Project III Report for COM S 4/5720 Spring 2025: A Q-Learing Implementation

Noah Miller[1]

***Abstract*— The problem of 3 chasing agents is difficult. These three agents, Tom, Jerry, and Spike, where Spike wants to attack Tom, Jerry wants to attack Spike, and Tom wants to attack Jerry. Not only is this problem being solved, but there is a chance that any action taken may be changed by an unknown probability. This paper details an implementation of Q-Learning as the planning algorithm of choice for these agents, as well as developments made along the way.**

I. INTRODUCTION

It is common for a path seeking problem involving agents to be used when understanding machine learning. Once past the basic understanding of one agent attempting to find the shortest path to a goal, is to introduce a moving goal. One step past that is to include someone to avoid as well. Once all that is met, we are left with the three agent problem. This is a problem in which the three agents, Tom, Jerry, and Spike, attempt to capture each other whilst avoiding obstacles on the map. Each round the map is different, so it needs to allow variance. From these turn, the things to consider are what moves are possible, and that Spike wants to capture Tom, Tom wants to capture Jerry, and Jerry wants to capture Spike. These considerations change the problem from a simple shortest path algorithm, to one needing more heuristics and understandings of the "best" path forward. The ultimate goal is to not be captured, but one can easily change the preference of actions based off scoring the winning states. These actions though, may be changed in an unknown probability by the managing algorithm. There are 3 changes, keeping the action the same, rotating to the left by 90-degree, and rotating to the right by 90-degree. As such the model must not calculate an algorithm for shortest path, but also one that will be the most likely to succeed given the randomness.

II. ALGORITHMIC DEVELOPMENT

The option considered for this problem was Q-Learning. Q-Learning is a learning algorithm that does not use a tree, but instead uses a table to store states, actions, and the scores of those combinations. As such, after learning, the best action is chosen with a lookup. Some of the benefits is that the training can be run again after learning more, and thus it can readjust its strategy as the probability charts get updated from attempted actions. This means that we can always keep updating the best case strategy rather quickly while still feeling confident in the models' ability to perform moves as the game continues. This table is updated by simulating games (50), at 30 total moves from all players combined to explore the next 10 rounds before retesting the next 10 rounds after updating probability spreads.

1) Implementation: This implementation again owes its credit to Professor Weng's sample code. This sample code was updated to no longer reflect a game of tic-tac-toe, but rather a game of agents chasing agents, but the template was used as the basis of the logic. The current implementation loops through 50 games, at 30 actions taking place. These games allow for the Q-table to be updated according to this scoring. 3 points if the acting player captured the pursued, -3 if they have been captured, 1 if the other agents have captured each other, 1 point if the number of rounds has been exceeded, -5 points if the agent runs into a wall, and finally 1 point if the other agents run into walls. These points (reward) is used in the following way.

$$Q(s,a,p) = Q(s,a,p) + \alpha(\text{reward} + \gamma q - Q(s,a,p))$$

s is the state, a is the action, p is the player taking the action, α is the learning constant, 0.5, γ is the discount factor, 0.9, q is the max next state score, and $Q(s,a,p)$ is the score of the current action. A state is this game is the position of each of the players, and an action is a movement 1 square in any direction, or standing still. This algorithm is once again naively implemented which will be discussed later, but that is how the Q-table gets updated. The first change from the tic-tac-toe implementation to the current was changing what actions can take place. Once these actions were understood, manipulating the other methods to be able to feed the information was important. Most methods needed to get updated to include the player acting as it is no longer just 2 and can be checked simply, but rather needed to be tracked throughout the entire process. The learning does still have a 10% chance to act randomly at any given point, allowing for training of non-optimal positions, as well as not always making non-optimal moves. The algorithm also implements the "mod_action" function much like in "main.py" during the testing rounds. The base probability is given at $\frac{1}{3}$ for each of the three options listen in the introduction. These probabilities get updated each time the model is called to generate one of the best move by updating the total count of each type of modification as well as the total number of rounds in which this information has been gathered. From these two facts, a new probability distribution can be generated. This method of updating is naive but very easy to keep track of, and more will be discussed in the discussion. The algorithm finally retrains itself every 10 times the model is called to generate the next action. This is done to take into account the new probability distribution

[1]Noah Miller with the Department of Computer Science, Iowa State University, Ames, IA 50013, USA nvmiller@iastate.edu

and update the states of each of the actors, giving a better chance the model can find better than random paths.

2) Testing: There was a very small amount of testing done in order to determine what iteration amounts to use. The solution started much higher, at 1000 games at 500 actions taking place, but this amount of computation was leading to results for the games taking far too long. As such the iteration amounts were slowly decreased first the number of games was reduced to 500 as well, and then down to 200, and then lowering the action amount to 300, and dropping again to 100 games at 30 rounds with retesting happening each call. The solution got updated one more time to where it currently stands at 50 games with 30 actions taking place, retraining every 10 calls. More testing could have been done to determine what learning constants, discount factors, randomness, and scoring algorithms allow for the most optimal model.

III. Discussion

While this solution has the most promise of determining good actions to take at any point in the board, there are still some inefficiencies and places for improvement if work were to continue towards solving this problem. One heuristic not included in training the model is how other players will act. Currently, other players react under the same model, but this will not be the case after submission, where it will instead be facing other models. As such the moves it believes to be optimal, may not be optimal or taken by other models leading to a diverging solution path very quickly. That is the reason for the ability to randomly sample, but is something of note. Another problem is that this model did not get as much testing as is available for it. The learning, discovery, and random sampling constants were simply used from Professor Weng's sample code. As such these constants may not be refined for the problem at hand. Another inefficiency is the speed. The speed is not as bad as my submission for Project-II having attempted to optimize it further and test on main to make sure nothing takes too long. This does not mean that it is quick though. I am sure through iteration, further research, and another developer the speed of the algorithm could be refined. Another problem is the naive approach to updating the probabilities of the modifications. Considering that the probabilities are not given, one must have an initial state that gets constantly updated. The base case decided on in this implementation was a count of 1 for each of the 3 modification types, with 3 as a total count. Each time the model gets called, the counter gets updated if one of the modifications occurred by comparing the action provided by the model compared to the action taken by the agent. If the counter has been updated, the total count gets updated, and then so do all the probabilities. This naive approach will over time update the probability to be closer to the actions, as long as the actions had the ability to be done. Because of the caveat, it may not always be a good implementation because it has the chance to never get updated. It also may fall behind because it takes multiple runs before the updated probability is used to train the model. By then the model may always be in a losing state.

IV. Conclusions

Q-learning is a great learning algorithm for solving the 3 agent chasing problem, especially with the added difficult of a random percentage option for the moves to stay the same, rotate 90-degree to the left, and 90-degree to the right. Q-Learning allows for the model to recalibrate itself often, as well as being the model that won out during Project-II. Because of those results, the model was tweaked in order to continuously update the predicted probability distribution as well as the optimal moves to make. This model is very naive in nature, leading to some non-optimal moves and training strategies, but it does solve the problem and give a solution that follows an optimal strategy, even if not implemented well.