NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

GAME THEORY AND APPLICATIONS

Lab Assignments

(CACSC16)

SUBMITTED BY: -

Name: *HARSH KUMAR*

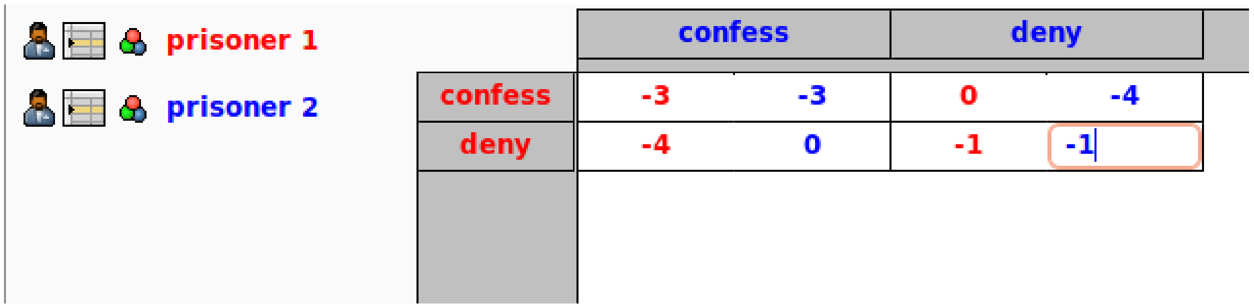
Branch: CSAI

Roll no.:2021UCA1829

TABLE OF CONTENT

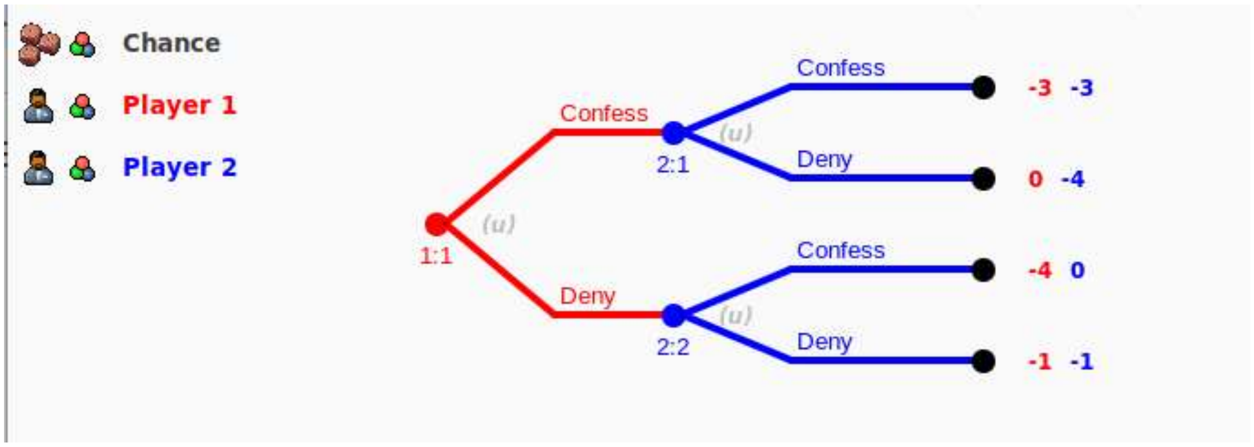
1.Construct the Game tree for the Prisoner's Dilemma in Gambit.

Table:



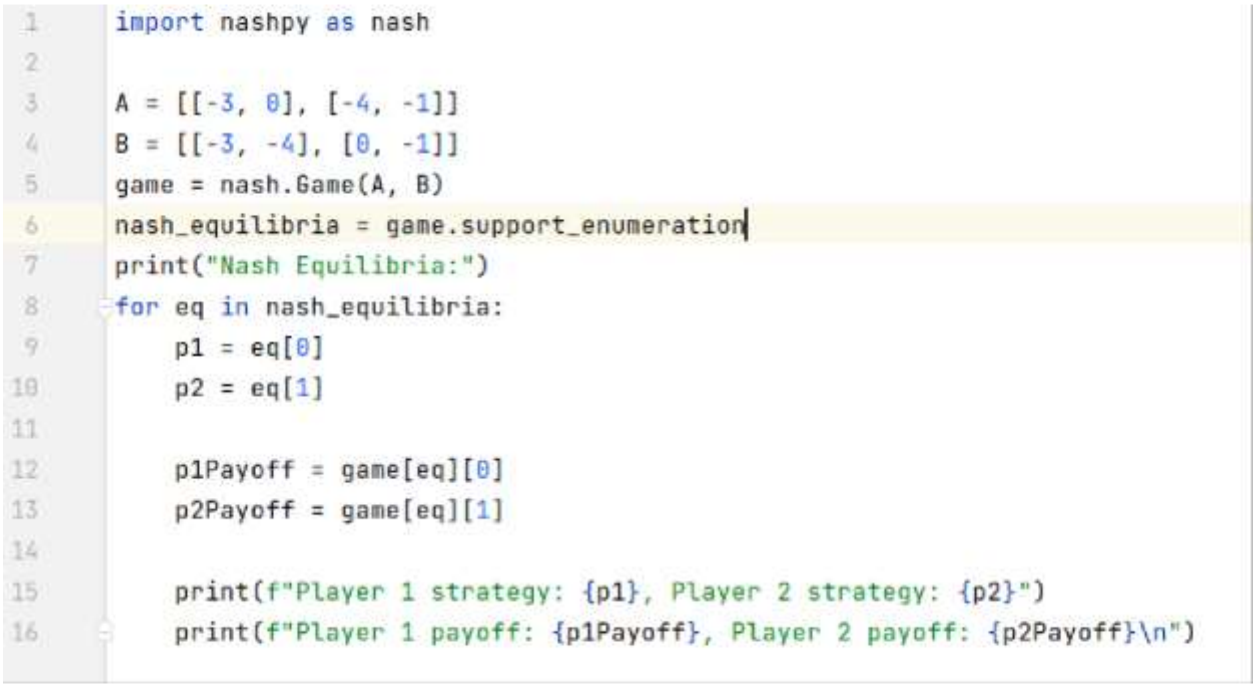
		confess		deny	
prisoner 1	confess	-3	-3	0	-4
prisoner 2	deny	-4	0	-1	-1

Graph:



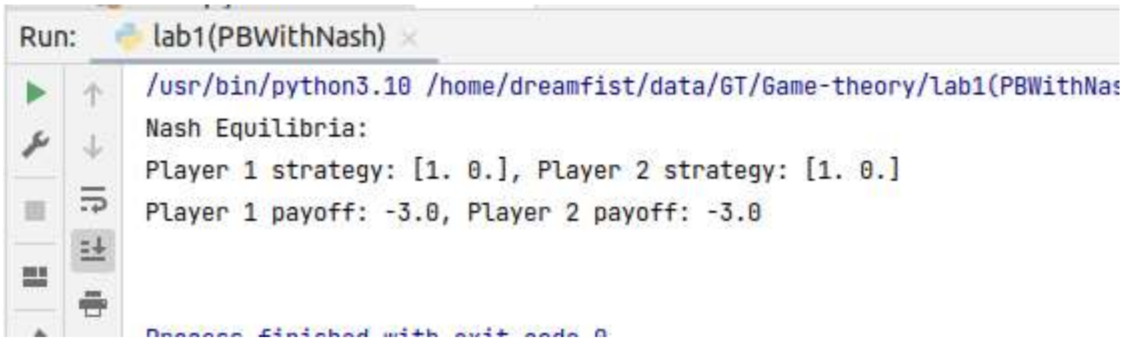
2.Evaluate Nash equilibrium for Prisoner's Dilemma using nashpy, without using nashpy

With nashpy:



```python
import nashpy as nash

A = [[-3, 0], [-4, -1]]
B = [[-3, -4], [0, -1]]
game = nash.Game(A, B)
nash_equilibria = game.support_enumeration
print("Nash Equilibria:")
for eq in nash_equilibria:
    p1 = eq[0]
    p2 = eq[1]

    p1Payoff = game[eq][0]
    p2Payoff = game[eq][1]

    print(f"Player 1 strategy: {p1}, Player 2 strategy: {p2}")
    print(f"Player 1 payoff: {p1Payoff}, Player 2 payoff: {p2Payoff}\n")
```

Output:

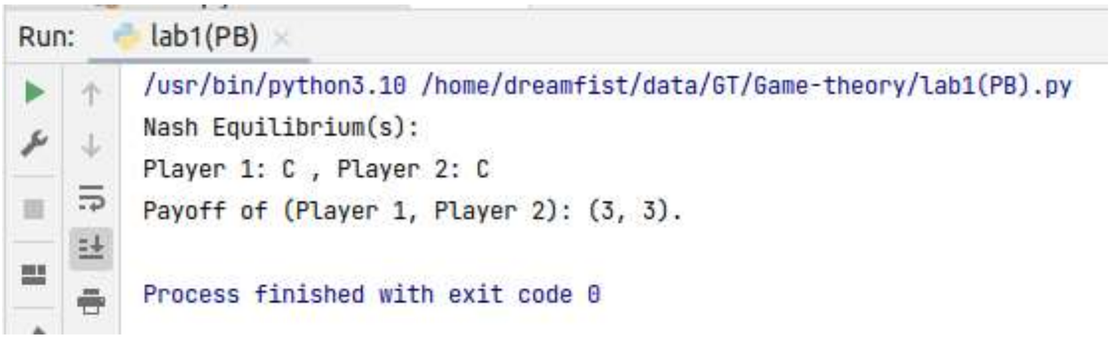


```
/usr/bin/python3.10 /home/dreamfist/data/GT/Game-theory/lab1(PBWithNas
Nash Equilibria:
Player 1 strategy: [1. 0.], Player 2 strategy: [1. 0.]
Player 1 payoff: -3.0, Player 2 payoff: -3.0


Process finished with exit code 0
```

Without nashpy:

```python
def calculate_payoff(player1_choice, player2_choice):
   payoff_matrix = {
        ('C', 'C'): (3, 3),
        ('C', 'D'): (0, 5),
        ('D', 'C'): (5, 0),
        ('D', 'D'): (1, 1),
   }
   return payoff_matrix[(player1_choice, player2_choice)]
def is_nash_equilibrium(player1_choice, player2_choice, strategy1, strategy2):
   player1_payoff = calculate_payoff(player1_choice, player2_choice)[0]
   player2_payoff = calculate_payoff(player1_choice, player2_choice)[1]

   if strategy1 == 'C' and player1_payoff < calculate_payoff('D', player2_choice)[0]:
       return False
   if strategy1 == 'D' and player1_payoff > calculate_payoff('C', player2_choice)[0]:
       return False
   if strategy2 == 'C' and player2_payoff < calculate_payoff(player1_choice, 'D')[1]:
       return False
   if strategy2 == 'D' and player2_payoff > calculate_payoff(player1_choice, 'C')[1]:
       return False
   return True

def find_nash_equilibrium(strategy1, strategy2):
   nash_equilibrium = []
   if is_nash_equilibrium('C', 'C', strategy1, strategy2):
       nash_equilibrium.append(('C', 'C'))
   if is_nash_equilibrium('C', 'D', strategy1, strategy2):
       nash_equilibrium.append(('C', 'D'))
   if is_nash_equilibrium('D', 'C', strategy1, strategy2):
       nash_equilibrium.append(('D', 'C'))
   if is_nash_equilibrium('D', 'D', strategy1, strategy2):
       nash_equilibrium.append(('D', 'D'))
   return nash_equilibrium

equilibria = find_nash_equilibrium('D', 'D')

if len(equilibria) > 0:
   print("Nash Equilibrium(s):")
   for equilibrium in equilibria:
       print(f"Player 1: {equilibrium[0]} , Player 2: {equilibrium[1]}")
else:
   print("No Nash Equilibrium found.")
```

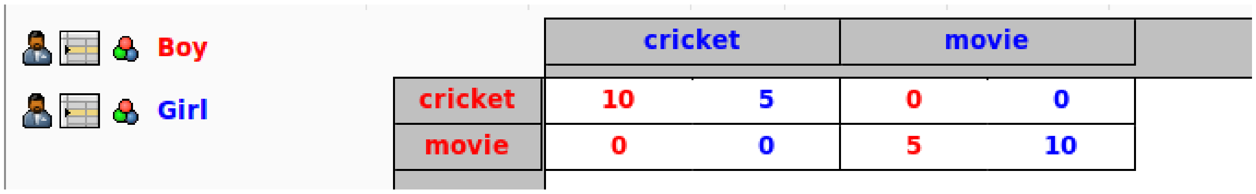
Output:



```
/usr/bin/python3.10 /home/dreamfist/data/GT/Game-theory/lab1(PB).py
Nash Equilibrium(s):
Player 1: C , Player 2: C
Payoff of (Player 1, Player 2): (3, 3).

Process finished with exit code 0
```

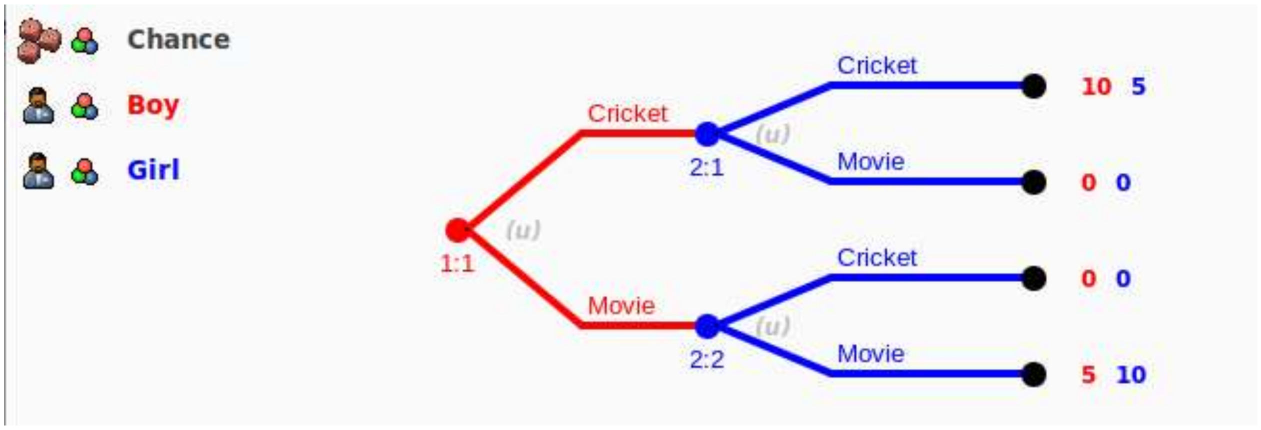
3.Construct the Game tree for the Battle of Sexes in Gambit.

Table:



		cricket		movie	
Boy					
Girl					
	cricket	10	5	0	0
	movie	0	0	5	10

Graph:



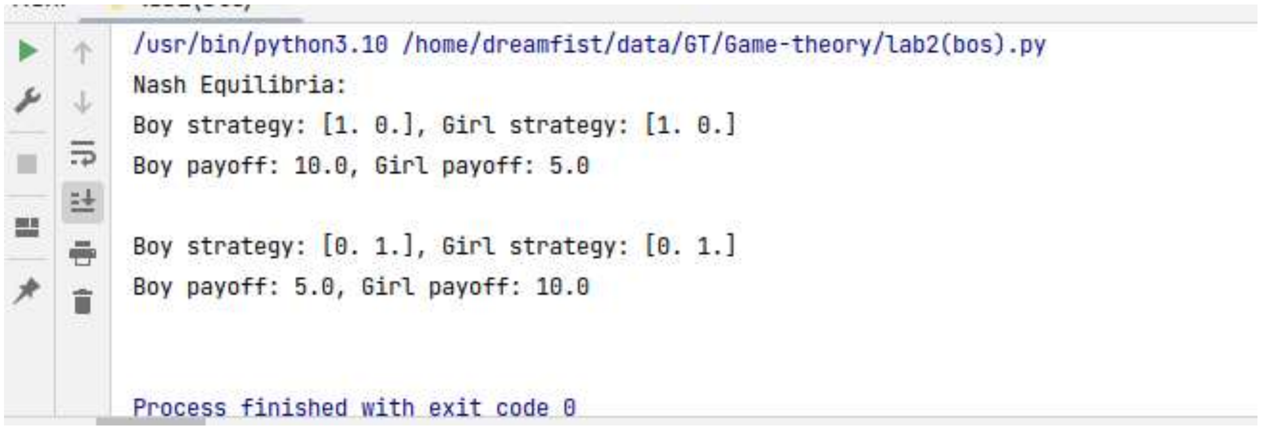
4.Evaluate Nash equilibrium for Battle of Sexes using nashpy, without using nashpy

With Nashpy:

```
import numpy as np
import nashpy as nash
boy_payoff = [[10, 0], [0, 5]]
girl_payoff = [[5, 0], [0, 10]]
game = nash.Game(boy_payoff, girl_payoff)
nash_equilibria = game.support_enumeration()
```

```
pure_st = []
for eq in nash_equilibria:
   boy_strategy = eq[0]
   girl_strategy = eq[1]
   if np.all(boy_strategy == np.round(boy_strategy)) and np.all(girl_strategy ==
np.round(girl_strategy)):
       pure_st.append(eq)

print("Nash Equilibria:")
for eq in pure_st:
   p1 = eq[0]
   p2 = eq[1]
   p1Payoff = game[eq][0]
   p2Payoff = game[eq][1]
   print(f"Boy strategy: {p1}, Girl strategy: {p2}")
   print(f"Boy payoff: {p1Payoff}, Girl payoff: {p2Payoff}\n")
```

Output:

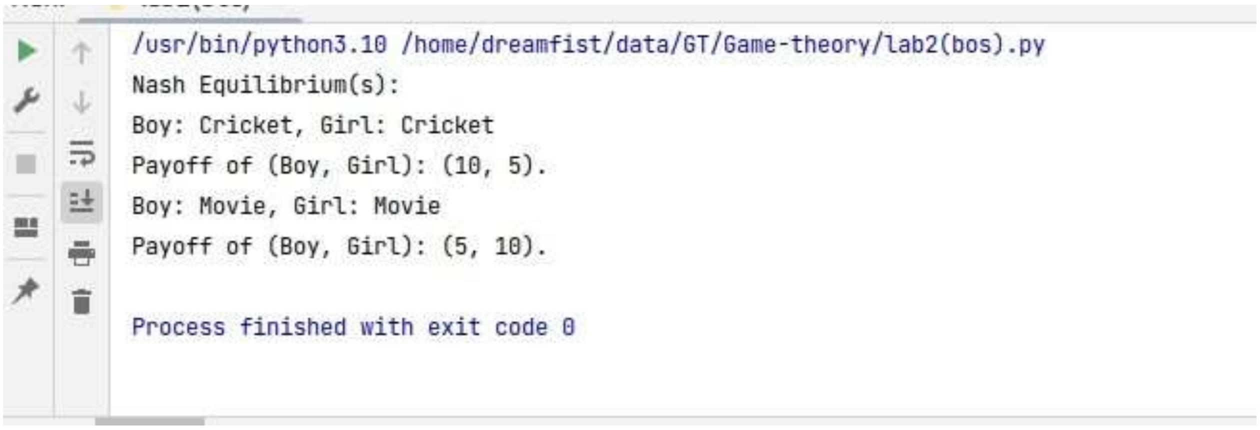


```
/usr/bin/python3.10 /home/dreamfist/data/GT/Game-theory/lab2(bos).py
Nash Equilibria:
Boy strategy: [1. 0.], Girl strategy: [1. 0.]
Boy payoff: 10.0, Girl payoff: 5.0

Boy strategy: [0. 1.], Girl strategy: [0. 1.]
Boy payoff: 5.0, Girl payoff: 10.0


Process finished with exit code 0
```

Without Nashpy:

```
def calculate_payoff(player1_choice, player2_choice):
   payoff_matrix = {
       ('Cricket', 'Cricket'): (10, 5),
       ('Cricket', 'Movie'): (0, 0),
       ('Movie', 'Cricket'): (0, 0),
       ('Movie', 'Movie'): (5, 10),
   }
   return payoff_matrix[(player1_choice, player2_choice)]

def is_nash_equilibrium(player1_choice, player2_choice, strategy1, strategy2):
   player1_payoff = calculate_payoff(player1_choice, player2_choice)[0]
   player2_payoff = calculate_payoff(player1_choice, player2_choice)[1]

   if strategy1 == 'Cricket' and player1_payoff < calculate_payoff('Movie',
player2_choice)[0]:
       return False
   if strategy1 == 'Movie' and player1_payoff > calculate_payoff('Cricket',
player2_choice)[0]:
       return False
   if strategy2 == 'Cricket' and player2_payoff < calculate_payoff(player1_choice,
'Movie')[1]:
       return False
   if strategy2 == 'Movie' and player2_payoff > calculate_payoff(player1_choice,
'Cricket')[1]:
       return False
   return True

def find_nash_equilibrium(strategy1, strategy2):
   nash_equilibrium = []
   if is_nash_equilibrium('Cricket', 'Cricket', strategy1, strategy2):
       nash_equilibrium.append(('Cricket', 'Cricket'))
   if is_nash_equilibrium('Cricket', 'Movie', strategy1, strategy2):
       nash_equilibrium.append(('Cricket', 'Movie'))
   if is_nash_equilibrium('Movie', 'Cricket', strategy1, strategy2):
       nash_equilibrium.append(('Movie', 'Cricket'))
   if is_nash_equilibrium('Movie', 'Movie', strategy1, strategy2):
       nash_equilibrium.append(('Movie', 'Movie'))
   return nash_equilibrium

equilibria = find_nash_equilibrium('Cricket', 'Cricket')
if len(equilibria) > 0:
   print("Nash Equilibrium(s):")
   for equilibrium in equilibria:
       print(f"Boy: {equilibrium[0]}, Girl: {equilibrium[1]}")
       print(f"Payoff of (Boy, Girl):
{calculate_payoff(equilibrium[0],equilibrium[1])}.")

else:
   print("No Nash Equilibrium found.")
```

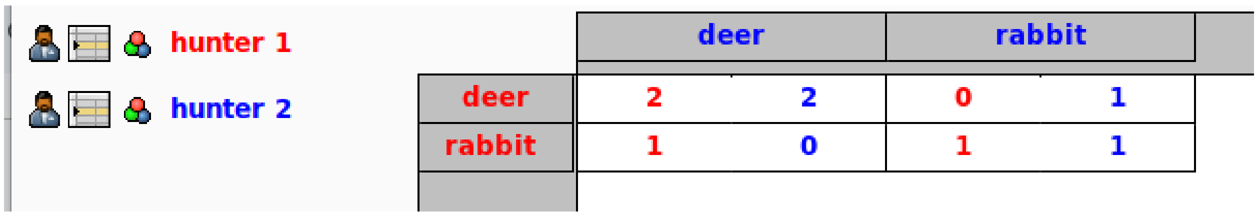
Output:



```
/usr/bin/python3.10 /home/dreamfist/data/GT/Game-theory/lab2(bos).py
Nash Equilibrium(s):
Boy: Cricket, Girl: Cricket
Payoff of (Boy, Girl): (10, 5).
Boy: Movie, Girl: Movie
Payoff of (Boy, Girl): (5, 10).

Process finished with exit code 0
```

5.Construct the Game tree for the Hunting Game in Gambit.

Table:

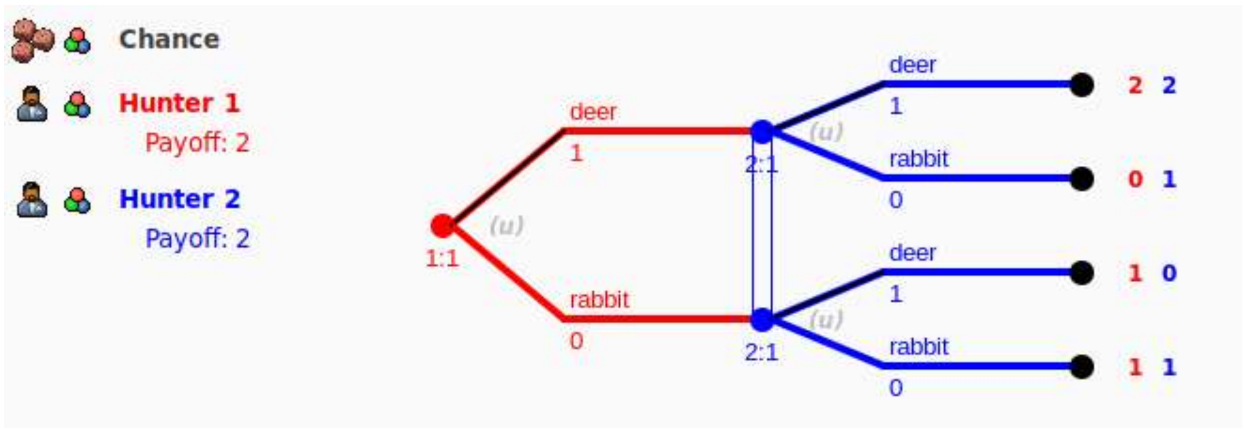


hunter 1

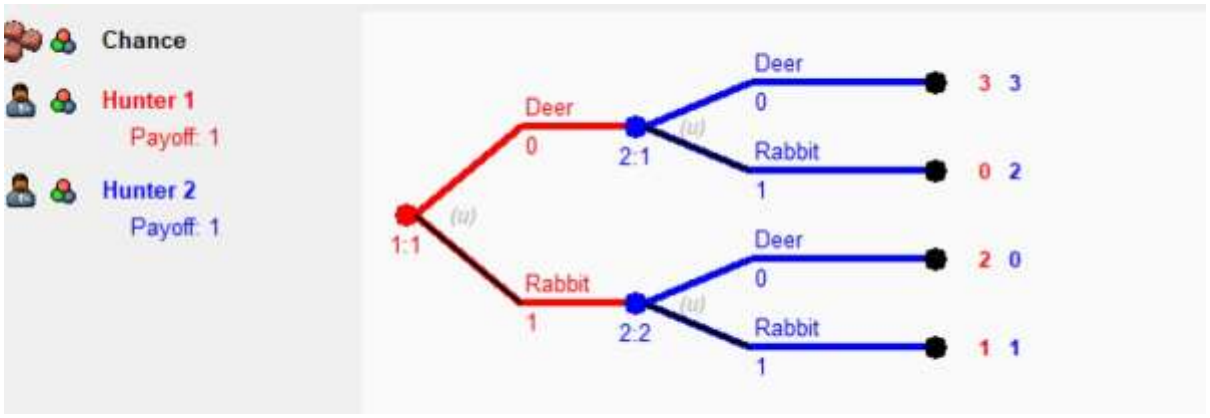
hunter 2

	deer		rabbit	
deer	2	2	0	1
rabbit	1	0	1	1

Graph:



6. Construct the Game tree for the Hunting Game in Gambit.

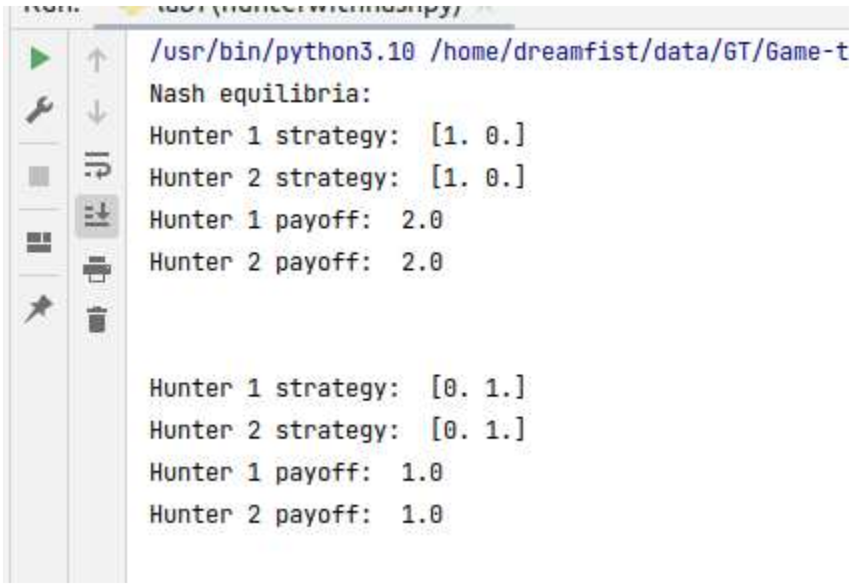


7.Evaluate Nash equilibrium for Hunting Game using nashpy, without using nashpy

With Nashpy:

```
import numpy as np
import nashpy as nash

# Define the payoff matrix
payoff_matrix_player_a = np.array([[2, 0], [1, 1]])
payoff_matrix_player_b = np.array([[2, 1], [0, 1]])

# Create the game
game = nash.Game(payoff_matrix_player_a, payoff_matrix_player_b)

# Find Nash equilibria using support_enumeration
equilibria = game.support_enumeration()

print("Nash equilibria:")
for eq in equilibria:
   p1 = eq[0]
   p2 = eq[1]

   p1Payoff = game[eq][0]
   p2Payoff = game[eq][0]

   print("Hunter 1 strategy: ", p1)
   print("Hunter 2 strategy: ", p2)
   print("Hunter 1 payoff: ", p1Payoff)
   print("Hunter 2 payoff: ", p2Payoff)
   print("\n")
```

Output:

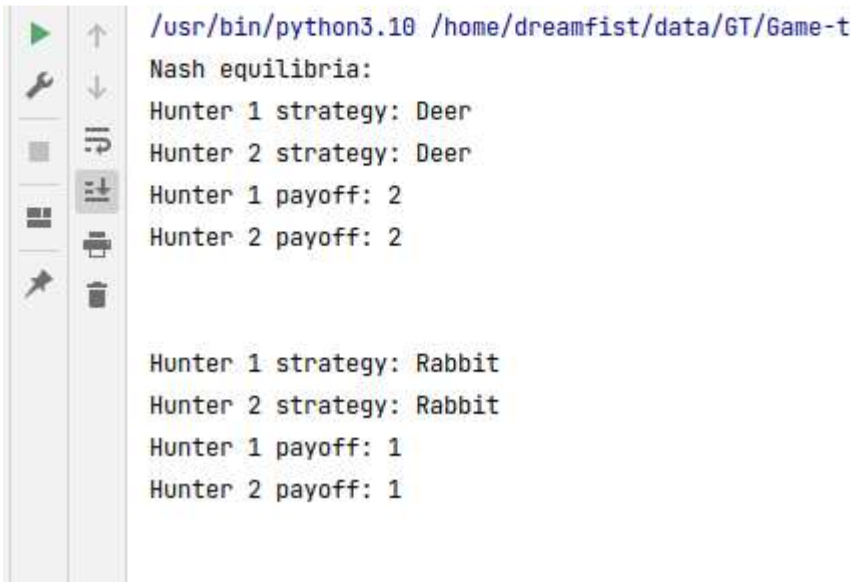


```
/usr/bin/python3.10 /home/dreamfist/data/GT/Game-t
Nash equilibria:
Hunter 1 strategy:  [1. 0.]
Hunter 2 strategy:  [1. 0.]
Hunter 1 payoff:  2.0
Hunter 2 payoff:  2.0


Hunter 1 strategy:  [0. 1.]
Hunter 2 strategy:  [0. 1.]
Hunter 1 payoff:  1.0
Hunter 2 payoff:  1.0
```

Without nashpy:

```
import numpy as np

# Define the payoff matrix
payoff_matrix_player_a = np.array([[2,  0],  [1, 1]])
payoff_matrix_player_b = np.array([[2,  1],  [0, 1]])
# Find Nash equilibria
equilibria = []
# Check for pure strategy Nash equilibria
for i in range(2):
   for j in range(2):
       if (
               payoff_matrix_player_a[i, j] >= payoff_matrix_player_a[1 - i, j]
               and payoff_matrix_player_b[i, j] >= payoff_matrix_player_b[i, 1 - j]
       ):
           equilibria.append((i, j))

print("Nash equilibria:")

for eq in equilibria:

   strategy={0:"Deer",1:"Rabbit"}
   p1, p2 = eq
   p1_payoff = payoff_matrix_player_a[p1, p2]
   p2_payoff = payoff_matrix_player_b[p1, p2]
   print("Hunter 1 strategy:", strategy[p1])
   print("Hunter 2 strategy:", strategy[p2])
   print("Hunter 1 payoff:", p1_payoff)
   print("Hunter 2 payoff:", p2_payoff)
   print("\n")
```

Output:



```
/usr/bin/python3.10 /home/dreamfist/data/GT/Game-t
Nash equilibria:
Hunter 1 strategy: Deer
Hunter 2 strategy: Deer
Hunter 1 payoff: 2
Hunter 2 payoff: 2


Hunter 1 strategy: Rabbit
Hunter 2 strategy: Rabbit
Hunter 1 payoff: 1
Hunter 2 payoff: 1
```

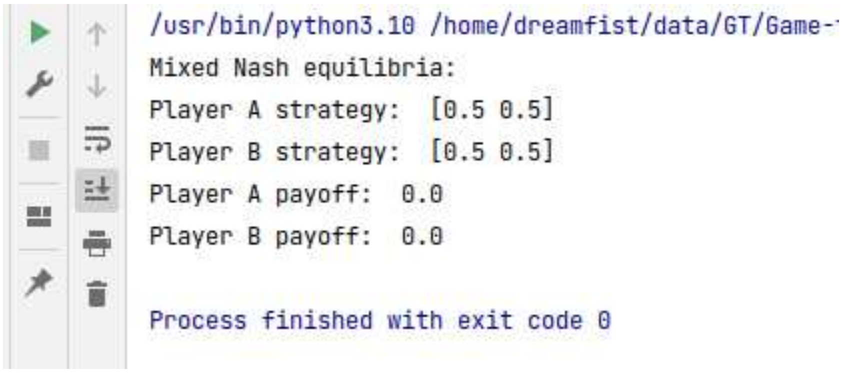
8.Matching Pennies: Two payers, each having a penny,

are asked to choose from among two strategies – heads (H) and tails (T). The row player wins if the two pennies match, while the column player wins if they do not. Find Nash Equilibrium nashpy, without using nashpy.

With nashpy:

```python
import numpy as np
import nashpy as nash

# Define the payoff matrix for Player A
payoff_matrix_player_a = np.array([[1, -1], [-1, 1]])
# Define the payoff matrix for Player B
payoff_matrix_player_b = np.array([[-1, 1], [1, -1]])
# Create the game
game = nash.Game(payoff_matrix_player_a, payoff_matrix_player_b)
# Find Nash equilibria using support_enumeration
equilibria = list(game.support_enumeration())

print("Mixed Nash equilibria:")
for eq in equilibria:
   p1 = eq[0]
   p2 = eq[1]

   p1_payoff = game[eq][0]
   p2_payoff = game[eq][1]

   print("Player A strategy: ", p1)
   print("Player B strategy: ", p2)
   print("Player A payoff: ", p1_payoff)
   print("Player B payoff: ", p2_payoff)
```

Output:



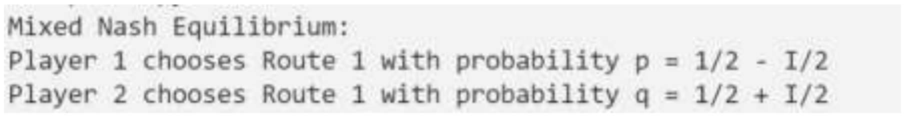
```
/usr/bin/python3.10 /home/dreamfist/data/GT/Game-
Mixed Nash equilibria:
Player A strategy:  [0.5 0.5]
Player B strategy:  [0.5 0.5]
Player A payoff:  0.0
Player B payoff:  0.0

Process finished with exit code 0
```

9. Routing congestion game: player 1 is interested in getting good service, hence would like the others to choose different routes, while the player 2 is interested only in disrupting the player 1’s service by trying to choose the same route. Find Nash Equilibrium nashpy, without using nashpy

CODE:

```python
from sympy import symbols, Eq, solve

# Define the symbols for the probabilities of Player 1 and Player 2
p, q = symbols('p q')

# Define the expected utility functions for both players
U1 = p * q
U2 = 1 - p * q

# Find the Nash Equilibrium by setting the expected utilities equal
equation = Eq(U1, U2)

# Solve for p and q
nash_equilibrium = solve((equation, p + q - 1), (p, q))

# Display the Nash Equilibrium
p_value, q_value = nash_equilibrium[0]
print("Mixed Nash Equilibrium:")
print(f"Player 1 chooses Route 1 with probability p = {p_value}")
print(f"Player 2 chooses Route 1 with probability q = {q_value}")
```

OUTPUT :

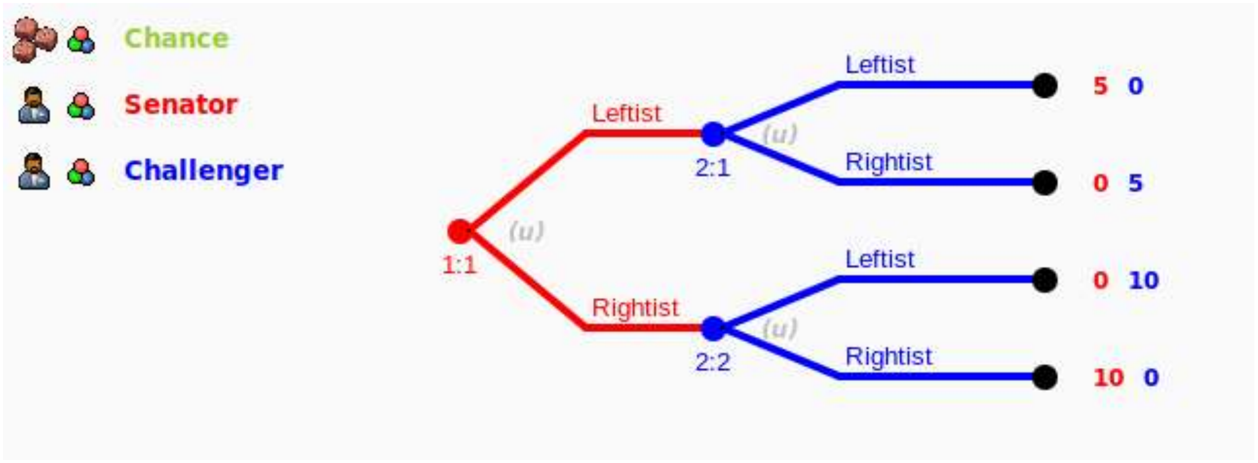


```
Mixed Nash Equilibrium:
Player 1 chooses Route 1 with probability p = 1/2 - I/2
Player 2 chooses Route 1 with probability q = 1/2 + I/2
```

10. SENATE RACE: An incumbent senator (from a rightist party) runs against a challenger (from a leftist party). They first choose a political platform, leftist or rightist, where the senator has to move first. If both choose the same platform, the incumbent wins, otherwise the challenger wins. Assume that the value of winning is 10, the value of compromising their political views (by choosing a platform not consistent with them) is 5, and the payoff is the sum of these values.

Use Gambit to illustrate Game tree.

Tree:



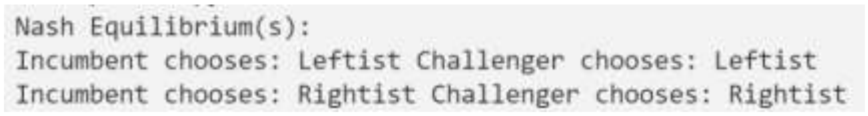
11. Implement the SENATE RACE and find out Nash equilibrium.

CODE :

```
# Define the Senate Race game with the correct payoff matrix dimensions
A = [[(10, 0), (0, 10)],  # Payoffs for Incumbent (Leftist, Rightist)
     [(0, 10), (10, 0)]]  # Payoffs for Challenger (Leftist, Rightist)
```

```
# Find the Nash Equilibrium by considering dominant strategies
def find_nash_equilibrium(payoff_matrix):
    equilibrium = []
    for i in range(2):
        for j in range(2):
            if payoff_matrix[i][j] == max(payoff_matrix[i]):
                equilibrium.append((i, j))
    return equilibrium

nash_equilibria = find_nash_equilibrium(A)

# Print the Nash Equilibrium (if it exists)
if nash_equilibria:
    print("Nash Equilibrium(s):")
    for eq in nash_equilibria:
        incumbent_choice = "Leftist" if eq[0] == 0 else "Rightist"
        challenger_choice = "Leftist" if eq[1] == 0 else "Rightist"
        print("Incumbent chooses:", incumbent_choice, "Challenger chooses:",
challenger_choice)
else:
    print("No Nash Equilibrium found for this game.")
```

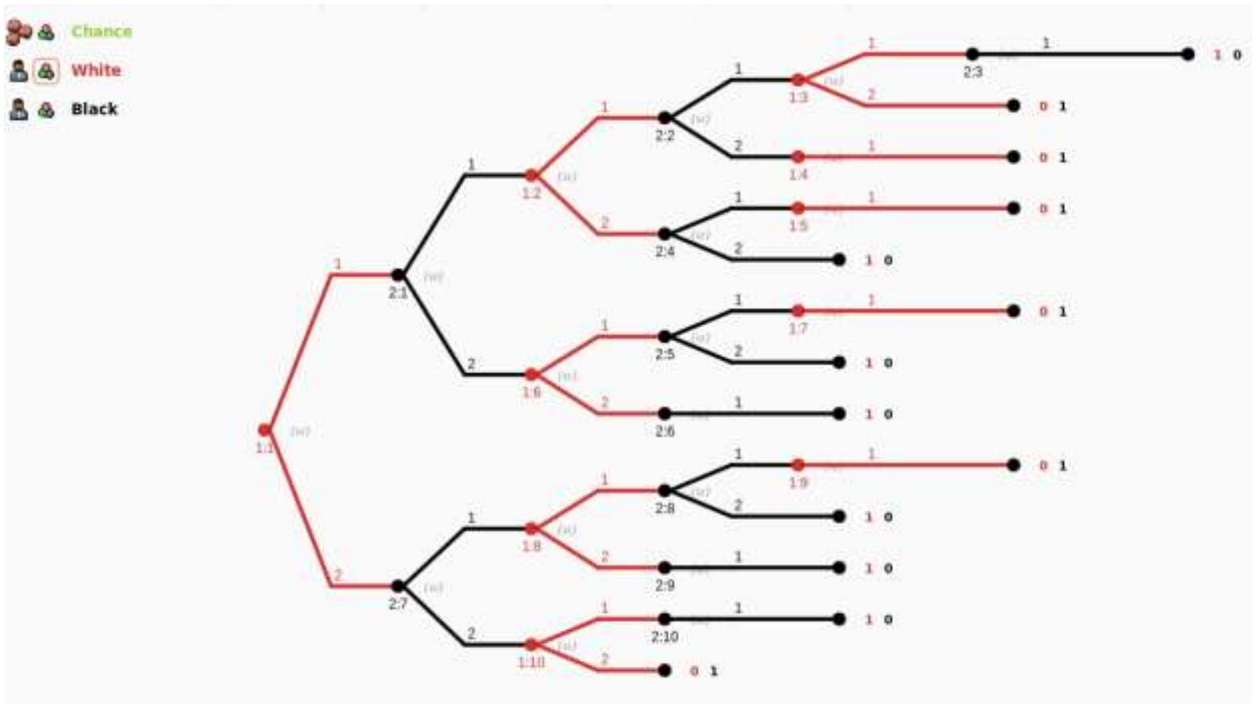
OUTPUT :



```
Nash Equilibrium(s):
Incumbent chooses: Leftist Challenger chooses: Leftist
Incumbent chooses: Rightist Challenger chooses: Rightist
```

12. BACKWARD INDUCTION: Six stones lie on the board. Black and White alternate to remove either one or two stones from the board, beginning with White. Whoever first faces an empty board when having to move loses. The winner gets $1, the loser loses $1. What are the best strategies for the players? Use Gambit to illustrate Game tree.

Tree:

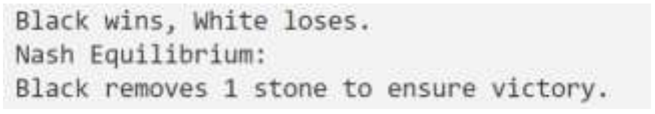


13. Implement the BACKWARD INDUCTION and find out Nash equilibrium.

CODE:

```
# Define the stone removal game
class StoneRemovalGame:
    def __init__(self, stones):
        self.stones = stones

    def play(self, player, stones_left):
        if stones_left == 0:
            return -1  # The player who makes the last move loses.

        if player == "White":
            for move in [1, 2]:
                if stones_left - move >= 0:
                    result = self.play("Black", stones_left - move)
                    if result == -1:
                        return 1  # White wins
            return -1  # White loses if no winning move is found
        else:  # player == "Black"
            for move in [1, 2]:
                if stones_left - move >= 0:
                    result = self.play("White", stones_left - move)
                    if result == -1:
                        return 1  # Black wins
            return -1  # Black loses if no winning move is found

# Create a Stone Removal Game with 6 stones
game = StoneRemovalGame(stones=6)

# Start the game with White's turn
result = game.play("White", game.stones)

# Determine the winner and Nash equilibrium
if result == 1:
    winner = "White"
    loser = "Black"
else:
    winner = "Black"
    loser = "White"

print(f"{winner} wins, {loser} loses.")

# Print Nash Equilibrium (there is only one)
print("Nash Equilibrium:")
print(f"{winner} removes 1 stone to ensure victory.")
```

OUTPUT :

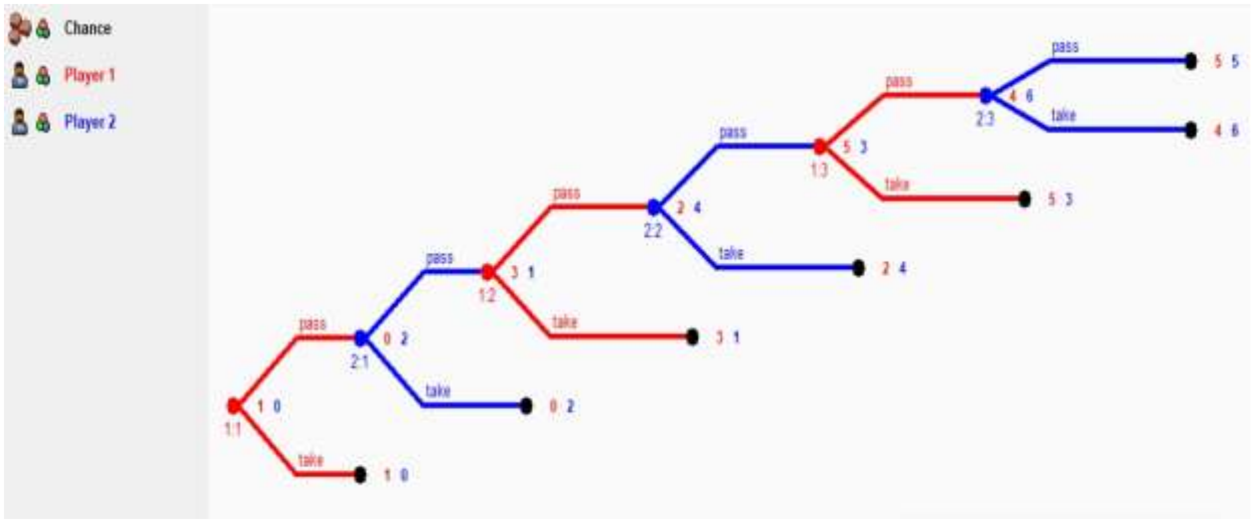


```
Black wins, White loses.
Nash Equilibrium:
Black removes 1 stone to ensure victory.
```

14. CENTIPEDE GAME In this game, two players alternately face two stacks of money. To make a move, a player has the choice either to pass, in which case both stacks grow slightly and the other player now must make a move facing slightly larger stacks, or to take the larger stack, in which case the other player gets the smaller one and the game ends. If it didn't end

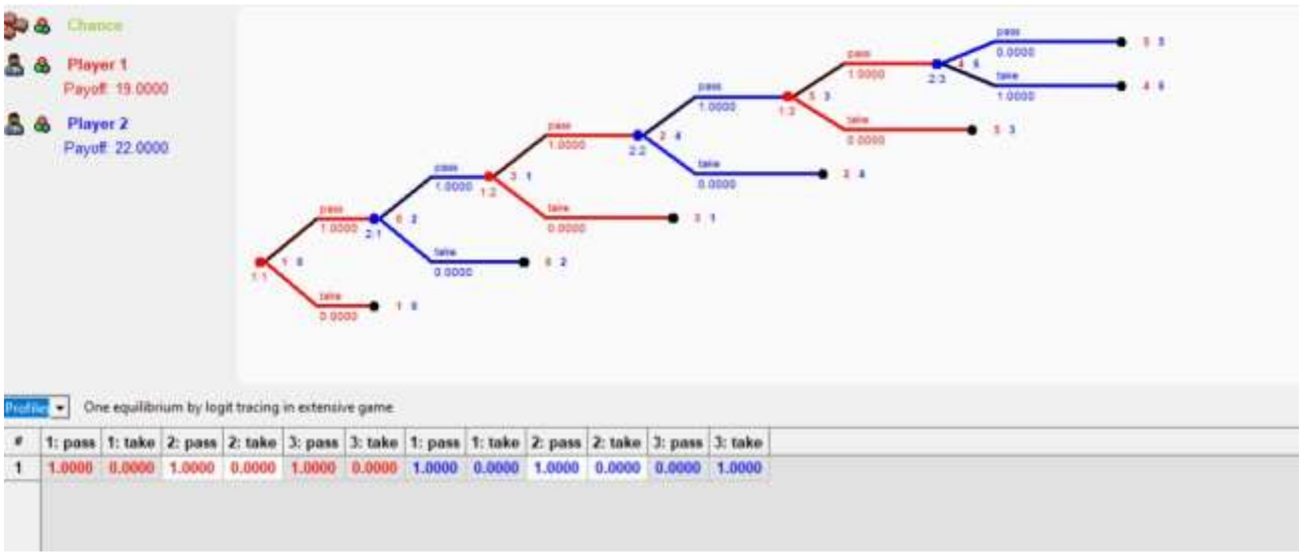
before, the game ends after a fixed number of rounds, in which case both players share the accumulated money evenly. Use Gambit to illustrate Game tree.

Code:



15. Implement the CENTIPEDE GAME and find out Nash equilibrium.

Code:



Profile One equilibrium by logit tracing in extensive game

#	1: pass	1: take	2: pass	2: take	3: pass	3: take	1: pass	1: take	2: pass	2: take	3: pass	3: take
1	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	0.0000	1.0000