Reinforcement and Online Learning Coursework

Dimitri Diomaiuta - 30598109

University of Southampton

1 Introduction

The aim of this project is to develop algorithmic strategies to play the Lemonade Stand Game (LSG). The LSG is a three agents game with a 12 positions setting, distributed like the hours on a clock. At every round, each player chooses a spot to put his stand on, $A_i = P_1, ..., P_{12}$. Every player chooses at the same time, without knowing the choice of the other players at the current round since communication between agents is not allowed. After each round, the utility of a player is calculated as the sum of the distances between the closest clockwise player and the closest anticlockwise player. The only exception to this is when two or all players are sitting on the same spot. In the first case the two players sitting in the same spot collect a quarter of the total revenue of a single round while the third collects half of it. In the second case each agent collects a third of the total revenue. The total points given at each round depend on how much points are assigned for a single spot. We use the U_{spot} variable to indicate this measure and R_{total} to indicate the total revenue of a single round. Equation 1 shows how the total number of points for a round are calculated. Equation 1 multiplies the total number of points that the spots generate by two since every point is collected by two agents. Assigning 1 to the unit spot variable leads to a total revenue per round of 24, while assigning 6 to the spot variable leads to a 144 total revenue.

$$R_{total} = U_{spot} \times positions \times 2 \tag{1}$$

The aim of each agent is, hence, to be as far as possible from the other players, in order to maximize his individual total revenue for the n rounds played. In this paper, we describe and evaluate different algorithmic strategies to play the Lemonade Stand Game.

2 Algorithm design

In this section we describe the design of the implemented algorithms. This section is divided into two subsection, each describing a different class of algorithms. The code of the implemented strategies can be found in section 5.

2.1 Game agnostic algorithms

This class of algorithms includes strategies that reduce the game to a multi-armed bandit setting. These algorithms are based on evaluative feedback of the actions taken and do not have any internal representation of the game or its rules. Given an array of actions, the value of each of them, $q_*(a)$, at time step t is the expected reward R_t returned when that action is taken, as equation 2 shows [1].

$$q_*(a) \doteq \mathbb{E}(R_t|A_t = a) \tag{2}$$

The aim of a multi-armed bandit algorithms is to maximize the total reward by deriving an estimated value for each action at time step t, $Q_t(a)$, as close as possible to the real value $q_*(a)$. Exploration and exploitation techniques are fundamental for this class of algorithms. The former allows to obtain more accurate actions estimators. The latter allows to maximize the reward by exploiting the current knowledge state.

Exp3 algorithm The Exp3 algorithm implements an instance of the multi-armed bandit algorithms family [2]. Given K actions, the Exp3 algorithm keeps track of K-length weight and probability vectors, initialized respectively at 1 and $1/K$. The probability vector assigns a probability to each action that is calculated according to the weight given to that action. Actions with higher weights obtain an higher probability. At every turn the action is selected according to the probability vector. The reward of the selected action is then used to update the weight of that action by using the exponential distribution. The exponential distribution increments the weights of the highest-reward actions guaranteeing exploitation. On the other hand, exploration is guaranteed by the gamma parameter, $\gamma \in (0, 1]$, which describes the probability to draw a uniform random action when selecting it. The external regret, the bound describing the difference from the optimal algorithm, of Exp3 is limited to $O(\sqrt{KTlog(K)})$, which makes it theoretically efficient.

Epsilon greedy algorithm The epsilon greedy algorithm (ϵ-greedy) is another type of multi-armed bandit algorithm [3]. Given K actions, ϵ-greedy keeps track of frequency and rewards vectors, initialized both at 0. The frequency vector keeps track of how many times an action has been selected. The rewards vector contains a value estimation of the actions. The epsilon parameter, $\epsilon \in (0, 1]$, guarantees that the algorithm balances between exploration and exploitation. Epsilon greedy selects a random action, exploration, with probability ϵ and greedily selects the action with higher value with probability $1-\epsilon$, exploitation. The reward of the selected action is then averaged with the frequency of that action and the previous estimated value to update the rewards vector.

2.2 Heuristic based algorithms

This class of algorithms includes strategies that have an internal representation of the game. The internal knowledge of the game is exploited, via heuristic functions, in a strategic manner in order to maximize the total reward.

Stick and follow algorithm This algorithm implements a simple, and yet, effective heuristic-based strategy. It is based on a simplified version of the ACT-R cognitive architecture [5]. In the initial phase of the game, the first 5 rounds, the algorithm chooses a random action (1 of the 12 available positions) and sticks to it. This is done in order to attract another cooperative player to select the opposite position and, as a result, maximize both utilities. The second phase of the game strategy is determined by an heuristic strategy. The algorithm analyzes the reward of the last performed action and compares it to a fixed utility threshold (implemented as 7 when $U_{spot} = 1$). This leads to the following two cases:

1. *Last reward* $>$ *threshold*: the algorithm sticks to the current position.
2. *Last reward* $\leq$ *threshold*: the algorithm picks an opponent player randomly and sits opposite to his last position.

The design of the heuristic function guarantees that the player does not continue sticking in a poor reward position and limits collusion of the other two players. In the case of the two other player colluding, the stick and follow strategy will break the their strategy by sitting on top of one of the two players (case number 2).

EA^2 algorithm This algorithm implements the winning strategy of the inaugural Lemonade Stand Game Tournament [4]. This heuristic based strategy has at its core the analysis of the opponents' history of game interactions in order to classify them and exploit their behaviour. The algorithm classifies the opponents' strategies into two generic classes: a stick and a follow strategy. The first one describes players that stick at one or a few locations for the entire duration of the game. The second one describes players that tend to sit opposite to other players in order to maximize their reward. The classification is not binary and is computed by keeping indices on the opponents behaviour. A stick index defines how close a player is to the ideal stick strategy, see equation 3. A general follow index defines how close a player is to the ideal follow strategy, see equation 5. Two additional player specific follow indices describe which of his opponents the player is following, see equation 4.

$$s_i = -\sum_{k=2}^{t-1} \frac{\gamma^{t-1-k}}{\Gamma} d(a_i(k), a_i(k-1))^p \tag{3}$$

$$f_{ij} = -\sum_{k=2}^{t-1} \frac{\gamma^{t-1-k}}{\Gamma} d(a_i(k), a_j^*(k-1))^p \tag{4}$$

$$f_i = -\sum_{k=2}^{t-1} \frac{\gamma^{t-1-k}}{\Gamma} \min_{j=N\setminus i} [d(a_i(k), a_j^*(k-1))]^p \tag{5}$$

where t represents the current time step, $\Gamma = \sum_{k=2}^{t-1} \gamma^{t-1-k}$, $d(a_i(k), a_j(k-1))$ represents the smaller distance between the current position of player i and the previous position of player j, while $a_j^*(k-1)$ describes the position opposite to player j. The γ response parameter is used to weight past events, like in the reinforcement learning temporal difference method. The parameter p defines the ideal type acceptance interval, whether to include in strategy classification noisy players that play similarly to an ideal strategy.

The indices are computed at each time step and they are needed to select the next action. The heuristics used to select the next action are described by the following ordered condition rules:

1. Player i is a sticker more than player j. Furthermore, his stick index is bigger than its follow index and player j one $\Rightarrow$ sit opposite to player i.
2. Both opposite players have high stick index $\Rightarrow$ stick if utility bigger than 8 otherwise sit opposite to the player with higher stick index.
3. Player i is a follower more than player j $\Rightarrow$ if player i is following j sit on player j otherwise stick.
4. Players i and j are following each other $\Rightarrow$ choose the opponent with highest follow index and sit on his position.
5. Players i and j are sitting opposite to each other $\Rightarrow$ sit on the opponent with lower stick index to make it move.
6. None of the above applies $\Rightarrow$ stick in current position.

The heuristic rules cover a large array of different possible situations and strategies to exploit them and maximize the agent total reward.

3 Evaluation

The implemented strategies have been extensively tested under different scenarios with different players. We chose a game setting consisting of 100 rounds and $U_{spot} = 6$, meaning that the total reward per round is 144. We used the following additional strategies for testing purposes:

– *Random*: Strategy that at every round selects an action following a uniform distribution.
– *Constant*: Strategy the sticks with the same action every round. If present, the subscribed number following the constant strategy indicates the action number.

The results were produced by iteratively running the same 100 rounds game setting for 50 times with different random seeds and then averaging the rewards. We can see the obtained results in table 1.

We can observe that in an ideal setting where both opponents sit on the same spot (Constant$_0$ + Constant$_0$) all the implemented strategies score close

Table 1. Evaluation results of the 4 implemented strategies

Opponents	Exp3	Egreedy	S&F	EA^2
Constant$_0$ + Constant$_0$	70.81	68.38	69.12	71.82
Constant$_0$ + Constant$_1$	61.09	61.84	65.96	65.78
Random + Random	48.18	48.29	48.20	49.27
Random + Constant	47.82	50.42	50.81	53.40
Random + EA^2	48.03	52.49	50.85	52.39
EA^2 + EA^2	40.35	43.89	48.82	48.21
Exp3 + Egreedy	48.09	50.37	50.26	52.87
S&F + EA^2	41.59	42.43	45.67	46.32
Constant$_0$ + Constant$_6$	36	36	36	36

to the maximum possible points (72). The difference from the maximum score can be address to the initial wait, for the heuristic-based algorithms, and to the exploration phase, for the multi-armed bandit algorithms. Both classes of algorithms perform well also against constant strategies playing different actions. Heuristic-based algorithms internal knowledge of the game allows them to realize the situation faster and sit opposite to one of the two constant players without having to test all of the possible positions. This explains their better performance in this scenario.

Random opponents make all the algorithm perform quite poorly and score, approximately, an average of a third of the total points. Random strategies, in fact, results in multi-armed bandit algorithms weighting all the actions the same and heuristic-based algorithms not being able to exploit the opponents behaviour (random strategy is both far from the sticker and the follower ideal strategy).

Interesting test scenarios are the ones that include, in the opponents, at least a sticker strategy or a cooperative one (Random + Constant and Random + EA^2). In these cases we can deduct that all of the algorithms, except of the Exp3 strategy, exploit this setting to coordinate with the right opponent. Exp3 action selection, subject to probabilities given by the weight vector, does not allow to continuously exploit the best position as Egreedy argmax action selection does. We can also note that EA^2 player indices allow the algorithm for faster convergence to collusion, while Egreedy and Stick and Follow strategies are slowed down, respectively, by exploration and random opponent selection.

Playing against two opponents that could possibly cooperate is another important test scenario (EA^2 + EA^2 and Stick and Follow + EA^2). We can observe heuristic-based strategies performing better at avoiding the collusion of the two opponents. This is because those strategies do not need to try different positions, once detected the situation of the utility drop they can immediately react with the best strategic action to break the opponents synchronization.

Overall, from the different scenarios outlined in table 1, we can infer that the heuristic-based methods perform better than the multi-armed bandit class. Game heuristics allow the algorithms to react to different type of situations with fixed effective strategies that lead to reward maximization. We also have to note that Egreedy algorithm shows a cooperative behaviour in a different array of situations even without having an internal game knowledge. Results also show that there is not effective strategy against two players colluding without changing their strategy or considering the third player.

4 Conclusion

In this paper we have described the design and evaluation of different strategies to play the Lemonade Stand Game. We have analyzed both multi-armed bandit and heuristic-based class of algorithms. The results suggest that having internal knowledge of the game increases the players performance. Complex heuristics lead to an even more consistent improvement, as the EA^2 algorithm performing better than all the other strategies shows.

Further research regarding multi-armed bandit algorithms for this setting should consider weighting the actions not only based on the obtained reward but also on the board configuration when that action was taken in order to exploit opponents patterns. On the other hand, the research regarding the heuristic-based algorithms should aim at increasing the classes of the type of players in order to model and exploit opponents that model strategies that differ from a sticker and a follower type.

References

1. Sutton, R.S. and Barto, A.G., 2011. Reinforcement learning: An introduction.
2. Auer, P., Cesa-Bianchi, N., Freund, Y. and Schapire, R.E., 2002. The nonstochastic multiarmed bandit problem. SIAM journal on computing, 32(1), pp.48-77.
3. Kuleshov, V. and Precup, D., 2014. Algorithms for multi-armed bandit problems. arXiv preprint arXiv:1402.6028.
4. Sykulski, A.M., Chapman, A.C., De Cote, E.M. and Jennings, N.R., 2010, January. EA2: The Winning Strategy for the Inaugural Lemonade Stand Game Tournament. In ECAI (pp. 209-214).
5. Reitter, D., Juvina, I., Stocco, A. and Lebiere, C., 2010. Resistance is futile: Winning lemonade market share through metacognitive reasoning in a three-agent cooperative game. Proceedings of the 19th behavior representation in modeling & simulation (BRIMS). Charleston, SC.

5 Appendix A: source code

This appendix section contains the source code of the implemented algorithms.

5.1 Exp3

```
package mystrategy;

import game.Game;
import game.Strategy;
import game.Utility;

import java.util.Random;
import java.util.Arrays;
import java.lang.Math;
import java.util.stream.*;


/**
 * Reducing the Lemonade Stand Game to a multi-armed bandit
     problem
 * and solve it via Exp3 method.
 * @author Dimitri Diomaiuta
 */
public class Exp3Strategy implements Strategy{

  static final double GAMMA = 0.2;
  int constantAction;
  Utility utility;
  int numActions; // actions
  double[] weights;
  double selectedProb;

  public Exp3Strategy(Game g, long seed, String options) {
    this.numActions = g.getNumActions();
    Random r = new Random(seed);
    constantAction = r.nextInt(numActions);
    this.utility = g.getUtility();
    weights = new double[numActions];
    Arrays.fill(weights, 1);
    selectedProb = 0;
  }

  // Uses categorical distribution to get the action index
  public int getActionIndexFromProbabilities(double[]
      probabilities) {
    double cumulative = 0;
    double rand = Math.random();
    for(int i = 0; i < probabilities.length; i++) {
      cumulative += probabilities[i];
      if(cumulative > rand) {
        selectedProb = probabilities[i];
        return i;
      }
    }
    selectedProb = probabilities[probabilities.length - 1];
    return probabilities.length - 1;
  }

  public int getAction() {
    double weightsSum = DoubleStream.of(weights).sum();
    double[] probabilities = new double[numActions];
    for(int i = 0; i < weights.length; i++) {
      probabilities[i] = (1 - this.GAMMA) * (weights[i] /
          weightsSum) + (this.GAMMA * 1.0 / numActions);
    }
    int action = getActionIndexFromProbabilities(probabilities);
    return action;
  }

  /* Given the actions of all the players it returns an array
      which
     size is the size of possible actions. At each index contains
         the
     possible revenue if that action number (which is the index) ,
         had
     been taken. Intuitevely we can access our utility by
     possibleUtilities[actionTaken] */
  public int[] getPossibleUtilities(int[] actions){
    int[] possibleUtilities=new int[numActions];
    for(int i=0;i<possibleUtilities.length;i++){
      actions[0]=i;
      possibleUtilities[i]=utility.getUtility(actions)[0];
    }
    return possibleUtilities;
  }

  public void updateWeights(int currentAction, int currentReward)
      {
    double estimatedReward = (currentReward / numActions) /
        selectedProb;
    weights[currentAction] *= Math.exp(estimatedReward * this.
        GAMMA / numActions);
  }

  public void observeOutcome(int[] actions) {
    // Trying to get the utility I got out of the selected actions
         (actions[0])
    int currentAction = actions[0];
    int[] possibleUtilities = getPossibleUtilities(actions);
    int currentReward = possibleUtilities[currentAction];
    updateWeights(currentAction, currentReward);
  }

}
```

5.2 Egreedy

```
package mystrategy;

import game.Game;
import game.Strategy;
import game.Utility;

import java.util.Random;
import java.util.Arrays;
import java.lang.Math;


/**
 * Reducing the Lemonade Stand Game to a multi-armed bandit
     problem
 * and solve it via Egreedy method.
 * @author Dimitri Diomaiuta
 */
public class Egreedy implements Strategy{

  int constantAction;
  Utility utility;
  int numActions;
  static final double EPSILON = 0.1;
  double[] rewards, myfrequencies;

  public Egreedy(Game g, long seed, String options) {
    this.numActions = g.getNumActions();
    Random r = new Random(seed);
    constantAction = r.nextInt(numActions);
    this.utility = g.getUtility();
    rewards = new double[numActions];
    myfrequencies = new double[numActions];
    Arrays.fill(rewards, 0);
    Arrays.fill(myfrequencies, 0);
  }

  private int argmax(double[] rewards) {
```

```
  int indexOfLargest = 0;
  for(int i = 1; i < rewards.length; i++) {
    if(rewards[i] > rewards[indexOfLargest]) {
      indexOfLargest = i;
    }
  }
  return indexOfLargest;
}

public int getAction() {
  if (Math.random() > this.EPSILON) {
    int indexOfLargest = argmax(rewards);
    myfrequencies[indexOfLargest] += 1;
    return indexOfLargest;
  }
  int rand = new Random().nextInt(numActions);
  myfrequencies[rand] += 1;
  return rand;
}

/* Given the actions of all the players it returns an array
    which
   size is the size of possible actions. At each index contains
       the
   possible revenue if that action number (which is the index) ,
       had
   been taken. Intuitevely we can access our utility by
   possibleUtilities[actionTaken] */
public int[] getPossibleUtilities(int[] actions){
  int[] possibleUtilities=new int[numActions];
  for(int i=0;i<possibleUtilities.length;i++){
    actions[0]=i;
    possibleUtilities[i]=utility.getUtility(actions)[0];
  }
  return possibleUtilities;
}

private void updateRewardVector(int currentReward, int action) {
  double n = myfrequencies[action];
  double prevReward = rewards[action];
  rewards[action] = ((n - 1) / n) * prevReward + (1 / n) *
      currentReward;
}

public void observeOutcome(int[] actions) {
    // Trying to get the utility I got out of the selected actions
         (actions[0])
    int currentAction = actions[0];
    int[] possibleUtilities = getPossibleUtilities(actions);
    updateRewardVector(possibleUtilities[currentAction],
        currentAction);
  }

}
```

5.3 Stick and Follow

```
package mystrategy;

import game.Game;
import game.Strategy;
import game.Utility;

import java.util.Random;
import java.lang.Math;


/**
 * A strategy sticks with and utility threshold. If the last
     utility
 * is less than the threshold the next action will be sitting
     opposite
 * to a random opponent.
 * @author Dimitri Diomaiuta
 */
public class StickAndFollow implements Strategy{
  Utility utility;
  int numActions, myLastUtility, numberOfPlays, myAction;
  static final int UTILITY_THRESHOLD = 7;
  int[] opponentsLastAction;

  public StickAndFollow(Game g, long seed, String options) {
    this.numActions = g.getNumActions();
    Random r = new Random(seed);
    myAction = r.nextInt(numActions);
    this.utility = g.getUtility();
    myLastUtility = numberOfPlays = 0;
    opponentsLastAction = new int[2];
  }

  public int getAction() {
    if(numberOfPlays < 1) {
      numberOfPlays++;
    } else if(myLastUtility <= UTILITY_THRESHOLD) {
      // Take a random opponent and sit opposite to its previous
          spot.
      if(Math.random() < 0.5) {
        myAction = (opponentsLastAction[0] + (numActions / 2)) %
            numActions;
      } else {
        myAction = (opponentsLastAction[1] + (numActions / 2)) %
            numActions;
      }
    }
    return myAction;
  }


  /* Given the actions of all the players it returns an array
      which
     size is the size of possible actions. At each index contains
         the
     possible revenue if that action number (which is the index) ,
          had
     been taken. Intuitevely we can access our utility by
     possibleUtilities[actionTaken] */
  public int[] getPossibleUtilities(int[] actions){
    int[] possibleUtilities=new int[numActions];
    for(int i=0;i<possibleUtilities.length;i++){
      actions[0]=i;
      possibleUtilities[i]=utility.getUtility(actions)[0];
    }
    return possibleUtilities;
  }

  public void observeOutcome(int[] actions) {
    int currentAction = actions[0];
    opponentsLastAction[0] = actions[1];
    opponentsLastAction[1] = actions[2];
    int[] possibleUtilities = getPossibleUtilities(actions);
    myLastUtility = possibleUtilities[currentAction];
  }

}
```

5.4 EA^2

```
package mystrategy;

import game.Game;
import game.Strategy;
import game.Utility;

import java.util.Random;
import java.util.ArrayList;
import java.lang.Math;


/**
 * Reimplementation of the EA2 strategy.
 * @author Dimitri Diomaiuta
 */
public class EA2 implements Strategy{
  int constantAction;
  Utility utility;
  int numActions;
  EA2Player playerI, playerJ;
  static final double smallGamma = 0.75; // The response rate
  static final double smallP = 0.5; // The scal parameter
  static final double tol = 0.1; // The tollerance for the
      conditions to hold
  static final int initialStick = 6;
  static final int T = 2;
  int stickCounter;
  int numberOfPlays; // The t parameter
  ArrayList<Integer> myHistory;
  double bigGamma;
  int currentUtility;
  int lastAction;

  public EA2(Game g, long seed, String options) {
    this.numActions = g.getNumActions();
    Random r = new Random(seed);
    lastAction = constantAction = r.nextInt(numActions);
    this.utility = g.getUtility();
    playerI = new EA2Player();
    playerJ = new EA2Player();
    stickCounter = initialStick; // stick for the first 5 moves
    numberOfPlays = 0;
    myHistory = new ArrayList<Integer>();
    bigGamma = 0;
    currentUtility = 0;
```

```
}

public int getAction() {
  // Resetting initial stickCounter
  if(stickCounter > 0) { // C0: stick
    stickCounter--;
    return lastAction;
  }
  // C1: Player i has higher stick index that its follow index
      and player j stick or follow index --> sit opposite player
       i
  if(playerI.stickIndex > playerI.followIndex - tol && (playerI.
      stickIndex > playerJ.stickIndex - tol || playerI.
      stickIndex > playerJ.followIndex - tol)) {
    return EA2Player.oppositeLocation(playerI.history.get(
        playerI.history.size()-1));
  }
  // C2: Both player i and j have high stick indices
  if(playerI.stickIndex > playerI.followIndex - tol && playerJ.
      stickIndex > playerJ.followIndex - tol) {
    // C2.1: utility > 8 --> Stick and set stickCount to T
    if(currentUtility > 8) {
      // We assume initialStick is T
      stickCounter = T;
      return lastAction;
    }
    // C2.2: utility <= 8 --> sit opposite the stickiest and set
         stickCount to T
    stickCounter = T;
    return playerI.stickIndex > playerJ.stickIndex ? EA2Player.
        oppositeLocation(playerI.history.get(playerI.history.
        size()-1)) :
       EA2Player.oppositeLocation(playerJ.history.get(playerJ.
           history.size()-1));
  }
  // C3: Player i has higher follow index than stick index and
      player j stick or follow index.
  if (playerI.followIndex > playerI.stickIndex - tol && (playerI
      .stickIndex > playerJ.stickIndex - tol || playerI.
      stickIndex > playerJ.followIndex - tol)) {
    // C3.1: player i is following me --> stick
    if(playerI.follow0 > playerI.followOther) {
      return lastAction;
    }
    // C3.2: player i is following j --> sit on j
    return playerJ.history.get(playerJ.history.size()-1);
  }
  // C4: Both player i and j have high follow indices and are
      following each other --> sit on the opponent with the
      highest follow index
  if(playerI.followIndex > playerI.stickIndex - tol && playerJ.
      followIndex > playerJ.stickIndex - tol
     && playerI.followOther > playerI.follow0 && playerJ.
        followOther > playerJ.follow0) {
    return playerI.followIndex > playerJ.followIndex ? playerI.
        history.get(playerI.history.size()-1) :
        playerJ.history.get(playerJ.history.size()-1);
  }
  // C5: Players i and j are sitting opposite to each other -->
      play carrot and stick --> sit on opponent with lower stick
       index
  if(EA2Player.oppositeLocation(playerI.history.get(playerI.
      history.size() - 1)) == EA2Player.oppositeLocation(playerJ
      .history.get(playerJ.history.size() - 1))) {
    return playerI.stickIndex < playerJ.stickIndex ? playerI.
        history.get(playerI.history.size()-1) :
        playerJ.history.get(playerJ.history.size()-1);
  }
  // C6: No condition satisfied
  return lastAction;
}

/* Given the actions of all the players it returns an array
    which
   size is the size of possible actions. At each index contains
       the
   possible revenue if that action number (which is the index) ,
        had
   been taken. Intuitevely we can access our utility by
   possibleUtilities[actionTaken] */
public int[] getPossibleUtilities(int[] actions){
  int[] possibleUtilities=new int[numActions];
  for(int i=0;i<possibleUtilities.length;i++){
    actions[0]=i;
    possibleUtilities[i]=utility.getUtility(actions)[0];
  }
  return possibleUtilities;
}

// Updates the follow and stick indices of the other two players
```

```
  public void updateIndeces() {
    this.bigGamma += Math.pow(smallGamma, numberOfPlays - 1);
    playerI.updateStickIndex(smallGamma, bigGamma, numberOfPlays,
        smallP);
    playerJ.updateStickIndex(smallGamma, bigGamma, numberOfPlays,
        smallP);
    playerI.updateFollowIndices(smallGamma, bigGamma,
        numberOfPlays, smallP, playerJ.history, myHistory);
    playerI.updateFollowIndices(smallGamma, bigGamma,
        numberOfPlays, smallP, playerI.history, myHistory);
  }

  public void observeOutcome(int[] actions) {
    myHistory.add(actions[0]);
    playerI.history.add(actions[1]);
    playerJ.history.add(actions[2]);
    numberOfPlays++;
    if(numberOfPlays > 1) {
      updateIndeces();
    }
    lastAction = actions[0];
    int[] possibleUtilities = getPossibleUtilities(actions);
    currentUtility = possibleUtilities[lastAction];
  }

}

package mystrategy;

import java.util.ArrayList;
import java.lang.Math;

public class EA2Player {

  double stickIndex, followIndex, followOther, follow0;
  ArrayList<Integer> history;
  static double upperLimit;

  public EA2Player() {
    stickIndex = followIndex = followOther = follow0 = 0;
    history = new ArrayList<Integer>();
    upperLimit = 12;
  }

  public static int oppositeLocation(int location) {
    int half = (int) upperLimit / 2;
  return location >= half ? location - half : location + half;
}

public void updateStickIndex(double smallGamma, double bigGamma,
     int numberOfPlays, double smallP) {
  double result = 0;
  for(int k = 1; k < numberOfPlays; k++) {
    double x = Math.pow(smallGamma, numberOfPlays - k) /
        bigGamma;
    double diff = (upperLimit + history.get(k) - history.get(k
        -1)) % upperLimit;
    double distance = diff < upperLimit - diff ? diff :
        upperLimit - diff;
    result += x * Math.pow(distance, smallP);
  }
  stickIndex = -result;
}

public void updateFollowIndices(double smallGamma, double
    bigGamma, int numberOfPlays, double smallP, ArrayList<
    Integer> historyOther, ArrayList<Integer> history0) {
  double followOtherResult = 0;
  double follow0Result = 0;
  double followIndexResult = 0;
  for(int k = 1; k < numberOfPlays; k++) {
    double x = Math.pow(smallGamma, numberOfPlays - k) /
        bigGamma;
    // Updating follow player J index
    double diffWithOther = (upperLimit + history.get(k) -
        oppositeLocation(historyOther.get(k-1))) % upperLimit;
    double distanceWithOther = diffWithOther < upperLimit -
        diffWithOther ? diffWithOther : upperLimit -
        diffWithOther;
    followOtherResult += x * Math.pow(distanceWithOther, smallP)
        ;
    // Updating follow player 0 index
    double diffWith0 = (upperLimit + history.get(k) -
        oppositeLocation(history0.get(k-1))) % upperLimit;
    double distanceWith0 = diffWith0 < upperLimit - diffWith0 ?
        diffWith0 : upperLimit - diffWith0;
    follow0Result += x * Math.pow(distanceWith0, smallP);
    // Updating general follow index
    followIndexResult += x * Math.pow(Math.min(distanceWithOther
        , distanceWith0) , smallP);
  }
```

```
    followOther = -followOtherResult;
    follow0 = -follow0Result;
    followIndex = -followIndexResult;
  }

  public String toString() {
    return "Player␣{\n␣␣␣␣stickIndex:␣" + stickIndex + "\n␣␣␣␣
        followIndex:␣" + followIndex + "\n␣␣␣␣followOther:␣" +
        followOther + "\n␣␣␣␣follow0:␣" + follow0 + "\n␣␣␣␣history
        :␣" + history + "\n}";
  }

}
```