CDS524 Assignment 1 - Reinforcement Learning Game Design

LUO Wenxuan SID:5581504

1. Game Design

1.1 Objective and Rules

In this assignment, I've developed a game where a predator pursues a prey through a maze.

Game Objective:

Predator: The predator's goal is to identify the optimal or shortest path to capture the prey within 70 steps.

```
# Check if the target turtle is reached
if ((prey_state[0]-predator_state[0])**2 + (prey_state[1]-predator_state[1])**2) <= 5000:

    done = True

    if randomm:
        prey_state = random.choice(goal_states)
    if show:
        score += 1
        target_turtle.hideturtle()
        target_turtle.goto(prey_state)
        target_turtle.showturtle()
```

Figure 1: Predator's winning rule

Prey: The prey aims to evade capture by the predator within 70 steps in each single game.

```
# Check if the target turtle escaped
if step == 70:
    done = True

    if randomm:
        prey_state = random.choice(goal_states)
    if show:
        prey_score +=1
        target_turtle.hideturtle()
        target_turtle.goto(prey_state)
        target_turtle.showturtle()
```

Figure 2: Prey's winning rule

Rules:

1. Movement Constraints: Both the predator and the prey must stay within the maze boundaries and avoid colliding with obstacles.
2. Predator's Score: If the predator manages to catch the prey within 70 steps, it earns 1 point.
3. Prey's Score: The prey gets 1 point if it avoids being caught within the 70 - step limit.
4. Round Structure: Each round consists of 70 steps. Once the predator catches the prey or the prey manages to escape, the next round starts. At the start of each new round, the prey will randomly appear in another permitted positions within the maze.

1.2 States and Action Space

This turtle-based game operates within a 650×650-pixel square bounded by black walls, visually constructed by the target_turtle through its path-drawing sequence (from (-325, -325) to corner coordinates). The playable state space is defined as a discrete grid ranging from -300 to 300 on both the x and y axis, with 50-pixel intervals (implemented via nested loops in the states list). This creates 169 potential grid positions (13×13 rows/columns). Crucially, the state space explicitly excludes red square blocks rendered by obstacle_turtle instances at predefined locations in the obstacles list. These obstacles, scaled to 50×50 pixels using shapesize(2.5), act as impassable barriers, narrowing valid states to 169 minus 30 hardcoded obstacle positions.

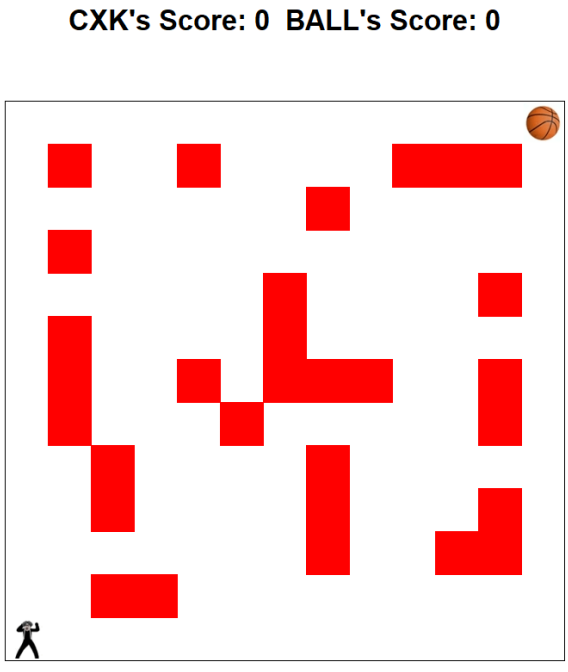

Figure 3: Game environment

The action space is limited to four directional movements: up, down, left, and right, each corresponding to a 50-pixel movement aligned with the grid. For instance, executing 'up' moves the agent_turtle (initially positioned at (-300, -300)) to (x, y+50), provided the new state remains within bounds and avoids obstacles.

```
# actions = ['up', 'down', 'left', 'right']
if action == 'up':
    next_state = (goal_x, goal_y + 50)
elif action == 'down':
    next_state = (goal_x, goal_y - 50)
elif action == 'left':
    next_state = (goal_x - 50, goal_y)
else:  # 'right'
    next_state = (goal_x + 50, goal_y)
```

Figure 4: Action space

1.3 Reward Function

Prey Reward Mechanism:

- The prey (controlled by agent_turtle) is incentivized to evade the predator and avoid hazards:
- Collision penalties: Receives -5 for hitting walls (next_state not in states) or obstacles (next_state in obstacles), with its position reset to prey_state to simulate blocked movement.
- Capture penalty: If the predator closes the prey 50 pixels (for horizontally or vertically adjacent positions) or 70.71 pixels (for diagonally adjacent positions), the prey is penalized with -10, marking immediate capture.
- Distance-based reward: When safe, the prey earns a reward proportional to its escape progress: (next_distance - current_distance)/10. Positive rewards encourage moving away from the predator, while negative values penalize moving closer.

```
# Prey reward
if next_state not in states:
    next_state = prey_state
    reward = -5  # Penalty for hitting the wall
elif ((next_state[0]-predator_state[0])**2 + (next_state[1]-predator_state[1])**2) <= 5000:
    reward = -10  # Reached the target turtle
elif next_state in obstacles:
    next_state = prey_state
    reward = -5  # Hit the obstacle turtle
else:
    current_distance= ((prey_state[0]-predator_state[0])**2 + (prey_state[1]-predator_state[1])**2)**(1/2)
    next_distance = ((next_state[0]-predator_state[0])**2 + (next_state[1]-predator_state[1])**2)**(1/2)
    reward = (next_distance - current_distance)/10
```

Figure 5: Prey reward

Predator Reward Mechanism:

- The predator (presumably target_turtle) is designed to pursue the prey:
- Collision penalties: Similarly penalized with -5 for wall/obstacle collisions, retaining its current position (predator_state).
- Capture success: Earns a high reward of +10 upon reaching the prey's coordinates (next_state == prey_state).
- Time pressure: Receives a constant -1 penalty per step to incentivize fast interception, discouraging idle behavior.

```
# Predator reward
if next_state not in states:
    next_state = predator_state
    reward = -5 # Penalty for hitting the wall
elif next_state == prey_state:
    reward = 10 # Reached the target turtle
elif next_state in obstacles:
    next_state = predator_state
    reward = -5  # Hit the obstacle turtle
else:
    reward = -1  # Time pressure penalty
```

Figure 6: Predator reward

2. Q-Learning Implementation

2.1 Parameter Definitions

```
93    # Define the parameters for Q-Learning
94    alpha = 0.1  # Learning rate
95    gamma = 0.9  # Discount rate
96    epsilon = 0.1  # Exploration rate
```

Figure 7: Q-Learning parameters

Alpha (Learning Rate): Controls how quickly the Q-values are updated with new information. A value of 0.1 means only 10% of the new estimated reward is incorporated into the current Q-value. A smaller alpha leads to slower but more stable learning, while a larger alpha prioritizes recent experiences but risks instability.

Gamma (Discount Rate): Determines the importance of future rewards. A value of 0.9 means future rewards are weighted heavily, encouraging the agent to prioritize long-term gains over immediate rewards. If gamma = 0, the agent would only care about immediate rewards.

Epsilon (Exploration Rate): Balances exploration vs. exploitation. With epsilon = 0.1, the agent has a 10% chance to choose a random action (explore) and a 90% chance to choose the best-known action (exploit). This ensures the agent occasionally tries new strategies to avoid local optima.

2.2 Epsilon-Greedy Policy

The prey has a 10% chance to explore (random action) and a 90% chance to exploit (choose the action with the highest Q-value for the current state). These balances learning new strategies while utilizing known good actions.

```
# Choose an action using epsilon-greedy policy
if np.random.uniform() < epsilon:
    action = np.random.choice(actions)  # Explore
else:
    action = max(predator_table[condition], key=predator_table[condition].get)  # Exploit
if np.random.uniform() < epsilon:
    action = np.random.choice(actions)  # Explore
else:
    action = max(prey_table[condition], key=prey_table[condition].get)  # Exploit
```

Figure 8: Epsilon-greedy policy

2.3 Q-Table Update for Prey and Predator

In the context of ***1.3 Reward Function***, we get the reward/penalty for every movement of Prey and Predator. Thus, we can update our Q-table file.

Update for Prey:

Alpha's Role: Scales how much the new TD target (reward + gamma * max_future_Q) updates the current Q-value.

Gamma's Role: Weights the future discounted reward (max_future_Q). A high gamma = 0.9 means the prey prioritizes future safety rather than the current chosen action chosen by Epsilon-Greedy Policy.

```
# Update the Q-table
prey_table[condition][action] += alpha * (
        reward + gamma * max(prey_table[next_condition].values()) -
        prey_table[condition][action])
```

Figure 9: Update the Q-table for Prey

Update for Predator:

Alpha and Gamma's Roles: Same as prey: alpha controls update magnitude, gamma emphasizes long-term rewards (e.g., planning a path to corner the prey).

```
# Update the Q-table
predator_table[condition][action] += alpha * (
        reward + gamma * max(predator_table[next_condition].values()) -
        predator_table[condition][action])
```

Figure 10: Update the Q-table for Predator

2.4 Training Loop

The first for loop is to go through every possible state of predator and the second for loop is to go through every possible state of prey. Thus, we can achieve every possible position combination of predator and prey.

```
for s in goal_states:  # s is to control the predator positions
    print(f"Training for CXK state {s}.")
    with open('predator.yaml', 'w') as f:
        yaml.dump(predator_table, f)
    with open('prey.yaml', 'w') as f:
        yaml.dump(prey_table, f)

    for g_state in goal_states:
        min_duration = 0  # This variety is to save one single time that Tom catch Jerry.
        min_duration_time = 0  # This variety is to save the times that Tom catch Jerry within a time limit.

        while min_duration_time != 500:...
```

Figure 11: Whole training process

```
duration = time.time() - start
if duration > min_duration:
    min_duration = duration
    min_duration_time = 0
else:
    min_duration_time += 1
```

Figure 12: First while control

This if judgement means that if the prey finds a better route to escape, then the predator needs to catch the prey for at least 500 times within the new min_duration again(min_duration_time will reset to 0 and count again). Importantly, in the while loop we update the Q-table.

3. Result Evaluation

3.1 Result Storage

```
In these new constructed dictionary, keys will be the position values of Tom and Jerry. And values will be the action
probabilities. After you have finished training, you can note these codes below.
"""
predator_table = {}
prey_table = {}
```

Figure 11: Create two new dictionaries

Before training, we don't have any data about which action we should take at every playable states. So we have to create two empty dictionaries to save action data of predator and prey separately.

```
"""Train Prey"""
x, y = predator_state
goal_x, goal_y = prey_state
condition = (x, y, goal_x, goal_y)
if condition not in prey_table.keys():
    prey_table[condition] = {action: 0 for action in actions}
"""Train Predator"""
x, y = predator_state
goal_x, goal_y = prey_state
condition = (x, y, goal_x, goal_y)
if condition not in predator_table.keys():
    predator_table[condition] = {action: 0 for action in actions}
```

Figure 12: Create new element in dictionaries

During training, if we encounter condition that doesn't exist in the dictionary, we will create a new element for this condition. If such condition exists, we will update its data by using functions in ***2.3 Q-Table Update for Prey and Predator***.

```
with open('predator.yaml', 'w') as f:
    yaml.dump(predator_table, f)
with open('prey.yaml', 'w') as f:
    yaml.dump(prey_table, f)
```

Figure 13: Saving data in yaml files

After the whole training process, we will get two completed dictionaries. Finally, we dump these data into two yaml files. Each yaml file contains every single condition (possible state combination of predator's position and prey's position) as follow:

```
? !!python/tuple
- -300
- -300
- -300
- -250
: down: 13.12903179454518
  left: 11.769751649580009
  right: 12.671701635177431
  up: 27.099276554160333
```

Figure 14: A condition in yaml file

The top 2 numbers in the Figure14 represent the x and y positions of predator, and the next two numbers represent those of prey. And every action has a real number used for choosing the optimal action in this condition.

The training result will be presented in my presentation video.

4. Challenges and Deficiencies

One of the most significant challenges encountered in this project stemmed from computational resource constraints, which severely impacted training efficiency. It takes me almost 3 hours to finished the first training but the agent still performed not well. The Q-learning algorithm's training time became very long due to two interrelated factors:

1. Exponential State-Space Complexity:

The environment's state space consists of 139 × 139 possible state combinations (e.g., grid positions or agent configurations). To ensure robust policy convergence, each state-action pair requires at least 500 training iterations for adequate exploration and exploitation. This results in a total of at least ~9.6 million training steps (139 × 139 × 500), creating an overwhelming computational burden.

2. Hardware Limitations:

Training was conducted on limited CPU capabilities and memory bandwidth. The sheer scale of the Q-table further strained system resources, causing: Slow Q-table convergence and Extended runtime per iteration.

After the first training, I turn off the display setting (score board update and movement display) and random setting (predator and prey appear in random position) and increase the speed of predator and prey. After such tunning, I finally increase my training speed.

The comparison will also be shown in my presentation video.