

## List of Tables

1	(Reward for Player A) . . . . .	2
2	(Reward for Player B) . . . . .	3
3	(Reward for Player A - Multiple action) . . . . .	8
4	(Reward for Player B - Multiple action) . . . . .	8

## List of Figures

1	Problem 1 Graphical Interface . . . . .	3
2	Result Two player with two action . . . . .	7
3	Problem 2 Graphical Interface . . . . .	9
4	Result Problem 2 . . . . .	12

# Contents

<b>I. Introduction</b>	<b>1</b>
1.1 Objectives . . . . .	1
1.2 Project structures . . . . .	1
<b>II. Programs and Materials</b>	<b>1</b>
<b>Methodology</b>	<b>2</b>
Problem 1 - game of 2 players with 2 strategies . . . . .	2
Zero sum . . . . .	4
Pure Nash equilibrium state . . . . .	4
Mixed Strategy . . . . .	5
Problem 2 - game of 2 players with multiple strategies . . . . .	8
Zero-sum . . . . .	9
Nash equilibrium . . . . .	10
Dominated strategy . . . . .	11
Multiple players - multiple choices . . . . .	13

# I. Introduction

## 1.1 Objectives

The game theory project consists of three problems and in all problems, I will determine:

- If the game is zero-sum or not
- Pure Nash equilibria
- The dominated strategies, and if there are the dominant strategies

## 1.2 Project structures

The project structure will be divided into three parts following the requirement and also an extra part. The first section is a game with two players and they have two strategies, the second part will be two players but multiple strategies per player. The next part is to make a three players game with three actions for each, and the extra part I will work with multiple player with multiple strategies for each player.

# II. Programs and Materials

For the graphical user interface part, I create a web application by using [React](#) for front-end part and [Golang](#) for backend server. The reason I chose this stack because it is easier to create UI in web base and also I am more familiar with Javascript and Golang.

## Requirement

- NodeJS
- Go

## Dependencies

- React
- Gin and Static middleware for Golang

### Installation, Running and testing

- Download and extract the project
- `npm install` (install dependencies)
- `npm run-script build`
- `go run ./main.go`
- Use your browser and go to `localhost:8082`

## Methodology

### Problem 1 - game of 2 players with 2 strategies

In this problem, There will be two player named as A and B. The strategy that a player can input utilizing number begun from 0, such as strategy "1", strategy "2",..., and it is interpreted as the reward tables below.

**Table 1:** (Reward for Player A)

Reward A		Player B	
Action		0	1
Player A	0	2	1
	1	0	1

**Table 2:** (Reward for Player B)

Reward B		Player B	
		Action	
Player A		0	1
	0	1	0
	1	0	2

User interface:

## Part 1

**RewardA**

2	1
0	1

**RewardB**

1	0
0	2

Check

**Figure 1:** Problem 1 Graphical Interface

We created the interface same as our input so that the strategy and reward will be represented as array  $[[2 \ 1] \ [0 \ 1]]$  for player A and  $[[1 \ 0] \ [0 \ 2]]$  for player B.

## Zero sum

In game theory, a zero-sum game is a mathematical representation of a situation in which an advantage that is won by one of two sides is lost by the other. In order to find out, we will check the reward of both players ( the total gains and the total losses), if the reward sum equal to zero (the total gains are ended up and total losses are subtracted ).

```
1 func CompareMatrix(mat1 [][]int, mat2 [][]int) bool {
2     row := len(mat1)
3     if row != len(mat2) {
4         return false
5     }
6     for i := 0; i < row; i++ {
7         col := len(mat1[i])
8         if col != len(mat2[i]) {
9             return false
10        }
11        for j := 0; j < col; j++ {
12            if mat1[i][j] != mat2[i][j] {
13                return false
14            }
15        }
16    }
17    return true
18 }
```

## Pure Nash equilibrium state

In pure strategy, if player A play a (with probability 1), player B can play for example the same action a but with probability 1 so that there will be no random play (pure nash equil state). A Nash equilibrium is a set of strategies, one for each player, such that no player has incentive to change his or her strategy given what the other players are doing. The first problem is two players versus game so that we can use brute force method to check every strategy, in each strategy, we find if there exist a better action for a player.

```

1 func CheckBetter(pA [][]int, pB [][]int, i int, j int) bool {
2     //pA check
3     for k := 0; k < 2; k++ {
4         if pA[k][j] > pA[i][j] {
5             return true
6         }
7     }
8     //pB check
9     for k := 0; k < 2; k++ {
10        if pB[i][k] > pB[i][j] {
11            return true
12        }
13    }
14    return false
15 }
16
17 result = result + "Nash equilibrium state <br/>player A, player B
18 :<br/>"
19 for i := 0; i < 2; i++ {
20     for j := 0; j < 2; j++ {
21         existBetter := CheckBetter(pA, pB, i, j)
22         if !existBetter {
23             fmt.Printf("%d,%d\n", i, j)
24             result = result + fmt.Sprintf("%d,%d<br/>", i, j)
25         }
26     }
27 }

```

## Mixed Strategy

A mixed strategy of player  $i$  is a measure of probability  $p_i$  defined on the set of pure strategies of player  $i$ . We denote the set  $P_i$  the set of mixed strategies of player  $i$ .  $p_i, s_i$  is the probability that  $i$  will play the pure strategy  $s_i$ .  $p_i$  in  $P_i$  therefore corresponds to a mixed strategy of player  $i$ .

For example, if we want to find a mixed strategy for player A, we will calculate the probability for each choice of player A so that the expectation values for every choice of B is equal. We applied the equation below to calculate the probability of player A pick 0 and the probability of player B pick 1 is equal to  $1-p_a$ .  $B_{\{(0,1)\}}$



is the award for player B when player A pick choice 0 and player B pick choice 1.

$$p_A = \frac{-B_{(0,1)} + B_{(1,1)}}{B_{(0,0)} - B_{(1,0)} - B_{(0,1)} + B_{(1,1)}} \quad (1)$$

$$p_B = \frac{-A_{(0,1)} + A_{(1,1)}}{A_{(0,0)} - A_{(1,0)} - A_{(0,1)} + A_{(1,1)}} \quad (2)$$

```

1 //find mixed Nash
2 var probA, probB float64
3 probA = float64(-pB[1][0] + pB[1][1]) / float64(pB[0][0] - pB
    [1][0] - pB[0][1] + pB[1][1])
4 probB = float64(-pA[0][1] + pA[1][1]) / float64(pA[0][0] - pA
    [0][1] - pA[1][0] + pA[1][1])
5 if probA >= 1 || probA <= 0 {
6     fmt.Println("Cannot find mixed strategy for player A because
    player B has dominated strategy")
7     result = result + "Player B has dominated strategy => No
    mixed strategy for player A<br/>"
8     _, y := IndexMaximalElement(pB)
9     fmt.Printf("Player B dominant strategy: %d\n", y)
10    result = result + fmt.Sprintf("Player B dominant strategy: %d
    <br/>", y)
11 } else {
12     fmt.Printf("Mixed strategy player A: [%g, %g]\n", probA, 1-
    probA)
13     result = result + fmt.Sprintf("Mixed strategy player A: [%g,
    %g]<br/>", probA, 1-probA)
14 }
15 if probB >= 1 || probB <= 0 {
16     fmt.Println("Cannot find mixed strategy for player B because
    player A has dominated strategy")
17     result = result + "Player A has dominated strategy => No
    mixed strategy for player B<br/>"
18     x, _ := IndexMaximalElement(pA)
19     fmt.Printf("Player A dominant strategy: %d\n", x)
20     result = result + fmt.Sprintf("Player A dominant strategy: %d
    <br/>", x)
21 } else {
22     fmt.Printf("Mixed strategy player B: [%g, %g]\n", probB, 1-

```

```

23     probB)
    result = result + fmt.Sprintf("Mixed strategy player B: [%g,
24     %g]<br/>", probB, 1-probB)
    }

```

## Result

### Part 1

#### RewardA

2	1
0	1

#### RewardB

1	0
0	2

Result:

Zero sum game ? false

Nash equilibrium state

player A, player B:

0,0

1,1

Mixed strategy player A: [0.6666666666666666, 0.33333333333333337]

Can't find mixed strategy for player B because player A has dominated strategy

Player A dominant strategy: 0

**Figure 2:** Result Two player with two action

## Problem 2 - game of 2 players with multiple strategies

In this problem, There will be two player named as A and B. The strategy that a player can input utilizing number begun from 0, such as strategy "1", strategy "2",..., The player A will have N choice from 0 to N-1, and player B will have M choice from 0 to M-1 and it is interpreted as the reward tables below.

**Table 3:** (Reward for Player A - Multiple action)

Reward A		Player B			
	Action	0	1	...	M-1
Player A	0	1	1	...	...
	1	2	0	...	...
	...	...	...	..	...
	n-1	...	...	...	...

**Table 4:** (Reward for Player B - Multiple action)

Reward B		Player B			
	Action	0	1	...	M-1
Player A	0	1	1	...	...
	1	2	0	...	...
	...	...	...	..	...
	n-1	...	...	...	...

**input** `SolvePart2(n, m int, pA, pB [][][]int)` where `n` and `m` is the number choices of player A and player B and `pA` and `pB` is the table reward of each player in

array datatype.

**User interface:**

## Part 2

Number of A's actions:

Number of B's actions:

### RewardA

Action	Player B		
Player A	1	1	1
	1	1	1

### RewardB

Action	Player B		
Player A	1	1	1
	1	1	1

**Figure 3:** Problem 2 Graphical Interface

The user will fill the number of action for both players and the application will generate the reward table following the players choice actions's input.

### Zero-sum

Im using the same function as the Part 1 because the checking process is similar.

## Nash equilibrium

In this project, we will find the best choice of A for each choice of B (max of each column) and do the same thing for B with max of each row.

```
1 func GetMaxAOneHot(n, m int, mat [][]int, max []int) [][]int {
2     maxOneHot := make([][]int, n)
3     for i := 0; i < n; i++ {
4         maxOneHot[i] = make([]int, m)
5     }
6
7     for j := 0; j < m; j++ {
8         for i := 0; i < n; i++ {
9             if mat[i][j] == max[j] {
10                 maxOneHot[i][j] = 1
11             }
12         }
13     }
14
15     return maxOneHot
16 }
```

We will mark the choice 1 if it is the best choice to response to the other player choice. After that we find the pure Nash by multiply two matrix cell by cell. If both players have a same case as their best choice, that case is considered as Nash equilibrium state.

We will mark the choice "1" if it is the best choice to response to the other player choice. There might be multiple best choices as they have equal value. If both players have a same case as their best choice, that case is considered as Nash equilibrium state. Considering the complexity of this process, we have the complexity of finding best choice for each player is  $O(n * m)$  and the complexity of cell-by-cell multiplication step  $O(n * m)$ . The complexity of finding pure Nash state(s) in this scenario is  $O(N * M)$

```
1 func MultiplyCellByCellMatrix(n, m int, pA [][]int, pB [][]int)
   [][]int {
2     mat := make([][]int, n)
3     for i := 0; i < n; i++ {
```

```
4     mat[i] = make([]int, m)
5     for j := 0; j < m; j++ {
6         mat[i][j] = pA[i][j]*pB[i][j]
7     }
8 }
9 return mat
10 }
```

## Dominated strategy

In the previous part, we already have the best choice which stamp all the best choice of a player for each choice of other players. On the off chance that, for each choice of B, the leading choice of A does not alter, it implies that the finest choice of A is the dominated strategy. So that we will find the product of all the column and row of the matrix.

PlayerA finds the product of all columns if that product = 1 means that if playerA performs that action, he will always get a score higher than B.

If the whole row of maxOne is = 1, it means that when performing this action, no matter what action B performs, A will at least draw or win.

```
1 //product of all the col
2 dominateA := ProductAllColMatrix(n, m, maxAOneHot)
3 fmt.Println(dominateA)
4 for i, v := range dominateA {
5     if v == 1 {
6         fmt.Printf("Player A dominated choice number: %d\n", i)
7         result += fmt.Sprintf("Player A dominated choice number:
8             %d<br/>", i)
9     }
10 }
11 //product of all the row
12 dominateB := ProductAllRowMatrix(n, m, maxBOneHot)
13 fmt.Println(dominateB)
14 for i, v := range dominateB {
15     if v == 1 {
16         fmt.Printf("Player B dominated choice number: %d\n", i)
17         result += fmt.Sprintf("Player B dominated choice number:
18             %d<br/>", i)
19     }
20 }
```

```
17 }
18 }
```

## Result

## Part 2

Number of A's actions:

Number of B's actions:

Create

### RewardA

Action	Player B			
Player A	2	0	1	1
	3	-2	4	0
	3	-5	1	0

### RewardB

Action	Player B			
Player A	1	3	0	-1
	-2	2	2	1
	-5	1	1	0

Check

Result:

Zero sum game ? false

Best choice index-matrix for A:

[[0 1 0 1] [1 0 1 0] [1 0 0 0]]

Best choice index-matrix for B:

[[0 1 0 0] [0 1 1 0] [0 1 1 0]]

Pure Nash Index:

[[0 1 0 0] [0 0 1 0] [0 0 0 0]]

Pure Nash Case(s):

0,1

1,2

Player B dominated choice number: 1

**Figure 4:** Result Problem 2

## Multiple players - multiple choices

For the input, we need the number player (denoted as  $N$ ) and for each player, we also need the maximum choice for them, represented as an array. For example, if we have 3 players, each of them has 3 maximum choices, so that  $N=3$  and the array represented the maximum choices of each players is  $[3,3,3]$  And for each permutation (for example  $[0,0,1]$ ) means that the player A picks choice 0, player B 0 and player C is 1 corresponding to the input has the reward  $[-1,0,2]$