Missouri University of Science & Technology | Department of Computer Science
Fall 2023 | **CS 5408: Game Theory for Computing**

Project 1

Instructor: *Sid Nadendla* | **Due:** *October 30, 2023*

Instructions: Students who did not follow any of the following instructions will be ignored and a zero grade will be rewarded accordingly.

- The main goal of this assignment is to implement a Python package `gtclab` for representing and solving both normal and extensive games from scratch.
- You are **not** allowed to import any other Python library, other than the ones that are already imported in the code-base.
- You are also **not** allowed to add, move, or remove any files, or even modify their names.
- You are also **not** allowed to change the signature (list of input attributes) of each function.

Problem 1 Game Representation *40 pts.*

Implement each of the following classes and methods listed below, to represent games in either normal[1], or extensive form representation:

(a) **Base Classes (20 pts):** These classes can be found in hw1/gtc-lab/base/

- `Player()`: This class contains the player label and their choice set. Therefore, define the following four static methods accordingly:
  - `set_choice_set()`: set the choice set for the player.
  - `get_choice_set()`: return the choice set.
  - `set_utility_matrix()`: set the utility matrix only for normal-form games.
  - `get_utility_matrix()`: return the utility matrix only for normal-form games.
- `State()`: This class is primarily used to define states in an extensive game. Therefore, define the following four static methods accordingly:
  - `set_player()`: set the player of a given state.
  - `get_player()`: return the player of a given state.
  - `set_info_set()`: set the label of information set to which the state belongs.
  - `get_info_set()`: return information set label for the given state.

[1] Representation of Bayesian normal-form games is beyond the scope of this assignment.

- `Tree()`: This class contains the tree data structure, along with player function, utilities, chance probabilities and chance probabilities. Therefore, define the following 18 static methods accordingly:
  - `create_state()`: Create a new state with a given `state_label` and include it in `self.states` dictionary only if the state is not already defined.
  - `add_state()`: If the given state is already defined outside the tree object, then add it to the `self.states` dictionary.
  - `add_player()`: Add the player with given label to `self.players` list.
  - `check_player_exists()`: Check if the player with a given label is already defined in `self.players` list.
  - `add_player_to_state()`: Find the player with the given player label. Then, using `set_player` method, set the found player to the given a state object.
  - `set_num_players()`: Set the given number of players to `self.num_players`, and also automatically create all the players and include them in `self.players` using `add_player()` method.
  - `get_num_players()`: Return `self.num_players`
  - `get_state()`: return the state for the given `state_label`.
  - `set_root()`: set the state of the given `state_label` as the tree's root.
  - `get_root()`: return the label of the tree's root node.
  - `is_parent()`: check if the state corresponding to a given `parent_state_label` is the parent of the state corresponding to a given `child_state_label`.
  - `set_child()`: set the state corresponding to the given `child_state_label` as a child to the state of the given `state_label`.
  - `get_children()`: return the labels of all child nodes for the state with a given `state_label`.
  - `is_leaf()`: check if the state corresponding to a given `state_label` is a leaf node in the tree.
  - `set_utilities()`: set the utilities of all players to a state corresponding to a given `state_label`, if it is a leaf node.
  - `get_utilities()`: return the utilities of all players, if the state corresponding to a given `state_label` is a leaf node.
  - `set_chance_prob()`: set the chance probability to the state corresponding to thee given `state_label`, if it is controlled by Nature.
  - `get_chance_prob()`: return the chance probability to the state of a given `state_label`, if it is controlled by Nature.

(b) **Models (20 pts):** These classes can be found in hw1/gtc-lab/models/

- `NormalGame()`: This class is designed to capture any complete-information normal-form game. Therefore, define the following two static methods accordingly:

- `__init__()`: This is a constructor that initializes an object of `NormalGame()` using three attributes, namely `num_players` (an integer that represents the total number of players in the game), `num_choices` (a dictionary that contains the total number of choices available at each player), and `utility_matrices` (a dictionary that contains utility matrices of each player). Formally, a normal form game contains three main attributes: Player set $\mathcal{N}$, Choice profile space $\mathcal{C}$ and utilities $\mathcal{U}$. The main goal of this function is to set these quantities within the objects of `Player()` class, using the respective attributes. Any errors within the representation (e.g. matrix dimensionality mismatch, incomplete representation) should be identified and appropriate flags need to be raised.
- `is_two_player_zero_sum()`: This static method checks whether/not the object of `NormalGame()` class represents a two-player, zero-sum game.

• `ExtensiveGame()`: This class is designed to capture any general extensive-form game. Therefore, define the following three static methods accordingly:

- `__init__()`: This is a constructor that initializes an object of `ExtensiveGame()` using one attributes, namely `tree` (an object of `Tree()` that contains the decision tree, player function, utilities, information sets as well as chance probabilities). Formally, an extensive form game contains three main attributes: Player set $\mathcal{N}$, Choice profile space $\mathcal{C}$ and utility space $\mathcal{U}$. The main goal of this function is to set these quantities within the objects of `Player()` class, using the respective attributes. Appropriate flags need to be raised whenever an error is identified by `is_tree_proper_extensive_game()`.
- `get_subgame()`: This method constructs a new extensive game using the subtree formed by the state corresponding to the given `state_label`.
- `is_tree_proper_extensive_game()`: This method is used to check if the input attribute `tree` in the class `ExtensiveGame()` represents a well-defined extensive game. Any error within the representation (e.g. incomplete representation, incorrect chance probabilities) should be identified and appropriate error type should be returned.

Problem 2 Solvers for Normal-Form Games ***30 pts.***

Implement each of the following classes and methods listed below, to solve normal-form games. Each of them are worth 6 points, and can be found in hw1/gtc-lab/solvers/

(a) `ieds()`: This class solves any complete-information normal-form game using *iterative elimination of dominated strategies* algorithm. Kindly implement the algorithm using the following two methods, as discussed below:

- `calc_reduced_game()`: In this method, calculate the reduced game upon completion of an entire round-robin of all players.
- `is_dominated()`: This method checks if a given choice is dominated by any other choice for a given player's utility matrix.

(b) `psme()`: This class solves two-player, zero-sum, complete-information normal-form games using *pure strategy minimax equilibrium* algorithm. If `is_two_player_zero_sum()` returns `FALSE`, return an error message stating that `psme()` cannot be used on the given game. Kindly implement the algorithm using the following method, as discussed below:

- `calc_psme()`: This method calculates psme and returns them. If the game does not have psme, return a flag with this information.

(c) `msme()`: This class solves two-player, zero-sum, complete-information normal-form games using *mixed strategy minimax equilibrium* algorithm. If `is_two_player_zero_sum()` returns `FALSE`, return an error message stating that `msme()` cannot be used on the given game. Kindly implement the algorithm using the following method, as discussed below:

- `calc_msme()`: This method calculates the mixed strategy minimax equilibrium for a given two-player zero-sum game.

(d) `psne()`: This class solves any complete-information normal-form game using *pure strategy Nash equilibrium* algorithm. Kindly implement the algorithm using the following method, as discussed below:

- `calc_psne()`: This method checks whether or not each choice profile is a Nash equilibrium using `is_best_response()` method, and returns all PSNE. If the game has no PSNE, then the method should print a message accordingly.
- `is_best_response()`: For a given utility matrix, this method checks if the given choice is the best response to the profile of all other players' choices.

(e) `msne_lp()`: This class solves two-player, zero-sum, complete-information normal-form games for *mixed strategy Nash equilibrium* using *linear programming* techniques. Kindly implement the algorithm using the following method, as discussed below:

- `calc_msne_lp()`: Using `cvxpy` package to solve linear programs, compute msne for two-player, zero-sum normal-form games. For more information, kindly refer to the following example found in cvx's github repository:

  `https://github.com/cvxpy/cvxpy/blob/master/examples/matrix_games_LP.py`

Problem 3 Validation ***30 pts.***

(a) **Iterative Elimination of Dominated Strategies (7 pts):** Write a Python script in Jupyter notebook that uses your implementation of `gtclab` package to compute

Iterative Elimination of Dominated Strategies algorithm for any general bimatrix game. Test your `ieds()` on the example provided in Problem 2 (shown below) in HW2. Validate your solution in HW2 using your own solver method.

	Left	Center	Right
Up	0,2	3,1	2,3
Middle	1,4	2,1	4,1
Down	2,1	4,4	3,2

(b) **Colonel Blotto Game (7 pts):** Write a Python script in Jupyter notebook that uses your implementation of `psne()` and `psme()` in `gtclab` package to compute both PSNE and PSME for the Colonel Blotto game discussed in Problem 1 of HW2. The program should print "This game has no PSNE/PSME," if the bimatrix game does not have a PSNE/PSME respectively. At the same time, if the game has multiple PSNE/PSME, it should report all the PSNE/PSME in the game. Compare your theoretical findings in HW2 to the output of your program.

(c) **Rock-Paper-Scissors (7 pts):** Write a Python script in Jupyter notebook that uses your implementation of `gtclab` package to compute *Mixed Strategy Nash equilibrium* and *Mixed Strategy minimax equilibrium* for rock-paper-scissors (RPS) game discussed in Problem 3 of HW2. Use `msne_lp()` and `msme()` solvers you implemented to find MSNE and MSME for RPS game respectively. Compare your theoretical findings in HW2 to the output of your program.

(d) **Extensive Game Representation (9 pts):** Write a Python script in Jupyter notebook that uses your implementation of `gtclab` package to implement the extensive game shown in Figure 1.

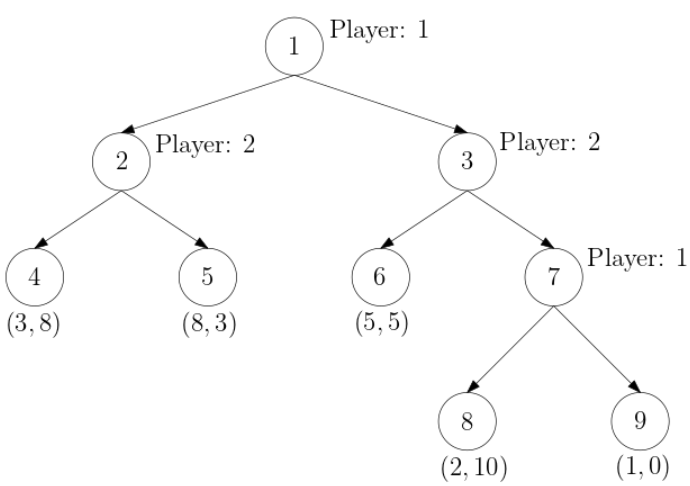

Figure 1: Extensive Game for Problem 3(d)