Vivekanand Education Society's
Institute of Technology
Approved by AICTE & Affiliated to University of Mumbai

Artificial Intelligence and Data Science Department

Subject/Odd Sem 2023-23/Experiment 1

Program :

1.	<pre>def prisoners_dilemma(player_a_choice, player_b_choice):
	    if player_a_choice == 'Cooperate' and player_b_choice == 'Cooperate':
	        return "Both players cooperate. Each gets 3 points."
	    elif player_a_choice == 'Cooperate' and player_b_choice == 'Betray':
	        return "Player A cooperates, but Player B betrays. Player A gets 0 points, Player B gets 5 points."
	    elif player_a_choice == 'Betray' and player_b_choice == 'Cooperate':
	        return "Player A betrays, but Player B cooperates. Player A gets 5 points, Player B gets 0 points."
	    elif player_a_choice == 'Betray' and player_b_choice == 'Betray':
	        return "Both players betray. Each gets 1 point."
	
	player_a_choice = input("Player A, choose 'Cooperate' or 'Betray': ")
	player_b_choice = input("Player B, choose 'Cooperate' or 'Betray': ")
	
	result = prisoners_dilemma(player_a_choice, player_b_choice)
	print(result)</pre>
	
	Output Screenshots :

```
PS C:\Users\Admin1\Desktop\Game Theory Lab> & "C:/Program Files/Python310/python.exe" "c:/Users/Admin1/Desktop/Game Theory Lab/dilemma.py"
Player A, choose 'Cooperate' or 'Betray': Cooperate
Player B, choose 'Cooperate' or 'Betray': Betray
Player A cooperates, but Player B betrays. Player A gets 0 points, Player B gets 5 points.
PS C:\Users\Admin1\Desktop\Game Theory Lab> & "C:/Program Files/Python310/python.exe" "c:/Users/Admin1/Desktop/Game Theory Lab/dilemma.py"
Player A, choose 'Cooperate' or 'Betray': Betray
Player B, choose 'Cooperate' or 'Betray': Cooperate
Player A betrays, but Player B cooperates. Player A gets 5 points, Player B gets 0 points.
PS C:\Users\Admin1\Desktop\Game Theory Lab> & "C:/Program Files/Python310/python.exe" "c:/Users/Admin1/Desktop/Game Theory Lab/dilemma.py"
Player A, choose 'Cooperate' or 'Betray': Cooperate
Player B, choose 'Cooperate' or 'Betray': Cooperate
Both players cooperate. Each gets 3 points.
PS C:\Users\Admin1\Desktop\Game Theory Lab> & "C:/Program Files/Python310/python.exe" "c:/Users/Admin1/Desktop/Game Theory Lab/dilemma.py"
Player A, choose 'Cooperate' or 'Betray': Betray
Player B, choose 'Cooperate' or 'Betray': Betray
Both players betray. Each gets 1 point.
PS C:\Users\Admin1\Desktop\Game Theory Lab>
```

Results and Discussions : The Prisoner's Dilemma illustrates the tension between individual self-interest and collective cooperation. Despite the rational choice for both players being to betray, the optimal outcome for both is achieved through cooperation. This implementation demonstrates how real-world scenarios can be modeled using game theory concepts and how outcomes can be affected by different choices, leading to insights into strategic decision-making and the interplay between competing interests.

Program :

1.

```
up_left = []
up_right = []
down_left = []
down_right = []

print('Please Enter the Values for your Matrix')
up_left.append(float(input('Please Enter A Value for A: ')))
up_left.append(float(input('Please Enter A Value for B: ')))
up_right.append(float(input('Please Enter A Value for C: ')))
up_right.append(float(input('Please Enter A Value for D: ')))
down_left.append(float(input('Please Enter A Value for E: ')))
down_left.append(float(input('Please Enter A Value for F: ')))
down_right.append(float(input('Please Enter A Value for G: ')))
down_right.append(float(input('Please Enter A Value for H: ')))

if up_left[0] >= down_left[0] and up_left[1] >= up_right[1]:
    up_left_bool = 1
else:
    up_left_bool = 0
```

```
if down_left[0] >= up_left[0] and down_left[1] >= down_right[1]:
    down_left_bool = 1
else:
    down_left_bool = 0

if up_right[0] >= down_right[0] and up_right[1] >= up_left[1]:
    up_right_bool = 1
else:
    up_right_bool = 0

if down_right[0] >= up_right[0] and down_right[1] >= down_left[1]:
    down_right_bool = 1
else:
    down_right_bool = 0

bool_values = [up_left_bool, up_right_bool, down_left_bool,
down_right_bool]

if up_left_bool == 1:
    print('Up, Left is a Nash Equilibrium')
else:
```

```
    pass


if up_right_bool == 1:

    print('Up, Right is a Nash Equilibrium')

else:

    pass


if down_left_bool == 1:

    print('Down, Left is a Nash Equilibrium')

else:

    pass


if down_right_bool == 1:

    print('Down, Right is a Nash Equilibrium')

else:

    pass


if 1 not in bool_values:

    print('There are no Nash Equilibria')

else:

    pass
```

Output Screenshots :

```
PS C:\Users\Admin1\Desktop\Game Theory Lab> & "C:/Program Files/Python310/python.exe" "c:/Users/Admin1/Desktop/Game Theory Lab/nash.py"
Please Enter the Values for your Matrix
Please Enter A Value for A: 3
Please Enter A Value for B: 3
Please Enter A Value for C: -1
Please Enter A Value for D: 5
Please Enter A Value for E: 5
Please Enter A Value for F: -1
Please Enter A Value for G: 0
Please Enter A Value for H: 0
Down, Right is a Nash Equilibrium
PS C:\Users\Admin1\Desktop\Game Theory Lab>
```

Results and Discussions : Nash Equilibrium in the context of the Prisoner's Dilemma demonstrates how rational decision-making can lead to stable outcomes even when individual choices conflict with optimal collective results.

Program :

```
def calculate_mixed_strategy_nash_equilibrium():
    print("Matching Pennies Game Nash Equilibrium Calculator!")
    print("Player 1 (Row Player) chooses H with probability p and T with probability 1-p.")
    print("Player 2 (Column Player) chooses H with probability q and T with probability 1-q.")

    # Get the input probabilities from the user for Player 1
    while True:
        try:
            p = float(input("Enter the probability (0 to 1) that Player 1 chooses H (p): "))
            if 0 <= p <= 1:
                break
            else:
                print("Invalid input. Please enter a probability between 0 and 1.")
        except ValueError:
            print("Invalid input. Please enter a valid number between 0 and 1.")

    # Calculate Player 2's mixed strategy based on Player 1's strategy
    q = (2 * p - 1) / (2 * p)

    # Check if q is a valid probability (between 0 and 1)
    if 0 <= q <= 1:
        print("\nMixed Strategy Nash Equilibrium:")
        print("Player 1's Mixed Strategy (p):", p)
        print("Player 2's Mixed Strategy (q):", q)
```

```python
    else:
        print("No valid Nash equilibrium exists.")

if __name__ == "__main__":
    calculate_mixed_strategy_nash_equilibrium()
```

Output

P = 0

```
Welcome to the Matching Pennies Game Nash Equilibrium Calculator!
Player 1 (Row Player) chooses H with probability p and T with probability 1-p.
Player 2 (Column Player) chooses H with probability q and T with probability 1-q.
Enter the probability (0 to 1) that Player 1 chooses H (p): 0.5

Mixed Strategy Nash Equilibrium:
Player 1's Mixed Strategy (p): 0.5
Player 2's Mixed Strategy (q): 0.0
```

P = 0.3

```
Matching Pennies Game Nash Equilibrium Calculator!
Player 1 (Row Player) chooses H with probability p and T with probability 1-p.
Player 2 (Column Player) chooses H with probability q and T with probability 1-q.
Enter the probability (0 to 1) that Player 1 chooses H (p): 0.3
No valid Nash equilibrium exists.
```

P = 0.9

```
Matching Pennies Game Nash Equilibrium Calculator!
Player 1 (Row Player) chooses H with probability p and T with probability 1-p.
Player 2 (Column Player) chooses H with probability q and T with probability 1-q.
Enter the probability (0 to 1) that Player 1 chooses H (p): 0.9

Mixed Strategy Nash Equilibrium:
Player 1's Mixed Strategy (p): 0.9
Player 2's Mixed Strategy (q): 0.4444444444444445
```

Program :

1.

```
import networkx as nx
import matplotlib.pyplot as plt

class GameNode:
    def __init__(self, player, label):
        self.player = player
        self.label = label
        self.children = []

    def add_child(self, child_node):
        self.children.append(child_node)

def build_game_tree():
    root = GameNode("Player 1", "Root")

    decision_node_1 = GameNode("Player 1", "Decision 1")
    root.add_child(decision_node_1)

    decision_node_2a = GameNode("Player 2", "Decision 2a")
    decision_node_2b = GameNode("Player 2", "Decision 2b")
    decision_node_1.add_child(decision_node_2a)
    decision_node_1.add_child(decision_node_2b)

    terminal_node_1a = GameNode("Player 1", "Outcome A (Player 1
wins 3)")
    terminal_node_1b = GameNode("Player 1", "Outcome B (Player 1
loses 1)")
    terminal_node_2a = GameNode("Player 2", "Outcome A (Player 2
loses 3)")
```

```
    terminal_node_2b = GameNode("Player 2", "Outcome B (Player 2
wins 1)")

    decision_node_2a.add_child(terminal_node_1a)
    decision_node_2a.add_child(terminal_node_2a)
    decision_node_2b.add_child(terminal_node_1b)
    decision_node_2b.add_child(terminal_node_2b)

    return root

def visualize_game_tree(node, graph, parent=None):
    graph.add_node(node.label, player=node.player)
    if parent is not None:
        graph.add_edge(parent.label, node.label)
    for child in node.children:
        visualize_game_tree(child, graph, node)

def display_game_tree(graph):
    pos = nx.spring_layout(graph)
    labels = {node: f"{node}\n({graph.nodes[node]['player']})" for
node in graph.nodes}
    nx.draw(graph, pos, with_labels=True, labels=labels,
node_size=800, node_color="lightblue", font_size=5)
    plt.title("Game Tree")
    plt.show()

def traverse_game_tree(node):
    print(f"Current node: {node.label} ({node.player})")
    if not node.children:
        return  # Reached a terminal node

    if node.player == "Player 1":
        print("Available choices:")
        for i, child in enumerate(node.children):
```

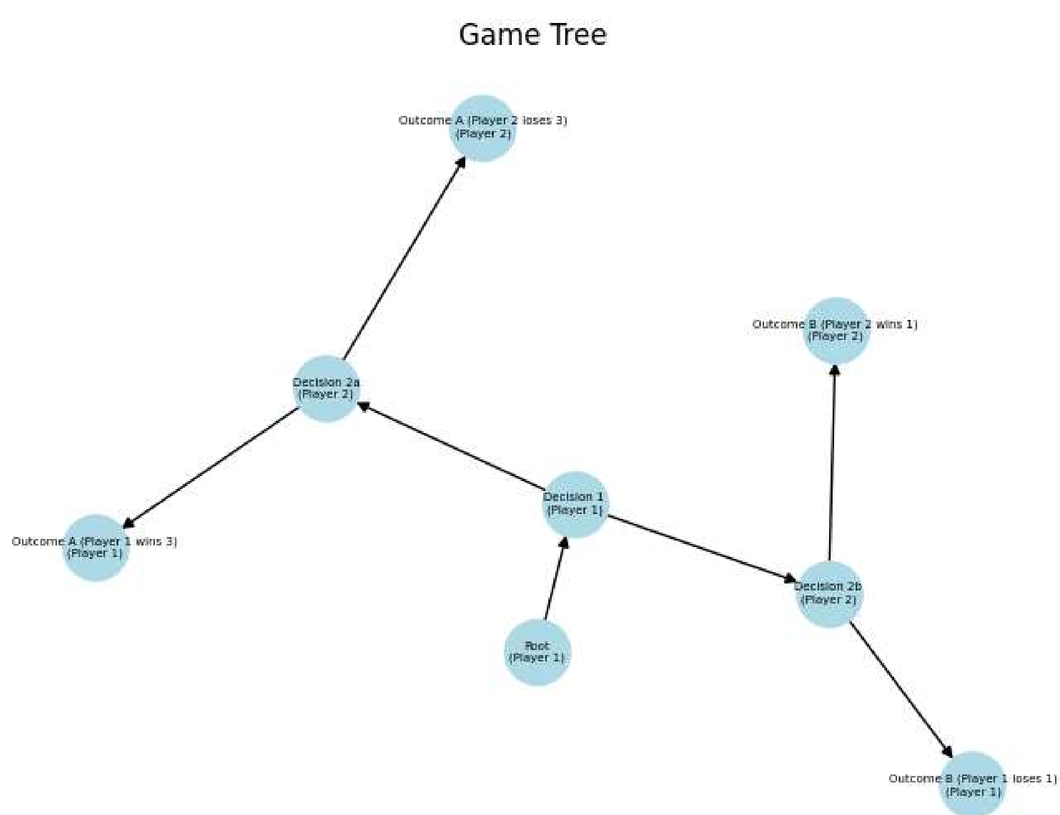
```
            print(f"{i + 1}: {child.label}")
        choice = int(input("Enter your choice (1/2): ")) - 1
        if 0 <= choice < len(node.children):
            traverse_game_tree(node.children[choice])
        else:
            print("Invalid choice. Please enter 1 or 2.")
    else:
        # Automatically choose a random option for Player 2 (you
can implement a strategy here)
        import random
        choice = random.randint(0, len(node.children) - 1)
        print(f"{node.player} chooses option {choice + 1}.")
        traverse_game_tree(node.children[choice])

if __name__ == "__main__":
    root_node = build_game_tree()
    game_tree_graph = nx.DiGraph()
    visualize_game_tree(root_node, game_tree_graph)
    display_game_tree(game_tree_graph)
    traverse_game_tree(root_node)
```

Output

```
Current node: Root (Player 1)
Available choices:
1: Decision 1
Enter your choice (1/2): 1
Current node: Decision 1 (Player 1)
Available choices:
1: Decision 2a
2: Decision 2b
Enter your choice (1/2): 2
Current node: Decision 2b (Player 2)
Player 2 chooses option 1.
Current node: Outcome B (Player 1 loses 1) (Player 1)
```

Game Tree



Program : **Imperfect information game - ROCK PAPER SCISSORS**

1.

```
import random

def get_player_choice():
    return random.choice(["Rock", "Paper", "Scissors"])

def determine_winner(player1_choice, player2_choice):
    if player1_choice == player2_choice:
        return "It's a tie!"
    elif (
        (player1_choice == "Rock" and player2_choice == "Scissors")
or
        (player1_choice == "Scissors" and player2_choice ==
"Paper") or
        (player1_choice == "Paper" and player2_choice == "Rock")
    ):
        return "Player 1 wins!"
    else:
        return "Player 2 wins!"

def play_imperfect_information_game():
    print("Welcome to the Rock, Paper, Scissors Imperfect
Information Game!")
    input("Press Enter to reveal your choices...")

    player1_choice = get_player_choice()
    player2_choice = get_player_choice()

    print(f"Player 1 chose: {player1_choice}")
    print(f"Player 2 chose: {player2_choice}")
```

```
    result = determine_winner(player1_choice, player2_choice)
    print(result)

if __name__ == "__main__":
    play_imperfect_information_game()
```

Output

```
Welcome to the Rock, Paper, Scissors Imperfect Information Game!
Press Enter to reveal your choices...Rock
Player 1 chose: Paper
Player 2 chose: Scissors
Player 2 wins!
```

```
Welcome to the Rock, Paper, Scissors Imperfect Information Game!
Press Enter to reveal your choices...Paper
Player 1 chose: Scissors
Player 2 chose: Paper
Player 1 wins!
```

```
Welcome to the Rock, Paper, Scissors Imperfect Information Game!
Press Enter to reveal your choices...
Player 1 chose: Rock
Player 2 chose: Rock
It's a tie!
```

Program : **Perfect information game - TIC TAC TOE**

1.

```
import random

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board, player):
    # Check rows, columns, and diagonals for a win
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or
all(board[j][i] == player for j in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)) or
all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

def is_full(board):
    return all(cell != " " for row in board for cell in row)

def ai_move(board):
    # Basic AI: Choose a random empty cell
    empty_cells = [(i, j) for i in range(3) for j in range(3) if
board[i][j] == " "]
    return random.choice(empty_cells)

def play_tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
```

```
    while True:
        print_board(board)

        if current_player == "X":
            row, col = map(int, input(f"Player {current_player}, 
enter your move (row and column): ").split())
        else:
            print(f"AI ({current_player}) is making a move...")
            row, col = ai_move(board)
            print(f"AI chooses row {row} and column {col}")

        if board[row][col] != " ":
            print("Invalid move. Cell already occupied. Try 
again.")
            continue

        board[row][col] = current_player

        if check_winner(board, current_player):
            print_board(board)
            if current_player == "X":
                print(f"Player {current_player} wins! 
Congratulations!")
            else:
                print(f"AI ({current_player}) wins! Better luck 
next time.")
            break

        if is_full(board):
            print_board(board)
            print("It's a draw! Nobody wins.")
            break
```

```python
        current_player = "O" if current_player == "X" else "X"

if __name__ == "__main__":
    print("Welcome to Tic-Tac-Toe!")
    play_tic_tac_toe()
```

Output

```
Welcome to Tic-Tac-Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row and column): 0 0
X |   |
---------
  |   |
---------
  |   |
---------
AI (O) is making a move...
AI chooses row 2 and column 0
X |   |
---------
  |   |
---------
O |   |
---------
Player X, enter your move (row and column): 0 2
X |   | X
---------
  |   |
---------
O |   |
---------
```

```
Player X, enter your move (row and column): 0 2
X |   | X
---------
  |   |
---------
O |   |
---------
AI (O) is making a move...
AI chooses row 1 and column 1
X |   | X
---------
  | O |
---------
O |   |
---------
Player X, enter your move (row and column): 0 1
X | X | X
---------
  | O |
---------
O |   |
---------
Player X wins! Congratulations!
```

Program :

1.

```
import random

# Function to simulate a coin toss
def coin_toss():
    return random.choice(["Heads", "Tails"])

# Function to calculate the utility of a player given their choice
and the coin toss result
def calculate_utility(player_choice, coin_result):
    if player_choice == coin_result:
        return 1
    else:
        return 0

# Main function to simulate the Bayesian Nash equilibrium
def simulate_bayesian_nash_equilibrium():
    num_simulations = 10000
    player_Alice_heads_count = 0
    player_Alice_tails_count = 0
    player_Bob_heads_count = 0
    player_Bob_tails_count = 0

    for _ in range(num_simulations):
        coin_result = coin_toss()

        # Alice's strategy (Heads or Tails)
        p_Alice_Heads = 0.6  # Adjust this probability as desired
        p_Alice_Tails = 1 - p_Alice_Heads

        # Bob's strategy (Heads or Tails)
```

```
        p_Bob_Heads = 0.4  # Adjust this probability as desired
        p_Bob_Tails = 1 - p_Bob_Heads

        # Choose based on probabilities
        alice_choice = "Heads" if random.random() < p_Alice_Heads
else "Tails"
        bob_choice = "Heads" if random.random() < p_Bob_Heads else
"Tails"

        # Calculate utilities
        alice_utility = calculate_utility(alice_choice,
coin_result)
        bob_utility = calculate_utility(bob_choice, coin_result)

        # Update counts
        if alice_choice == "Heads":
            player_Alice_heads_count += 1
        else:
            player_Alice_tails_count += 1

        if bob_choice == "Heads":
            player_Bob_heads_count += 1
        else:
            player_Bob_tails_count += 1

    # Calculate probabilities from counts
    p_Alice_Heads_eq = player_Alice_heads_count / num_simulations
    p_Alice_Tails_eq = player_Alice_tails_count / num_simulations
    p_Bob_Heads_eq = player_Bob_heads_count / num_simulations
    p_Bob_Tails_eq = player_Bob_tails_count / num_simulations

    print("Bayesian Nash Equilibrium:")
    print(f"Alice chooses Heads with probability:
{p_Alice_Heads_eq}")
```

```
    print(f"Alice chooses Tails with probability:
{p_Alice_Tails_eq}")
    print(f"Bob chooses Heads with probability: {p_Bob_Heads_eq}")
    print(f"Bob chooses Tails with probability: {p_Bob_Tails_eq}")

if __name__ == "__main__":
    print("Simulating Bayesian Nash Equilibrium in a Coin Toss
Game...")
    simulate_bayesian_nash_equilibrium()
```

Output

```
Simulating Bayesian Nash Equilibrium in a Coin Toss Game...
Bayesian Nash Equilibrium:
Alice chooses Heads with probability: 0.5977
Alice chooses Tails with probability: 0.4023
Bob chooses Heads with probability: 0.4016
Bob chooses Tails with probability: 0.5984
```