Game Theory: Homework #2

Farrukh Baratov (14636816) and Satchit Chatterji (14428237)

Collaboration. The formulation of the theorem in Exercise 2 was discussed with Fátima González-Novo López, but was not included here since only 3 exercises were to be solved. Farrukh came up with the concrete examples in Exercise 1 and 3, and Satchit wrote the concrete code for Exercise 4. We both discussed, verified and corrected each other's work and the code was tested to our best efforts.

Exercise 1

	L	R
T	3, 2	3, 2
B	1, 1	2, 1

Consider the normal form game above. B is a *strictly* dominated strategy for Rowena, and L is a *weakly* dominated strategy for Colin.

If we start elimination with Rowena, we first eliminate B, leaving only the top row. The remaining strategies are equivalent for both Colin and Rowena, and no more elimination occurs. This leaves the strategies (T, L) and (T, R).

If we start elimination with Colin, L is eliminated first, leaving only R. Since T is strictly dominant for Rowena, B is eliminated. This results in the strategy (T, R).

This example shows that changing the order of weakly dominated strategy elimination also changed the final outcome. Therefore, iterated elimination of weakly dominated strategies is **not** order-independent.

Exercise 3

	L	R
T	1, 1	0, 5
B	5, 0	2, 2

Consider the above example of a normal form game. We can see that, based on the payouts for each action, Colin's dominant strategy is playing L, while Rowena's is T. Therefore, (T, L) is a pure NE, as both players will choose the action with the guaranteed higher payoff. The sum of expected utilities is 2. Note that there are no truly mixed Nash equilibria. Thus, (T, L) is the only NE here.

Suppose the players coordinate, resulting in a correlated equilibrium that allows only for BL and TR to occur. In this case, the expected utility for each player is as follows:

$$\frac{5+0}{2} = 2.5$$

Therefore, the sum of expected utilities is as follows:

$$2.5 + 2.5 = 5$$

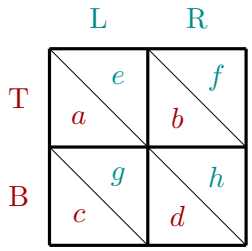
This results in both a higher expected utility and a higher sum of expected utilities than the pure NE allows. Therefore, this correlated equilibrium results in the best expected utility for both players.

Exercise 4

The code for this exercise was written in Python 3.10.7 and only requires NumPy as a dependency. The code can be found in the appendix and in the separate `solve_nash.py` file submitted alongside this document.

Representation

The utility matrix is represented as a single string, with integer payoff values delimited by commas: `a,b,c,d,e,f,g,h` . The first four values are the payoffs for row player, and the next four are the payoffs for the column player. The values in this string should correspond to the following payoff matrix:



Thus, the code should be run by calling in the terminal:

```
python solve_nash.py a,b,c,d,e,f,g,h
```

For example, running the code on the example question 2 (a) from Homework 1 gives us the following output:

```
$ python solve_nash.py 5,6,2,9,9,3,2,5
Payoffs: 5,6,2,9,9,3,2,5
 Table form:
                | 5\9 | 6\3 |
                |-----|-----|
                | 2\2 | 9\5 |

Pure Nash equilibium : [(0, 0), (1, 1)]
Mixed Nash equilibium: (0.3333333333333333, 0.5)
```

The pure Nash equilibrium are coordinates for the matrix (in this case the cells at the top left and the bottom right) and the mixed NE are the probability values of the players for respectively playing their first action (i.e. top and left respectively for the row and column player). For this game, the output matches the expected result when computing NE by hand. The program was verified in this way for several known games successfully.

Finding Pure Nash Equilibria

The function `find_pure_ne` computes all the pure Nash equilibria in the input 2x2 game. It does this by following the definition of a pure NE. It finds these by iterating through each cell in the matrix and verifying whether the payoff for the row player in that cell is *at least* as high as the payoff in the cells above or below it. This is equivalent to checking whether the row player has an incentive to change their play, given the column player has played their action. Similarly, the column player's payoff is compared to the payoffs to the right or left of it for the same reason. If this cell's payoff provides a utility that is at least as high as it's respective neighbouring cells' utilities for each player, then it is considered a Nash equilibrium. Since this is a small game (2x2) iterating through all possibilities is trivial – however, for a larger game (more players or actions), the space of possibilities grows exponentially. Thus, in this case, better heuristics or strategies such as IESDS may be implemented to assist in reducing computation time.

Finding Mixed Nash Equilibria

The function `find_mixed_ne` computes the finite mixed Nash equilibria in the input 2x2 game if it exists. Since we are looking for a Nash equilibrium, the column player must be indifferent to the action he takes: i.e. his utilities for choosing L and R should be the same, regardless of what the row player plays. If the probability of choosing L is p, this can be computed directly as:

$$e \cdot p + g \cdot (1 - p) = f \cdot p + h \cdot (1 - p)$$
$$p = \frac{h - g}{e - f - g + h}$$

Similar reasoning may be conducted for the row player. If the probability of them choosing T if q, it may be computed as:

$$a \cdot q + b \cdot (1 - q) = c \cdot q + d \cdot (1 - q)$$
$$q = \frac{d - b}{a - c - b + d}$$

However, this direct computation has a few caveats, since there is no restriction on the payoffs in the 2x2 matrix. Firstly, the denominators in both of these fractions should not be zero. In this case, no mixed NE exist. Secondly, if the resulting values of p or q are greater than 1 or less than 0, this cannot be a probability, and thus no mixed NE exist. Finally, if $p \in [0, 1]$ and $q \in [0, 1]$, and we compute and output it as such.

In this final case, it may not be the case that there only exists one Nash equilibria, but infinitely many, e.g. due to the fact that one player may extract the same utility by playing their actions within a continuous range of probabilities (e.g. in the case of equal strictly dominating strategies). This is not taken into account in the submitted program. However, it may be generalised in a number of ways. One way is to compute the mixed NE, and then simulate playing for values $p \pm \epsilon$ and $q \pm \epsilon$ for some small value(s) ϵ. This simulates human reasoning (*"What if Colin plays L more often than the critical value. Does Rowina's strategy change?"*). However, depending on the choice of ϵ, there may be a tradeoff between not finding the bounds of the infinite mixed strategies (since ϵ is necessarily rational in the case of computers), and computation time.

Other ways that may be possible to implement were found in papers such as Avis *et al.* (2010) (polyhedral vertex enumeration), Widger & Grosu (2008) (parallel support enumeration) or Lemke & Howson, (1964) (the Lemke-Howson algorithm). However, these were not explored further during this assignment.

References

Avis, D., Rosenberg, G. D., Savani, R., & Von Stengel, B. (2010). *Enumeration of Nash Equilibria for Two-player Games.* Economic Theory, 42, 9-37.

Widger, J., & Grosu, D. (2008, July). *Computing Equilibria in Bimatrix Games by Parallel Support Enumeration.* In 2008 International Symposium on Parallel and Distributed Computing (pp. 250-256). IEEE.

Lemke, C. E., & Howson, Jr, J. T. (1964). *Equilibrium Points of Bimatrix Games.* Journal of the Society for Industrial and Applied Mathematics, 12(2), 413-423.

Appendix: Code for `solve_nash.py`

The source code is also submitted alongside this document.

```
import numpy as np
import argparse

def find_pure_ne(payoffs):
  """ Finds pure Nash equilibria for a 2 player, 2 action
    game by iteratively verifying that each cell is/isn't
    dominant for each player in the respective situation.
    For each cell, the row player's payoff must be greater
    or equal to the cell below/above it, and the column
    player's payoff should be greater or equal to the cells
    to the left/right of itself.
  """

  possible_nes = [(0,0), (0,1), (1,0), (1,1)]
  found_nes = []

  # reshape the payoff matrix
  # new shape of payoff matrix: rows, columns, payoffs
  new_payoffs = np.empty((payoffs[0].size + payoffs[1].size,))
  new_payoffs[0::2] = payoffs[0]
  new_payoffs[1::2] = payoffs[1]
  payoffs = new_payoffs.reshape(-1,2,2)

  # loop over all cells
  for p_ne in possible_nes:
    p1, p2 = payoffs[p_ne]
    # get cells to compare the current cell to
    p1_comp = payoffs[:,p_ne[1],0]
    p2_comp = payoffs[p_ne[0],:,1]
    # verify the current cell is dominant
    if p1 == np.max(p1_comp) and p2 == np.max(p2_comp):
      found_nes.append(p_ne)

  return found_nes
```

```
def get_ne_outcome(num, den):
  """ Checks mixed NE computation for bad probability form,
    i.e. den < 0, and 0<=num/den<=1.
  """
  if den == 0: # no mixed strategy exists
    return None

  if 0 <= num/den <= 1: # mixed strategy exists
    return abs(num/den) # abs for the py repr of '-0'

  else: # no mixed strategy exists
    return None

def find_mixed_ne(payoffs):
  """ Finds mixed Nash equilibria from payoff
    matrix by solving for each player's
    probability of play, assuming what the other
    player does makes them indifferent to what
    they play.
  """
  a,b,c,d = payoffs[0]
  e,f,g,h = payoffs[1]

  num_p = h-g
  den_p = e-f-g+h
  p = get_ne_outcome(num_p, den_p)

  num_q = d-b
  den_q = a-c-b+d
  q = get_ne_outcome(num_q, den_q)

  if p is None or q is None:
    return None, None

  return p, q

def parse_game(inp_str):
  """ Parses a string of ints such as a,b,c,d,e,f,g,h
    to a matrix of payoffs [[a b c d] [e f g h]]
  """
  payoffs = [int(x) for x in inp_str.split(",")]
  payoffs = np.array(payoffs).reshape(2,4)

  return payoffs

def print_table(test_str):
  a,b,c,d,e,f,g,h = test_str.split(",")
  table = f""" Table form:
```

```
    | {a}\\{e} | {b}\\{f} |
    |-----|-----|
    | {c}\\{g} | {d}\\{h} |
  """
  print(table)

def compute_all(test_str):
  """ Computes and prints finitely many Nash equilibria
    for a 2 player, 2 action game.
  """
  print("Payoffs:", test_str)
  print_table(test_str)
  parsed = parse_game(test_str)
  pure = find_pure_ne(parsed)
  print("Pure Nash equilibium :", pure)
  mixed = find_mixed_ne(parsed)
  if None in mixed:
    mixed = "None found!"
  print("Mixed Nash equilibium:", mixed)

def run_tests():
  # a few tests with known NE to test the system
  test = "5,6,2,9,9,3,2,5"
  compute_all(test)
  test = "4,6,6,1,7,3,3,3"
  compute_all(test)
  test = "4,6,6,1,3,7,0,1"
  compute_all(test)
  test = "4,6,6,1,3,7,1,1"
  compute_all(test)
  test = "1,5,0,2,1,0,5,2"
  compute_all(test)

parser = argparse.ArgumentParser()
parser.add_argument("input_str")

if __name__ == '__main__':
  args = parser.parse_args()
  compute_all(args.input_str)
```