lab7-ai

October 15, 2024

[2]:
```python
import random

class Card:
    """Represents a card with value and suit."""
    SUIT_PRIORITY = {'Spades': 4, 'Hearts': 3, 'Diamonds': 2, 'Clubs': 1}

    def __init__(self, value, suit):
        self.value = value
        self.suit = suit

    def __repr__(self):
        return f"{self.value} of {self.suit}"

    def compare(self, other):
        """Compare two cards based on value and suit priority."""
        if self.value > other.value:
            return True
        elif self.value == other.value:
            return Card.SUIT_PRIORITY[self.suit] > Card.SUIT_PRIORITY[other.
 ↪suit]
        return False


class Player:
    """Represents a player in the game."""
    def __init__(self, id):
        self.id = id
        self.card = None

    def assign_card(self, card):
        self.card = card

    def __repr__(self):
        return f"Player {self.id} with card {self.card}"


class CasinoAgent:
```

```python
    """The AI Agent that manages the casino game."""
    def __init__(self, num_players):
        self.players = [Player(i + 1) for i in range(num_players)]
        self.cards = self.generate_cards(num_players)
        self.available_players = set(range(1, num_players + 1))
        self.available_cards = set(range(1, num_players + 1))

    def generate_cards(self, num_cards):
        """Generate a deck with specified number of cards."""
        suits = ['Spades', 'Hearts', 'Diamonds', 'Clubs']
        cards = []
        for i in range(1, num_cards + 1):
            suit = suits[(i - 1) % 4]  # Cycle through suits
            cards.append(Card(i, suit))
        return cards

    def roll_dice(self, n):
        """Simulate a dice roll with n faces."""
        return random.randint(1, n)

    def assign_cards(self):
        """Assign cards to players based on dice rolls."""
        while self.available_players and self.available_cards:
            player_roll = self.roll_dice(len(self.players))
            card_roll = self.roll_dice(len(self.cards))

            if player_roll in self.available_players and card_roll in self.
↪available_cards:
                player = self.players[player_roll - 1]
                card = self.cards[card_roll - 1]

                player.assign_card(card)
                self.available_players.remove(player_roll)
                self.available_cards.remove(card_roll)
                print(f"Assigned {card} to Player {player.id}")

    def announce_winner(self):
        """Announce the winner based on the highest card value."""
        winner = None
        for player in self.players:
            if player.card:
                if winner is None or player.card.compare(winner.card):
                    winner = player
        print(f"The winner is: {winner}")

    def play_game(self):
        """Run the entire game process."""
```

```
        self.assign_cards()
        self.announce_winner()


# Main function to run the game
if __name__ == "__main__":
    num_players = int(input("Enter the number of players: "))
    agent = CasinoAgent(num_players)
    agent.play_game()
```

```
Enter the number of players:  7

Assigned 5 of Spades to Player 6
Assigned 4 of Clubs to Player 3
Assigned 1 of Spades to Player 1
Assigned 6 of Hearts to Player 7
Assigned 2 of Hearts to Player 5
Assigned 3 of Diamonds to Player 4
Assigned 7 of Diamonds to Player 2
The winner is: Player 2 with card 7 of Diamonds
```

[3]:

```
import random

# Base class for all agents
class Agent:
    """A base agent with a sense and act mechanism."""
    def sense(self, environment):
        raise NotImplementedError

    def act(self):
        raise NotImplementedError


# 1. Goal-Based Agent: Tries to achieve a specific goal (e.g., reaching a␣
↪target position in a grid)
class GoalBasedAgent(Agent):
    """Agent that tries to achieve a goal in a grid environment."""
    def __init__(self, start, goal):
        self.position = start
        self.goal = goal

    def sense(self, environment):
        """Sense the environment to find available moves."""
        return environment.get_possible_moves(self.position)

    def act(self, environment):
        """Move towards the goal if not already there."""
        while self.position != self.goal:
```

```python
            moves = self.sense(environment)
            self.position = random.choice(moves)  # Randomly pick one valid move
            print(f"Moved to {self.position}")
        print("Reached the goal!")


# 2. Model-Based Agent: Uses an internal model to predict future states and act␣
 ↪accordingly.
class ModelBasedAgent(Agent):
    """Agent that navigates using an internal model."""
    def __init__(self):
        self.internal_model = {}  # Store visited states

    def sense(self, environment):
        """Sense the current environment and update the internal model."""
        state = environment.get_current_state()
        self.internal_model[state] = True  # Mark state as visited
        print(f"Sensed state: {state}")

    def act(self, environment):
        """Act based on the internal model."""
        for _ in range(5):  # Perform 5 actions
            self.sense(environment)
            action = random.choice(["Move Left", "Move Right", "Stay"])
            print(f"Performed action: {action}")


# 3. Utility-Based Agent: Chooses actions based on the utility of outcomes.
class UtilityBasedAgent(Agent):
    """Agent that picks the action with the highest utility."""
    def __init__(self, actions):
        self.actions = actions

    def sense(self, environment):
        """Get possible actions and their utilities."""
        utilities = {action: random.randint(1, 10) for action in self.actions}
        print(f"Action utilities: {utilities}")
        return utilities

    def act(self, environment):
        """Choose the action with the highest utility."""
        utilities = self.sense(environment)
        best_action = max(utilities, key=utilities.get)
        print(f"Chose action: {best_action}")


# Environment for agents to interact with
```

```python
class Environment:
    def __init__(self):
        self.grid = [(0, 0), (0, 1), (1, 0), (1, 1)]  # Example grid

    def get_possible_moves(self, position):
        """Return all valid moves from the current position."""
        x, y = position
        moves = [(x + dx, y + dy) for dx, dy in [(-1, 0), (1, 0), (0, -1), (0,␣
↪1)]]
        return [move for move in moves if move in self.grid]

    def get_current_state(self):
        """Simulate returning a current state of the environment."""
        return random.choice(["Idle", "Busy", "Error"])


# Main program to demonstrate the three agents
if __name__ == "__main__":
    # Goal-Based Agent
    print("\n--- Goal-Based Agent ---")
    env = Environment()
    goal_agent = GoalBasedAgent(start=(0, 0), goal=(1, 1))
    goal_agent.act(env)

    # Model-Based Agent
    print("\n--- Model-Based Agent ---")
    model_agent = ModelBasedAgent()
    model_agent.act(env)

    # Utility-Based Agent
    print("\n--- Utility-Based Agent ---")
    actions = ["Move Left", "Move Right", "Collect Item", "Stay"]
    utility_agent = UtilityBasedAgent(actions)
    utility_agent.act(env)
```

```
--- Goal-Based Agent ---
Moved to (1, 0)
Moved to (1, 1)
Reached the goal!

--- Model-Based Agent ---
Sensed state: Busy
Performed action: Stay
Sensed state: Busy
Performed action: Move Right
Sensed state: Error
```

```
Performed action: Stay
Sensed state: Idle
Performed action: Move Left
Sensed state: Error
Performed action: Stay

--- Utility-Based Agent ---
Action utilities: {'Move Left': 10, 'Move Right': 3, 'Collect Item': 8, 'Stay':
8}
Chose action: Move Left
```

[ ]: