List of Tables

List of Figures

Contents

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 familiar with Javascript and Golang. More important, Golang is much faster than Python (I did a version in Python but it is slower than 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	0	1
Player A	0	1	0
	1	0	2

User interface:

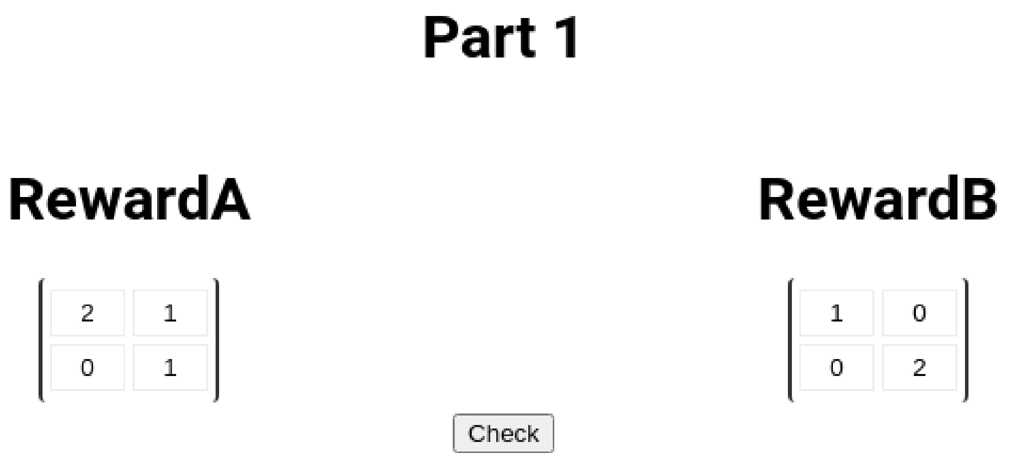


Figure 1: Problem 1 Graphical Interface

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 lesses are subtracted).

```go
func CompareMatrix(mat1 [][]int, mat2 [][]int) bool {
    row := len(mat1)
    if row != len(mat2) {
        return false
    }
    for i := 0; i < row; i++ {
        col := len(mat1[i])
        if col != len(mat2[i]) {
            return false
        }
        for j := 0; j < col; j++ {
            if mat1[i][j] != mat2[i][j] {
                return false
            }
        }
    }
    return true
}
```

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.

```go
func CheckBetter(pA [][]int, pB [][]int, i int, j int) bool {
    //pA check
    for k := 0; k < 2; k++ {
        if pA[k][j] > pA[i][j] {
            return true
        }
    }
    //pB check
    for k := 0; k < 2; k++ {
        if pB[i][k] > pB[i][j] {
            return true
        }
    }
    return false
}

result = result + "Nash equilibrium state <br/>player A, player B
    :<br/>"
for i := 0; i < 2; i++ {
    for j := 0; j < 2; j++ {
        existBetter := CheckBetter(pA, pB, i,j)
        if !existBetter {
            fmt.Printf("%d,%d\n", i, j)
            result = result + fmt.Sprintf("%d,%d<br/>", i, j)
        }
    }
}
```

Mixed Strategy

A mixed strategy of player i is a measure of probability pi 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 si. p_i in P_i therefore corresponds to a mixed strategy of player i.

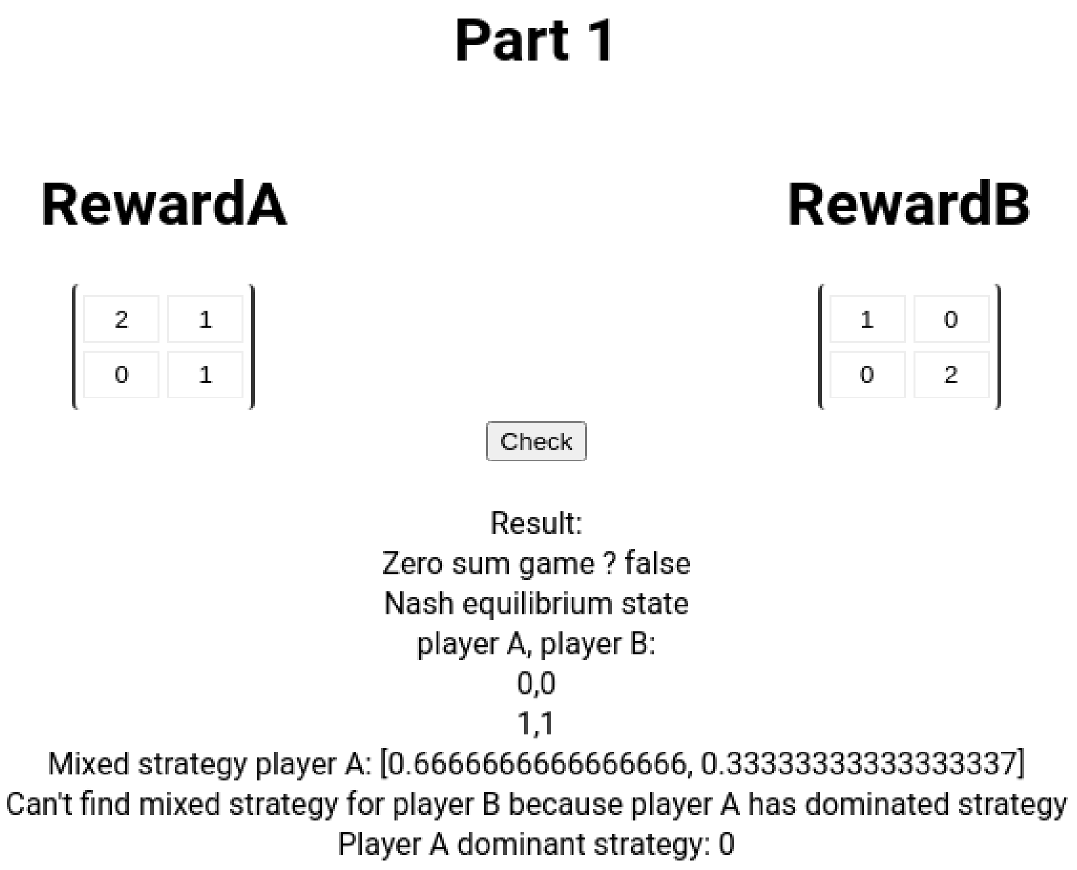
For example, if we want to find a mixed strategy for player A, we will calculate the probability for each choice of player A so that the expectation values for every choice of B is equal. We applied the equation below to calculate the probability of player A pick `0` and the probability of player B pick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

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

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

```go
//find mixed Nash
var probA, probB float64
probA = float64(-pB[1][0] + pB[1][1]) / float64(pB[0][0] - pB[1][0] - pB[0][1] + pB[1][1])
probB = float64(-pA[0][1] + pA[1][1]) / float64(pA[0][0] - pA[0][1] - pA[1][0] + pA[1][1])
if probA >= 1 || probA <= 0 {
    fmt.Println("Can't find mixed strategy for player A because player B has dominated strategy")
    result = result + "Can't find mixed strategy for player A because player B has dominated strategy<br/>"
    _, y := IndexMaximalElement(pB)
    fmt.Printf("Player B dominant strategy: %d\n", y)
    result = result + fmt.Sprintf("Player B dominant strategy: %d<br/>", y)
} else {
    fmt.Printf("Mixed strategy player A: [%g, %g]\n", probA, 1-probA)
    result = result + fmt.Sprintf("Mixed strategy player A: [%g, %g]<br/>", probA, 1-probA)
}
if probB >= 1 || probB <= 0 {
    fmt.Println("Can't find mixed strategy for player B because player A has dominated strategy")
    result = result + "Can't find mixed strategy for player B because player A has dominated strategy<br/>"
    x, _ := IndexMaximalElement(pA)
    fmt.Printf("Player A dominant strategy: %d\n", x)
    result = result + fmt.Sprintf("Player A dominant strategy: %d<br/>", x)
} else {
    fmt.Printf("Mixed strategy player B: [%g, %g]\n", probB, 1-
        probB)
    result = result + fmt.Sprintf("Mixed strategy player B: [%g,
        %g]<br/>", probB, 1-probB)
}
```

Result



2	1
0	1

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)

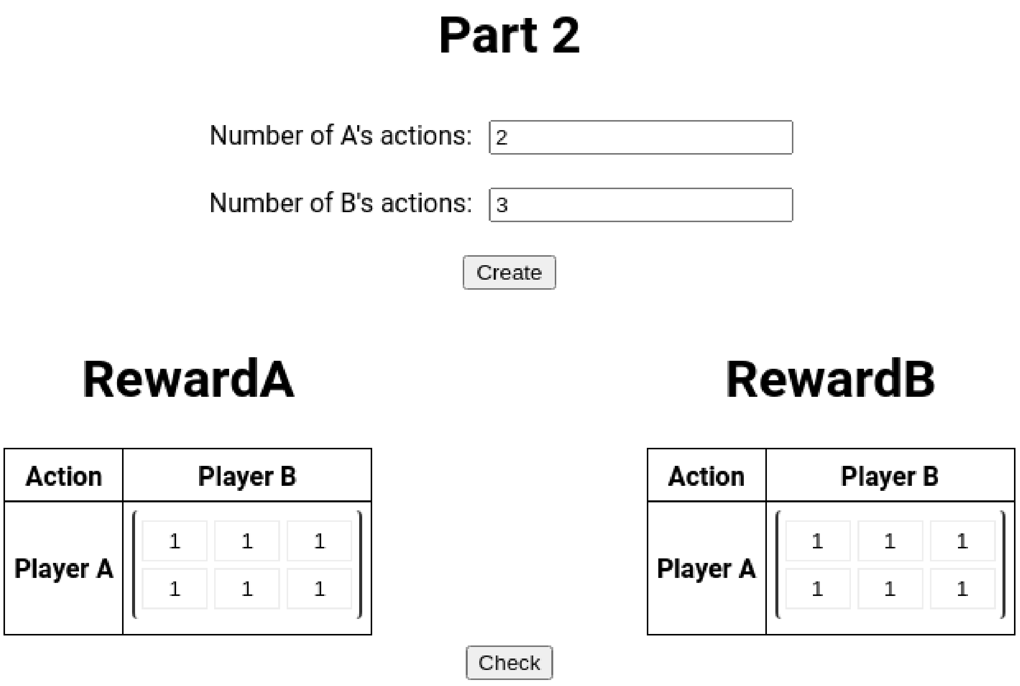
Reward A			Player B		
	Action	0	1	...	M-1
	0	1	1	...	...
Player A	1	2	0	...	...
	...	...	...	..	...
	n-1	...	...	...	...

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

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

Number of B's actions: 3

RewardA

Action	Player B
Player A	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$

RewardB

Action	Player B
Player A	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$

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 equilabrium

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.

```go
func GetMaxAOneHot(n, m int, mat [][]int, max []int) [][]int {
    maxOneHot := make([][]int, n)
    for i := 0; i < n; i++ {
        maxOneHot[i] = make([]int, m)
    }

    for j := 0; j < m; j++ {
        for i := 0; i < n; i++ {
            if mat[i][j] == max[j] {
                maxOneHot[i][j] = 1
            }
        }
    }

    return maxOneHot
}
```

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 Nashequilibrium 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)

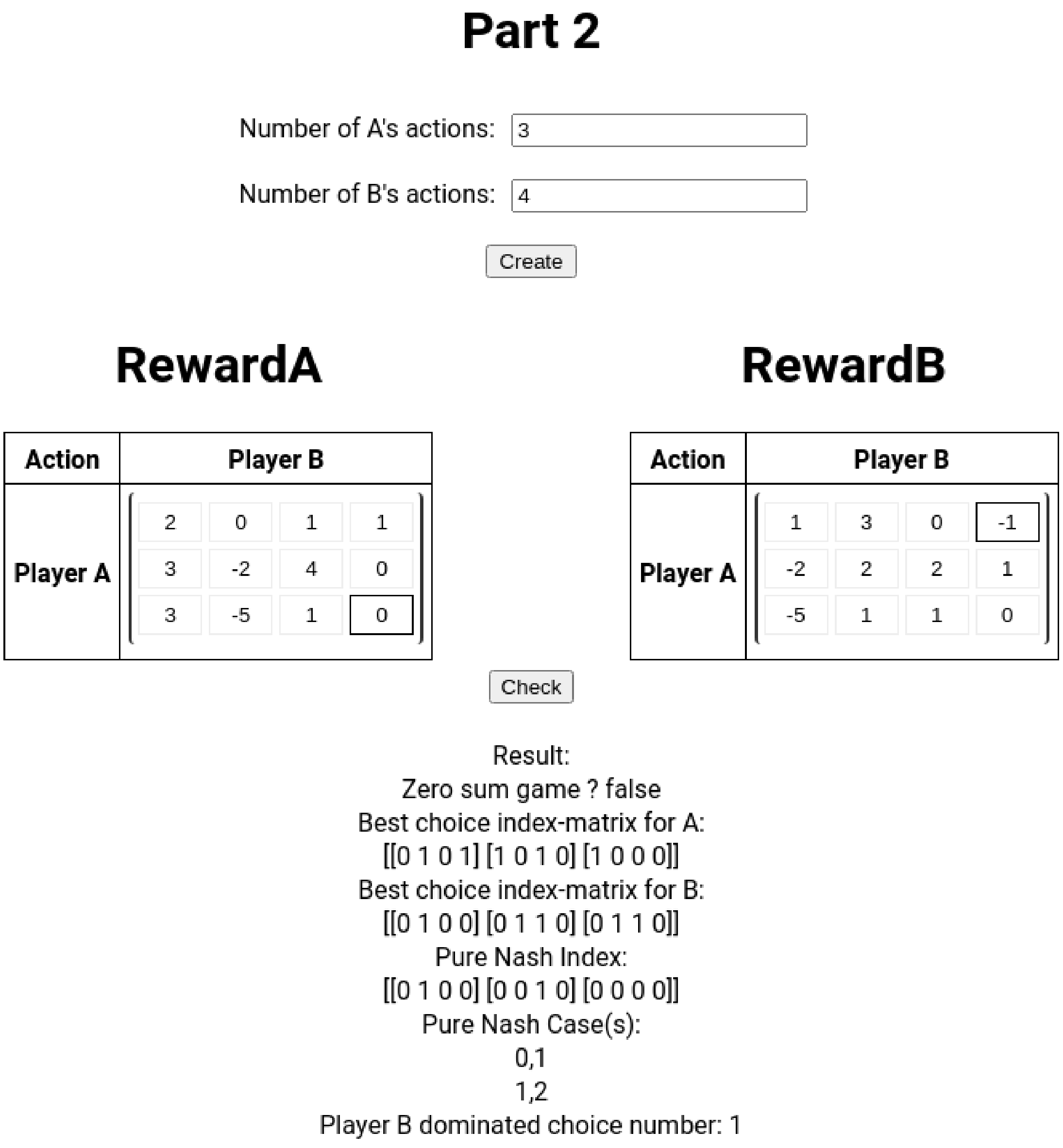
```go
func MultiplyCellByCellMatrix(n, m int, pA [][]int, pB [][]int)
    [][]int {
    mat := make([][]int, n)
    for i := 0; i < n; i++ {
        mat[i] = make([]int, m)
        for j := 0; j < m; j++ {
            mat[i][j] = pA[i][j]*pB[i][j]
        }
    }
    return mat
}
```

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

```go
//product of all the col
dominateA := ProductAllColMatrix(n, m, maxAOneHot)
fmt.Println(dominateA)
for i, v := range dominateA {
    if v == 1 {
        fmt.Printf("Player A dominated choice number: %d\n", i)
        result += fmt.Sprintf("Player A dominated choice number:
           %d<br/>", i)
    }
}
//product of all the row
dominateB := ProductAllRowMatrix(n, m, maxBOneHot)
fmt.Println(dominateB)
for i, v := range dominateB {
    if v == 1 {
        fmt.Printf("Player B dominated choice number: %d\n", i)
        result += fmt.Sprintf("Player B dominated choice number:
           %d<br/>", i)
    }
}
```

Result



Part 2

Number of A's actions: 3

Number of B's actions: 4

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

```
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

Based on the game defined in the section 3, the best method to input the game data is by text method. The first line will be the number of player (N). The second line will be an array showing the max choice of each player ([C0, C1, C2, ..., CN −1]). The code will generate a text file with the permutation of all possible cases for user to input the data in the format Listing 1.9.

```
1
 # Case [0 ,0 ,0]
2
 < user_input_reward_here >
3
 # Case [1 ,0 ,0]
4
 10 20 30
5
 # Case [2 ,0 ,0]
```

Pure nash

For this section, I have developed a pivot-comparing algorithm to check every possible cases for Pure Nash Equilibrium. Therefore, I confidence that the code works with any size of N players, unlimited number of actions per player. The method uses one case as a pivot and find other cases to compare with the pivot. The worse case will be mark as "non-Nash" and the better case will be a new pivot. Using pivot method with marking table makes the algorithm better than brute-force method. to check every cases at least once. If N is too large, the time to run will be low as the complexity is exponential time.