Report: Project of Game Theory

Nguyen Viet Sang
viet-sang.nguyen@etu.unilim.fr

Master 1 Cryptis, Faculty of Science and Technique, University of Limoges

1 Introduction

In this report, we present an implementation of problems in Game Theory. Although there are 3 “volets” in the requirement, we combine them to a general problem. In the very first week, we implement a specific game with 2 Players - 2 Actions. However, after that, we define a general problem which is able to give solutions for all 3 “volets”. Our program works well with this idea. We also implement a User Graphic Interface (GUI) to demonstrate our work. The video of demo can be found at https://youtu.be/blwCa2zhBn8.

More specifically, our objective is to build a Python program which is capable of:

- checking if the game is zero-sum or not for games of Many Players - Many Actions,
- finding pure Nash Equilibrium (if existed) for games of Many Players - Many Actions,
- finding dominated strategies (if existed) for games of Many Players - Many Actions,
- finding mixed Nash Equilibrium for games of 2 Players

Since the general program works smoothly with games of Many Players - Many Actions, it could easily adapt to games of 2 Players - 2 Actions or 2 Players - Many Actions. The most important thing that contributes to the success of this general solution is the way we design data structure. Thanks to the convenient data structure, we could derive many ways of calculation to solve the general problem.

This report is organised as follows:

- Section 2 analyses the data structure,
- Section 3 presents the algorithm of zero-sum checking,
- Section 4 provides a method to find pure Nash Equilibrium,
- Section 5 presents a way to find dominated strategies,
- Section 6 presents the calculation of mixed Nash Equilibrium,
- Section 7 shows three examples corresponding to each “volet”,
- Section 8 discusses about GUI,
- Section 9 concludes the work.

2 Data Structure

Since we aim to solve the general game, it should be defined an optimized data structure. This helps to reduce difficulties when working with different problems such as determining Nash equilibrium or dominated strategies. We design our data structure in a way which is similar to represent a list of consecutive numbers in binary. For example, decimal numbers from 0 to 7 is represented in binary as shown in table 1.

We denote that *n_players*, *n_actions* are the number of players and the list containing the number of actions for these players. For instance, there is 3 players (P_0, P_1, P_2). Player P_0 has 2 actions $(a_0^{(0)}, a_1^{(0)})$, player P_1 has 3 actions $(a_0^{(1)}, a_1^{(1)}, a_2^{(1)})$ and player P_2 has 2 actions $(a_0^{(2)}, a_1^{(2)})$. Notice that all indexes start from 0. We denote as listing 1.2.

```python
n_players = 3 # P0, P1, P2
n_actions = [2, 3, 2] # [NbActionsOf P0, NbActionsOf P1, NbActionsOf P2]
```

Listing 1.1: Notation of players and actions

Number	bit 0	bit 1	bit 2
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Table 1: Listing decimal number in binary

No.	Action_P0	Action_P1	Action_P2
0	$a_0^{(0)}$	$\uparrow a_0^{(1)}$	$a_0^{(2)}$
1	$a_0^{(0)}$	$a_0^{(1)}$	$a_1^{(2)}$
2	$a_0^{(0)}$	$a_1^{(1)} \uparrow$	$a_0^{(2)}$
3	$a_0^{(0)}$ jr=6	$a_1^{(1)} \downarrow$ cr=2	$a_1^{(2)}$
4	$a_0^{(0)}$	$a_2^{(1)}$	$a_0^{(2)}$
5	$a_0^{(0)}$	$\downarrow a_2^{(1)}$	$a_1^{(2)}$
6	$a_1^{(0)}$	$a_0^{(1)}$	$a_0^{(2)}$
7	$a_1^{(0)}$	$a_0^{(1)}$	$a_1^{(2)}$
8	$a_1^{(0)}$	$a_1^{(1)}$	$a_0^{(2)}$
9	$a_1^{(0)}$	$a_1^{(1)}$	$a_1^{(2)}$
10	$a_1^{(0)}$	$a_2^{(1)}$	$a_0^{(2)}$
11	$a_1^{(0)}$	$a_2^{(1)}$	$a_1^{(2)}$

No.	Gain_P0	Gain_P1	Gain_P2
0	9	5	8
1	2	1	−2
2	3	−1	−3
3	0	−2	5
4	0	−3	0
5	0	−3	0
6	−1	3	0
7	−1	1	3
8	5	−2	0
9	0	−1	6
10	7	−3	1
11	0	−2	0

Table 2: Listing combinations of actions and gains

There are totally $2 \times 3 \times 2 = 12$ combinations (indexed from 0 to 11) of actions among these three players. Similar to represent a list of consecutive numbers in binary, we represent the combinations as shown in table 2 (left). This is the structure that how we store the gains of players' actions (right). We utilise the structure of actions as indexes to retrieve corresponding gains.

A certain line in table 2 (left) is a combination of the three players' actions. For example, we consider line 10 where the combination is $(a_1^{(0)}, a_2^{(1)}, a_0^{(2)})$. It means that in this case, player 0 chooses action $a_1^{(0)}$ (among his two possible actions $a_0^{(0)}$ and $a_1^{(0)}$), player 1 chooses action $a_2^{(1)}$ (among his three possible actions $a_0^{(1)}$, $a_1^{(1)}$ and $a_2^{(1)}$), player 2 chooses action $a_0^{(2)}$ (among his two possible actions $a_0^{(2)}$ and $a_1^{(2)}$).

We consider a combination as a tuple of actions. A tuple of gains corresponding these actions is defined in the same way. Line 10 in table 2 (right) contains a tuple $(7, -3, 1)$. It means that in this case, the gain of player 0 when choosing action $a_1^{(0)}$ is 7. Similarly for action $a_2^{(1)}$ of player 1 and action $a_0^{(2)}$ of player 2, the gains are -3 and 1, respectively.

In doing so, we generalize the problem with N players who have their own number of actions. In Python, we use the following formats:

```
n_players = N # P0, P1, ..., PN
n_actions = [na0, na1, ..., na(N-1)] # [NbActionsOf P0, NbActionsOf P1, ...,
    NbActionsOf P(N-1)]
```

Listing 1.2: Notation of players and actions

In practice, we only store the gains in table 2 (right) in Python as a list of tuples as show in listing 1.3.

```
normal\_form = [
    (9 , 5, 8),
    (2 , 1,-2),
    (3 ,-1,-3),
    (0 ,-2, 5),
    (0 ,-3, 0),
    (0 ,-3, 0),
    (-1, 3, 0),
    (-1, 1, 3),
    (5 ,-2, 0),
    (0 ,-1, 6),
    (7 ,-3, 1),
    (0 ,-2, 0)]
```

Listing 1.3: Gains structure in Python

3 Zero-sum Checking

It is quite simple to determine whether a game is zero-sum or not by traversing the list of tuples such as listing 1.3. If there is any tuple in which sum of all gains is not equivalent to 0, the game is not zero-sum.

```python
def check_zero_sum(self):
  for element in self.normal_form:
    if sum(element) != 0:
      return False
  return True
```

Listing 1.4: Zero-sum checking

4 Pure Nash Equilibrium (NE)

To find pure NEs, we sequentially examine tuple by tuple in the list. We consider tuple of gains $(7, -3, 1)$ in line 10 from table 2. It corresponds to tuple of actions $(a_1^{(0)}, a_2^{(1)}, a_0^{(2)})$. The actions of player 1 and player 2 are fixed. We then retrieve all tuples of gains which at the positions $(a_i^{(0)}, a_2^{(1)}, a_0^{(2)})$ with i in the range of player 0's actions. This results line 10 $(i = 1)$ and line 4 $(i = 0)$ in the table 2. We check that if 7 is the highest gain among $a_i^{(0)}$ or not (equivalent is also acceptable in case of non-strict). We then do the same work to check that if -3 is the highest value among gains of actions $a_j^{(1)}$ of player 1 in $(a_1^{(0)}, a_j^{(1)}, a_0^{(2)})$ or not. And the same is for player 2 with value 1 among gains of actions $a_k^{(2)}$ in $(a_1^{(0)}, a_2^{(1)}, a_k^{(2)})$. If three of those are all *True*, we claim that this tuple is a pure NE.

```python
def find_Nash_equilibrium_pure(self):
  nash_idx = []
  nash_tuples = []
  for idx, element in enumerate(self.nform):
    actions = self.get_actions_from_index(idx)
    is_nash = True

    for player, action in enumerate(actions):
      const_actions = [True] * self.n_players
      const_actions[player] = False

      gains = self.get_gains_const_actions(player, actions, const_actions)
      is_geq = self.compare_geq_element_to_list(element[player], gains)
      if not is_geq:
        is_nash = False
        break

    if is_nash:
      nash_idx.append(idx)
      nash_tuples.append(element)
  return nash_idx, nash_tuples
```

Listing 1.5: Find pure Nash Equilibrium

Listing 1.5 shows the algorithm of finding pure NEs. Given the index (idx) of a tuple in normal form, line 5 gets its actions (for example, $(a_1^{(0)}, a_2^{(1)}, a_0^{(2)})$ corresponds to $idx = 10$). Lines 9 and 10 present the actions which are fixed in the currently considered tuple. Line 12 retrieves all gains needed to compare (for example, $a_i^{(0)}$). Line 13 does the work of comparison. We now discuss some support functions.

Support functions:

Functions `get_actions_from_index(idx)` (line 5) and `get_gains_const_actions(player, actions, const_actions)` (line 12) are implemented based on mathematics. We observe some rules in our data structure and apply mathematics for calculation relating to indexes. More specifically, we define a new term called **consecutive range** (denoted as cr) which is maximum number of the consecutively same actions in a column (for a player). The other term is **jump range** (denoted as jr) which is the number of actions in a repeated cycle in a column (see the table 2). Listing 1.6 shows the examples of cr and jr considering table 2.

```
jr = [12, 6, 2] # [jr of P1, jr of P2, jr of P3]
cr = [6, 2, 1]  # [cr of P1, cr of P2, cr of P3]
```

Listing 1.6: Consecutive Range (cr) and Jump Range (jr)

Now we show the method to calculate jr and cr when knowing $n\_players = N$ and $n\_actions = [na_0, na_1, ..., na_{N-1}]$. First of all, we define a function named `mul_to_end(i)` (denoted as $MTE(i)$ for short). Given an index i of an element in $n\_actions$, this function return the product of elements from i to the end of list $(i, i+1, ..., N-1)$.

$$MTE(i) = na_i \times na_{i+1} \times ... \times na_{N-1}, \qquad 0 \leq i \leq N-1 \tag{1}$$

Consider a player P_i, $cr_i = MTE(i+1)$. This works for all players except the last player P_{N-1}. For this player, $cr_{N-1} = 1$. The calculations of **consecutive range** cr and **jump range** jr are shown in equation 2 and 3.

$$cr_i = \begin{cases} MTE(i+1), & \text{if } 0 \leq i \leq N-1 \\ 1, & \text{if } i = N-1 \end{cases} \tag{2}$$

$$jr_i = MTE(i), \qquad 0 \leq i \leq N-1 \tag{3}$$

We discuss about the function `get_actions_from_index(self, idx)` in listing 1.7. Given an index of a tuple of gains in list of normal form, we would like to get the tuple of actions corresponding to these gains. For example, $(7, -3, 1)$ in table 2 has the index 10. From this index, we calculate (i, j, k) representing actions in $(a_i^{(0)}, a_j^{(1)}, a_k^{(2)})$. In other words, we would like to know which action that each player chooses with this tuple of gains. We desire that the result is $(i, j, k) = (1, 2, 0)$. In $a_l^{(p)}$, given p and idx, we calculate l by using the equation 4.

$$l = \left\lfloor \frac{idx \ \% \ jr[p]}{cr[p]} \right\rfloor \tag{4}$$

```
def get_actions_from_index(self, idx):
  actions = []
  for player in range(self.n_players):
    idx_b = idx % self.jr[player]
    idx_p = int(idx_b / cr[player])
    actions.append(idx_p)
  return tuple(actions)
```

Listing 1.7: Fucntion `get_actions_from_index(self, idx)`

We discuss about the function `get_gains_const_actions(self, player, actions, const_actions)` shown in listing 1.8. In the example which we consider earlier, line 10 in table 2 gives the tuple of actions $(a_1^{(0)}, a_2^{(1)}, a_0^{(2)})$ (parameter `actions`) and the tuple of gains $(7, -3, 1)$. This function returns a list of gains $a_i^{(0)}$ satisfying $(a_i^{(0)}, a_2^{(1)}, a_0^{(2)})$ in the table when fixing actions of player 1 and player 2. The desired result is $[0, 7]$. We then compare 7 of player 0 (parameter `player`) with each element in this result as presented in algorithm of finding pure NE in listing 1.5. Parameter `const_actions` holds value $(False, True, True)$ which indicates that actions of player 1 and 2 are fixed. The idea is to find possible indexes which have actions $a_2^{(1)}$ of player 1 and $a_1^{(2)}$ of player 2. This results two lists of indexes $[4, 5, 10, 11]$ and $[0, 2, 4, 6, 8, 10]$, respectively (line 7 in listing 1.8). The desired result is the intersection of these two lists $[4, 10]$ (line 8 in listing 1.8) corresponding the gains $[0, 7$ of player 0.

```
1  def get_gains_const_actions(self, player, actions, const_actions):
2    '''Get list of gains to compare to determine a pure Nash equilibrium'''
3    n_tuples = len(self.nform)
4    idxs = [i for i in range(n_tuples)]
5    for player_i, is_const in enumerate(const_actions):
6      if is_const:
7        idxs_i = self.get_indexes_from_action(player_i, actions[player_i])
8        idxs = list(set(idxs).intersection(set(idxs_i)))
9    gains = []
10   for idx in idxs:
11     gains.append(self.nform[idx][player])
12   return gains
```

Listing 1.8: Fucntion `get_gains_const_actions(self, player, actions, const_actions)`

In line 7 of listing 1.8, we mention a function named `get_indexes_from_action(self, player, action)`. We now discuss this function which is shown in listing 1.9. Given a certain action of a certain player, it returns all possible indexes in data structure containing this action. For example, possible indexes of player 1's action 2 are $[4, 5, 10, 11]$. The calculation is based on cr and jr as shown in listing 1.9 from line 5 to line 8. However, in this algorithm, we use a dictionary named `sp`. We now discuss this new term.

```
1  def get_indexes_from_action(self, player, action):
2    s_idx = self.sp[player][action]
3    n_tuples = len(self.nform)
4    idxs = []
5    while s_idx < n_tuples:
6      for i in range(s_idx, s_idx + self.cr[player]):
7        idxs.append(i)
8      s_idx += self.jr[player]
9    return idxs
```

Listing 1.9: Fucntion `get_indexes_from_action(self, player, action)`

Besides, we define a new term called **starting positions** (denoted as sp). It contains starting indexes of each players' action. For instance, action $a_1^{(1)}$ of player 1 has starting index 2. In Python, we store these starting positions as a dictionary as listing 1.10. For example, we get the starting position of player 1's action 2 in the following syntax: `sp[player][action] = sp[1][2] = 4`.

```
sp = {
  0: {0: 0, 1: 6},
  1: {0: 0, 1: 2, 2: 4},
  2: {0: 0, 1: 1}
}
```

Listing 1.10: Dictionary of Starting Positions (sp)

5 Dominated Strategies

To find dominated strategies, we sequentially examine each action of each player to determine that whether it is a dominated strategy or not. We consider action $a_1^{(1)}$ of player 1. We would like to examine if this is a dominated one. We firstly get all gains of this player corresponding to action $a_1^{(1)}$ (top-down order). It is

$$gains(a_1^{(1)}) = [5, 1, 3, 1]$$

We then get the gains of player 1's other actions $a_j^{(1)}$ $(j \neq 1)$. In the example in table 2, the possible values of j can be 0 or 2. These gains are

$$gains(a_0^{(1)}) = [-1, -2, -2, -1]$$

$$gains(a_0^{(1)}) = [-3, -3, -3, -2]$$

We compare gains in currently considered action $gains(a_1^{(1)})$ with those ones in $gains(a_0^{(1)})$ and $gains(a_2^{(1)})$, in turn. If the results are all less than or equal $(\leq)$, $a_1^{(1)}$ is a dominated strategy.

```
 1 def find_dominated_strategies(self):
 2   '''
 3   Output form: [(player, dominated_action, dominating_action)]
 4   [(1, 1, 0)]: player 1 has dominated action 1 by dominating action 0
 5   '''
 6   dominated_strategies = []
 7   has_dominated = True
 8   while has_dominated:
 9     has_dominated = False
10 
11     for player in range(self.n_players):
12       for action in range(self.n_actions[player]):
13         if not self.dmnt[player][action]:
14           gains = self.get_gains(player, action)
15 
16           for cmp_action in range(self.n_actions[player]):
17             if cmp_action != action:
18               cmp_gains = self.get_gains(player, cmp_action)
19 
20               is_dominated = self.compare_leq_list_to_list(gains, cmp_gains)
21               if is_dominated:
22                 has_dominated = True
23                 self.dmnt[player][action] = True
24                 self.remove_dominated_strategy(player, action)
25                 dominated_strategies.append((player, action, cmp_action))
26                 break
27 
28   return dominated_strategies
```

Listing 1.11: Find dominated strategies

Listing 1.11 presents the algorithm to find dominated strategies. Line 14 gets the gains of currently considered action $(gains(a_1^{(1)}))$. Lines 16 gets gains of other actions $(gains(a_0^{(1)})$ and $gains(a_2^{(1)}))$. Line 20 does the comparison work. If the strategy is dominated, it should be removed from normal form (line 24). It leads to a new normal form after removing and we have to find dominated strategies in this new normal form. That is the purpose of the loop `while` in line 8.

However, in practice, we remove dominated strategy by simply replacing its gains with negative infinitive (`-math.inf` in Python 3). There are two reasons why we do that. On one hand, it does not affect to further comparisons. On the other hand, our data structure does not break by applying this method.

Besides, in listing 1.11, we use an another dictionary named *dmnt*. This contains boolean values to indicate which actions are dominated and removed from normal form. An example of this dictionary is shown in listing 1.12. We consider action 2 of player 1. `dmnt[player][action] = dmnt[1][2] = True` means this strategy of player 1 is dominated and has been remove from the normal form.

```
dmnt = {
  0: {0: False, 1: Fasle},
  1: {0: False, 1: False, 2: True},
  2: {0: False, 1: False}
}
```

Listing 1.12: Dictionary of Dominated Strategies

Function `get_gains(player, action)` in line 14 and 18 is based on mathematics. We use the terms **consecutive range** (*cr*), **jump range** (*jr*) and `starting positions` (*sp*) mentioned earlier. Given an action of a player, we retrieve its gains as the the way shown in listing 1.13.

```python
def get_gains(self, player, action):
  s_idx = self.sp[player][action]
  n_tuples = len(self.nform)
  res = []
  while s_idx < n_tuples:
    for i in range(s_idx, s_idx + self.cr[player]):
      res.append(self.nform_dmnt[i][player])
    s_idx += self.jr[player]
  return res
```

Listing 1.13: Function `get_gains(self, player, action)`

Definition of *cr*, *jr* and *sp* are very useful and convenient in handling indexes of iteration.

6 Mixed Nash Equilibrium

There are two cases that we need to determine mixed NE. The first one is 2 Players - 2 Actions when having no dominated strategies. And the second one is 2 Players - Many Actions when existing dominated strategies and then we could reduce the game to 2 Players - 2 Actions.

We consider notations of gains in table 3. If there is no dominated strategy in this game, we could calculate p and q by the equation5. From p and q, we could easily calculate mixed NE.

$$p = \frac{g_{11}^{(1)} - g_{10}^{(1)}}{g_{00}^{(1)} - g_{01}^{(1)} - g_{10}^{(1)} + g_{11}^{(1)}} \qquad p = \frac{g_{11}^{(0)} - g_{01}^{(0)}}{g_{00}^{(0)} - g_{01}^{(0)} - g_{10}^{(0)} + g_{11}^{(0)}} \tag{5}$$

Player 0\Player 1	Action 0 (q)	Action 1 ($1-q$)
Action 0 (p)	$(g_{00}^{(0)}, g_{00}^{(1)})$	$(g_{01}^{(0)}, g_{01}^{(1)})$
Action 1 ($1-p$)	$(g_{10}^{(0)}, g_{10}^{(1)})$	$(g_{11}^{(0)}, g_{11}^{(1)})$

Table 3: Notations of gains in case 2 Players - 2 Actions

7 Verify with examples

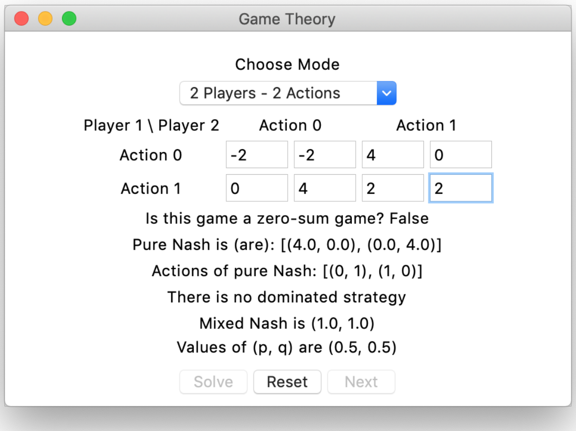
We run the examples in this section and show in the demo which can be found at https://youtu.be/blwCa2zhBn8.

7.1 Verify with 2 Players - 2 Actions

Player 0\Player 1	Action 0	Action 1
Action 0	$g(a_0^{(0)}, a_0^{(1)}) = (-2, -2)$	$g(a_0^{(0)}, a_1^{(1)}) = (4, 0)$
Action 1	$g(a_1^{(0)}, a_0^{(1)}) = (0, 4)$	$g(a_1^{(0)}, a_1^{(1)}) = (2, 2)$

Table 4: Example for verification with 2 Players - 2 Actions

We consider the example shown in table 4. This example is taken from an exercise in TD. The expected results are as screenshot in figure 1.



Choose Mode

2 Players - 2 Actions

Player 1 \ Player 2	Action 0		Action 1	
Action 0	-2	-2	4	0
Action 1	0	4	2	2

Is this game a zero-sum game? False

Pure Nash is (are): [(4.0, 0.0), (0.0, 4.0)]

Actions of pure Nash: [(0, 1), (1, 0)]

There is no dominated strategy

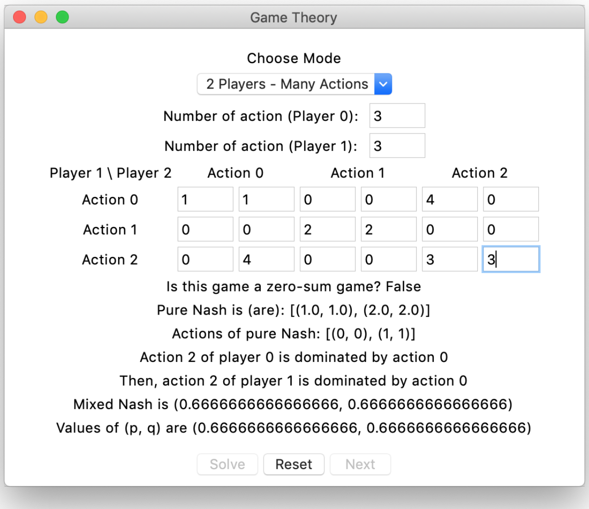
Mixed Nash is (1.0, 1.0)

Values of (p, q) are (0.5, 0.5)

Fig. 1: GUI of example shown in table 4

Player 0\Player 1	Action 0	Action 1	Action 2
Action 0	$g(a_0^{(0)}, a_0^{(1)}) = (1,1)$	$g(a_0^{(0)}, a_1^{(1)}) = (0,0)$	$g(a_0^{(0)}, a_2^{(1)}) = (4,0)$
Action 1	$g(a_1^{(0)}, a_0^{(1)}) = (0,0)$	$g(a_1^{(0)}, a_1^{(1)}) = (2,2)$	$g(a_1^{(0)}, a_2^{(1)}) = (0,0)$
Action 2	$g(a_2^{(0)}, a_0^{(1)}) = (0,4)$	$g(a_2^{(0)}, a_1^{(1)}) = (0,0)$	$g(a_2^{(0)}, a_2^{(1)}) = (3,3)$

Table 5: Example for verification with 2 Players - Many Actions



Choose Mode

2 Players - Many Actions

Number of action (Player 0): 3

Number of action (Player 1): 3

Player 1 \ Player 2	Action 0		Action 1		Action 2	
Action 0	1	1	0	0	4	0
Action 1	0	0	2	2	0	0
Action 2	0	4	0	0	3	3

Is this game a zero-sum game? False

Pure Nash is (are): [(1.0, 1.0), (2.0, 2.0)]

Actions of pure Nash: [(0, 0), (1, 1)]

Action 2 of player 0 is dominated by action 0

Then, action 2 of player 1 is dominated by action 0

Mixed Nash is (0.6666666666666666, 0.6666666666666666)

Values of (p, q) are (0.6666666666666666, 0.6666666666666666)

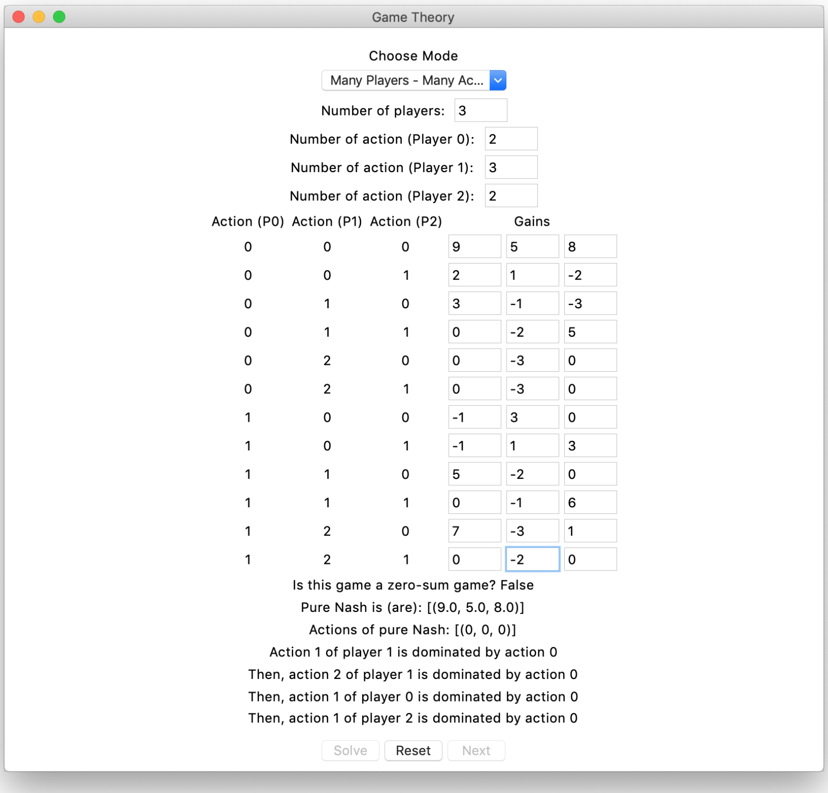
Fig. 2: GUI of example shown in table 5

7.2 Verify with 2 Players - Many Actions

We use some exercises in TD class to test the program. Figure 2 shows the results after solving the problem in table 5. We notice that in this example, after removing dominated strategies, we have a game of 2 Players - 2 Actions. Then, we could calculate mixed NE.

7.3 Verify with Many Players - Many Actions

We verify the game shown in 2 earlier. Recall that there are 3 players with 2, 3 and 2 actions, respectively. The results shown in figure 3.



Choose Mode

Many Players - Many Ac...

Number of players: 3

Number of action (Player 0): 2

Number of action (Player 1): 3

Number of action (Player 2): 2

Action (P0)	Action (P1)	Action (P2)	Gains		
0	0	0	9	5	8
0	0	1	2	1	-2
0	1	0	3	-1	-3
0	1	1	0	-2	5
0	2	0	0	-3	0
0	2	1	0	-3	0
1	0	0	-1	3	0
1	0	1	-1	1	3
1	1	0	5	-2	0
1	1	1	0	-1	6
1	2	0	7	-3	1
1	2	1	0	-2	0

```
Is this game a zero-sum game? False
Pure Nash is (are): [(9.0, 5.0, 8.0)]
Actions of pure Nash: [(0, 0, 0)]
Action 1 of player 1 is dominated by action 0
Then, action 2 of player 1 is dominated by action 0
Then, action 1 of player 0 is dominated by action 0
Then, action 1 of player 2 is dominated by action 0
```

Fig. 3: GUI of example shown in table 2

8 Graphic User Interface

As we can see in figures 1, 2 and 3, we build a GUI for users to input gains as well as number of actions or players in case Many Players - Many Actions. After clicking “Solve” button, this GUI shows the results. This GUI is built based on library Tkinter in Python.

9 Conclusion

In this project, we successfully implement a program to solve almost required problems. Our approach is so special because we aim to solve the general problem and hence, it is able to provide solutions for the sub-problems. More specifically, we propose a convenient data structure and methods of finding pure NEs, dominated strategies, mixed NE and checking zero-sum. Not only that, we aslo build a GUI to friendly communicate with users.