https://github.gatech.edu/jliu727/Project3

# *RoboCup – Correlated-Q*

Jiaxin Liu
Department of Computer Science
Georgia Institute of Technology
Georgia, U.S.
jliu727@gatech.edu

***Abstract*—This article looks into Correlated-Q Learning method introduced by Amy Greenwald in 2003 in paper *Correlated-Q Learning*. Here we use the method described in the paper to replicate the result on a grid game -- Soccer Game mentioned in part 5 of the paper.**

***Keywords—correlated-q, friend-q, foe-q, q-learning, markov game, correlated equilibria.***

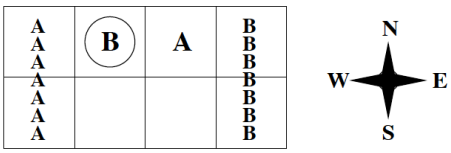
## I. INTRODUCTION

In this part of course, we learned Markov Games and methods of finding the best policies using different methods. In this paper, the author introduces CE-Q (Correlated-Q), a Q-learning algorithm for multi-agents, utilizing correlated equilibria solution to find equilibrium policies. CE-Q generalizes both Nash-Q and Friend-Foe-Q each in general-sum game and constant-sum game. Therefore, it is applicable in more classes of games than other algorithms. Also CE (Correlated Equilibria) can be computed easily using linear programming and achieve higher rewards than NE. There are four variants in choosing from multiple equilibrium and payoff – utilitarian, egalitarian, republican and libertarian. In this soccer game, we are using utilitarian in choosing the correlated equilibrium. In this project, we will reproduce the soccer game Amy did in her paper to show performance of CE-Q compared to other methods.

## II. ROBOCUP ENVIRONMENT

In this paper, the problem Amy looked at is the Soccer Game. It is a zero-sum game where there exist no deterministic equilibria. Unlike the previous projects, we need to construct the environment in this case.

The Soccer Game consists of a soccer field (the environment) and two players. The soccer field is a 2x4 grid field as shown below. The two players are A and B and the initial state of the game is B at (0,1) holding the ball and A at (0,2). There are five possible actions for each player, heading North, South, East, West or Stick (stay at current position). Grids (0,0) and (1,0) are goals for A and grids (0,3), (1,3) are goals for B. If any of the player holding the ball moves into the goal area, the player corresponding to the goal will score 100 points while the other player will get -100 points. Whenever a goal is scored, the game ends. At time t, both players initiate an action and the two actions are executed in random order. When a collision occurs, whoever take the position first stay at the position and the other player remains at its original position. Note here that a special case is that if the player trying to move into an occupied grid while holding a ball, the ball would be transferred to the other player. Also, I am treating bouncing into the wall as staying still at its original position.




In order to implement this game, we need to define two classes in environments, the Soccer_Game and the Player. Player is a simple class and has 4 attributes: x, y (position), ball (whether the player has the ball or not) and pid (the player's id). Soccer_Game realizes the main function of this environment. First, we initialize soccer game with several attributes to define the size of the field, two players (initialized as the graph above) and their respective goals in the field. Also, we define action space to be 0 (N), 1 (S), 2 (E), 3 (W), 4 (Stick) and a flag done to monitor if the current game is done or not.

In this class, there are 4 methods. First, since we need to run this game multiple episodes for the experiment, method *reset()* help reset everything to the initial state (graph above). Then, we need to check if a goal has been scored for each move. Method *check_goal(pid, y)* checks status of each player after each move to see if they have scored or not. If there is a score, the done flag is set to different values according to the type of scoring made. There are four of them: A scores goal A, A scores goal B,

B scores goal A and B scores goal B. We need rewards to update Q values in the learning process, method *get_reward()* returns reward for both A and B according to the scoring type revealed from the done value.

Finally, the most important method *move(pid, direction)* takes in a pid (indicating which player to move) and its moving direction and outputs information on pid, previous position, current position (after moving), rewards and whether or not the game is done. This method handles all the edge cases such as collision, ball transfer and hitting boundary mentioned above.

## III. IMPLEMENTATION & ALGORITHM

In this experiment, we want to compare performance of CE-Q with Friend-Q, Foe-Q and Q-Learning. Therefore, we need implement agent for each of these algorithms.

First, we need to determine that the total number of states. We label position in the soccer field as 0,1,2,3,4,5,6,7:

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |

There are three variables defining a unique state on the field, the position of A, the position of B and possession of the ball. In our setting, if A holds the ball, we mark ball possession as 0, otherwise, 1. An example would be the start state, A at (0,2) and B at (0,1) and B holding the ball. The state would be coded as 112. In this case, we have 8*7*2 possible states. We label state number from 0,1 … 110, 111 corresponding to state coding 001, 002 … 175, 176.

We implement Q-Learning based on the multi-agent O-Learning provided in the paper. There are two Q tables of size 112x5 (number of possible states x number of possible actions) each for player A and player B to track its own Q values. It is very similar to the simple Q-learning where agents choose random action epsilon percent of time and choose the optimal action according to the Q table otherwise. We execute pairs of actions and update Q value by the following equation:

$$Q_A(s, a_A) = (1-\alpha) * Q_A(s, a_A) + \alpha * ((1-\gamma) * R_A + \gamma * V^*(s'))$$

where $V^*(s') = \max_{a \in A(s')} Q^*(s', a)$

We then check the goal condition to see if the game ends or not.

We implement Friend-Q based on logic from Michael Littman's paper *Friend-or-Foe Q-learning in General-Sum Games*. In this algorithm, the other player is identified as friendly, therefore, seeking coordination equilibrium. This algorithm is off-policy so we take uniformly random actions. In this algorithm, we has a Q table of size 112x5x5 (number of possible states x number of possible A actions x number of possible B actions) as we need to consider movements of both A and B. while updating the Q value, we simply pick $\max_{\vec{a} \in A(s')} Q^*(s', \vec{a})$. Note here different from the general Q learning, we are using action pairs $\vec{a}$ which is $(a_A, a_B)$ to get and update Q values. After updating, we follow the same routine as Q-Learning to check if the game ends or not.

Foe-Q is implemented assuming the other player is foe and would always try to minimize its opponents score. In this case, we need to solve a minmax problem. We need to find the maximum of all the minimum rewards from each action for a particular state. Similar to Friend-Q algorithm, we need to create a Q table of size 112x5x5. The Q value for a single state looks like:

| | $0_B$ | $1_B$ | $2_B$ | $3_B$ | $4_B$ |
|---|---|---|---|---|---|
| $0_A$ | $Q_{00}$ | $Q_{01}$ | $Q_{02}$ | $Q_{03}$ | $Q_{04}$ |
| $1_A$ | $Q_{10}$ | $Q_{11}$ | $Q_{12}$ | $Q_{13}$ | $Q_{14}$ |
| $2_A$ | $Q_{20}$ | $Q_{21}$ | $Q_{22}$ | $Q_{23}$ | $Q_{24}$ |
| $3_A$ | $Q_{30}$ | $Q_{31}$ | $Q_{32}$ | $Q_{33}$ | $Q_{34}$ |
| $4_A$ | $Q_{40}$ | $Q_{41}$ | $Q_{42}$ | $Q_{43}$ | $Q_{44}$ |

The first column and first rows are action 0,1,2,3,4 (N, S, E, W, Stick) for player A and B. Here we would like to look for $P_{0A}, P_{1B}, P_{2A}, P_{3A}, P_{4A}$ to achieve the max V assuming B is foe. We have the following constraints:

$$\begin{gathered}
Q_{00} * P_{0A} + Q_{10} * P_{1A} + Q_{20} * P_{2A} + Q_{30} * P_{3A} + Q_{40} * P_{4A} \geq x \\
Q_{01} * P_{0A} + Q_{11} * P_{1A} + Q_{21} * P_{2A} + Q_{31} * P_{3A} + Q_{41} * P_{4A} \geq x \\
Q_{02} * P_{0A} + Q_{12} * P_{1A} + Q_{22} * P_{2A} + Q_{32} * P_{3A} + Q_{42} * P_{4A} \geq x \\
Q_{03} * P_{0A} + Q_{13} * P_{1A} + Q_{23} * P_{2A} + Q_{33} * P_{3A} + Q_{43} * P_{4A} \geq x \\
Q_{04} * P_{0A} + Q_{14} * P_{1A} + Q_{24} * P_{2A} + Q_{34} * P_{3A} + Q_{44} * P_{4A} \geq x \\
P_{0A}, P_{1B}, P_{2A}, P_{3A}, P_{4A} \geq 0
\end{gathered}$$

https://github.gatech.edu/jliu727/Project3

$$P_{0A} + P_{1A} + P_{2A} + P_{3A} + P_{4A} = 1$$

We solve these set of constraints using linear programming from cvxopt to find the $P_{0A}, P_{1B}, P_{2A}, P_{3A}, P_{4A}$ that would maximize x. This x is the V we are looking for to update Q value.

Now we are left with the last implementation of Correlated-Q learning. In this algorithm, we need to two Q tables of size 112x5x5 each for player A and player B because they will need to come up with a correlated equilibrium together. The Q table for a single state looks like this:

| | $0_B$ | $1_B$ | $2_B$ | $3_B$ | $4_B$ |
|---|---|---|---|---|---|
| $0_A$ | $Q_{00}$ | $Q_{01}$ | $Q_{02}$ | $Q_{03}$ | $Q_{04}$ |
| $1_A$ | $Q_{10}$ | $Q_{11}$ | $Q_{12}$ | $Q_{13}$ | $Q_{14}$ |
| $2_A$ | $Q_{20}$ | $Q_{21}$ | $Q_{22}$ | $Q_{23}$ | $Q_{24}$ |
| $3_A$ | $Q_{30}$ | $Q_{31}$ | $Q_{32}$ | $Q_{33}$ | $Q_{34}$ |
| $4_A$ | $Q_{40}$ | $Q_{41}$ | $Q_{42}$ | $Q_{43}$ | $Q_{44}$ |

Q-Table A

| | $0_B$ | $1_B$ | $2_B$ | $3_B$ | $4_B$ |
|---|---|---|---|---|---|
| $0_A$ | $Q_{00}$ | $Q_{01}$ | $Q_{02}$ | $Q_{03}$ | $Q_{04}$ |
| $1_A$ | $Q_{10}$ | $Q_{11}$ | $Q_{12}$ | $Q_{13}$ | $Q_{14}$ |
| $2_A$ | $Q_{20}$ | $Q_{21}$ | $Q_{22}$ | $Q_{23}$ | $Q_{24}$ |
| $3_A$ | $Q_{30}$ | $Q_{31}$ | $Q_{32}$ | $Q_{33}$ | $Q_{34}$ |
| $4_A$ | $Q_{40}$ | $Q_{41}$ | $Q_{42}$ | $Q_{43}$ | $Q_{44}$ |

Q-Table B

We want to find correlated equilibrium using linear programming again from cvxopt. The first constrain looks like this:

$$Q_{00} * P_{00} + Q_{01} * P_{01} + Q_{02} * P_{02} + Q_{03} * P_{03} + Q_{04} * P_{04} \geq Q_{10} * P_{00} + Q_{11} * P_{01} + Q_{12} * P_{12} + Q_{13} * P_{13} + Q_{14} * P_{14}$$

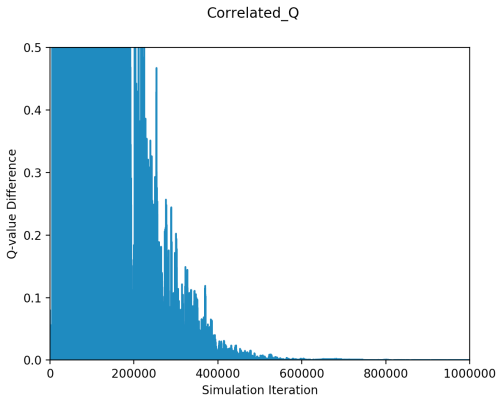
If we move left of the constraint to the right, we get:

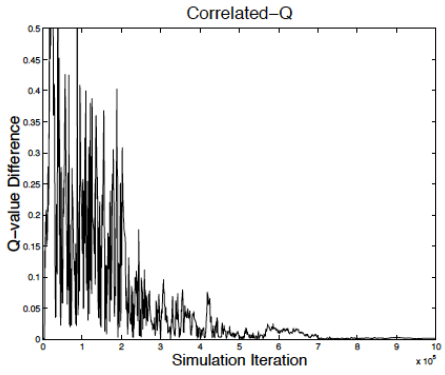
$$(Q_{00} - Q_{10}) * P_{00} + (Q_{01} - Q_{11}) * P_{01} + (Q_{02} - Q_{12}) * P_{02} + (Q_{03} - Q_{13}) * P_{03} + (Q_{04} - Q_{14}) * P_{04} \geq 0$$

We create one constraint for each pair of rows and end up with 4*5 constraints for each Q-table. Combining with the constraints of all probabilities larger or equal to 0 and all probabilities add up to 1. We result in a total of 4*5*2 + 25 = 65 constraints and one equation. Here we are trying to maximize the sum of the players' rewards. Therefore we solve for the $P_{00}$, $P_{01}$, $P_{02}$ … $P_{42}$, $P_{43}$, $P_{44}$ that maximizes $V = (Q_{00B} + Q_{00A}) * P_{00} + (Q_{01B} + Q_{01A}) * P_{01} \ldots + (Q_{43B} + Q_{43A}) * P_{43} + (Q_{44B} + Q_{44A}) * P_{44}$. Finally, we use the calculated V to update both Q-tables.

## IV. Result Analysis

### Correlated-Q:

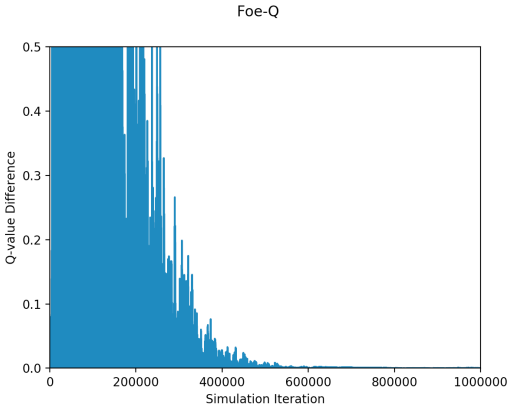


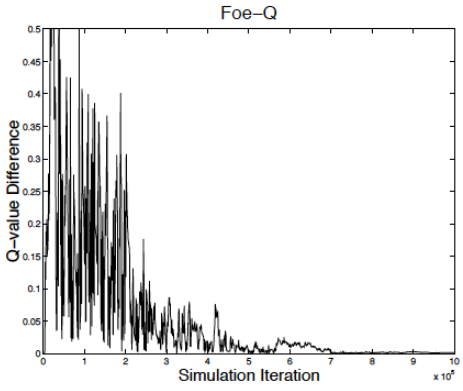




In this experiment, we want to investigate the Q-value convergence using Correlated-Q. The graph on the left is a replication of Amy's Figure 3(a) in the paper. The x axis is the number of iterations. The y axis is the Q-value absolute value difference. Similar to the graph from Amy's, we can see that the Q-value difference starts out very large and converges towards 0 after ~400000 iterations as expected. Here, we are using gamma = 0.9 and alpha starting from 0.8. According to the paper, the alpha gradually decreases to 0.001 during the learning process around 40000 - 500000 iterations. Therefore, we set the alpha decay exponentially around 0.99996 every episode. Looking more into the learned behavior, we see that in the initial state, the agent would primarily choose to head north(stick) or south. In this graph, we can see that our graph is denser than example graph from Amy's. It is possibly because Amy has eliminated some values (possibly all zero values) while we are plotting all the values from learning, creating a denser distribution. Correlated-Q also runs the slowest among all learning algorithms because of its calculation on all the constraints.

https://github.gatech.edu/jliu727/Project3

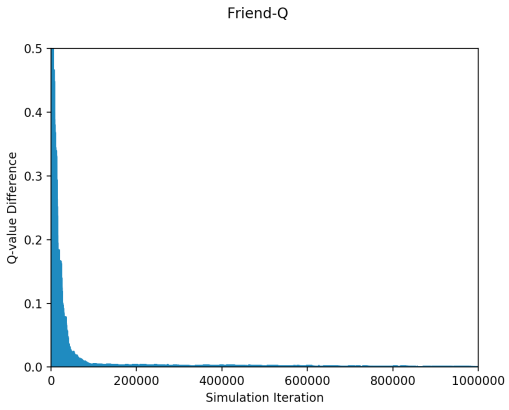
## Foe-Q:

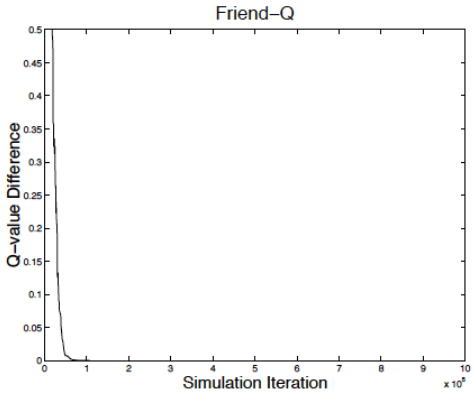



This experiment investigates the Q-value convergence using Foe-Q. The graph on the left is a replication of Amy's Figure 3(b) in the paper. We can see that the resulting graph shares a very similar trend to the graph from Amy's where the Q-value difference starts out high and exponentially decay to 0 around 400000-500000 iterations. Here, we are still setting gamma to 0.9 and alpha starting from 0.8. We are using a faster decay rate of 0.99995 until alpha decays to 0.01 as the update iterations for initial state per episodes is larger than Correlated-Q, hence requiring fewer episodes to converge. Here, we are seeing similar denser distribution as we see in Correlated-Q due to the same reason above. This algorithm generates same optimal policies as in Correlated-Q for initial states. Player A chooses to go south or stick(north) to gain max Q value in adversarial environment. Foe-Q runs the second slowest in all four algorithms also because of the linear programming calculation. It is a little faster than Correlated-Q due to the small amount of constraints it needs to calculate compared to Correlated-Q.

## Friend-Q:

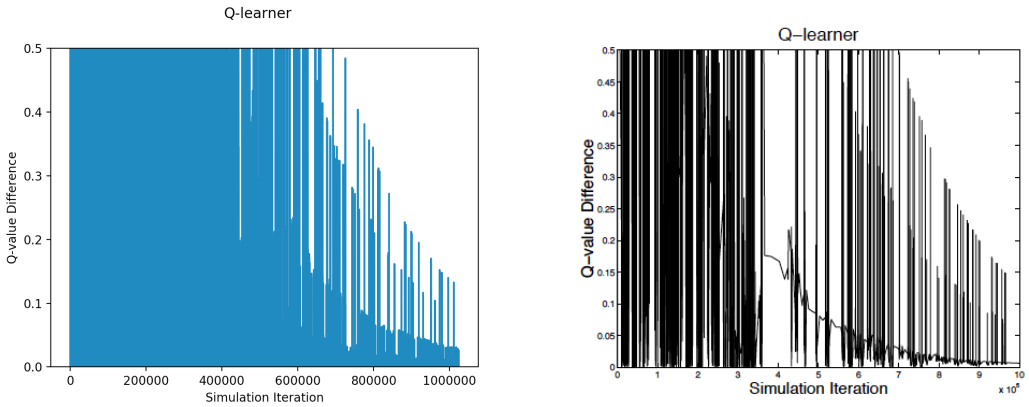



This experiment investigates the Q-value convergence using Friend-Q. The graph on the left is the replication of Amy's Figure 3(c) in the paper. We can see Friend-Q also converges exponentially as expected. It converges rather quickly around 100000 iterations similar to the graph from Amy's. Here we are setting gamma to 0.9 and alpha starting from 0.03. We exponentially decay alpha by a factor of 0. 9998 until it reaches 0.01 as specified in the paper. However, although it converges rather quickly, by looking into the Q value table for initial state, we can see that A randomly chooses any of the 5 actions. It is because in Friend-Q algorithm, A assumes that B would help it achieve a higher score and expects B to west directly to score A's goal. Therefore, it did not matter which direction A moves. Therefore, although it converges, it is converging to the wrong optimal policy as the soccer game is not a coordination game. Friend-Q runs very fast as there is no complicated calculation involved.

https://github.gatech.edu/jliu727/Project3

**Q-Learning:**




This experiment investigates the Q-value convergence using Q-learning. The graph on the left is a replication of Amy's Figure 3(d). We ca see that Q-value difference is not converging in the graph. It just gets cut off by the decaying learning rate. Here we are using a gamma of 0.9, alpha staring from 1 and epsilon starting from 1. We are exponentially decaying both alpha and gamma by a factor of 0.999964 each episode until they reach 0.001. The fact that it does not converge makes sense because Q-Learning only compute Q-value of its own. The final possible outcome for it would be a deterministic policy because it does not consider its opponent's behavior. However, soccer game is a zero-sum game that does not have a deterministic policy. Q learning also runs very fast as there is no complicated calculation and only simple updates to the Q table.

## V. Pitfalls & Problems

Among the four graphs, FoeQ and Correlated-Q is the hardest to implement for two reasons. First, the constraints need to be expressed in matrix form to be processed by cvxopt linear programming and the transform from constraints to matrix can be tedious. The second reason is that it takes a relatively long time for FoeQ and Correlated-Q to run and we do not see indication of a successful convergence until the middle of the process. Also, the paper did not clearly define if the iteration in the graph is a single step or an entire episode. At first, I thought it would be an entire episode, however, by looking closely to the graph for Q-learner, I decided it is a single step as in an episode, there could be several steps that updates a particular state, in that case, the total update of an episode could be outside the bound of the learning rate envelope so it makes more sense to treat iteration as single step.

## VI. Conclusion

From replication of this experiment, we can see that different scenario should require different algorithms. Correlated-Q apparently is applicable in a broader type of games but it takes too long to run. Q-Learner and Friend-Q can be very fast but they do not make sense outside a particular set of problems. Knowing the advantages and disadvantage of an algorithm can help us solve problems faster and more accurately.

## References

[1] A. Greenwald, K. Hall, "Correlated-Q Learning," 2003

[2] M. Littman. Friend or foe Q-learning in general-sum Markov games. In *Proceedings of Eighteenth International Conference on Machine Learning*, pages 322-328, June 2001.