Game Theory Framework Help

Fabien Dufoulon

23/07/2015

Contents

1 Introduction

This work was written to help in the use of the Game Theory Framework application.

The Game Theory Framework application focuses on providing simulations of games, with players simulated by learning algorithms implemented in the application itself. It allows to play out scenarios which would be complicated to visualize theoretically, mostly in the case of multi-player scenarios, but also when the players or/and actions for each player are numerous.

The aim of this help is to explain the underlying structure used in the framework, as well as the way games and learning algorithms should be understood and expanded upon. Moreover, it should give an idea of why each structure of the project were used, and what compromise was struck in doing so.

2 Games

The Games are focused around the playTurn(), getRewards() and utility() functions. Another important functionality is the part that links games to the GameFactory. This part needs to be implemented in all instantiable derived classes, in order for GameFactory can instantiate this class properly.
The random generator of a game is default_random_engine, from the $C++$ STL, as it is rarely used.

2.1 Utility

The utility function takes in a vector of strategies describing what each player has chosen to do this turn, and outputs the rewards for each player in this given situation.

The utility function is specific to the problem and will need to be implemented in each class directly inheriting from the Game class.

As said above, the utility requires game-specific information. This information will need to be given to the constructor of the class, and stored in member variables. This is done mainly by giving the name of text files to the constructor of the class, in which the text files will be parsed.

It is possible to change the state of the game in utility, the playTurn function makes sure the updates are correct.

```
1   std::vector<double> Game2by2::utility(const std::vector<int> &strategies){
2       int k = (strategies[0]-1) * 2 + (strategies[1]-1);
3       std::vector<double> v = {payoffMatrixP1[k], payoffMatrixP2[k]};
4       return v;  }
```

2.2 GetReward

The getReward function takes in a vector of strategies describing what each player has chosen to do this turn, and a player, and outputs all the possible rewards the player could have received by changing the action chosen(amongst the available actions) and the reward returned by his chosen action, with every other player having the same action.

Normally, this function does not need to be overridden in derived classes. However, it happens when you need rewards from one arm to depend on the rewards from the other arms. An example is the ServerDistribution game, where the rewards are normalized so that the worst arm receives a loss of 1, therefore a reward of 0.

```
1 //Here, player is index, returns this player's possible rewards according to his available strategies.
2 std::vector<double> Game::getReward(int player, std::vector<int> strategies){
3     auto rewards = std::vector<double>(players[player].playerMaximumStrategyNumber, 0.);
4     for (auto &l : players[player].availableStrategies){ //Test out only the strategies available to the
          player. Other are given reward 0.
5         strategies[player] = l;
6         rewards[l-1] = utility(strategies)[player];
7     }
8     return rewards;
9 }
```

```
1  std::vector<double> ServerDistribution::getReward(int player, std::vector<int> strategies){
2      std::vector<double> rewards(Game::getReward(player, strategies));
3
4      //Normalize depending on trace or cost here
5      if (rewardTypeServerActivation == "COST") { //Take distance from smallest cost
6          double minEl = *std::min_element(std::begin(rewards), std::end(rewards));
7          std::transform(std::begin(rewards), std::end(rewards), std::begin(rewards),
8                         [&] (double d) {return d - minEl;}  );
9      }
10     //now normalize
```

```
    double maxEl = *std::max_element(std::begin(rewards), std::end(rewards));
    std::transform(std::begin(rewards), std::end(rewards), std::begin(rewards),
                   [&] (double d) {return 1 - d / maxEl;}  );
    return rewards;
}
```

2.3 PlayTurn

The playTurn function takes in a vector of strategies describing what each player has chosen to do this turn. It polls each player to obtain the strategy chosen by their current learning algorithm. It then updates every player with the vector of his possible rewards, then makes sure utility is called one last time to update the game properly if so needed.

This function can be overridden in derived classes, however it is interesting to call Game::playTurn in the override then code whatever is needed afterwards (Examples : MOA, MetaGame and ServerDistributionGame).

```
void Game::playTurn(){
    std::vector<int> strategies;
    strategies.reserve(players.size());
    //Get all of the strategies chosen by the current learning algorithm of each player.
    for (auto &p : players){
        //Update available strategies for sleeping bandits
        std::vector<int> availableStrategies(p.playerMaximumStrategyNumber);
        std::iota(std::begin(availableStrategies), std::end(availableStrategies), 1); //Range from 1 to
            maxStrategyNumber
        //Act upon available strategies here
        p.createAvailableStrategies(availableStrategies); //Update player available action set.

        //Get the strategy chosen by this player.
        strategies.push_back( p.getCurrentLearningAlgorithm()->choice(p) );
    }

    //Utility called a lot more than the timeStep
    //Return to player k all possible rewards for his action, knowing that all other players do not change
        their own actions.
    for (std::size_t k = 0; k < players.size(); ++k){
        players[k].updateRewards(getReward(k, strategies), strategies[k]);
    }
    //To have the game finish with the correct update if utility modifies the game state to return the
        appropriate information.
    utility(strategies);

    //Store the current time step.
    ++timeStep;
}
```

2.4 Factory Part

The derived classes who want to be instantiable by call to GameFactory need to possess a public static member function create with the correct signature as well as private static member variable from AlgorithmRegister templated with the derived class.
Normally, it is mostly sufficient to take the existing code from already existing header files and implementations files and modify it slightly to fit the needs of the new derived class(especially in the create function() definition, where some conditions can be defined on the number of inputs and so on).

The detailed explanation is that the GameFactory class holds a static member variable describing the mapping from a name to a function returning a newly created corresponding game instance. That function is the corresponding create function defined in the corresponding game derived class(Static member variables are accessible to all instances of a class, therefore the mapping is the same for all GameFactory instances no matter which one updates it).
This static member variable is also accessible by GameRegister<Template>, a derived class of GameFactory. Moreover, the constructor of GameRegister<Template> takes in a vector of names(ids) and associates the corresponding create function

defined in the Template class(This is why the signature of create is very important).
The constructor of all GameRegister instances is called when member variables of class GameRegister are defined, and static member variables require to be defined, therefore all static member variables of class GameRegister in game classes need to be defined in the implementation file so that they update the mapping in GameFactory.

After all this work, all that is needed to create a game instance is to call GameFactory::createInstance correctly(As a bonus, it is possible to associate multiple names to a single class so that it's easier to use).

The following code example is from Game2by2 :

```
//Header file
public:
  static Game* create(std::vector<std::string> inputFiles, int seed);
private:
  static GameRegister<Game2by2> reg;

//Implementation file
//If not enough input text files, return nullptr.
Game* Game2by2::create(std::vector<std::string> inputFiles, int seed){
    if (inputFiles.size() >= 2) return new Game2by2(inputFiles[0], inputFiles[1], seed);
    else return nullptr;
} //Initialize identifier of this game.
GameRegister<Game2by2> Game2by2::reg({"Game2by2", "G2x2"});
```

2.5 Regret print functions

The printRegret function takes in a file name and a sampling integer, and outputs the external regret of all players in the file. This regret is computed by comparing the highest cumulative reward possible by choosing the same action over and over again, to the cumulative reward of the player's algorithm.

The printInternalRegret has exactly the same input and outputs the regret of the player. This regret is computed by accumulating the comparative loss at each time step between the player's choice and the optimal action within the current game state.

2.6 Game information print function

The Game class holds a printInformation function, with the same input as above, that is to say a file name and sampling. As implemented in the base class, it does nothing.

This function is to be implemented if wanted in the derived base classes, in order to get information about the evolution of the game state for example. It can even be used to print the regret again in a more compatible format for the user.

2.7 Game basic initialization information

Any game simulating class needs a basic amount of information. This information is treated in the base Game class. The constructor takes in a text file describing the initialization parameters, the text file is parsed, and the information is used and stored in game. The constructor can also take in a extra parameter, a seed, to allow for fast instantiation of the same game with different seeds by the GameFactory.

The parameters common to every game are the maximum number of players at any time in the game, a seed, and for each player the maximum number of strategy available to him at any time step followed by the learning algorithms he has access to. The first learning algorithm in the list is chosen as his current algorithm, and currently there is no way to customize the change of algorithms from the text file.

```
number of players
seed
MaxStrategy player 1 + whitespace + LearningAlgorithms list(separated by spaces)
....
same thing for last player
```

Therefore, all derived classes' constructors require such a text file as input, but will call upon a base class constructor to process the file.

When writing new game derived classes, it is very helpful to check out the other parse constructors in order to get a good idea of what methods are given by the Game class(for example a method to split string according to a certain character) and to see what kind of parse treatment has already been done before. The constructors already implemented work on the following principle :

1. First, surround the parse of a file itself by a try-catch block, which will allow you to know which file exactly is non conform, if exceptions are prepared in the surrounded code (That means you need to put the exceptions yourself for special stuff, like making sure that the number of channels and the number of descriptions of these channels match).

2. Then, inside that try-catch block, use the getLine and split method to efficiently parse the text file. These methods will allow other users to have an easy time following your code.

3. Last, throw exceptions when there is an implicit constraint in the game parameters (With : throw std::logic_ error(string)) or in the previous step parse(for example if you know the line is supposed to be split into exactly four parameters).

An example of a constructor parsing extra information in a relatively simple manner can be found in Game2by2.

```
1  CongestionGame::CongestionGame(const std::string &initSettings, const std::string &userSettings, const std::
       string &weightSettings, int seed) : ResourceAccessBased(initSettings, userSettings, seed)
2  {
3      //Buffer for parsing text files
4      std::string strOneLine;
5      std::vector<std::string> vect;
6
7      try{
8          std::ifstream inFile(weightSettings.c_str());
9
10         //Player weights parse
11         for (int i = 0; i < numberOfPlayers(); i++){
12             getline(inFile, strOneLine);
13             vect = split(strOneLine, ' ');
14             //get mapping for each player
15             if (vect.size() != 1 ) throw std::logic_error( "Weights not well defined." );
16             playerWeights.push_back( std::stoi(vect[0]) );
17         }
18     } catch(const std::exception& e){
19         std::cout << "Check out Weight Settings file." << std::endl;
20         std::cout << e.what() << std::endl;
21     }
22 }
```

3 Player

The player has a very specific goal in the Game Theory Framework. His goal is to store all information on the rewards, the strategies he has chosen and the algorithms he has access to. However, he has no information on the game played, or on how to take advantage of the structure of the game. The mapping from the finite state space the player has access to, to the real actions in the game is implemented in the game class(in each derived game class).

The player receives feedback from the game in the form of rewards. That means the learning algorithms associated to a player will seek to maximize the rewards received by the player. The player will also store the regret(external and internal) throughout the game. All these informations are stored in order have a better understanding of how the game developed. The player has access to multiple algorithms but only has one current algorithm(defined by index). However, he can be made to change learning algorithms.
So that algorithms are not shared between players through the copy of pointers, player instance are forbidden from being copied.

Finally, in order to implement a sleeping bandit variation, the player holds a vector describing all the available actions, that is to say some actions can be forbidden during a certain period of time.

```
class Player
{
    public:
        /// Maximum number of strategies available to player
        int playerMaximumStrategyNumber;
        /// Strategies available to the player.
        std::vector<int> availableStrategies;

        /// Current time step of the player in the game (different from that of the game if the player went to sleep).
        int timeStep;
        /// Vector of the number of times the player has played one strategy(=index+1).
        std::vector<int> numberOfTimesPlayed;
        /// Vector of average rewards of player when he has played one strategy(=index+1).
        std::vector<double> averageRewards;

        /// Vector of the strategies chosen in the past.
        std::vector<double> pastStrategies;
        /// Vector of the possible payoffs if only this player had changed his move. [0] is zero payoff, [t] corresponds to the t turn payoff.
        std::vector<std::vector<double>> possibleRewards;
        /// Vector of the sum of the rewards if the player only played a specific strategy.
        std::vector<double> sumRewards;
        /// Reward accumulated by player with his algorithm.
        double sumRewardsAlgorithm;

        /// Vector of learning algorithms available to the player.
        std::vector<LearningAlgorithms*> playerLearningAlgorithms;
        /// Index of the current learning algorithm of the player(0 to n-1).
        int playerCurrentLearningAlgorithm;
        /// Vector of probability distribution of the player.
        std::vector<std::vector<double>> probabilities;

        /// Vector of the external regret of the player over time.
        std::vector<double> regrets;
        /// Vector of the average external regret of the player over time.
        std::vector<double> averageRegrets;
        /// Vector of the internal regret of the player over time.
        std::vector<double> intRegrets;
        /// Vector of the internal regret of the player over time.
        std::vector<double> averageIntRegrets;
        /// Vector of the internal regret of the player over time.
        std::vector<double> worstRegrets;
        /// Vector of the internal regret of the player over time.
        std::vector<double> averageWorstRegrets;
};
```

```
class Player
{
    public:
        /// The player is initialised with the maximum number of strategies available to him and the
              learning algorithms he has access to.
        /// He starts here with the first algorithm by default.
        Player(int maxStrategy, std::vector<LearningAlgorithms*> learningAlgorithms, int startAlgorithmIndex
              = 0);
        virtual ~Player() = default;

        //Move operators defined as default
        Player(Player&&) = default;
        Player& operator=(Player&&) = default;

        //Forbid Copying players
        Player(const Player&) = delete;
        Player& operator=(const Player&) = delete;

        /// Returns a pointer to the currently used learning algorithm.
        LearningAlgorithms* getCurrentLearningAlgorithm() const;
        /// Returns a pointer to the currently used learning algorithm.
        int getLastStrategyChosen() const;
        /// Changes the vector of available strategies.
        void createAvailableStrategies(std::vector<int> availableStrategies);

        /**
         Updates most of the local variables, that is to say the knowledge of the player,
         so that learning algorithms have access to that through a reference to the player.
        */
        void updateRewards(std::vector<double> reward, int correspondingStrategy);

        /// Calculates the external regret based on past history and possible rewards.
        void updateExternalRegret();
        /// Calculates the internal regret based on past history and possible rewards.
        void updateInternalRegret();
        /// Calculates worst possible internal regret
        void updateWorstRegret();
        ///Change the current learning algori
};
```

4 Learning Algorithms

Learning algorithms are all based on the same structure. The core function of any learning algorithm is choice(). It returns a strategy (Warning : strategy and the index corresponding to a strategy are to be understood as $index = strategy - 1$).
Moreover, another important fact is that learning algorithms possess a probability vector as a member variable(p), this allows the player to retrieve this information for his own use. The random generator of the learning algorithms is pcg32, from the PCG C++ library.

The main structure of the choice function is :

- Use the getVector() function to retrieve the appropriate information from the corresponding player, and change the probability vector accordingly.
- Then, transform the probability vector through transformVector() (For example, here Hedge uses an exponential).
- After that, normalize the probability vector through normalizeVector(). This gives us a vector whose elements sum up to one, so it can be treated as a probability vector.
- Finally, decide according to the probability vector.

```
int LearningAlgorithms::choice(const Player &player){
    if (player.timeStep == 0) p = std::vector<double>(player.playerMaximumStrategyNumber,0.);

    getVector(player);
    transformVector(p);
    normalizeVector(p);
    return decide(p);
}
```

The learning algorithms classes inheritance tree is structured according to the getVector() function. For example, learning algorithms using the average rewards of the player derive from BestResponseAverage, whereas learning algorithms using the vector of possible rewards of a player derive from RandomizedMajorityMyopic. Moreover, if the vector does not need to be transformed after the getVector() function, then either use the LearningAlgorithms implementation of it or override a previous implementation by writing an empty transformVector() function.

A possible idea would be to implement templates, in order to avoid this kind of inheritance tree, as it could be argued that inheritance over getVector() will cause problem if we want to reduce the number of times we use the same transformVector() function.

4.1 GetVector

This function takes a player reference as input and extracts information from it. Its return type is void, that means that it is supposed to modify p. It is also supposed to take care of the initialization conditions and of the fact that non available actions have a probability set to 0.
A sliding-window mechanism can be implemented here (Ex : RandomizedMajorityMyopic, FollowTheLeader and Hedge). The following convention has been used : a negative or null window is used to denote the fact that a window is not used.

The following code examples are from BestResponseAverage and FollowTheLeader respectively :

```
void BestResponseAverage::getVector(const Player &player){
    auto initNecessary = false;
    for(auto &strat : player.availableStrategies){
        if (player.numberOfTimesPlayed[strat-1] == 0){
            initNecessary = true;
            break;
        }
    }
    if (initNecessary) getInitialVector(player);
    else{
        std::fill(std::begin(p), std::end(p), 0.);
        for(auto &strat : player.availableStrategies){
            p[strat-1] = player.averageRewards[strat-1];
        }
    }
}
```

This function can do more than just recover information from a player. For example, UCB and Exp3 do not hold the same internal structure.

4.2 TransformVector

This function takes a vector reference and transforms it. This can generally done nicely through application of a std::transform operation and the use of a lambda function.

A code example can be found in the description of Hedge. Moreover, if the learning algorithm needs to always return the element with the highest probability, then it is possible to use convertVectorMaxProbabilist() :

```
void LearningAlgorithms::convertVectorMaxProbabilist(std::vector<double> &v){
    // Iterator to max element
    auto MaxEl = *std::max_element(std::begin(v), std::end(v));

    //Choose probabilistically between the tied max elements (uniform)
    for (int i = 0; i < v.size(); i++){
        v[i] = (v[i] == MaxEl) ? 1 : 0;
    }
    normalizeVector(v);
}
```

4.3 NormalizeVector

This function takes a vector reference and normalizes it. It is implemented in LearningAlgorithms and thus does not need to be reimplemented again in the derived classes.

The following code example is from LearningAlgorithms :

```
void LearningAlgorithms::normalizeVector(std::vector<double> &v){
    //Takes highest element then normalizes vector by the highest element.
    auto normalTerm = std::accumulate(std::begin(v), std::end(v), 0.0 );
    if (normalTerm == 1) return;
    std::transform(std::begin(v), std::end(v), std::begin(v),
                   [&](double d){ return d / normalTerm; } );
}
```

4.4 Decide

This functions decide which strategy to return according to the input probability distribution. There also shouldn't be any need to override this function.

The following code example is from LearningAlgorithms :

```
int LearningAlgorithms::decide(const std::vector<double> &v){
    std::uniform_real_distribution<double> distribution(0.0,1.0);

    double proba = distribution(rng);

    double sumProba = 0;
    int l = 1;
    for (auto it : v){
        sumProba += it;
        if (sumProba > proba) {
                strategyChosen = l;
                return strategyChosen;
        }
        l++;
    }

    //not supposed to reach here
    strategyChosen = v.size();
    //std::cout << "out" << p.size() << std::endl;
    return strategyChosen;
}
```

4.5 Update

The learning algorithms all possess an update() function. The base implementation is an empty function.
It is used for example in Exp3 to update the weights according to the reward given by the previous chosen arm.

4.6 Initialization

The base LearningAlgorithms class implements a method called getInitialVector(). Basically, if there are actions that have not been played, this function will give equal probability to all such actions. This needs to be called in getVector().

```
void LearningAlgorithms::getInitialVector(const Player &player){
    std::fill(std::begin(p), std::end(p), 0.);

    for(auto &strat : player.availableStrategies){
        if (player.numberOfTimesPlayed[strat-1] == 0) p[strat-1] = 1;
    }
}
```

4.7 Factory Part

There is also a Factory mechanism which allows the games to easily instantiate learning algorithms. It works similarly to the mechanism in the game classes, however the signature for the create function is a bit different. Moreover, the corresponding classes are now AlgorithmFactory and AlgorithmRegister<Template>. However the basic principle is the same.

The following code example is from Best Response Average :

```
//Header file
public:
  static LearningAlgorithms* create(Game* creator, std::vector<double> params);
private:
  static AlgorithmRegister<BestResponseAverage> reg;

//Implementation file
//Always create instance.
LearningAlgorithms* BestResponseAverage::create(Game* creator, std::vector<double> params){
    return new BestResponseAverage(creator);
}  //Initialize identifiers of this algorithm.
AlgorithmRegister<BestResponseAverage> BestResponseAverage::reg({"BRA", "BestResponseAverage", "
      BESTRESPONSEAVERAGE"});
```

5 Learning Algorithms Encyclopedia

Here, we will describe what parameters the implemented learning algorithms need, so that users who wish to use the Game Theory Framework do not need to look in the source code for the specific behaviours.

5.1 Partial Information

5.1.1 BestResponseAverage

This learning algorithm derives directly from LearningAlgorithms(for the moment at least). It requires each arm to be played at least once at the beginning. Afterwards, it chooses the arm with the highest average reward. It overrides both getVector() and transformVector().

5.1.2 Logit

This learning algorithm derives from Logit. It also requires each arm to be played at least once at the beginning. Instead of choosing the highest average reward, it chooses probabilistically after doing an exponential transformation on the average rewards(Overrides transformVector).

5.1.3 UCB family

5.1.3.a ScoreUCB

This class derives from LearningAlgorithms. It requires each arm to be played at least once at the beginning. It transforms the average rewards to model a confidence bound. It chooses probabilistically. It overrides getVector().

5.1.3.b UCB

This class derives from ScoreUCB, and is representative of the basic UCB algorithm. It also requires each arm to be played at least once at the beginning. It transforms the average rewards to model a confidence bound and it chooses the highest score value. It overrides transformVector().

5.1.3.c TemporalUCB

This class derives from ScoreUCB, and is representative of the basic UCB algorithm. It also requires each arm to be played at least once at the beginning. It transforms the average rewards to model a confidence bound and it chooses the highest score value. It overrides getVector().

5.1.3.d DiscountUCB

This class is representative of the UCB algorithm with discounted variables. Therefore it derives from UCB and overrides both getVector() and update(). It has a parameter gamma, which should be between 0 and 1.

5.1.3.e KLUCB

This class is an abstract class. It allows to derive multiple algorithms with different distances but the same concept. There are currently two derived classes. The first implements the Bernoulli distance(KLUCBBernoulli), and the second the Gaussian distance(KLUCBGaussian).

5.1.4 Exp3 family

5.1.4.a Exp3

This class derives from LearningAlgorithms. It is representative of the Exp3 algorithm.

5.1.4.b Exp31

This class derives from Exp3. It is representative of the Exp31 algorithm.

5.1.5 Epsilon-greedy family

5.1.5.a Epsilon-greedy

This class derives from LearningAlgorithms. It is representative of the ϵ-greedy algorithm. It has one parameter, ϵ, which is contained between 0 and 1 and is the probability of choosing one random available action. Otherwise, it will choose the action with the highest average rewards.

5.1.5.b EpsilonN-greedy

This class derives from EpsilonGreedy. It is representative of the ϵN-greedy algorithm. This class' epsilon parameter decreases over time proportional to the square of the time. It has two parameters, c and d, which model the rate of decrease of epsilon. The rest of the algorithm works just like the ϵ-greedy algorithm.

5.2 Full Information

5.2.1 RandomizedMajorityMyopic

This class is the base class of all classes needing to extracts information from the vector of all the possible rewards the player could have received when changing his action compared to a certain set of actions chosen by the other players. It sums the possible rewards over time for each arm, and adds a window mechanism to the sum. This sum is then averaged by either the time step or the window. It also takes care of choosing only available actions.

This class overrides the getVector() function to extract this information and have a window. If the class has a window member variable of 0 or less, then there is no window mechanism.

5.2.2 FollowTheLeader

This class derives from RandomizedMajorityMyopic. It overrides the tranformVector() function, where it gives the arm with the highest sum of possible rewards a probability of one.

```
1  void FollowTheLeader::transformVector(std::vector<double> &v){
2      convertVectorMaxProbabilist(v);
3  }
```

5.2.3 Hedge

This class also derives from RandomizedMajorityMyopic. Like FollowTheLeader, it also overrides the tranformVector() function, where it shifts to an exponential representation of the sum of all possible rewards for each arm.

```
1  void Hedge::transformVector(std::vector<double> &v){
2      std::transform(std::begin(v), std::end(v), std::begin(v),
3              [&](double d) -> double { return exp(eta * d); } );
4  }
```

6 Game Encyclopedia

Here, we will describe what file format the implemented games need, so that users who wish to use the Game Theory Framework do not need to look in the source code for the specific behaviours.

6.1 Two player Game

This game simulates a two player game(strategic form). The utility function is implemented through pay-off matrices. The elements of the matrices need to be written in a text file in the same format as below :

```
u1(1,1) u1(1,2)
u1(2,1) u1(2,2)
u2(1,1) u2(1,2)
u1(2,1) u2(2,2)
```

The first four elements are the rewards for the first player, and the others for the second player. $u(i, j)$ is the reward if the first player plays action i and the second action j.

6.2 ResourceAccessBased Game

Abstract class used to model games using resources. The main purpose of this class is to factor the code of the constructor, in order for it not to be copied across multiple resource based games. This class takes in an extra text file in the constructor, which defines the number of resources and the user access to resources.

```
  5 (Total number of resources)
1 2 5 (Set of resources accessible to the first player)
....
1 2 3 4 5 (Set of resources accessible to the last player)
```

6.3 Congestion Game

This class models congestion games, and derives from ResourceAccessBased. Each player have a given weight in this game, non-dependant on the particular resource accessed. When multiple players access the same resource, the given reward is divided. This class takes in an extra text file in the constructor, which defines the weights of every player.

```
1 (Weight of the first player)
....
6 (Weight of the last player)
```

6.4 Multichannel Opportunistic Game (MOA)

This class models the Multichannel Opportunistic Access Game, also referred to as Dynamic Spectrum Access. It derives from ResourceAccessBased. This is supposed to model a network where a set of primary users are already using some channels, with their actions changing over time, and where we are studying a secondary set of players with lesser rights trying to exploit the remaining capacity of the network channels without collisions.

There are multiple resources available and the players need to choose one amongst those they can access. The resources are modeled by two-state Markov Chains, with one state being the "Bad State"(0) and the other the "Good State"(1).
If multiple players choose the same resource, they both receive no reward at all(There is also theoretically a variant where they can share the resources). Moreover, even if there is only one player on a resource but that resource is in state 0, then they receive nothing.
Finally, at the end of a turn, all resources undergo a transformation based on their underlying Markov Chain transition probabilities.

The specific elements to this game in the constructor is a single text file, which describes the Markov Chain transition probabilities of the resources.

```
  p00 p01 p10 p11 (First resource)
...
  p00 p01 p10 p11 (Last resource)
```

With $p_{0,0}$ the transition from state 0 to itself, $p_{0,1}$ from 0 to 1, $p_{1,0}$ from 1 to 0, and $p_{1,1}$ the transition from state 1 to itself.

6.5 Random Multichannel Opportunistic Game (RMOA)

This class is derived from the previous class, the Multichannel Opportunistic Access game. The only change is that the transitions are not done on resources which are not chosen by any player.
There are no changes on the input text files compared to MOA.

6.6 ResourceSimplePayoff

This class models a game where players have access to a set of resources which always give out a certain pay-off, regardless of the number of players(for each player). It derives from ResourceAccessBased.

The specific elements to this game in the constructor is a single text file, which describes the pay-offs of all resources.

```
  p_1
...
  p_n
```

With p_i the pay-off of resource i.

6.7 Server Distribution

6.7.1 Server Distribution with predictions variant

6.8 MetaGame

This class is derived from Game directly. It is also a friend of the Game class. This class only requires a Game object pointer in addition to the normal text file required for games.

The concept behind this class is that each player of this game has a number of actions equal to the number of learning algorithms of the corresponding player in the game inside the pointer. Therefore, he chooses which algorithm the corresponding player will use at each time step.

The rewards for each action corresponding to a learning algorithm is the rewards received by the action chosen by the corresponding player with this specific learning algorithm. (Not up to date)

6.9 MassLaunchGame

Although technically not a game, this class allows the user to launch a game multiple times with a variable amount of seeds. This class therefore does not need the normal game initialization file, however it is given a file with the seeds it will use later on, as input of the constructor.
MassLaunchGame has a createSeeds static member function. This allows users to easily create files with a huge number of seeds automatically, without having to instantiate a MassLaunchGame instance(but still store the function somewhere near MassLaunchGame). The file will contain the number of seeds specified, as a sequence beginning at the starting seed input, with an unitary increment.
The most important function of MassLaunchGame is play(). In play(), MassLaunchGame will create the first game with the first seed. It will be played the total number of turns, the information taken care of, then MassLaunchGame will throw the game away and start using the next seed to create another instance. The averaging the internal and external regret over all the various games is done little by little, before MassLaunchGame throws away each game.

Using MassLaunchGame is very easy. If a file with seeds has not been created, it can be created through the createSeeds static member function, then we can instantiate a MassLaunchGame instance and use play. Finally, using printRegret() and printInternalRegret() will output text files we can then plot. The following example is used to launch a Multichannel Opportunistic Access game with 1000 seeds and 1000 turns, and then print out 100 points amongst the regrets.

```
1     std::string initSettings = "initSettings.txt";
2     std::string userSettings = "userSettings.txt";
3     std::string channelSettings = "channelSettings.txt";
4     std::string resourceSettings = "resourceSettings.txt";
5     std::string extraSeedSettings = "extraSeedSettings.txt";
6
7     MassLaunchGame::createSeeds(extraSeedSettings, 1000, 200);
8     std::vector<std::string> inputFiles = {initSettings, userSettings, channelSettings};
9
10    MassLaunchGame massLaunch = MassLaunchGame(extraSeedSettings);
11    int playSize = 1000;
12    massLaunch.play("MOA", inputFiles, playSize, 1);
13
14    massLaunch.printInternalRegret("intDataMass.txt", std::ceil(playSize/100.));
15    massLaunch.printRegret("intDataMassExt.txt", std::ceil(playSize/100.));
```

The first file "initSettings.txt" is written according to the following format.

```
number of players
seed
MaxStrategy player 1 + whitespace + LearningAlgorithms list(separated by spaces)
....
same thing for last player
```

The second "userSettings.txt" has this format :

```
5 (Total number of resources)
1 2 5 (Set of resources accessible to the first player)
....
1 2 3 4 5 (Set of resources accessible to the last player)
```

The third "channelSettings.txt" has this format :

```
p00 p01 p10 p11 (Probability transitions of first channel)
...
p00 p01 p10 p11 (Probability transitions of last channel)
```

And "extraSeedSettings.txt" has this format :

```
10 12 35 46 98 85 14 (seed list)
```