroadcastingAgent)
            ])
            return np.abs(agent.message - average) <= self.done_tolerance
        else:
            return False

    def get_all_done(self, **kwargs):
        for agent in self.agents.values():
```

(continues on next page)

(continued from previous page)

```
            if isinstance(agent, BroadcastingAgent):
                if not self.get_done(agent):
                    return False
        return True
```

Building and running the simulation

Now that all the components have been created, we can create the full simulation:

```
from abmarl.sim.gridworld.base import GridWorldSimulation

class BroadcastSim(GridWorldSimulation):
    def __init__(self, **kwargs):
        self.agents = kwargs['agents']

        self.position_state = PositionState(**kwargs)
        self.broadcasting_state = BroadcastingState(**kwargs)

        self.move_actor = MoveActor(**kwargs)
        self.broadcast_actor = BroadcastingActor(**kwargs)

        self.grid_observer = SingleGridObserver(**kwargs)
        self.broadcast_observer = BroadcastObserver(broadcasting_state=self.broadcasting_
↪state, **kwargs)

        self.done = AverageMessageDone(**kwargs)

        self.finalize()

    def reset(self, **kwargs):
        self.position_state.reset(**kwargs)
        self.broadcasting_state.reset(**kwargs)

        self.rewards = {agent.id: 0 for agent in self.agents.values()}

    def step(self, action_dict, **kwargs):
        # process broadcasts
        for agent_id, action in action_dict.items():
            agent = self.agents[agent_id]
            receiving_agents = self.broadcast_actor.process_action(agent, action,␣
↪**kwargs)
            if receiving_agents is not None:
                self.broadcasting_state.update_receipients(agent, receiving_agents)

        # process moves
        for agent_id, action in action_dict.items():
            agent = self.agents[agent_id]
            move_result = self.move_actor.process_action(agent, action, **kwargs)
            if not move_result:
                self.rewards[agent.id] -= 0.1
```

(continues on next page)

(continued from previous page)

```
        # Entropy penalty
        for agent_id in action_dict:
            self.rewards[agent_id] -= 0.01

    def render(self, **kwargs):
        super().render(**kwargs)
        for agent in self.agents.values():
            if isinstance(agent, BroadcastingAgent):
                print(f"{agent.id}: {agent.message}")
        print()

    def get_obs(self, agent_id, **kwargs):
        agent = self.agents[agent_id]
        return {
            **self.grid_observer.get_obs(agent, **kwargs),
            **self.broadcast_observer.get_obs(agent, **kwargs)
        }

    def get_reward(self, agent_id, **kwargs):
        reward = self.rewards[agent_id]
        self.rewards[agent_id] = 0
        return reward

    def get_done(self, agent_id, **kwargs):
        return self.done.get_done(agent_id, **kwargs)

    def get_all_done(self, **kwargs):
        return self.done.get_all_done(**kwargs)

    def get_info(self, **kwargs):
        return {}
```

Let's initialize our simulation and run it. We initialize some BroadcastingAgents and some BlockingAgents. Then we initialize the simulation with a *broadcast mapping* that specifies that broadcasts can only be made amont agents with encoding 1, which are the BroadcastingAgents.

```
agents = {
    'broadcaster0': BroadcastingAgent(id='broadcaster0', encoding=1, broadcast_range=6,␣
↪render_color='green'),
    'broadcaster1': BroadcastingAgent(id='broadcaster1', encoding=1, broadcast_range=6,␣
↪render_color='green'),
    'broadcaster2': BroadcastingAgent(id='broadcaster2', encoding=1, broadcast_range=6,␣
↪render_color='green'),
    'broadcaster3': BroadcastingAgent(id='broadcaster3', encoding=1, broadcast_range=6,␣
↪render_color='green'),
    'blocker0': BlockingAgent(id='blocker0', encoding=2, move_range=2, view_range=3,␣
↪render_color='black'),
    'blocker1': BlockingAgent(id='blocker1', encoding=2, move_range=1, view_range=3,␣
↪render_color='black'),
    'blocker2': BlockingAgent(id='blocker2', encoding=2, move_range=1, view_range=3,␣
↪render_color='black'),
}
```

(continues on next page)

(continued from previous page)

```
sim = BroadcastSim.build_sim(
    7, 7,
    agents=agents,
    broadcast_mapping={1: [1]},
    done_tolerance=5e-10
)

sim.reset()
fig = plt.figure()
sim.render(fig=fig)

done_agents = set()
for i in range(50):
    action = {
        agent.id: agent.action_space.sample() for agent in agents.values() if agent.id␣
↪not in done_agents
    }
    sim.step(action)
    for agent in agents:
        if agent not in done_agents:
            obs = sim.get_obs(agent)
        if sim.get_done(agent):
            done_agents.add(agent)

    sim.render(fig=fig)
    if sim.get_all_done():
        break
```

The visualization produces an animation like the one at the top of this page. We can see the "path towards consensus" among the BroadcastingAgents in the output. Keep your eye open for the effects of blocking.

```
Step 1
broadcaster0: 0.5936447861764813
broadcaster1: -0.8344218389696239
broadcaster2: 0.09891331950679949
broadcaster3: 0.32590416873488093

Step 2
broadcaster0: 0.028375705313912796
broadcaster1: -0.25425883511737146
broadcaster2: -0.13653478357598114
broadcaster3: -0.25425883511737146

For steps 3-5, notice that Broadcaster3 is blocked. The other broadcasters
have reached a consensus, but the simulation does not end becaue they must all
agree.

Step 3
broadcaster0: -0.12080597112647994
broadcaster1: -0.12080597112647994
broadcaster2: -0.12080597112647995
broadcaster3: -0.15416918712420283
```

(continues on next page)

(continued from previous page)

```
Step 4
broadcaster0: -0.12080597112647994
broadcaster1: -0.12080597112647994
broadcaster2: -0.12080597112647995
broadcaster3: -0.15416918712420283

Step 5
broadcaster0: -0.12080597112647994
broadcaster1: -0.12080597112647994
broadcaster2: -0.12080597112647995
broadcaster3: -0.15416918712420283

Broadcaster3 is no longer blocked
Step 6
broadcaster0: -0.12080597112647995
broadcaster1: -0.12080597112647995
broadcaster2: -0.12080597112647995
broadcaster3: -0.1319270431257209

...

Step 16
broadcaster0: -0.1241744002450772
broadcaster1: -0.12417639653661512
broadcaster2: -0.12417523451616769
broadcaster3: -0.12417511533458334

Step 17
broadcaster0: -0.12417528665811084
broadcaster1: -0.12417528665811083
broadcaster2: -0.12417528665811083
broadcaster3: -0.12417528665811084
```

Extra Challenges

Having successfully created new components and fit them into the GridWorld Simulation Framework, we can create a vast variety of different simulations, constrained primarily by our own imagination. We leave the extra challenges up to you and what you can think of.

CHAPTER

SIX

ABMARL API SPECIFICATION

6.1 Abmarl Simulations

`class abmarl.sim.`**`PrincipleAgent`**`(`*id=None*, *seed=None*, ***kwargs*`)`

Principle Agent class for agents in a simulation.

`property active`

True if the agent is still active in the simulation.

Active means that the agent is in a valid state. For example, suppose agents in our Simulation can die. Then active is True if the agents are alive or False if they're dead.

`property configured`

All agents must have an id.

`finalize``(`***kwargs*`)`

`property id`

The agent's unique identifier.

`property seed`

Seed for random number generation.

`class abmarl.sim.`**`ObservingAgent`**`(`*observation_space=None*, ***kwargs*`)`

ObservingAgents can observe the state of the simulation.

The agent's observation must be *in* its observation space. The SimulationManager will send the observation to the Trainer, which will use it to produce actions.

`property configured`

Observing agents must have an observation space.

`finalize``(`***kwargs*`)`

Wrap all the observation spaces with a Dict and seed it if the agent was created with a seed.

`property observation_space`

`class abmarl.sim.`**`ActingAgent`**`(`*action_space=None*, ***kwargs*`)`

ActingAgents can act in the simulation.

The Trainer will produce actions for the agents and send them to the SimulationManager, which will process those actions in its step function.

`property action_space`

property configured

Acting agents must have an action space.

finalize(***kwargs*)

Wrap all the action spaces with a Dict if applicable and seed it if the agent was created with a seed.

class abmarl.sim.**Agent**(*observation_space=None*, ***kwargs*)

Bases: *ObservingAgent*, *ActingAgent*

An Agent that can both observe and act.

class abmarl.sim.**AgentBasedSimulation**

AgentBasedSimulation interface.

Under this design model the observations, rewards, and done conditions of the agents is treated as part of the simulations internal state instead of as output from reset and step. Thus, it is the simulations responsibility to manage rewards and dones as part of its state (e.g. via self.rewards dictionary).

This interface supports both single- and multi-agent simulations by treating the single-agent simulation as a special case of the multi-agent, where there is only a single agent in the agents dictionary.

property agents

A dict that maps the Agent's id to the Agent object. An Agent must be an instance of PrincipleAgent. A multi-agent simulation is expected to have multiple entries in the dictionary, whereas a single-agent simulation should only have a single entry in the dictionary.

finalize()

Finalize the initialization process. At this point, every agent should be configured with action and observation spaces, which we convert into Dict spaces for interfacing with the trainer.

abstract get_all_done(***kwargs*)

Return the simulation's done status.

abstract get_done(*agent_id*, ***kwargs*)

Return the agent's done status.

abstract get_info(*agent_id*, ***kwargs*)

Return the agent's info.

abstract get_obs(*agent_id*, ***kwargs*)

Return the agent's observation.

abstract get_reward(*agent_id*, ***kwargs*)

Return the agent's reward.

abstract render(***kwargs*)

Render the simulation for vizualization.

abstract reset(***kwargs*)

Reset the simulation simulation to a start state, which may be randomly generated.

abstract step(*action*, ***kwargs*)

Step the simulation forward one discrete time-step. The action is a dictionary that contains the action of each agent in this time-step.

6.2 Abmarl Simulation Managers

class `abmarl.managers.`**`SimulationManager`**(*sim*)

Control interaction between Trainer and AgentBasedSimulation.

A Manager implmenents the reset and step API, by which it calls the AgentBasedSimulation API, using the getters within reset and step to accomplish the desired control flow.

`sim`

The AgentBasedSimulation.

`agents`

The agents that are in the AgentBasedSimulation.

`render`(***kwargs*)

abstract **`reset`**(***kwargs*)

Reset the simulation.

Returns
The first observation of the agent(s).

abstract **`step`**(*action_dict*, ***kwargs*)

Step the simulation forward one discrete time-step.

Parameters
`action_dict` – Dictionary mapping agent(s) to their actions in this time step.

Returns

The observations, rewards, done status, and info for the agent(s) whose actions we expect to receive next.

Note: We do not necessarily return anything for the agent whose actions we just received in this time-step. This behavior is defined by each Manager.

class `abmarl.managers.`**`TurnBasedManager`**(*sim*)

The TurnBasedManager allows agents to take turns. The order of the agents is stored and the obs of the first agent is returned at reset. Each step returns the info of the next agent "in line". Agents who are done are removed from this line. Once all the agents are done, the manager returns all done.

`reset`(***kwargs*)

Reset the simulation and return the observation of the first agent.

`step`(*action_dict*, ***kwargs*)

Assert that the incoming action does not come from an agent who is recorded as done. Step the simulation forward and return the observation, reward, done, and info of the next agent. If that next agent finished in this turn, then include the obs for the following agent, and so on until an agent is found that is not done. If all agents are done in this turn, then the wrapper returns all done.

class `abmarl.managers.`**`AllStepManager`**(*sim*)

The AllStepManager gets the observations of all agents at reset. At step, it gets the observations of all the agents that are not done. Once all the agents are done, the manager returns all done.

`reset`(***kwargs*)

Reset the simulation and return the observation of all the agents.

`step`(*action_dict*, ***kwargs*)

Assert that the incoming action does not come from an agent who is recorded as done. Step the simulation forward and return the observation, reward, done, and info of all the non-done agents, including the agents that were done in this step. If all agents are done in this turn, then the manager returns all done.

6.3 Abmarl External Integration

`class abmarl.external.`**`GymWrapper`**(*sim*)

Wrap an AgentBasedSimulation object with only a single agent to the gym.Env interface. This wrapper exposes the single agent's observation and action space directly in the simulation.

`property action_space`

The agent's action space is the environment's action space.

`property observation_space`

The agent's observation space is the environment's observation space.

`render`(***kwargs*)

Forward render calls to the composed simulation.

`reset`(***kwargs*)

Return the observation from the single agent.

`step`(*action*, ***kwargs*)

Wrap the action by storing it in a dict that maps the agent's id to the action. Pass to sim.step. Return the observation, reward, done, and info from the single agent.

`property unwrapped`

Fall through all the wrappers and obtain the original, completely unwrapped simulation.

`class abmarl.external.`**`MultiAgentWrapper`**(*sim*)

Enable connection between SimulationManager and RLlib Trainer.

Wraps a SimulationManager and forwards all calls to the manager. This class is boilerplate and needed because RLlib checks that the simulation is an instance of MultiAgentEnv.

`sim`

The SimulationManager.

`render`(**args*, ***kwargs*)

See SimulationManager.

`reset`()

See SimulationManager.

`step`(*actions*)

See SimulationManager.

`property unwrapped`

Fall through all the wrappers to the SimulationManager.

Returns

The wrapped SimulationManager.

6.4 Abmarl GridWorld Simulation Framework

6.4.1 Base

`class abmarl.sim.gridworld.base.GridWorldSimulation`

GridWorldSimulation interface.

Extends the AgentBasedSimulation interface for the GridWorld. We provide builders for streamlining the building process.

`classmethod build_sim`(*rows*, *cols*, ***kwargs*)

Build a GridSimulation.

Specify the number of row, the number of cols, a dictionary of agents, and any additional parameters.

Parameters

- `rows` – The number of rows in the grid. Must be a positive integer.
- `cols` – The number of cols in the grid. Must be a positive integer.
- `agents` – The dictionary of agents in the grid.

Returns

A GridSimulation configured as specified.

`classmethod build_sim_from_file`(*file_name*, *object_registry*, ***kwargs*)

Build a GridSimulation from a text file.

Parameters

- `file_name` – Name of the file that specifies the initial grid setup. In the file, each cell should be a single alphanumeric character indicating which agent will be at that position (from the perspective of looking down on the grid). That agent will be given that initial position. 0's are reserved for empty space.
- `object_registry` – A dictionary that maps characters from the file to a function that generates the agent. This must be a function because each agent must have unique id, which is generated here.

Returns

A GridSimulation built from the file.

`render`(*fig=None*, ***kwargs*)

Draw the grid and all active agents in the grid.

Agents are drawn at their positions using their respective shape and color.

Parameters

`fig` – The figure on which to draw the grid. It's important to provide this figure because the same figure must be used when drawing each state of the simulation. Otherwise, a ton of figures will pop up, which is very annoying.

`class abmarl.sim.gridworld.base.GridWorldBaseComponent`(*agents=None*, *grid=None*, ***kwargs*)

Component base class from which all components will inherit.

Every component has access to the dictionary of agents and the grid.

`property agents`

A dict that maps the Agent's id to the Agent object. All agents must be GridWorldAgents.

property cols

The number of columns in the grid.

property grid

The grid indexes the agents by their position.

For example, an agent whose position is (3, 2) can be accessed through the grid with `self.grid[3, 2]`. Components are responsible for maintaining the connection between agent position and grid index.

property rows

The number of rows in the grid.

class abmarl.sim.gridworld.grid.Grid(*rows*, *cols*, *overlapping=None*, ***kwargs*)

A Grid stores the agents at indices in a numpy array.

Components can interface with the Grid. Each index in the grid is a dictionary that maps the agent id to the agent object itself. If agents can overlap, then there may be more than one agent per cell.

Parameters

- **rows** – The number of rows in the grid.
- **cols** – The number of columns in the grid.
- **overlapping** – Dictionary that maps the agents' encodings to a list of encodings with which they can occupy the same cell. To avoid undefined behavior, the overlapping should be symmetric, so that if 2 can overlap with 3, then 3 can also overlap with 2.

property cols

The number of columns in the grid.

place(*agent*, *ndx*)

Place an agent at an index.

If the cell is available, the agent will be placed at that index in the grid and the agent's position will be updated. The placement is successful if the new position is unoccupied or if the agent already occupying that position is overlappable AND this agent is overlappable.

Parameters

- **agent** – The agent to place.
- **ndx** – The new index for this agent.

Returns

The successfulness of the placement.

query(*agent*, *ndx*)

Query a cell in the grid to see if is available to this agent.

The cell is available for the agent if it is empty or if both the occupying agent and the querying agent are overlappable.

Parameters

- **agent** – The agent for which we are checking availabilty.
- **ndx** – The cell to query.

Returns

The availability of this cell.

`remove`(*agent*, *ndx*)

Remove an agent from an index.

Parameters

- **agent** – The agent to remove
- **ndx** – The old index for this agent

`reset`(***kwargs*)

Reset the grid to an empty state.

`property rows`

The number of rows in the grid.

6.4.2 Agents

`class abmarl.sim.gridworld.agent.GridWorldAgent`(*initial_position=None*, *blocking=False*, *encoding=None*, *render_shape='o'*, *render_color='gray'*, ***kwargs*)

The base agent in the GridWorld.

`property blocking`

Specify if this agent blocks other agent's observations and actions.

`property configured`

All agents must have an id.

`property encoding`

The numerical value that identifies the type of agent.

The value does not necessarily identify the agent itself. For example, other agents who observe this agent will see this value.

`property initial_position`

The agent's initial position at reset.

`property position`

The agent's position in the grid.

`property render_color`

The agent's color in the rendered grid.

`property render_shape`

The agent's shape in the rendered grid.

`class abmarl.sim.gridworld.agent.GridObservingAgent`(*view_range=None*, ***kwargs*)

Observe the grid up to view range cells away.

`property configured`

Observing agents must have an observation space.

`property view_range`

The number of cells away this agent can observe in each step.

`class abmarl.sim.gridworld.agent.MovingAgent`(*move_range=None*, ***kwargs*)

Move up to move_range cells.

property configured

Acting agents must have an action space.

property move_range

The maximum number of cells away that the agent can move.

class abmarl.sim.gridworld.agent.**HealthAgent**(*initial_health=None*, ***kwargs*)

Agents have health points and can die.

Health is bounded between 0 and 1.

property active

The agent is active if its health is greater than 0.

property health

The agent's health throughout the simulation trajectory.

The health will always be between 0 and 1.

property initial_health

The agent's initial health between 0 and 1.

class abmarl.sim.gridworld.agent.**AttackingAgent**(*attack_range=None*, *attack_strength=None*, *attack_accuracy=None*, ***kwargs*)

Agents that can attack other agents.

property attack_accuracy

The effective accuracy of the agent's attack.

Should be between 0 and 1. To make deterministic attacks, use 1.

property attack_range

The maximum range of the attack.

property attack_strength

The strength of the attack.

Should be between 0 and 1.

property configured

Acting agents must have an action space.

6.4.3 State

class abmarl.sim.gridworld.state.**StateBaseComponent**(*agents=None*, *grid=None*, ***kwargs*)

Abstract State Component base from which all state components will inherit.

abstract reset(***kwargs*)

Resets the part of the state for which it is responsible.

class abmarl.sim.gridworld.state.**PositionState**(*agents=None*, *grid=None*, ***kwargs*)

Manage the agents' positions in the grid.

reset(***kwargs*)

Give agents their starting positions.

We use the agent's initial position if it exists. Otherwise, we randomly place the agents in the grid.

class `abmarl.sim.gridworld.state.`**`HealthState`**(*agents=None*, *grid=None*, ***kwargs*)

Manage the state of the agents' healths.

Every HealthAgent has a health. If that health falls to zero, that agent dies and is remove from the grid.

`reset`(***kwargs*)

Give HealthAgents their starting healths.

We use the agent's initial health if it exists. Otherwise, we randomly assign a value between 0 and 1.

6.4.4 Actors

class `abmarl.sim.gridworld.actor.`**`ActorBaseComponent`**(*agents=None*, *grid=None*, ***kwargs*)

Abstract Actor Component class from which all Actor Components will inherit.

abstract property key

The key in the action dictionary.

The action space of all acting agents in the gridworld framework is a dict. We can build up complex action spaces with multiple components by assigning each component an entry in the action dictionary. Actions will be a dictionary even if your simulation only has one Actor.

abstract **`process_action`**(*agent*, *action_dict*, ***kwargs*)

Process the agent's action.

Parameters

- **agent** – The acting agent.
- **action_dict** – The action dictionary for this agent in this step. The dictionary may have different entries, each of which will be processed by different Actors.

abstract property supported_agent_type

The type of Agent that this Actor works with.

If an agent is this type, the Actor will add its entry to the agent's action space and will process actions for this agent.

class `abmarl.sim.gridworld.actor.`**`MoveActor`**(***kwargs*)

Agents can move to unoccupied nearby squares.

property key

This Actor's key is "move".

`process_action`(*agent*, *action_dict*, ***kwargs*)

The agent can move to nearby squares.

The agent's new position must be within the grid and the cell-occupation rules must be met.

Parameters

- **agent** – Move the agent if it is a MovingAgent.
- **action_dict** – The action dictionary for this agent in this step. If the agent is a MovingAgent, then the action dictionary will contain the "move" entry.

Returns

True if the move is successful, False otherwise.

property supported_agent_type

This Actor works with MovingAgents.

class `abmarl.sim.gridworld.actor.`**AttackActor**(*attack_mapping=None*, ***kwargs*)

Agents can attack other agents.

property attack_mapping

Dict that dictates which agents the attacking agent can attack.

The dictionary maps the attacking agents' encodings to a list of encodings that they can attack.

property key

This Actor's key is "attack".

process_action(*attacking_agent*, *action_dict*, ***kwargs*)

If the agent has chosen to attack, then we process their attack.

The processing goes through a series of checks. The attack is possible if there is an attacked agent such that:

1. The attacked agent is active.
2. The attacked agent is within range.
3. The attacked agent is valid according to the attack_mapping.

If the attack is possible, then we determine the success of the attack based on the attacking agent's accuracy. If the attack is successful, then the attacked agent's health is depleted by the attacking agent's strength, possibly resulting in its death.

property supported_agent_type

This Actor works with AttackingAgents.

6.4.5 Observers

class `abmarl.sim.gridworld.observer.`**ObserverBaseComponent**(*agents=None*, *grid=None*, ***kwargs*)

Abstract Observer Component base from which all observer components will inherit.

abstract get_obs(*agent*, ***kwargs*)

Observe the state of the simulation.

Parameters
`agent` – The agent for which we return an observation.

Returns
This agent's observation.

abstract property key

The key in the observation dictionary.

The observation space of all observing agents in the gridworld framework is a dict. We can build up complex observation spaces with multiple components by assigning each component an entry in the observation dictionary. Observations will be a dictionary even if your simulation only has one Observer.

abstract property supported_agent_type

The type of Agent that this Observer works with.

If an agent is this type, the Observer will add its entry to the agent's observation space and will produce observations for this agent.

`class abmarl.sim.gridworld.observer.SingleGridObserver`(*observe_self=True*, ***kwargs*)

Observe a subset of the grid centered on the agent's position.

The observation is centered around the observing agent's position. Each agent in the "observation window" is recorded in the relative cell using its encoding. If there are multiple agents on a single cell with different encodings, the agent will observe only one of them chosen at random.

`get_obs`(*agent*, ***kwargs*)

The agent observes a sub-grid centered on its position.

The observation may include other agents, empty spaces, out of bounds, and masked cells, which can be blocked from view by other blocking agents.

Returns
The observation as a dictionary.

`property key`

This Observer's key is "grid".

`property observe_self`

Agents can observe themselves, which may hide important information if overlapping is important. This can be turned off by setting observe_self to False.

`property supported_agent_type`

This Observer works with GridObservingAgents.

`class abmarl.sim.gridworld.observer.MultiGridObserver`(***kwargs*)

Observe a subset of the grid centered on the agent's position.

The observation is centered around the observing agent's position. The observing agent sees a stack of observations, one for each positive encoding, where the number of agents of each encoding is given rather than the encoding itself. Out of bounds and masked indicators appear in every grid.

`get_obs`(*agent*, ***kwargs*)

The agent observes one or more sub-grids centered on its position.

The observation may include other agents, empty spaces, out of bounds, and masked cells, which can be blocked from view by other blocking agents. Each grid records the number of agents on a particular cell correlated to a specific encoding.

Returns
The observation as a dictionary.

`property key`

This Observer's key is "grid".

`property supported_agent_type`

This Observer works with GridObservingAgents.

6.4.6 Done

class abmarl.sim.gridworld.done.**DoneBaseComponent**(*agents=None*, *grid=None*, ***kwargs*)

Abstract Done Component class from which all Done Components will inherit.

abstract get_all_done(***kwargs*)

Determine if all the agents are done and/or if the simulation is done.

Returns
True if all agents are done or if the simulation is done. Otherwise False.

abstract get_done(*agent*, ***kwargs*)

Determine if an agent is done in this step.

Parameters
agent – The agent we are querying.

Returns
True if the agent is done, otherwise False.

class abmarl.sim.gridworld.done.**ActiveDone**(*agents=None*, *grid=None*, ***kwargs*)

Inactive agents are indicated as done.

get_all_done(***kwargs*)

Return True if all agents are inactive. Otherwise, return False.

get_done(*agent*, ***kwargs*)

Return True if the agent is inactive. Otherwise, return False.

class abmarl.sim.gridworld.done.**OneTeamRemainingDone**(*agents=None*, *grid=None*, ***kwargs*)

Inactive agents are indicated as done.

If the only active agents are those who are all of the same encoding, then the simulation ends.

get_all_done(***kwargs*)

Return true if all active agents have the same encoding. Otherwise, return false.

6.4.7 Wrappers

class abmarl.sim.gridworld.wrapper.**ComponentWrapper**(*agents=None*, *grid=None*, ***kwargs*)

Wraps GridWorldBaseComponent.

Every wrapper must be able to wrap the respective space and points to/from that space. Agents and Grid are referenced directly from the wrapped component rather than received as initialization parameters.

property agents

The agent dictionary is directly taken from the wrapped component.

abstract check_space(*space*)

Verify that the space can be wrapped.

property grid

The grid is directly taken from the wrapped component.

property unwrapped

Fall through all the wrappers and obtain the original, completely unwrapped component.

abstract wrap_point(*space*, *point*)

Wrap a point to the space.

Parameters

- **space** – The space into which to wrap the point.
- **point** – The point to wrap.

abstract wrap_space(*space*)

Wrap the space.

Parameters

space – The space to wrap.

abstract property wrapped_component

Get the first-level wrapped component.

class abmarl.sim.gridworld.wrapper.**ActorWrapper**(*component*)

Wraps an ActorComponent.

Modify the action space of the agents involved with the Actor, namely the specific actor's channel. The actions recieved from the trainer are in the wrapped space, so we need to unwrap them to send them to the actor. This is the opposite from how we wrap and unwrap observations.

property key

The key is the same as the wrapped actor's key.

process_action(*agent*, *action_dict*, ***kwargs*)

Unwrap the action and pass it to the wrapped actor to process.

Parameters

- **agent** – The acting agent.
- **action_dict** – The action dictionary for this agent in this step. The action in this channel comes in the wrapped space.

property supported_agent_type

The supported agent type is the same as the wrapped actor's supported agent type.

property wrapped_component

Get the wrapped actor.

class abmarl.sim.gridworld.wrapper.**RavelActionWrapper**(*component*)

Use numpy's ravel capabilities to convert space and points to Discrete.

check_space(*space*)

Ensure that the space is of type that can be ravelled to discrete value.

wrap_point(*space*, *point*)

Unravel a single discrete point to a value in the space.

Recall that the action from the trainer arrives in the wrapped discrete space, so we need to unravel it so that it is in the unwrapped space before giving it to the actor.

wrap_space(*space*)

Convert the space into a Discrete space.

CHAPTER

SEVEN

CITATION

Abmarl has been published in the Journal of Open Source Software. It can be cited using the following bibtex entry:

```
@article{Rusu2021,
  doi = {10.21105/joss.03424},
  url = {https://doi.org/10.21105/joss.03424},
  year = {2021},
  publisher = {The Open Journal},
  volume = {6},
  number = {64},
  pages = {3424},
  author = {Edward Rusu and Ruben Glatt},
  title = {Abmarl: Connecting Agent-Based Simulations with Multi-Agent Reinforcement␣
↪Learning},
  journal = {Journal of Open Source Software}
}
```

INDEX