CS 4187: Game Theory

Assignment

Release date:18th and 25th November, 2024

Due date: 20th December, 204, 11:59 pm (IST)

Maximum Score: 100 marks (this assignment is graded)

Read the following before you move forward:

1. This programming assignment is worth **15% of the total marks** of this course.

2. The submission deadline is **20th December**. No extension. But it is **highly suggested that you complete it before the end-semester exam for your own good**.

3. Plagiarism, if detected, no matter how big or small, will lead to a score of ZERO for all the team member; no exceptions! It is ok to take help from AI tools or talk to other teams. But, you cannot copy from another team or any AI tools.
    a. It is down right stupid to copy-paste from AI tools and think that is acceptable.
    b. Also, it is a bad idea to keep open another teams work in front of you while you do the assignment. While you may think you are just taking help, your reports and your codes will get heavily influenced by the other team to the point that it will qualify as being plagiarized.

4. This is a team project. You have to do the project with the team that you selected in the excel sheet that was shared with you.
    a. **Only one submission per team**.
    b. Submission must be made using the google drive link that was shared with you.
    c. **Your submission in the google drive link must contain ONLY TWO files:**
        i. **Either *task1_option1.py* OR *task1_option2.py*.**
        ii. **Either *task2_option1.py*, OR *task2_option2.py*, OR *task2_option3.pdf*.**

    Read the deliverables carefully to understand what needs to be submitted in the Python scripts and/or the report. **DON'T submit any other files. You MUST NOT zip/compress these two files. You MUST NOT put these two files in a subfolder inside the google drive link.** Directly drop these two files in the google drive folder corresponding to your team.

In this assignment, there are **two tasks**. **Both the tasks are mandatory**. For each of the task, you have two options. **You have to do one option per task**. For these tasks, you are required to **implement the code from scratch**. To ensure this, I recommend you don't use any other libraries but *Numpy*, *Scipy*, and *Matplotlib*. If you use any other library, the final judgement about whether you implemented the code from scratch is mine.

Task 1:

You are given a perfect information **extended form game** (PIEFG) with/without chance moves. You have to do ONLY ONE of the following options:

Option 1: Convert the PIEFG into a strategic form game (SFG). This code must be implemented in the Python script *task1_option1.py* that is provided to you. This Python script has a function called *to_sfg*. You have to write a code for *to_sfg*. **Don't change the overall skeleton of the Python script**.

Option 2: Compute a subgame-perfect Nash equilibrium (SPNE). This code must be implemented in the Python script *task1_option2.py* that is provided to you. This Python script has a function called *compute_SPNE*. You have to write a code for *compute_SPNE*. **Don't change the overall skeleton of the Python script**. I want to make a few points clear:

- You have to compute only one SPNE.
- If your code does not account for chance moves, you get ZERO.

In order to explain the input and the output format, I have recorded a small YouTube video:

https://youtu.be/_psRT-Cmd-g

You must watch this video only after reading through the description of the input and output format given below. The video is to clarify a few points and NOT meant to be a substitute of reading the description.

Input format: The input to the functions *to_sfg* and *compute_SPNE* is a PIEFG. PIEFG is a tree. In what follows we describe the data structure used to capture this tree. We do this using an example to make it easier to understand. The example is shown in *Figure 1* and the corresponding data structure in *Figure 2*.

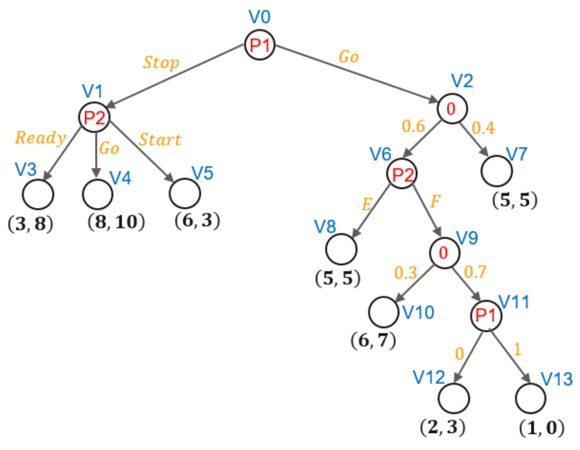
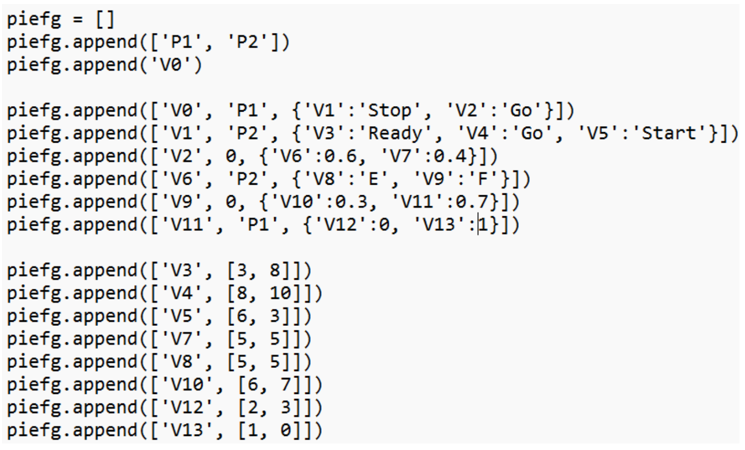


Figure 1



```
piefg = []
piefg.append(['P1', 'P2'])
piefg.append('V0')

piefg.append(['V0', 'P1', {'V1':'Stop', 'V2':'Go'}])
piefg.append(['V1', 'P2', {'V3':'Ready', 'V4':'Go', 'V5':'Start'}])
piefg.append(['V2', 0, {'V6':0.6, 'V7':0.4}])
piefg.append(['V6', 'P2', {'V8':'E', 'V9':'F'}])
piefg.append(['V9', 0, {'V10':0.3, 'V11':0.7}])
piefg.append(['V11', 'P1', {'V12':0, 'V13':1}])

piefg.append(['V3', [3, 8]])
piefg.append(['V4', [8, 10]])
piefg.append(['V5', [6, 3]])
piefg.append(['V7', [5, 5]])
piefg.append(['V8', [5, 5]])
piefg.append(['V10', [6, 7]])
piefg.append(['V12', [2, 3]])
piefg.append(['V13', [1, 0]])
```

Figure 2

The data structure is basically a **list-of-lists**:

1. The first element of the list is a list containing the **all the players**. This list MUST not contain "nature" player. Recall that nature is used as a player in those vertices containing **chance moves**. We use 0 to capture nature and hence **0 can't be a player in the list**. Also, 0 here is an integer (not a string or a float).

2. The second element of the list is the name of the **root vertex** of the tree.

3. The remaining elements of the list are lists containing the information of terminal and non-terminal vertices; **one list per vertex**:
   - **Non-terminal vertices:** The information of each non-terminal vertex is captured in a list. The first element of the list is the vertex name. The second element of the list is the player who is playing in this vertex. The third element of this list is a dictionary. The content of this dictionary depends on the player who is playing in this vertex:
     i. **Nature:** The dictionary contains the successor vertex and the probability of going to that vertex. The successor vertex is the **key** and the probability of going to that vertex is the **value** of the dictionary.

     ii. **Players other than nature:** The dictionary contains the successor vertex and the action which the player has to take in order to go to that vertex. The successor vertex is the **key** and the corresponding action is the **value** of the dictionary.

   - **Terminal vertices:** The information of each terminal vertex is captured in a list. The first element of the list is the vertex name. The second element of the list is a list containing player's payoff. **The order of the payoff should match the order of players which is the first element of the list**.

NOTE: I would like to be explicit about three points. *First,* the datatype of the player, vertex, and action can be either string, integer, or float. *Second,* you can have more than two players. *Third,* in Figure 2 we listed the non-terminal vertices first and then the terminal ones. But, the order in which you mention the non-terminal and terminal vertices does not matter. This is because the length of the lists for non-terminal and terminal vertices are three and two respectively and hence we can easily decipher which list is for which type (terminal or non-terminal) of vertex.

Output format: For the PIEFG shown in Figure 1, an example of the output for options 1 and 2 are.

Option 1: [((‘V0’, ‘V11’), (‘V1’, ‘V6’)), U]

Option 2: [((‘V0’, ‘V11’), (‘V1’, ‘V6’)), ((‘Stop’, 0), (‘Go’, ‘E’))]

For both options 1 and 2, the output is a list. The first element in the list for both options are ((‘V0’, ‘V11’), (‘V1’, ‘V6’)) which is a **tuple of tuples**; one tuple per player. So, each tuple is associated with a player. The tuple corresponding the player contains the vertices where the player makes decision, e.g. (‘V0’, ‘V11’) is for player P1 and the second tuple (‘V1’, ‘V6’) is for player P2. **The order of the vertices in the tuple does NOT matter**, e.g. for P1, the tuple can be (‘V11’, ‘V0’) as well. **But, the order of the tuple matters; it should**

match the order of the players given in the input. Therefore, the tuple corresponding to player P1 comes before P2 because the order of the players in input is [‘P1’, ‘P2’].

For option 1, the second element of the list is U, the utility function of the corresponding SFG. U is a dictionary whose **keys** are the strategy profile and whose **values** are the utility of the players corresponding to the strategy profile. For the PIEFG shown in Figure 1 there are $6 \times 4 = 24$ strategy profiles ($3 \times 2 = 6$ strategies for player P1 and $2 \times 2 = 4$ strategies for P2). Therefore, U is a dictionary of 24 key-value pairs. Few of these key value pairs are as follows,

U[((‘Stop’, 0), (‘Go’, ‘E’))] = (8, 10)

U[((‘Go’, 0), (‘Go’, ‘E’))] = (5, 5)

U[((‘Go’, 1), (‘Go’, ‘F’))] = (3.50, 3.26)

Consider the first key-value pair. For this, the key ((‘Stop’, 0), (‘Go’, ‘E’)) is the strategy profile and (8, 10) is the payoff. **The order of the elements in the strategy profile must match the order of vertices** in ((‘V0’, ‘V11’), (‘V1’, ‘V6’)). **The order of the payoff must match the order of the players that is mentioned in the input**.

For option 2, the second element of the list is a SPNE of the PIEFG. For the given PIEFG ((‘Stop’, 0), (‘Go’, ‘E’)) is a SPNE. Just like option 1, SPNE is a strategy profile and hence the order of the elements in SPNE must match the order of vertices in ((‘V0’, ‘V11’), (‘V1’, ‘V6’)).

Deliverable: Either *task1_option1.py* OR *task1_option2.py* **but not both**.

Task 2:

You have to do ONLY ONE of the following three options:

1. **Option 1:** You are given a strategic form game (SFG). Compute the pure strategies of each player that survives iterated removal of strictly dominated strategies (IRSDS). This code must be implemented in the Python script *task2_option1.py* that is provided to you. This Python script has a function called *IRSDS*. You have to write a code for *IRSDS*. **Don’t change the overall skeleton of the Python script**. **Read the description in page 5 of this document before attempting this problem**.
    - You must account for strict domination by MIXED STRATEGIES otherwise you get ZERO.

2. **Option 2:** You are given a SFG. Compute a correlated equilibrium of the game. This code must be implemented in the Python script *task2_option2.py* that is provided to you. This Python script has a function called *compute_CE.py*. You have to write a code for *compute_CE.py*. **Don’t change the overall skeleton of the Python script**. **Read the description in page 5 of this document before attempting this problem**.

3. **Option 3:** Solve ALL the problems in page 6 of this document. The answers to this problem must be submitted as a pdf titled *task2_option3.pdf*.

Instructions for Options 1 and 2 of Task 2

Linear Programming in Python: For options 1 and 2, you have to solve a linear program in Python (preferably Scipy). Here is a YouTube video where I demonstrate how to use Scipy to compute MSNE of a two-player zero-sum game using linear programming in Python:

https://youtu.be/-NPAUhVYHkU

Input format for options 1 and 2: The input to the function *IRSDS* and *compute_CE* is a **Numpy matrix** capturing the utility of each player for different strategy profile. To elaborate, let there be n players indexed $i = 1, 2, \cdots, n$. Player i has m_i pure strategies indexed $k_i = 1, 2, \cdots, m_i$. Then the input to *IRSDS* is a **Numpy matrix** of dimensions,

$$m_1 \times m_2 \times \cdots \times m_n \times n$$

Consequently, the utility of player i corresponding to the strategy profile $(k_1,\ k_2,\ \cdots,\ k_n)$ is the value of the Numpy matrix corresponding to the index $(k_1 - 1,\ k_2 - 1,\ \cdots,\ k_n - 1,\ i - 1)$ *(the -1's are there because Python is 0-indexed)*.

Output format for option 1: The output of the function *IRSDS* is a **dictionary** whose **keys** are the players and the **value** corresponding to the key is a **list** containing all the pure strategies of the player that survived IRSDS.

Output format for option 2: The output of the function *compute_CE* is a **dictionary** whose **keys** are the players and the **value** corresponding to the key is a **list** containing a correlated equilibrium of the player. The k^{th} element of the list is probability that the player will play pure strategy $k + 1$ ($+1$ *because Python is* 0-*indexed*).

		Player B			
		1	2	3	4
Player A	1	3, 2	4, 1	2, 3	0, 4
	2	4, 4	2, 5	1, 2	1.5, 4
	3	1, 3	3, 1	3, 1	4, 2
	4	5, 1	3, 1	2, 3	1, 4

Figure 3

Example for options 1 and 2: The utility matrix of a 2-player SFG is shown in Figure 3. The input to the functions *IRSDS* and *compute_CE* for the game shown in Figure 3 is $4 \times 4 \times 2$ Numpy matrix which is included in this folder as *task2_example.npy*.

For option 1, the output of the function *IRSDS* is the following dictionary,

$$\{1{:}\ [3, 4], 2{:}\ [1, 4]\}$$

For option 2, one possible output of the function *compute_CE* is the following dictionary,

$$\{1{:}\ [0, 0, 0.75, 0.25], 2{:}\ [0.43, 0, 0, 0.57]\}$$

Problems for Option 3 of Task 2

REMEMBER: You have to solve all of them.

Problem 1: Prove or disprove this statement: It possible that a SFG don't have a strictly dominant **pure** strategy but has a strictly dominant **mixed** strategy.

- *NOTE:* The above statement is different from the statement: A strategy (pure or mixed) may not be strictly dominated by a pure strategy but can be strictly dominated by a mixed strategy.

Problem 2: Compute a sequential equilibrium (SE) of the imperfect information extended form game (IIEFG) shown in Figure 4. The three nodes of player P3 forms one information set (this should be obvious but I am telling it just in case there is a confusion). Also, the top-right node where player P2 plays has only one action B (it is not a typo). You may want to look into **lectures 19, 20, 28, 29, 30, and 42 slides** (there is also a video lecture in it that show how to compute **mixed strategy SE**; you may not have to compute mixed strategy SE for this problem but mixed strategy SE is included in the final exam).

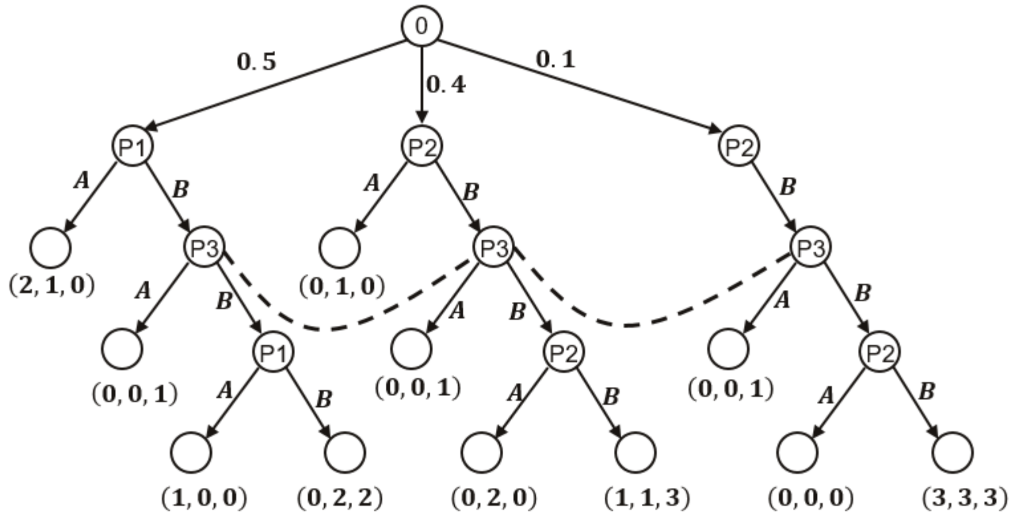


Figure 4

Problem 3: A consumer needs 1 unit of a certain good. This consumer can procure this good from 2 companies. The production cost of this good for companies 1 and 2 are c_1 and c_2 respectively. The cost of production are **private** information of the companies. Both the companies set the price of the good **simultaneously**. Let p_1 and p_2 be the price set by companies 1 and 2 respectively. The customer buys from the company with the lower price (if both the companies sell at the same price, it will choose a company uniformly at random). The utility of company i is $p_i - c_i$ if it sells and zero otherwise.

You are given that c_1 and c_2 are chosen **independently** and **uniformly** between 0 to 1. Compute a **symmetric** Bayesian Nash equilibrium of this game. You may assume that the function mapping c_1 to p_1 and c_2 to p_2 are **strictly-increasing** and **differentiable**.