NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

GAME THEORY

PROF: MR. RUDRESH DWIVEDI

PRACTICAL- END SEMESTER

SNEHA GUPTA

2021UCA1859

INDEX

0.1 Practical 1:

Intro to Gambit's GUI

Gambit is an open-source collection of tools for doing computation in game theory. With Gambit, you can build, analyse, and explore game models. Use Gambit's graphical interface to get intuition about simple games, or the command-line tools and Python extension to support your world-class research and practical applications. Gambit is cross-platform: Get it for Microsoft Windows, Mac OS X, or Linux.

0.1.1 What is Gambit?

Gambit is a set of software tools for doing computation on finite, noncooperative games. These comprise a graphical interface for interactively building and analysing general games in extensive or strategy form; a number of command-line tools for computing Nash equilibria and other solution concepts in games; and, a set of file formats for storing and communicating games to external tools.

0.1.2 Key Features Of Gambit:

Gambit has a number of features useful both for the researcher and the instructor i.e.

1. Inter□active, cross-platform graphical interface. All Gambit features are available through the use of a graphical interface, which runs under multiple operating systems: Windows, various flavours of Un*x (including Linux), and Mac OS X. The interface offers flexible methods for creating extensive and strategic games. The interface is useful for the advanced researcher, but is intended to be accessible for students taking a first course in game theory as well.

2. Command-line tools for computing equilibria. More advanced applications often require extensive computing time and/or the ability to script computations. All algorithms in Gambit are packaged as individual, command□line programs, whose operation and output are configurable.

3. Extensibility and interoperability. The Gambit tools read and write file formats which are textual and documented, making them portable across systems and able to interact with external tools. It is therefore straightforward to extend the capabilities of Gambit by, for example, implementing a new method for computing equilibria, reimplementing an existing one more efficiently, or creating tools to programmatically create, manipulate, and transform games, or for econometric analysis on games.

0.1.3 Limitations Of Gambit

Gambit has a few limitations that may be important in some applications. We outline them here1. G. Gambit is for finite games only. Because of the mathematical structure of finite games, it is possible to write many general- purpose routines for analysing these games. Thus, Gambit

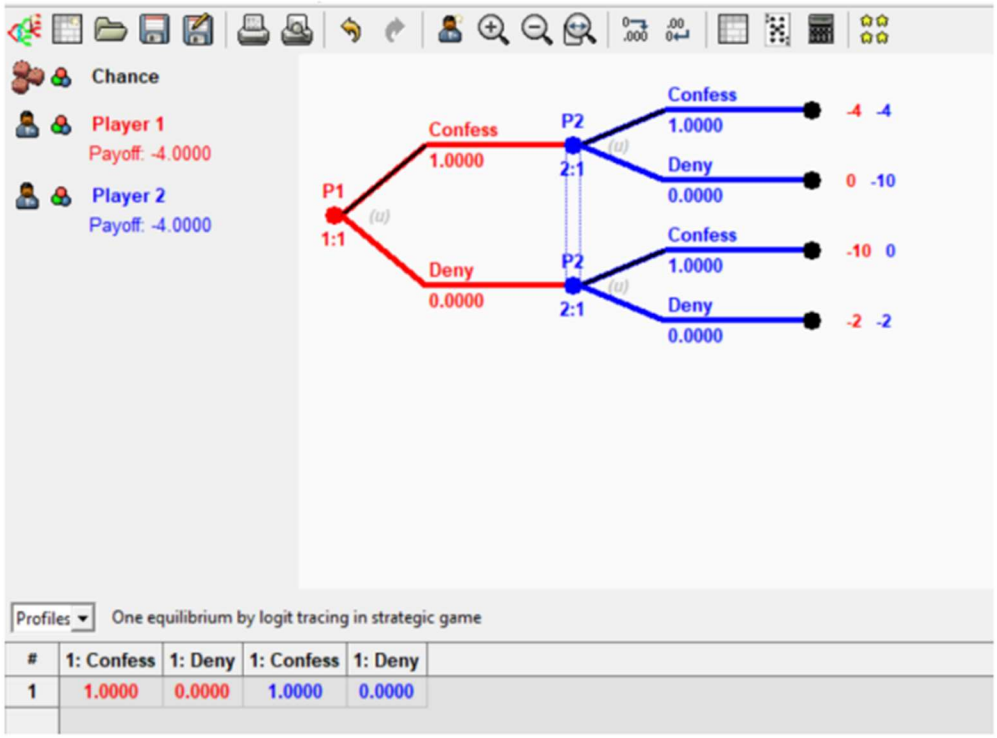
can be used in a wide variety of applications of game theory. However, games that are not finite, that is, games in which players may choose from a continuum of actions, or in which players may have a continuum of types, do not admit the same general-purpose 2. Gambit is for noncooperative game theory only. Gambit focuses on the branch of game theory in which the rules of the game are written down explicitly, and in which players choose their actions independently. Gambit's analytical tools centre primarily around Nash equilibrium, and related concepts of bounded rationality such as quantal response equilibrium. Gambit does not at this time provide any representations of, or methods for, analysing games written in cooperative form. (It should be noted that some problems in cooperative game theory do not suffer from the computational complexity that the Nash equilibrium problem does, and thus cooperative concepts could be an interesting future direction of development.) of development.)

0.2 Practical 2:

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

```
[3]: from IPython.display import Image
     Image('Prisoner dilemma.png')
```

[3]:



0.3 Practical: 3

Evaluate Nash equilibrium for Prisoner's Dilemma using nashpy

```
import numpy as np
import nashpy as nash

P1 = np.array([[-3, 0], [-4, -1]])
P2 = np.array([[-3, -4], [0, -1]])
```

```
game1 = nash.Game(P1, P2)
game1
```

```
Bi matrix game with payoff matrices:

Row player:
[[-3  0]
 [-4 -1]]

Column player:
[[-3 -4]
 [ 0 -1]]
```

```
eq = game1.support_enumeration()
```

```
for ans in eq:
    print(ans)
```

```
(array([1., 0.]), array([1., 0.]))
```

Which means player one chooses strategy 1 over 2 i.e. Confess always and player rwo chooses startegy 1 over 2 i.e. Confess always.

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

```
P1 = np.array([[-3, 0], [-4, -1]])
P2 = np.array([[-3, -4], [0, -1]])
```

```
def findNE(payoff1, payoff2):
    strategyCount = P1.shape[0]
    eql = []

    for A in range(strategyCount):
        for B in range(strategyCount):
            currA = payoff1[A][B]
            currB = payoff2[A][B]

            isNE = True
            for a in range(strategyCount):
                for b in range(strategyCount):
```

```
                    if(currA<payoff1[a][B] or currB<payoff2[A][b]): # it is a
↪way to check: if other player chooses some strategy, is there an advantage
↪to move away from this state
                        isNE = False
                        break
            if isNE:
                eql.append((A, B))
    return eql
```

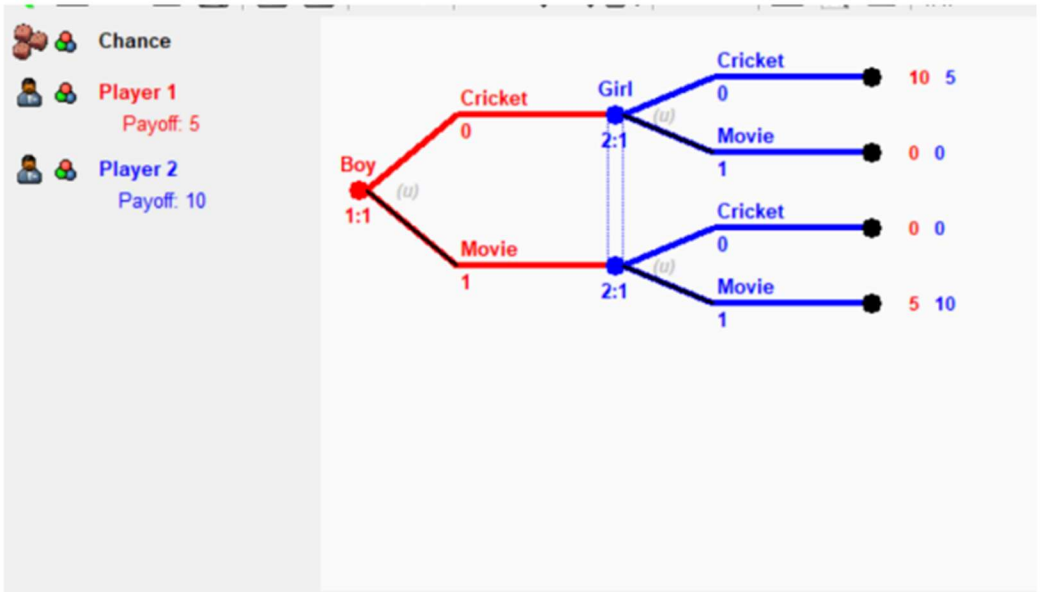
```
findNE(P1, P2)
```

```
[(0, 0)]
```

0.4 Practical: 4

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

```
from IPython.display import Image
Image("Battle of Sexes.png")
```



Profiles All equilibria by enumeration of mixed strategies in strategic game

#	1: Cricket	1: Movie	1: Cricket	1: Movie
1	1	0	1	0
2	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{2}{3}$
3	0	1	0	1

0.5 Practical: 5

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

```
P1 = np.array([[10, 0], [0, 5]])
P2 = np.array([[5, 0], [0, 10]])
```

```
game2 = nash.Game(P1, P2)
game2
```

```
Bi matrix game with payoff matrices:

Row player:
[[10  0]
 [ 0  5]]

Column player:
[[ 5  0]
 [ 0 10]]
```

```
eql = game2.support_enumeration()
for ans in eql:
    print(ans)
```

```
(array([1., 0.]), array([1., 0.]))
(array([0., 1.]), array([0., 1.]))
(array([0.66666667, 0.33333333]), array([0.33333333, 0.66666667]))
```

Without nashpy

```
def findNE(payoff1, payoff2):
    strategyCount = P1.shape[0]
    eql = []

    for A in range(strategyCount):
        for B in range(strategyCount):
            currA = payoff1[A][B]
            currB = payoff2[A][B]

            isNE = True
            for a in range(strategyCount):
                for b in range(strategyCount):
                    if(currA<payoff1[a][B] or currB<payoff2[A][b]): # it is a␣
 ↪way to check: if other player chooses some strategy, is there an advantage␣
 ↪to move away from this state
                        isNE = False
                        break
```

```
            if isNE:
                eql.append((A, B))
    return eql
```

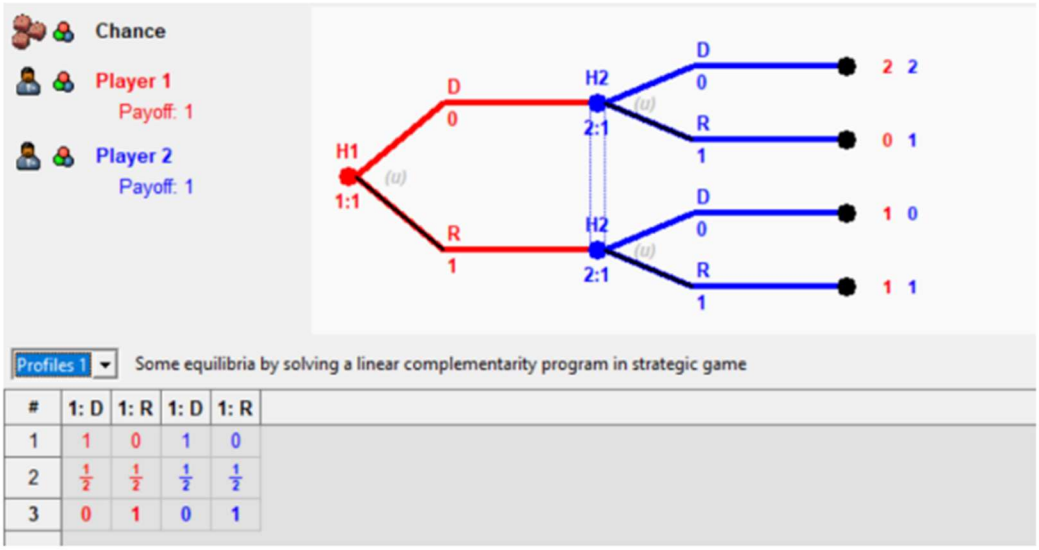
```
findNE(P1, P2)
```

```
[(0, 0), (1, 1)]
```

0.6 Practical: 6

Construct the Game tree for the Hunting Game in Gambit

```
from IPython.display import Image
Image("I am a hunter.png")
```



#	1: D	1: R	1: D	1: R
1	1	0	1	0
2	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
3	0	1	0	1

0.7 Practical: 7

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

```
P1 = np.array([[2, 0], [1, 1]])
P2 = np.array([[2, 1], [0, 1]])
```

```
game3 = nash.Game(P1,P2)
game3
```

```
Bi matrix game with payoff matrices:

Row player:
[[2 0]
 [1 1]]

Column player:
[[2 1]
 [0 1]]
```

```
eql = game3.support_enumeration()
for ans in eql:
    print(ans)
```

```
(array([1., 0.]), array([1., 0.]))
(array([0., 1.]), array([0., 1.]))
(array([0.5, 0.5]), array([0.5, 0.5]))
```

without nashpy

```
def findNE(payoff1, payoff2):
    strategyCount = P1.shape[0]
    eql = []

    for A in range(strategyCount):
        for B in range(strategyCount):
            currA = payoff1[A][B]
            currB = payoff2[A][B]

            isNE = True
            for a in range(strategyCount):
                for b in range(strategyCount):
                    if(currA<payoff1[a][B] or currB<payoff2[A][b]): # it is a␣
 ↪way to check: if other player chooses some strategy, is there an advantage␣
 ↪to move away from this state
                        isNE = False
                        break
            if isNE:
                eql.append((A, B))
    return eql
```

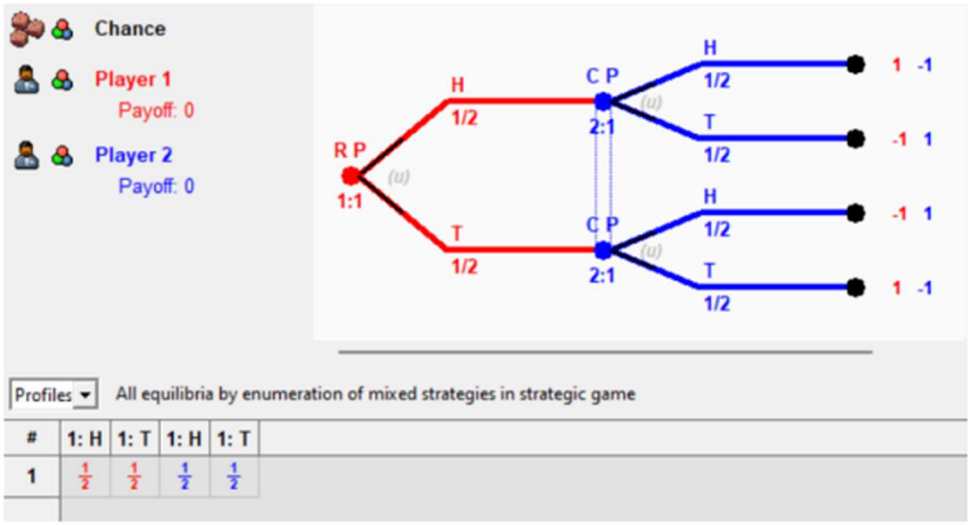
```
findNE(P1, P2)
```

```
[(0, 0), (1, 1)]
```

0.8 Practical 8:

Matching Pennies: Two players, 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.

```
Image('Pennies game.png')
```



With nashpy

```
P1 = np.array([[1, -1], [-1, 1]])
P2 = np.array([[-1, 1], [1, -1]])
```

```
game4 = nash.Game(P1, P2)
game4
```

```
Zero sum game with payoff matrices:

Row player:
[[ 1 -1]
 [-1  1]]

Column player:
[[-1  1]
 [ 1 -1]]
```

```
eql = game4.support_enumeration()
for ans in eql:
    print(ans)
```

```
(array([0.5, 0.5]), array([0.5, 0.5]))
```

Wihtout nashpy

```
def findNE(payoff1, payoff2):
    strategyCount = P1.shape[0]
    eql = []

    for A in range(strategyCount):
        for B in range(strategyCount):
            currA = payoff1[A][B]
            currB = payoff2[A][B]

            isNE = True
            for a in range(strategyCount):
                for b in range(strategyCount):
                    if(currA<payoff1[a][B] or currB<payoff2[A][b]): # it is a
 ↪way to check: if other player chooses some strategy, is there an advantage
 ↪to move away from this state
                        isNE = False
                        break
            if isNE:
                eql.append((A, B))
    return eql
```

```
findNE(P1, P2)
```

```
[]
```

Mixed strategy

```
def best_response(player_payoff_matrix, opponent_strategy):
    return np.argmax(np.dot(player_payoff_matrix, opponent_strategy))

player1_strategy = np.array([0.5, 0.5]) # Player 1's mixed strategy
player2_best_response = best_response(P2, player1_strategy)

player2_strategy = np.array([0.5, 0.5]) # Player 2's mixed strategy
player1_best_response = best_response(P1, player2_strategy)

if player2_best_response == 0 and player1_best_response == 0: # equillibrium
 ↪condition
 print("Nash Equilibrium:")
 print("Player 1's strategy:", player1_strategy)
 print("Player 2's strategy:", player2_strategy)
else:
```

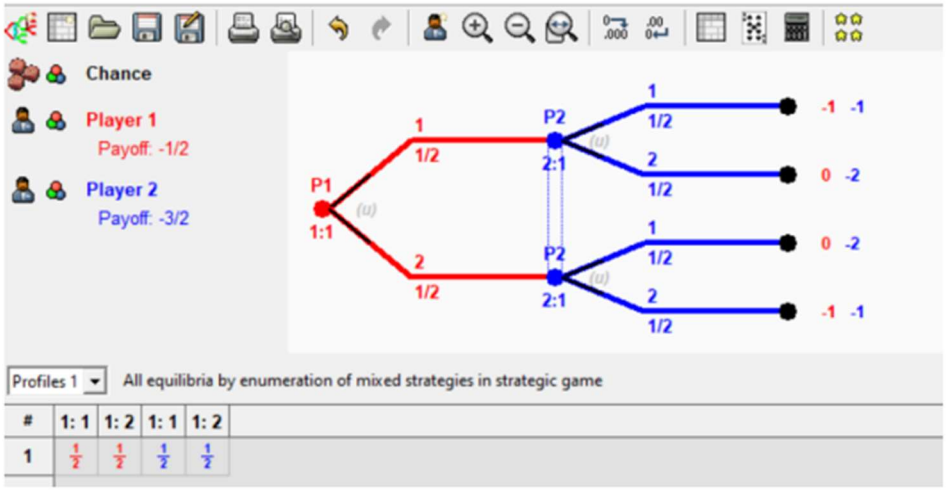
```
 print("No Nash Equilibrium found.")
```

```
Nash Equilibrium:
Player 1's strategy: [0.5 0.5]
Player 2's strategy: [0.5 0.5]
```

0.9 Practical 9:

Routing congestion game: player 1 is interested in getting good service, hence would like the others to choose different routes, whil player 2 is interested only in disrupting player I's service by trying o choose the same route. Find Nash Equilibrium nashpy, without us ng na.shpy

```
Image('Routing game.png')
```



With nashpy

```
P1 = np.array([[-1, 0], [0, -1]])
P2 = np.array([[-1, -2], [-2, -1]])
```

```
game5 = nash.Game(P1, P2)
game5
```

```
Bi matrix game with payoff matrices:

Row player:
[[-1  0]
 [ 0 -1]]

Column player:

 [[-1 -2]
  [-2 -1]]
```

```
eql = game5.support_enumeration()
for ans in eql:
    print(ans)
```

```
(array([0.5, 0.5]), array([0.5, 0.5]))
```

Without nashpy

```
def findNE(payoff1, payoff2):
    strategyCount = P1.shape[0]
    eql = []

    for A in range(strategyCount):
        for B in range(strategyCount):
            currA = payoff1[A][B]
            currB = payoff2[A][B]

            isNE = True
            for a in range(strategyCount):
                for b in range(strategyCount):
                    if(currA<payoff1[a][B] or currB<payoff2[A][b]): # it is a␣
 ↪way to check: if other player chooses some strategy, is there an advantage␣
 ↪to move away from this state
                        isNE = False
                        break
            if isNE:
                eql.append((A, B))
    return eql
```

```
findNE(P1, P2)
```

```
[]
```

```
def best_response(player_payoff_matrix, opponent_strategy):
    return np.argmax(np.dot(player_payoff_matrix, opponent_strategy))

player1_strategy = np.array([0.5, 0.5]) # Player 1's mixed strategy
player2_best_response = best_response(P2, player1_strategy)

player2_strategy = np.array([0.5, 0.5]) # Player 2's mixed strategy
player1_best_response = best_response(P1, player2_strategy)

if player2_best_response == 0 and player1_best_response == 0: # equillibrium␣
 ↪condition
 print("Nash Equilibrium:")
```

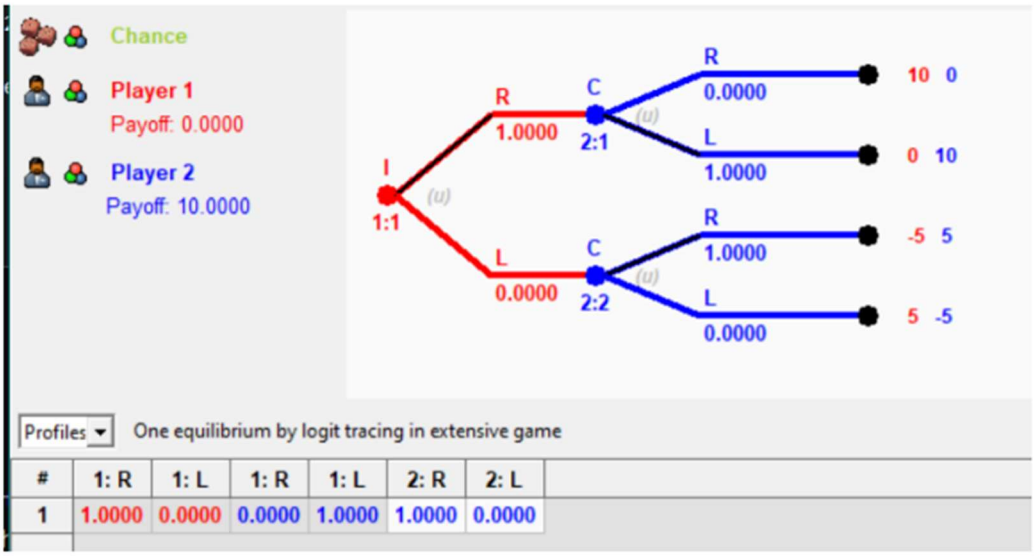
```
 print("Player 1's strategy:", player1_strategy)
 print("Player 2's strategy:", player2_strategy)
else:
 print("No Nash Equilibrium found.")
```

```
Nash Equilibrium:
Player 1's strategy: [0.5 0.5]
Player 2's strategy: [0.5 0.5]
```

0.10 Practical 10 and 11:

ZSENATE RACE: An incumbent senator (from a rightist party) runs against a challenger (from a leftist party). They first choose a political platform, leftist o rightist, where the senator has to move first. If both choose the same platform, t e 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 Gamb t to illustrgamea Game tree.

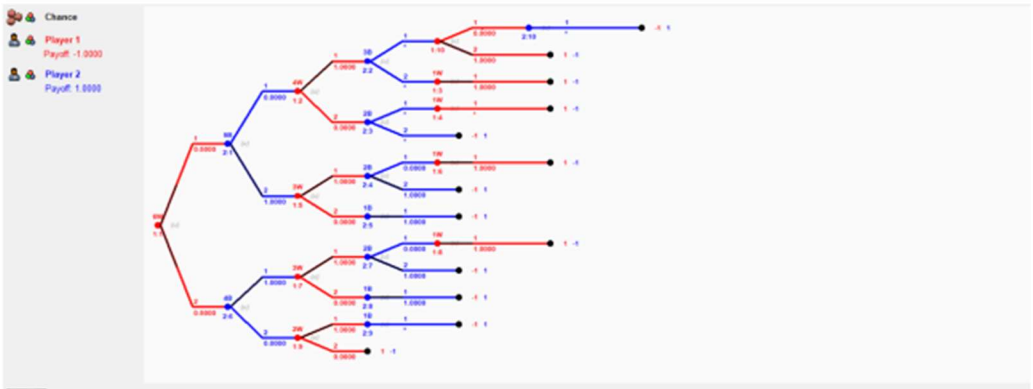
```
Image("senate race.png")
```



0.11 Practical 12 & 13:

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 havi g to move loses. The winner ges $1, the loser loss $1. What are he best strategies for the players? Use Gambit to illustratgameGame tree.

```
Image('racism.png')
```



Profile One equilibrium by simplicial subdivision in strategic game

#	1: 1	1: 2	2: 1	2: 2	3: 1	4: 1	5: 1	5: 2	6: 1	7: 1	7: 2	8: 1	9: 1	9: 2	10: 1	10: 2	1: 1	1: 2	2: 1	2: 2	3: 1	3: 2	4: 1	4: 2	5: 1	6: 1	6: 2	7: 1	7: 2	8: 1	9: 1	10: 1
1	0.5000	0.5000	1.0000	0.0000	1.0000	*	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	0.0000	1.0000	0.0000	1.0000	*	*	*	*	0.0000	1.0000	1.0000	1.0000	0.0000	0.0000	1.0000	1.0000	*	*