Artificial Intelligence

Lab: 7

15/10/2024

Task 1:

Write a Python code using object-oriented classes to make a Simple Reflex Agent, as we have been doing in class. Below are the tasks that the agent must perform.

You manage a casino where a probabilistic game of wages is played. There are 'n' players and 'n' cards involved in this game. A card is mapped to a player based on the outcome of a roll of dicewith-'n'-faces. The problem is, you want to save as much money as you want therefore you want to fire your employee who hosts this game and rolls the dice. To save money, make an AI agent to replace the casino employee who performs the tasks below to host the game.

1. Identify number of the contestants
2. Add a same number of cards to the game
3. Perform the roll of two dice; one for players and the other for cards.
4. As per the value of the rolls, assign the corresponding card to the corresponding player.
5. Once a card is assigned to a players, both are invalid if the rolls of dice calls them again.
6. Announce the winner – the player with the highest card value, as per the legend below.

   a. Cards Legend:

i. The bigger the number, the higher the priority

ii. Spades > Hearts > Diamonds > Clubs

CODE:


```
{
 "cells": [
  {
   "metadata": {},
   "source": [
    "## Task 1"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 28,
   "id": "683a066c",
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "Starting the game...\n",
      "Player 1 is assigned 2 of Clubs.\n",
      "Player 3 is assigned 13 of Diamonds.\n",
      "Player 2 is assigned 12 of Hearts.\n",
      "Player 4 is assigned 10 of Clubs.\n",
```

```
    "The winner is Player 3 with the card 13 of Diamonds.\n"
   ]
  }
 ],
 "source": [
  "import random\n",
  "\n",
  "class Card:\n",
  "    def __init__(self, suit, value):\n",
  "        self.suit = suit\n",
  "        self.value = value\n",
  "\n",
  "    def __repr__(self):\n",
  "        return f\"{self.value} of {self.suit}\"  # Return a string representation of the card\n",
  "\n",
  "class Player:\n",
  "    def __init__(self, name):\n",
  "        self.name = name  \n",
  "        self.card = None  # Initially, the player has no card\n",
"\n",
  "    def assign_card(self, card):\n",
  "        self.card = card  # Assign a card to the player\n",
  "\n",
  "class Casino:\n",
  "    def __init__(self, num_players):\n",
```

```
"       self.players = [Player(f\"Player {i + 1}\") for i in range(num_players)]  # Create players\n",
"       self.cards = self.create_cards()  # Create cards\n",
"       self.assigned = set()  # To keep track of assigned players and cards\n",
"\n",
"    def create_cards(self):\n",
"       suits = ['Spades', 'Hearts', 'Diamonds', 'Clubs']\n",
"       values = list(range(1, 14))  # Values from 1 to 13\n",
"       return [Card(suit, value) for suit in suits for value in values]  \n",
"\n",
"    def roll_dice(self):\n",
"       return random.randint(1, len(self.players)), random.randint(1, len(self.cards))  # Roll two dice\n",
"\n",
"    def assign_cards(self):\n",
"       while len(self.assigned) < len(self.players) * 2:  \n",
"          player_roll, card_roll = self.roll_dice()  # Roll dice\n",
"          player_index = player_roll - 1  # Adjust for 0-indexing\n",
"          card_index = card_roll - 1  # Adjust for 0-indexing\n",
"\n",
"          if player_index not in self.assigned and card_index not in self.assigned:\n",
"             self.players[player_index].assign_card(self.cards[card_index])  # Assign card to player\n",
"             self.assigned.add(player_index)  # Mark player as assigned\n",
"             self.assigned.add(card_index)  # Mark card as assigned\n",
"             print(f\"{self.players[player_index].name} is assigned {self.cards[card_index]}.\")\n",
```

```
    "\n",
    "    def determine_winner(self):\n",
    "        # Check if all players have assigned cards before determining the winner\n",
    "        if all(player.card is not None for player in self.players):\n",
    "            # Determine the winner based on the assigned cards\n",
    "            winner = max(self.players, key=lambda player: (player.card.value, self.card_priority(player.card.suit)))\n",
    "            print(f\"The winner is {winner.name} with the card {winner.card}.\")\n",
    "        else:\n",
    "            print(\"Not all players have been assigned cards. Cannot determine a winner.\")\n",
    "\n",
    "    @staticmethod\n",
    "    def card_priority(suit):\n",
    "        # Define priority for the suits\n",
    "        priorities = {'Spades': 4, 'Hearts': 3, 'Diamonds': 2, 'Clubs': 1}\n",
    "        return priorities[suit]  # Return the priority of the suit\n",
"\n",
    "    def host_game(self):\n",
    "        print(\"Starting the game...\")\n",
    "        self.assign_cards()  # Assign cards to players\n",
    "        self.determine_winner()  # Determine the winner\n",
    "\n",
    "# Example usage\n",
    "num_players = 4  # Number of players\n",
```

"casino_game = Casino(num_players) # Create an instance of the Casino\n",

"casino_game.host_game() # Host the game\n"

]

},

Task 2:

Write a program that includes the three different types of agents and their implementation in different scenarios.

- The program includes the implementation of goal-based agents.
- The program includes the implementation of the model-based agent.
- The program includes the implementation of a utility-based agent.

CODE:

{

"metadata": {},

"source": [

"\n",

"\n",

"## Task 2"

]

},

{

"cell_type": "code",

"execution_count": 36,

"id": "f06595c9",

"metadata": {},

```
  "outputs": [
   {
    "name": "stdout",
    "output_type": "stream",
    "text": [
     "\n",
     "\t MODEL BASED AGENT \n",
     "\n",
     "Scenario 1 - Delivery Robot \n",
     "\n",
     "Initial Warehouse states (1 indicates pick package from location, 0 indicates no package):\n",
     "[1, 1, 0, 0, 0, 0, 1, 1, 0, 1]\n",
     "Current Warehouse locations: [0, 1, 6, 7, 9]\n",
     "Moved from location 0 to 0\n",
     "Package picked from location: 0\n",
     "Current Warehouse locations: [1, 6, 7, 9]\n",
     "Moved from location 0 to 1\n",
     "Package picked from location: 1\n",
     "Current Warehouse locations: [6, 7, 9]\n",
     "Moved from location 1 to 6\n",
     "Package picked from location: 6\n",
     "Current Warehouse locations: [7, 9]\n",
     "Moved from location 6 to 7\n",
     "Package picked from location: 7\n",
     "Current Warehouse locations: [9]\n",
```

```
    "Moved from location 7 to 9\n",
    "Package picked from location: 9\n",
    "Current Warehouse locations: []\n",
    "All Packages are picked from locations!\n",
    "\n",
    "Final Warehouse States:\n",
    "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]\n",
    "History of moves: [0, 1, 6, 7, 9]\n",
    "\n",
"\n",     "\n",
    "\n",
    "\n",
    "\t GOAL BASED AGENT \n",
    "\n",
    "Scenario 2 - Diswashing Robot \n",
    "\n",
    "Current Dish Rack State: [0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1]\n",
    "Starting dishwashing process...\n",
    "Dish at position 0 is already clean, skipping.\n",
    "Dish washed at position 1.\n",
"Dish washed at position 2.\n",     "Dish
washed at position 3.\n",
    "Dish at position 4 is already clean, skipping.\n",
"Dish at position 5 is already clean, skipping.\n",
    "Dish washed at position 6.\n",
    "Dish at position 7 is already clean, skipping.\n",
```

"Dish at position 8 is already clean, skipping.\n",
"Dish at position 9 is already clean, skipping.\n",
"Dish washed at position 10.\n",
"Dishwashing process completed.\n",
"Current Dish Rack State: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]\n",
"\n",
"\n", "\n",
"\n",
"\n",
"\t UILITY BASED AGENT \n",
"\n",
"Scenario 3 - Garden Maintenance Robot \n",
"\n",
"Starting watering process...\n",
"Watered plant at position 3.\n",
"Updated garden state: [0, 1, 1, 0, 0, 0, 1, 1]\n",
"Watered plant at position 6.\n",
"Updated garden state: [0, 1, 1, 0, 0, 0, 0, 1]\n",
"Watered plant at position 7.\n",
"Updated garden state: [0, 1, 1, 0, 0, 0, 0, 0]\n",
"Watered plant at position 2.\n",
"Updated garden state: [0, 1, 0, 0, 0, 0, 0, 0]\n",
"Watered plant at position 1.\n",
"Updated garden state: [0, 0, 0, 0, 0, 0, 0, 0]\n",
"No dry plants to water.\n",
"Watering process completed.\n"

```
    ]
   }
  ],
  "source": [
   "# Scenario 1 - Delivery Robot (Model Based Agent)\n",
   "#    The Delivery Robot picks up packages from various Warehouses and delivers them to a specific area. The agent will keep \n",
   "#     track of warehouse locations from where the packhages are picked. (1 for picking packages from location and 0 for no \n",
   "#      package).\n",
   "\n",
   "print('\\n\\t MODEL BASED AGENT \\n')\n",
   "print('Scenario 1 - Delivery Robot \\n')\n",
   "\n",
   "import random\n",
   "\n",
   "class Locations:\n",
   "    def __init__(self, size):\n",
   "        self.size = size\n",
   "        self.location = self.generate_locations()  # These locations are of warehouses\n",
   "        \n",
   "    def generate_locations(self):\n",
   "        return [random.choice([0, 1]) for _ in range(self.size)]\n",
   "    \n",
   "    def package_present(self, position):\n",
   "        return self.location[position] == 1\n",
```

```
"    \n",
"    def pickPackage(self, position):\n",
"        if self.package_present(position):\n",
"            self.location[position] = 0  \n",
"            print(f\"Package picked from location: {position}\")\n", "\n",
"    def get_package_locations(self):\n",
"        return [i for i in range(self.size) if self.package_present(i)]\n",
"    \n",
"class DeliveryAgent:\n",
"    def __init__(self, locations):\n",
"        self.locations = locations\n",
"        self.current_location = 0  \n",
"        self.history = []  \n",
"        \n",
"    def pick_packages(self):   \n",
"        while True:\n",
"            package_locations = self.locations.get_package_locations()\n",
"            print(\"Current Warehouse locations:\", package_locations)\n",
"            if not  package_locations:\n",
"                print(\"All Packages are picked from locations!\")\n",
"                break\n",
"           \n",
"            next_location =  package_locations[0]  # Move to the first location\n",
"            self.move_to(next_location)\n",
"            self.locations.pickPackage(next_location)\n",
```

```
"          \n",
"    def move_to(self, location):\n",
"        print(f\"Moved from location {self.current_location} to {location}\")\n",
"        self.current_location = location  # Update the current position\n",
"        self.history.append(location)  # Record the move in history\n",
"        \n",
"    def print_history(self):\n",
"        print(\"History of moves:\", self.history)\n",
"\n",
"warehouse_numbers = 10\n",
"locations = Locations(warehouse_numbers)\n",
"print(\"Initial Warehouse states (1 indicates pick package from location, 0 indicates no package):\")\n",
"print(locations.location)\n",
"delivery_agent = DeliveryAgent(locations)\n",
"delivery_agent.pick_packages()\n",
"print(\"\\nFinal Warehouse States:\")\n",
"print(locations.location)\n",
"delivery_agent.print_history()\n",
"print('\\n\\n\\n')\n",
"\n",
"\n",
"\n",
"\n",
"# Scenario 2 - Dishwashing robot (Goal based agent)\n",
"#    The robot's task is to clean a set of dirty dishes represented in a list. Each dish can either be dirty \n",
```

```
"#       (represented by 1) or clean (represented by 0).\n",
"\n",
"\n",
"print('\\n\\t GOAL BASED AGENT \\n')\n",
"print('Scenario 2 - Diswashing Robot \\n')\n",
"\n",
"class DishRack:\n",
"    def __init__(self, initial_state):\n",
"        self.state = initial_state  \n",
"    \n",
"    def wash(self, position):\n",
"        if self.state[position] == 1:  \n",
"            self.state[position] = 0  \n",
"            print(f\"Dish washed at position {position}.\")\n",
"    \n",
"    def display(self):\n",
"        print(\"Current Dish Rack State:\", self.state)\n",
"\n",
"class DishwashingAgent:\n",
"    def __init__(self, dish_rack):\n",
"        self.dish_rack = dish_rack\n",
"    \n",
"    def wash_dishes(self):\n",
"        print(\"Starting dishwashing process...\")\n",
"        for position in range(len(self.dish_rack.state)):\n",
"            if self.dish_rack.state[position] == 1:\n",
```

```
   "              self.dish_rack.wash(position)  \n",
   "          else:\n",
   "              print(f\"Dish at position {position} is already clean, skipping.\")\n",
   "       print(\"Dishwashing process completed.\")\n",
   "       self.dish_rack.display()\n",
   "\n",
   "\n",
   "initial_state = [0 , 1, 1, 1, 0, 0, 1, 0, 0, 0, 1]\n",
   "dish_rack = DishRack(initial_state)  \n",
   "dish_rack.display() \n",
   "\n",
   "agent = DishwashingAgent(dish_rack)  \n",
   "agent.wash_dishes()  \n",
   "print('\\n\\n\\n')\n",
   "\n",
"\n",    "\n",
   "\n",
   "#  Scenario 3 - Garden Maintenance Robot (Utility based agent)\n",
   "#    The robot's task is to water plants in a garden, where each plant can be either dry (represented by 1) or already \n",
   "#     watered (represented by 0). Each plant has a different utility value indicating its importance based on the type of plant.\n",
   "\n",
   "\n",
   "print('\\n\\t UILITY BASED AGENT \\n')\n",
   "print('Scenario 3 - Garden Maintenance Robot \\n')\n",
   "\n",
```

```
"class Garden:\n",
"    def __init__(self):\n",
"        self.state = [0, 1, 1, 1, 0, 0, 1, 1] \n",
"        self.utilities = [0, 2, 5, 10, 0, 0, 7, 6] \n", "
\n",
"    def water(self, position):\n",
"        if self.state[position] == 1:  \n",
"            self.state[position] = 0  \n",
"            print(f\"Watered plant at position {position}.\")\n",
"            print(f\"Updated garden state: {self.state}\")\n",
"    \n",
"    def get_utility(self, position):\n",
"        return self.utilities[position]\n",
"\n",
"class UtilityBasedGardenMaintenanceAgent:\n",
"    def __init__(self, garden):\n",
"        self.garden = garden\n",
"    \n",
"    def water_plants(self):\n",
"        print(\"Starting watering process...\")\n",
"        while True:\n",
"            highest_utility = -1\n",
"            best_position = -1\n",
"            for position in range(len(self.garden.state)):\n",
"                if self.garden.state[position] == 1:  # If dry\n",
"                    utility = self.garden.get_utility(position)                \n",
```

```
    "                    if utility > highest_utility: \n",
    "                        highest_utility = utility\n",
    "                        best_position = position        \n",
    "            if best_position != -1:  \n",
    "                self.garden.water(best_position)\n",
    "            else:\n",
    "                print(\"No dry plants to water.\")\n",
    "                break \n",
    "        print(\"Watering process completed.\")\n",
    "\n",
"\n",
    "garden = Garden()\n",
    "agent = UtilityBasedGardenMaintenanceAgent(garden)\n",
    "agent.water_plants()\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "d774d8ba",
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
 "metadata": {
```

```
 "kernelspec": {
  "display_name": "Python 3 (ipykernel)",
  "language": "python",
  "name": "python3"
 },
 "language_info": {
  "codemirror_mode": {
   "name": "ipython",
   "version": 3
  },
  "file_extension": ".py",
  "mimetype": "text/x-python",
  "name": "python",
  "nbconvert_exporter": "python",
  "pygments_lexer": "ipython3",
  "version": "3.11.5"
 }
},
"nbformat": 4,
"nbformat_minor": 5
}
```