CS210 : ARTIFICIAL INTELLIGENCE LAB

LAB ASSIGNMENT 6: AI & Python

Submitted By:

Name: AKSHAT SAHU

Roll No: U22CS034

Branch: CSE

Semester: 4th Sem

Division : A

Submitted To: Dr. Chandra Prakash

Department of Computer Science and Engineering

SV NATIONAL INSTITUTE OF TECHNOLOGY SURAT

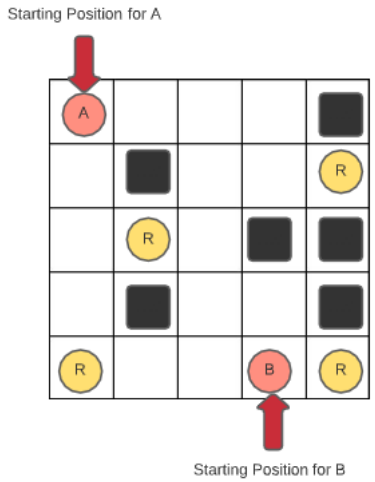
2024

PART A : Adversarial Problem [25 Marks]

1. Fastest Multi-Agent Reward Collection [5 marks]

Inputs: Consider the maze given in the figure below. The walled tiles are marked in black and your agent A and B cannot move to or through those positions.
Write a python/C program that takes the maze as a 5x5 matrix input where 0 denotes an empty tile, 1 denotes an obstruction/wall, 2 denotes the start state and 3 denotes the reward. Assume valid actions as L,R,U,D,S,N where L=move_left, R=move_right, U=move_up, D=move_down.

Your code should help the agents collect all the rewards individually and record the steps in doing so. The agent with the minimum number of steps to collect the rewards wins that round of the game. Run this game for 10 rounds/Episodes, the agent with the most number of wins after 10 rounds is declared as the winner.



Hints: a) To achieve this you can use any search algorithm eg. BFS/ DFS /A*.
b) Your program should create the appropriate data structure that can capture problem states, asmentioned in the problem.
c) Once the all the goals are reached (i.e. Reward position), program should terminate.

Outputs: The output should contain the number of tiles visited by each agent and the winner for each round. It should also declare the winner of all the rounds combined as "out_advsearch.txt".

```
import heapq

class MazeSolver:
    def __init__(self, maze):
        self.maze = maze
        self.rows = len(maze)
        self.cols = len(maze[0])
        self.agent_A_start = self.find_start(2)
        self.agent_B_start = self.find_start(4)
        self.rewards = self.find_rewards()
        self.visited_A = set()
        self.visited_B = set()
```

```
    def find_start(self, agent_id):
        for i in range(self.rows):
            for j in range(self.cols):
                if self.maze[i][j] == agent_id:
                    return (i, j)

    def find_rewards(self):
        rewards = []
        for i in range(self.rows):
            for j in range(self.cols):
                if self.maze[i][j] == 3:
                    rewards.append((i, j))
        return rewards

    def heuristic(self, current, goal):
        return abs(current[0] - goal[0]) + abs(current[1] - goal[1])

    def is_valid_move(self, position):
        x, y = position
        return 0 <= x < self.rows and 0 <= y < self.cols and self.maze[x][y] != 1

    def astar(self, start, goal, visited):
        heap = [(0, start, [])]

        while heap:
            _, current, path = heapq.heappop(heap)

            if current == goal:
                visited.update(path)
                return path

            if current not in visited:
                visited.add(current)

                moves = [(-1, 0, 'U'), (1, 0, 'D'), (0, -1, 'L'), (0, 1, 'R')]

                for dx, dy, action in moves:
                    next_position = (current[0] + dx, current[1] + dy)
                    if self.is_valid_move(next_position):
                        heapq.heappush(heap, (self.heuristic(next_position, goal)
+ len(path), next_position, path + [(next_position, action)]))

        return []

    def solve_maze(self, agent_start, agent_rewards, visited_set):
        start = agent_start
```

```
        total_steps = 0

        for reward in agent_rewards:
            path = self.astar(start, reward, visited_set)
            total_steps += len(path) - 1
            start = reward

        return total_steps

    def play_game(self):
        wins_A = 0
        wins_B = 0

        for _ in range(10):
            steps_A = self.solve_maze(self.agent_A_start, self.rewards,
self.visited_A)
            steps_B = self.solve_maze(self.agent_B_start, self.rewards,
self.visited_B)

            if steps_A < steps_B:
                wins_A += 1
            else:
                wins_B += 1

            print(f"Round {_ + 1}:")
            print(f"Agent A steps: {steps_A}")
            print(f"Agent B steps: {steps_B}")
            print()

        # Declare the winner of all rounds combined
        if wins_A > wins_B:
            print("Agent A is the winner!")
        elif wins_B > wins_A:
            print("Agent B is the winner!")
        else:
            print("It's a tie!")

if __name__ == "__main__":
    maze = [
        [2, 0, 0, 0, 1],
        [0, 1, 0, 0, 3],
        [0, 3, 0, 1, 1],
        [0, 1, 0, 0, 1],
        [3, 0, 0, 0, 4]
    ]

    solver = MazeSolver(maze)
```

```
    solver.play_game()
```

2. Fastest Multi-Agent Reward Collection Using Minimax algorithm [10 Marks] Inputs:
   In the above problem, we make a small modification by making the game a turn based one. Agent A will have the first turn, then B and so on till one of them ends up collecting all the rewards.

   a)Use MiniMax algorithm to achieve this and declare the winner of the game. You need to do this only for 1 round.
   b) Explore Alpha-beta algorithm.
   In your output file, include the visiting sequence for each agent and the eventual winner of the game.

```
# Define the initial game state
initial_state = {
    'agents': ['A', 'B'],  # Agents participating in the game
    'current_agent': 'A',  # Agent starting the game
    'rewards_collected': {'A': 0, 'B': 0}  # Rewards collected by each agent
}

# Define a function to generate successor states
def successors(state):
    current_agent = state['current_agent']
    next_agent = 'A' if current_agent == 'B' else 'B'

    # Generate successor states for the current agent's turn
    successor_states = []
    for reward in range(1, 11):
        successor_state = state.copy()
        successor_state['rewards_collected'][current_agent] += reward
        successor_state['current_agent'] = next_agent
        successor_states.append(successor_state)

    return successor_states

# Define a function to evaluate the utility of a state
def evaluate(state):
    # Difference in rewards collected by agents
    return state['rewards_collected']['A'] - state['rewards_collected']['B']

# Minimax algorithm
def minimax(state, depth, maximizing_agent):
    if depth == 0 or game_over(state):
        return evaluate(state)

    if maximizing_agent:
        max_eval = float('-inf')
        for successor_state in successors(state):
            eval = minimax(successor_state, depth - 1, False)
```

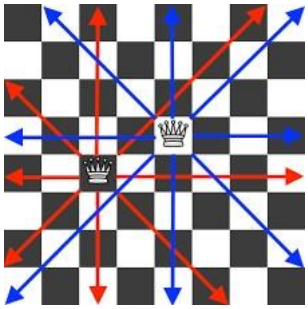
```
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float('inf')
        for successor_state in successors(state):
            eval = minimax(successor_state, depth - 1, True)
            min_eval = min(min_eval, eval)
        return min_eval

# Define a function to check if the game is over
def game_over(state):
    # Check if any agent has collected all rewards
    return max(state['rewards_collected'].values()) >= 10

# Main code
winner_score = minimax(initial_state, 1, True)  # One round of the game

# Determine the winner based on the final game state
winner = 'A' if winner_score > 0 else 'B'
print("Winner of the game:", winner)
```

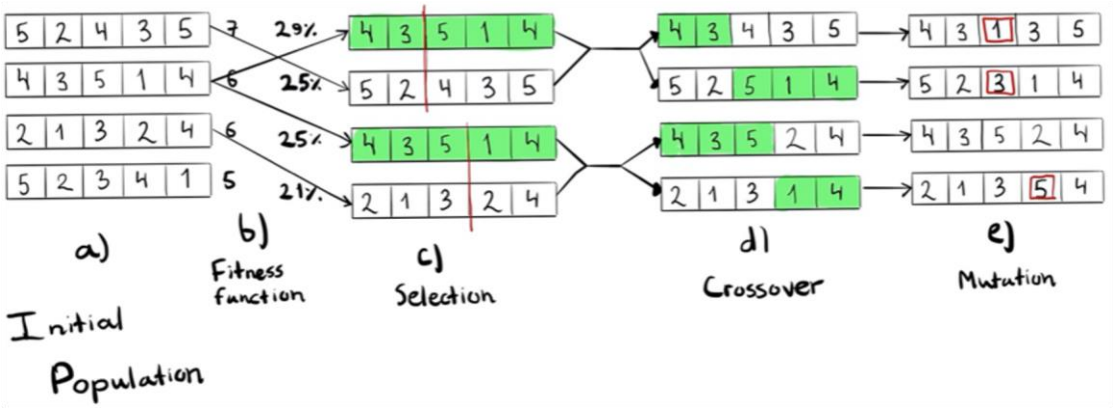
3. Evolutionary algorithms can be seen as a type of random search that tries to solve problems with the help of nature. Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. [10 Marks]



In a 8-queen problem, the goal is to place eight queens on an 8x8 chessboard so that no two queens threaten or attack each other. To prevent the queens from attacking one another, no two queens should be in the same row, column, or diagonal.

Apply GA to solve the given 5-queen problem in python. Accordingly give the of solutions. Comment on the average fitness value of the population after every iteration.

Steps in the Genetic Algorithm:



1. Chromosome design
2. Initialization: Create a randomized population of potential solutions (board states).
3. Fitness evaluation:
4. Selection
5. Crossover
6. Mutation
7. Update Generation
8. Go back to step 3 or Termination

```
import random

def create_board(size):
    board = [0] * size
    for i in range(size):
        board[i] = random.randint(0, size - 1)
    return board

def fitness(board):
    conflicts = 0
    size = len(board)
    for i in range(size):
        for j in range(i + 1, size):
            if board[i] == board[j] or abs(i - j) == abs(board[i] -
board[j]):
                conflicts += 1
    return conflicts

def crossover(parent1, parent2):
    size = len(parent1)
    crossover_point = random.randint(1, size - 1)
    child = parent1[:crossover_point] + parent2[crossover_point:]
    return child

def mutate(board):
    size = len(board)
```

```
        mutation_point = random.randint(0, size - 1)
        new_value = random.randint(0, size - 1)
        board[mutation_point] = new_value
        return board

    def genetic_algorithm(population_size, generations):
        board_size = 5
        population = [create_board(board_size) for _ in
range(population_size)]

        for _ in range(generations):
            population = sorted(population, key=lambda x: fitness(x))
            if fitness(population[0]) == 0:
                return population[0]

            new_population = []

            for _ in range(population_size // 2):
                parent1 = random.choice(population[:population_size // 2])
                parent2 = random.choice(population[:population_size // 2])
                child = crossover(parent1, parent2)
                if random.random() < 0.1:
                    child = mutate(child)
                new_population.append(child)

            population = new_population

        return None

    solution = genetic_algorithm(population_size=100, generations=1000)
    if solution:
        print("Solution found:", solution)
    else:
        print("Solution not found. Try increasing the population size or
number of generations.")
```

PART B : Exploratory Problem [15 Marks]

4. Watch the AlphaGo - The Movie | Full award-winning documentary on youtube (link : https://www.youtube.com/watch?v=WXuK6gekU1Y) and answer the following :

   (a) What this documentary is all about. Who were the player/s competing in this. On which operating system Alpha-go was running.

   (b) I was't able to foresee. What we can infer from this statement at 43.10 by the player.

(c) How moves ahead / many ply, *Alpha-Go* can foresee according to the documentary. What do you think about human foresee capability in this regard.

(d) How many total Episode(full game) was played in the game between Alpha-Go and human player. What was the result finally.

(e) Game 2, move 37 was played by Alpha-go. What was the expert comment on this? What did Lee Sedol view on this.

(f) What was the comment of Director, Stanford A.I Lab. What ML Techniques she worked on.

(g) What is a slack-move. What we can learn form this move from game 5.

(h) What do you think about the max-min approach contribution in this game. Is it of some use.

(i) What was the algorithm Aplha-go, implemented against the world best go player.

```python
# Define the initial game state
initial_state = {
    'agents': ['A', 'B'],  # Agents participating in the game
    'current_agent': 'A',  # Agent starting the game
    'rewards_collected': {'A': 0, 'B': 0},  # Rewards collected by each agent
    'visiting_sequence': []  # Visiting sequence for each agent
}

# Define a function to generate successor states
def successors(state):
    current_agent = state['current_agent']
    next_agent = 'A' if current_agent == 'B' else 'B'

    # Generate successor states for the current agent's turn
    successor_states = []
    for reward in range(1, 11):
        successor_state = state.copy()
        successor_state['rewards_collected'][current_agent] += reward
        successor_state['current_agent'] = next_agent
        successor_state['visiting_sequence'].append((current_agent, reward))
        successor_states.append(successor_state)

    return successor_states

# Define a function to evaluate the utility of a state
def evaluate(state):
    # Difference in rewards collected by agents
    return state['rewards_collected']['A'] - state['rewards_collected']['B']

# Alpha-beta algorithm
def alpha_beta(state, depth, alpha, beta, maximizing_agent):
    if depth == 0 or game_over(state):
        return evaluate(state)

    if maximizing_agent:
```

```
        max_eval = float('-inf')
        for successor_state in successors(state):
            eval = alpha_beta(successor_state, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cutoff
        return max_eval
    else:
        min_eval = float('inf')
        for successor_state in successors(state):
            eval = alpha_beta(successor_state, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cutoff
        return min_eval

# Define a function to check if the game is over
def game_over(state):
    # Check if any agent has collected all rewards
    return max(state['rewards_collected'].values()) >= 10

# Main code
winner_score = alpha_beta(initial_state, 3, float('-inf'), float('inf'), True)  #
Search depth: 3

# Determine the winner based on the final game state
winner = 'A' if winner_score > 0 else 'B'
print("Winner of the game:", winner)
print("Visiting sequence for each agent:", initial_state['visiting_sequence'])
```