### Code Modifications

#### 1. Update

crowd\_sim.py

```python

# crowd\_sim.py

from crowd\_sim.envs.utils.info import \*

from crowd\_sim.envs.utils.utils import point\_to\_segment\_dist

class CrowdSim(gym.Env):

# ... existing code ...

def step(self, actions, update=True):

"""

Compute actions for all agents, detect collision, update environment, and return (ob, reward, done, info)

:param actions: List of actions for each robot [action1, action2]

"""

human\_actions = []

for human in self.humans:

# Observation for humans is always coordinates

ob = [other\_human.get\_observable\_state() for other\_human in self.humans if other\_human != human]

ob += [robot.get\_observable\_state() for robot in [self.robot1, self.robot2] if robot.visible]

human\_actions.append(human.act(ob))

# Collision detection

collision = False

hose\_violation = False

dmin = float('inf')

robots = [self.robot1, self.robot2]

for i, robot in enumerate(robots):

action = actions[i]

# Check collision with humans

for human in self.humans:

px = human.px - robot.px

py = human.py - robot.py

dist = np.hypot(px, py) - human.radius - robot.radius

if dist < 0:

collision = True

break

elif dist < self.discomfort\_dist:

# Discomfort penalty can be added here

pass

if dist < dmin:

dmin = dist

if collision:

break

# Hose length constraint

robot\_distance = np.hypot(self.robot1.px - self.robot2.px, self.robot1.py - self.robot2.py)

if robot\_distance > 0.6:

hose\_violation = True

# Compute rewards

if collision:

reward = self.collision\_penalty

done = True

info = Collision()

elif hose\_violation:

reward = -0.5 # Penalty for violating hose constraint

done = False

info = HoseViolation()

else:

# Reward for making progress towards the goal can be added here

reward = 0

done = False

info = Nothing()

# Update agents if not looking ahead

if update:

# Update robot positions

for i, robot in enumerate(robots):

robot.step(actions[i])

# Update human positions

for i, human in enumerate(self.humans):

human.step(human\_actions[i])

self.global\_time += self.time\_step

# Get new observations

ob = self.\_get\_observation()

return ob, reward, done, info

def \_get\_observation(self):

if self.robot1.sensor == 'coordinates':

ob1 = [human.get\_observable\_state() for human in self.humans]

ob2 = [human.get\_observable\_state() for human in self.humans]

ob = [ob1, ob2]

elif self.robot1.sensor == 'RGB':

raise NotImplementedError

return ob

def set\_robot1(self, robot):

self.robot1 = robot

self.robot1.set\_env(self)

def set\_robot2(self, robot):

self.robot2 = robot

self.robot2.set\_env(self)

```

#### 2. Update

info.py

```python

# info.py

class HoseViolation(object):

def \_\_init\_\_(self):

pass

def \_\_str\_\_(self):

return 'Hose length constraint violated'

```

#### 3. Update

robot.py

```python

# robot.py

class Robot(Agent):

def \_\_init\_\_(self, config, section, robot\_index):

super().\_\_init\_\_(config, section)

self.robot\_index = robot\_index

self.env = None

def set\_env(self, env):

self.env = env

def get\_other\_robot\_state(self):

if self.robot\_index == 0:

return self.env.robot2.get\_full\_state()

else:

return self.env.robot1.get\_full\_state()

def act(self, ob):

if self.policy is None:

raise AttributeError('Policy attribute has to be set!')

other\_robot\_state = self.get\_other\_robot\_state()

state = JointState(self.get\_full\_state(), other\_robot\_state, ob)

action = self.policy.predict(state)

return action

```

#### 4. Update

explorer.py

```python

# explorer.py

class Explorer(object):

def \_\_init\_\_(self, env, robot1, robot2, device, memory=None, gamma=None, target\_policy=None):

self.env = env

self.robot1 = robot1

self.robot2 = robot2

self.device = device

self.memory = memory

self.gamma = gamma

self.target\_policy = target\_policy

def run\_k\_episodes(self, k, phase, update\_memory=False, imitation\_learning=False, episode=None,

print\_failure=False):

self.robot1.policy.set\_phase(phase)

self.robot2.policy.set\_phase(phase)

hose\_violations = 0

# ... existing code ...

for i in range(k):

ob = self.env.reset(phase)

done = False

# ... existing code ...

while not done:

action1 = self.robot1.act(ob[0])

action2 = self.robot2.act(ob[1])

actions = [action1, action2]

ob, reward, done, info = self.env.step(actions)

# ... existing code ...

if isinstance(info, HoseViolation):

hose\_violations += 1

# You can handle additional logic here

# ... existing code ...

# After episodes, you can log hose violation statistics

logging.info('Hose violations: {}'.format(hose\_violations))

```

#### 5. Update

multi\_human\_rl.py

```python

# multi\_human\_rl.py

class MultiHumanRL(CADRL):

def predict(self, state):

"""

Modify the input state to include other robot's state

"""

if self.phase is None or self.device is None:

raise AttributeError('Phase, device attributes have to be set!')

# ... existing code ...

# Combine self state and other robot state

combined\_self\_state = state.self\_state + state.other\_robot\_state

# Transform state

state\_tensor = torch.cat([torch.Tensor([combined\_self\_state + human\_state]).to(self.device)

for human\_state in state.human\_states], dim=0)

# ... existing code ...

return max\_action

```

#### 6. Update

state.py

```python

# state.py

class JointState(object):

def \_\_init\_\_(self, self\_state, other\_robot\_state, human\_states):

self.self\_state = self\_state

self.other\_robot\_state = other\_robot\_state

self.human\_states = human\_states

```

#### 7. Update `train.py`

```python

# train.py

# Initialize robots

robot1 = Robot(env\_config, 'robot', robot\_index=0)

robot2 = Robot(env\_config, 'robot', robot\_index=1)

# Set policies

robot1.set\_policy(policy)

robot2.set\_policy(policy)

# Set robots in the environment

env.set\_robot1(robot1)

env.set\_robot2(robot2)

# Initialize explorer with both robots

explorer = Explorer(env, robot1, robot2, device, memory=memory, gamma=0.9)

```

#### 8. Update `sarl.py`

Modify the

ValueNetwork

to accept the combined self state.

```python

# sarl.py

class ValueNetwork(nn.Module):

def forward(self, state):

"""

Modify to handle combined self state (self + other robot)

"""

size = state.shape

combined\_self\_state = state[:, 0, :self.self\_state\_dim]

# ... existing code ...

```

---

Please make sure to test each component thoroughly to ensure that the hose constraint is correctly enforced and that the robots behave as expected in the simulation.