Project II Report for COM S 4/5720 Spring 2025: A Q-Learning 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. 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. The scoring is as follows. If any hit an obstacle, they receive a 0. If only one of the agents captures another, and does not get captured, they receive three points, otherwise if no captures were taken, every agent that can receives 1 point.

II. ALGORITHMIC DEVELOPMENT

As the learning concerning combative agents took place, Professor Weng graciously offered three key ideas. These ideas were MiniMax, Monte Carlo Tree Searching (MCTS), and Q-Learning. MiniMax is an attempt at building a tree of each possible action and then either minimizing, or maximizing the ”score” of the action based off which player. One will notice rather quickly that is only two options, as such it seems best suited for questions of 2 agents. The scoring algorithm can be changed in order to incentive actions more or less, which is a great benefit. One of the problems is that it only concerns 2 agents, and this problem holds 3. Because of this, a MiniMax implementation did not seem like a great possibility. Monte Carlo Tree Searching was another possible answer to these types of problems. This solution once again generates a tree of actions. These actions are not all actions possible though, but rather a sampling of actions. As such it reduces the complexity of the tree. Another key distinction is that at any point the algorithm is simply attempting to play out a game state until a predefined winning position or until a certain depth has been reached. Then and only then does the score propagate up the tree. It similarly struggles with 3 agent games as opposed to 2 agent games, but that can be accounted for in the way the back propagation happens, as well as how the rewards are given out. This option was attempted and will be discussed later. The final 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 must only be run once, and from there the learning will have explored a great depth of options and have scores for many actions regarding each state. It also does not attempt to be fully deterministic, but instead keeps some components of randomness that allow for non-optimal moves to be taken and checked against. The final benefit is that the model can simply store what agent made the move, and the scoring for the model can be taken from the scoring of the winning states as provided in the documentation.

A. MCTS

As discussed above Monte Carlo Tree Search (MTCS) is a tree based searching algorithm to attempt to find the optimal move for a given state based off a statistical sampling of actions and playing them out until an end state. From there, the score is added to the value of the node, which is later used in an Upper Confidence Bound calculation to find the best action.

1) Implementation: The attempted implementation utilized the sample code provided by Professor Weng in class. From this sample code many updates were made, but the overall logic was kept throughout the process. The first overall was understanding what actions can be taken, and what to define as the state. The state as defined here and from now on will a list of the three players positions. The world will define the board underlying the players that they move on. All legal actions to randomly iterate through in sampling are found by listing all moves the other two agents could take in the world by examining the state. Once all the actions have been found, they are randomly sampled to build down the tree. This process happens until an end state is reached. In this implementation, an end state is that a capture has occurred, or a certain amount of actions has taken place.

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

From there a score gets decided as described. If a capture has happened, the player to capture another receives 3 points, the agent captured receives -3 points, and the agent not involved receives 1 point. If multiple captures have taken place, then each player receives 1 point. Otherwise, if the round count has become too high each node receives 0 points as there is no clear evidence that it is good or bad. These scores would then propagate up the tree, as well as incrementing the number of visits at the node. From there, the best child of the root is chosen as the best action.

2) Testing: Testing was not extensive on this method. As implemented calculations must be called at each instance of deciding actions. The method was also implemented naively and as such it led to a very slow algorithm of deciding what action to take. Because of this inefficiency, testing could not be done to the degree that would be suitable to determine the algorithm with the best score.

3) Discussion: This MCTS solution was naively implemented. The most obvious point of that was the way in which a child node was created. These nodes were created by examing any action that can be done by any of the two agents and choosing one at random. This means that there are cases that the nodes simply switch between two agents, not allowing the third to move. As such the model lacks the ability to model the game to the fullest potential. This non-rotating creation of child nodes also gets in the way of the current implementation of back-propagation. This propagation will simply continuously iterate through a list of [3, -3, 1], but this iteration assumes that there is a fair rotation of actions which is not guaranteed by the model. Another problem is that is does not score any stalling, or disincentive getting captured if other captures occured. As such the model does not gather a good heuristic to develop the best action off of. For these reasons, and the overall slowness of the algorithm as I implemented it, this solution was not used.

B. Q-Learning

As discussed above, Q-Learning is a non model based learning algorithm to determine best action. The algorithm instead updates a table that is used to determine the best action based upon the state, the position of players, and what player is acting. The algorithm, much like MCTS runs a series of simulations, but this time it is instead entire games simulated. Once these games have been simulated, a lookup table with scores of actions will exist and be used to determine the best action at each stage of the actual game being played, although there is still a chance for the agent to act randomly.

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 100 games, at 100 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, and finally 1 point if the number of rounds has been exceeded. 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.

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 finally where the solution currenlty stands at 100 games with 100 actions taking place. More testing could have been done to determine what learning constants, discount factors, randomness, and scoring algorithms allow for the most optimal model.

3) Discussion: While this solution has the most promise of determining good actions to take at any point in the board, there are still some inefficenieces 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 avaible 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 inefficenciey is the speed. The speed is faster than MCTS, but because of the depth of the problem, there are still a lot of actions to check and calculate scores upon. The depth of the problem is the biggest issue, as there is no way to explore all possible options. As such this method explores

those that are most likely to be explored by others. The depth problem also only occurs once at the start as opposed to potentially happening on every action as in MCTS

III. Conclusions

Based off the limited testing and naive attempts at implementation of Q-learning and MCTS, Q-learning is more efficient on a time scale capacity without large noticeable loss in the algorithmic succcess. This claim is made very loosely though as sufficient testing has not been done. Q-Learning allows for the quickest responses and has been tested the most for depth to test, and how many loops to test upon. Q-Learning also seems the most adoptable towards the 3 agent problem from the simpilier 2 agent implementation detailed in Professor Weng's sample code.