Comparative Analysis of Reinforcement Learning Algorithms with Optimal Solutions on Three Open AI Gym Games

Collin Price, Brandon Hills, Sam Oh

December 7, 2017

1 Introduction

In this project, we conducted a comparative study of different algorithms on three different games in the OpenAI gym environments: Taxi, Frozen Lake, and N-Chain. Taxi is a game where the player spawns randomly on a 5x5 grid, and there are 4 locations. Two of these locations are randomly chosen as a pick-up and drop-off point, and the player must go to the pick-up location to get a passenger, and then move to the drop-off point. Frozen Lake comes in two forms: a 4x4 grid and an 8x8 grid, where there are frozen locations and holes. The player must move from the top left corner to the bottom right corner, without falling into a hole (in which case they lose). However, each time a player moves in a direction, they have a one-third probability of going in that direction, and also going in the two adjacent directions. N-Chain is a simple game where you can go forward one step or back to the beginning of the chain. You can choose to go to the beginning and gain a small reward, or try reaching the end of the chain, where you can repeatedly gain larger rewards. However, there is a twenty percent chance that you take the opposite action you choose.

The purpose of our system is to compare the performance of a reinforcement-learning algorithm to an optimal policy that we compute from four variations of games on the OpenAI Gym. Our goals are to see whether the Q-learning agents will converge to the optimal policy we compute for each stochastic/deterministic game, and if so, how long the Q-learning agents will take. If the agents take a significant number of iterations, is there a way that we can lower the number of iterations by tweaking the underlying algorithm?

In order to answer these questions, we split the coding up into two parts: (1) model each of the games either with an MDP or as a search problem and apply different algorithms to each game to compute an optimal policy, and (2) apply a basic Q-learning algorithm to each of the games, tune the hyper-parameters for the algorithm, and if convergence takes too long, then add additional incentives to the game to improve results.

We would like to mention that, initially, our project began with a focus on the Atari game Space Invaders (one of the environments provided by OpenAI). Ultimately, we switched our project to the ones listed above, which had much smaller and more manageable state spaces. This allowed us to explore the performance of certain reinforcement algorithms we learned from class, Q-learning algorithms, and also to tweak the algorithms in order to improve performance. Similarly, this allowed us to take advantage of concepts such as Markov Decision Processes and Search problems as we represented different games in this way in order to compute an optimal policy with which we could compare results [4]. More about what the original project was and why the switch is necessary is included in the Background and Related Work section of the paper.

2 Background and Related Work

Initially, we began our project with the goal of working in the openAI gym environment specifically with the Atari game, Space Invaders. The goal of the project was to create multiple learning algorithms for the game that we could compare the results based on play. This project would have allowed us to explore more practical parts in reinforcement learning by exposing us to deep learning. For the challenge of Space Invaders, we noticed that Asynchronous Actor-Critic Agents (A3C), Deep Q-Learning, and Optimized Deep Q-Learning) [1] were all the top performers for Atari games and seemed incredibly interesting, touching on some material from class but expanding our knowledge past the material we explicitly covering in class. For example, each of these algorithms used a convolutional neural network in order to take in the input in a manner that would allow the algorithm to learn. For this reason, we decided to attempt the A3C algorithm in trying to solve Space Invaders. We followed a walkthrough on Asynchronous Actor-Critic Agents, and tried to implement the method to work for the Space Invaders environment, but ran into the problem where bullets were not being registered by our Neural Network. This caused our agent's policy to converge to simply shooting (and not dodging to survive). To combat this issue, we would have had to somehow parse the pixel array for ship positions, which would have been a difficult task, especially since the ships move throughout the game. We determined that a move to a comparative study of a reinforcement algorithm across multiple stochastic/deterministic OpenAI games would allow us to more directly employ concepts we learned from class. Moreover, the additional time and difficulties we ran into during the creation of these machine learning algorithms were difficult to overcome because of our limited knowledge in this area.

Ultimately, we decided that the switch to our present project was the right move to make because (1) the new project more explicitly explores most of the course topics we originally intended on exploring and actually covers additional topics such as informed search and MDPs, (2) our original project required too much outside knowledge that was beyond the scope of this class's final project, and (3) despite much time spent on learning these new concepts, our progress was slow going and it seemed as if it would be difficult to fully achieve the project goals set out in our proposal unless we slightly modified our project to focus on different openAI gym games other than Atari.

In order to see our original work with A3C, you can look at the ”a3c” folder, which we kept in our github (see Appendix 1).

3 Problem Specification

Taxi Game:
In the Taxi Game, OpenAI represents the current state of the game with a number between 0-500. For learning, the environment returns a reward every time a passenger is picked up or dropped off. The taxi environment provided by OpenAI Gym takes 6 different integers as actions, representing Up, Down, Left, Right, Pickup, and Dropoff. Thus, the coding problem we needed to solve is to return the correct array of commands to the environment, given whichever state the Taxi is in.

Frozen Lakes:
In the Frozen Lakes game, the player must move from the top left corner to the bottom right corner, without falling into a hole (in which case they lose). OpenAI represents states for this game also as integers, where the top left corner is 0, and each state is incremented by one moving from left to right, and down the grid. The layout of the game that our agent is interacting with remains constant, i.e. the start and goal locations, as well as the hole locations, stay the same every time you play the game. In the 4x4 version of Frozen Lakes, there are 16 separate states to provide policies for. An for the 8x8 version, there are 64 states. The Frozen Lake environment has 4 actions, encoded as integers representing the 4 cardinal directions. The trickiest aspect of this game, however, is that the actions are non-deterministic. For any of the 4 given cardinal directions, there is a $\frac{1}{3}$ chance that the agent will move in the intended direction, and a $\frac{1}{3}$ probability that the agent will take an adjacent action instead. (i.e. if our algorithm inputs an UP command, there is a $\frac{1}{3}$ chance the agent will move RIGHT, and a $\frac{1}{3}$ chance that it will move LEFT). For learning, a positive reward (of one) is given only when the player has reached the goal state, otherwise all rewards from actions are zero.

N-Chain:
In the N-Chain Game, you start at the beginning of the chain and have two action-options: you can go forward one step towards the goal (the end of the chain where you receive a large award) or go back to the beginning of the chain and receive a small reward. However, there is a small chance that you take the opposite action you choose (twenty percent). The N-Chain environment returns the state the agent is in (represented as an integer from 0 to n-1, in the case we worked with, it was a 5-Chain). A small reward is granted for going backwards (2), while a large reward is granted for reaching the end of the chain and going forward (10). All other actions receive no reward.

4 Approach

We approached modeling and optimization in the following ways: With the taxi game, we represented the space as an (x,y) coordinate plane by parsing the integers representing the states. The tens digits tell the x position and hundreds digits tell the y position of the taxi. Also, the ones digit, along with whether the tens digit is odd or even, represents where the passenger pick up or drop off point is. Once we gained this information, we implemented A* search with the Manhattan distance as a heuristic to find the optimal paths to the pick-up and drop-off locations. The A* search algorithm we used is defined in searchTaxi.py.

We modeled the frozen lake games and N-Chain game as Markov Decision Processes (MDPs). Similarly to Taxi, in our modelling we converted the provided states to (x,y) coordinates. Once we did this, we could easily right transition functions for both games, as we just manipulate the coordinates and use the probabilities provided by the game. We also followed the same reward schemes for the games, which were provided by OpenAI, in our reward functions. To compute optimal policies for these games, we used value iteration on our created models, determing the optimal policy of a state as the action of the maximum Q-value for that state.

For the reinforcement learning part of our project, we implemented the Q-learning algorithm, with which we tested on all of the games. For taxi and frozen lake, we also created a variant of Q-learning that added extra rewards for getting closer to the goal, and penalties for getting farther away (or a penalty for reaching a terminal state in the case of frozen lake). For each game, we tuned the hyper-parameters to give maximal performance: epsilon for a epsilon-greedy policy, the learning rate (alpha), and gamma (discount rate).

In terms of the specific data structures used in the implementation, we used the Counter data structure which we had used in homework 3 [2] in order to store the different values for the value iteration problem as well as the Q values in the Q-learning agents for each of the games. Additionally, we used the Priority Queue class from homework 1 in order to keep track of the values in the fringe for our A* search algorithm [3]. The implementation of these two data structures can be found in our utils.py file in our github (see Appendix 1).

In terms of a clear specification of the algorithms we ultimately used in the implementation, we used value iteration, Q-learning, what we call incentivized Q-learning, A* search, and a random algorithm. Similarly, we attempted to create an approximate Q-learning algorithm but had difficulties defining features in a way that allowed the agent to learn effectively, and so we scrapped it.

5 Experiments

For each environment, we trained the algorithms for a certain number of iterations, where each iteration was one full game interaction. We tested to find how many iterations it took for each algorithm to converge to a near-optimal result. We then tested each algorithm on the environment 1000 times and averaged the results in order to see trends in performance. For Taxi, we compared the results of our Q-learning agents with the optimal results from our A* search. For Frozen Lakes and N-Chain, we compared the performance of our agents with that of an optimal policy calculated with value iteration through our MDP models.

Before collecting data for Taxi, we tuned hyper-parameters to best suit the game. We did this by changing one parameter at a time to find an approximate maximal allocation. In the case of Taxi, we found got the following values: our probability of taking a random action in Q-learning ϵ = 0.15, the discount rate γ = 0.9, and the learning rate α = 0.1. For the Taxi game, a "successful" game play interaction is when the taxi agent performs completely optimally. Figure 1 represents the average number of successes out of 1000 game play interactions for different training iterations.

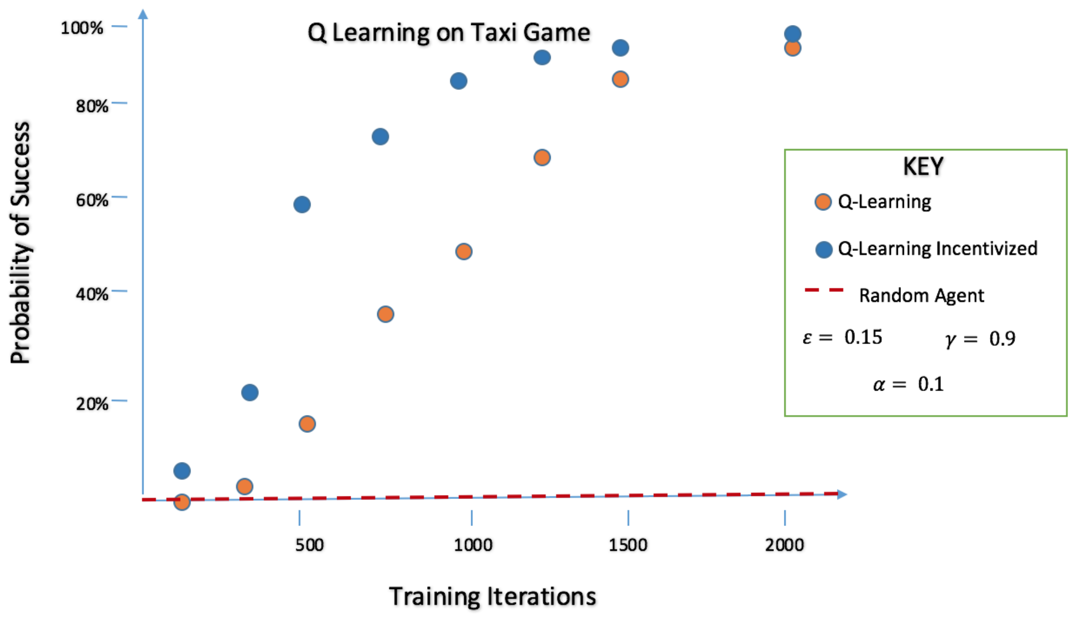

Figure 1: Data for Q-Learning on Taxi Game

5.1 Results

In Taxi, we found that our incentivized algorithm converged on the optimal policy in around 28% fewer iterations than the non-incentivized algorithm. The data that we collected for this game, as shown in Figure 1, shows, on the x-axis, the number of training iterations that we trained the algorithm on.

The data that we collected for Frozen Lakes, similarly shows the number of training iterations of our Q-learning agents vs. the average number of successes out of 1000 game play interactions. We ended up tuning our hyper-parameters to get an ϵ of 0.2, γ of 0.99, and α of 0.1 for both the 4x4 version and the 8x8 versions of the game. For the Frozen Lakes Games, a ”successful” game play is defined as the agent making it from the start state to the end state without falling into a pit. In order to know if the Q-Learning algorithm converged to the optimal path, we printed the policies for each of the environments and compared it to the optimal policy (as solved by using value iteration on our MDP model). See figures 3 and 4 below:

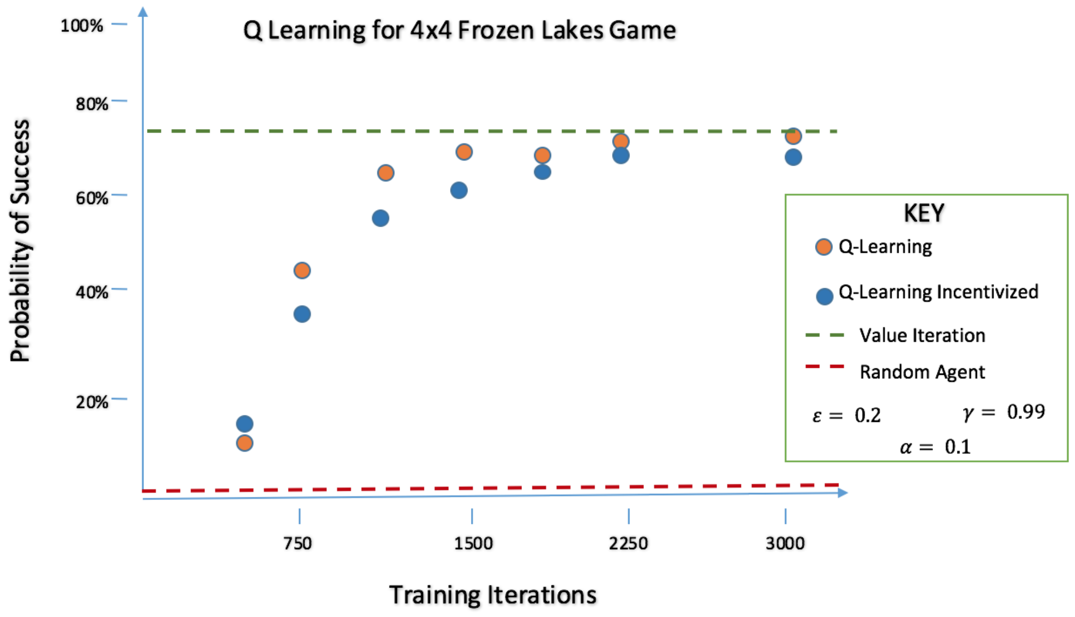

Figure 2: Data for Q-Learning on 4x4 Environment

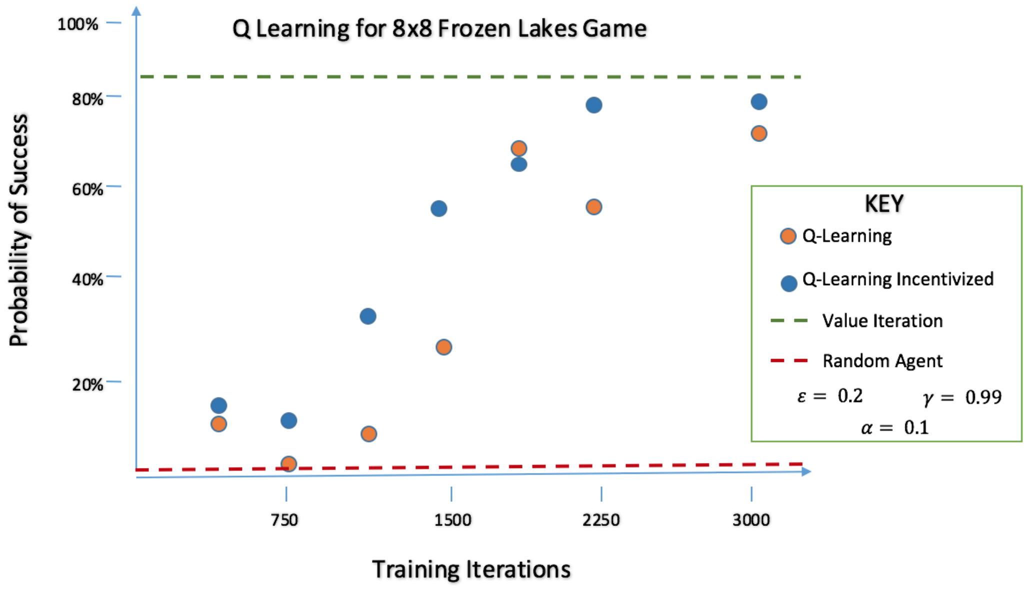

Figure 3: Data for Q-Learning on 8x8 Environment

We found that on the 8x8 Frozen Lakes game, our incentivized algorithm converged faster than the regular Q-Learning algorithm. However, on the 4x4 Frozen Lakes game, our incentivized Q-Learning agent performed slightly worse than regular Q-learning agent. We were at first surprised by these results, as we exected our incentivized Q-learning agent to perform better, similarly to Taxi. However, upon further evaluation, we realized that there was a reasonable explanation to these non-intuitive results. The reason we created the incentives was to guide the agent to the goal at the end of the game (to avoid our Q-learning agent from never experience the end goal state). However, adding incentives for moving towards the goal actually created adverse incentives in terms of an optimal policy: by giving higher rewards based on the proximity to the goal, the incentivized Q-learning agent often overlooks tactical moves that would keep the agent alive and safer. In a smaller Frozen Lakes game, such an incentive can prove to actually hinder learning the optimal policy, because often the optimal policy requires some moves that are not directed towards the goal in order to guarantee safety of the agent and avoid falling in the hole. With the 8x8 game, however, where reaching the end goal is much harder when the Q-learning agent doesn't have any rewards to bring it in that direction, the added incentives helped the agent locate the goal. Meanwhile, the regular Q-learning agent could spend considerable time wandering the map before finally experience the end goal reward.

In N-Chain, because there is no terminal state, we measured success based on how many rewards were gained by our agents in one round of the game (which happened to be 1000 actions).

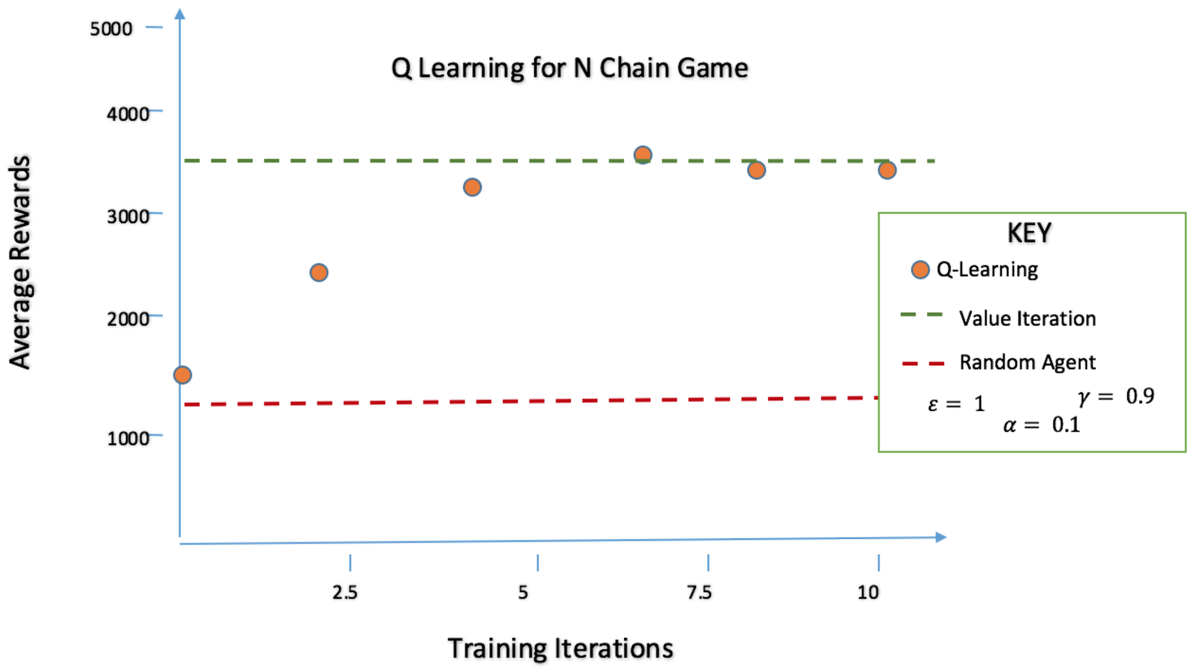

Figure 4: Data for Q-Learning on NChain

N-Chain:
In our N-Chain game, we found that the Q-Learning algorithm converged to the optimal policy after only roughly 5 iterations of gameplay, as seen in Figure 4. We had calculated from value iteration that the optimal policy for our specific instantiation of N-Chain (5-Chain) was to move forward at any state. This posed as a potential challenge for Q-learning with low epsilon values, as the Q-learning agent would experience the backward reward soon and never explore enough to experience the large reward from moving forward five states. For this reason, we posited that the best strategy was to use an epsilon value of 1, in order to maximize the likelihood of experiencing the large rewards from repeated forward actions. Our N-Chain Q-Learning algorithm converged much earlier than the other algorithms due to the smaller size of the state space and the existence of only two action options (only 10 total Q-values).

6 Discussion

The Taxi game behaved as expected, where the incentivized Q-learning agent outperformed the normal Q-learning agent. This was because the incentivized agent focused more on actions that were optimal, and because the state space was deterministic, there were no complications with creating these incentives.

Our incentivized Q-learning agent performed worse in a smaller grid in frozen lake because the extra incentives we create to get the agent to experience the end reward also distract the agent from exploring the optimal path. In a smaller grid, where traversing to the end randomly is more likely, our regular Q-learning agent could not be distracted by the noise of these extra incentives. However, in a larger 8x8 grid, the regular Q-learning agent could not experience the reward at the end of the grid as fast as the incentivized Q-learning agent. Thus, for the larger grid, incentivized Q-learning was a better method, while in the smaller grid normal Q-learning was supreme.

The optimal policy for N-Chain converged rather quickly, as each game iteration contained 1000 total actions and the state space was quite small. Since we used an epsilon of 1, we ignored any learned policy that would make the agent always go back to the beginning of the chain. Thus, repeating enough times, the agent could eventually experience the large end reward through repeated forward actions (which we had calculated as the optimal policy for any state with value iteration). If OpenAI were to make an N-Chain game with a large state space (say 100 states), Q-learning would be impractical. This is because the probability of reaching the end reward would fall dramatically for a random action policy, making the probability of ever experiencing the end reward very low.

In future work we could improve and further our work by implementing and comparing additional learning algorithms other than Q-learning in order to see how these new algorithms compare in terms of the number of iterations needed to obtain the optimal solution. Moreover, we could improve the analysis of our existing algorithms by applying these algorithms to more, different games on the openAI gym.

A System Description (Appendix 1)

In order to run our system and generate the output we discussed in the write-up, simply follow the steps below (or on our github readme!).

1. Clone the github repository locally. It can be found here (it is a public repository): https://github.com/Sam(gym-algorithms
2. Make sure you are using python 2.7 instead of 3.x
3. Download all of the necessary requirements in the requirements.txt folder with a simple "pip install -r requirements.txt". The only requirement there is the gym module.
4. Now, run the following four files in order to get the following output:
   (a) "python testFrozenLake.py": this will return the results of a regular Q-learning agent, an incentivized Q-learning agent, Value Iteration on the MDP, and a Random agent for the 4x4 Frozen Lake game grid first, then the 8x8 after all of the agents run for the 4x4. It will also display the optimal policies for each state, as a list of integers representing the actions for a state which is represented by the index of the list. We have action meanings written in comments.
   (b) "python testNChain.py": this will return the results of a regular Q-learning agent (which converges to optimal quickly) and also both Value Iteration on the MDP as well

as a random agent for the NChain OpenAI Gym game. Like FrozenLake, we also display the learned and optimal policies for all states.

(c) ”python testTaxi.py”: this will return the results of a Q-learning agent, an incentivized Q-learning agent and also a random agent on the Taxi game (explained in the intro).

(d) ”python searchTaxi.py”: this will return the results of an optimal solution that is found by A* search on the Taxi game. This solution always passes, as A* is optimal.

B Group Makeup (Appendix 2)

The three project participants are Sam Oh, Collin Price, and Brandon Hills. Here is the link to our public github repository: https://github.com/SamOh/openAI-gym-algorithms. Below, the contributions of each participant are included.

The three of us worked together on most parts, and split up the work according to our project proposal. But because we decided to modify our original project a bit, we ended up varying slightly from the proposal. Collin Price contributed heavily to the A3C algorithm prior to the switch, and then again contributed heavily to the overall code base after the switch, working on the MDP and value iteration process for Frozen lake as well as the search problem and A* search for the Taxi game and the Q-learning agents for each of the games as well as the incentivized Q-learning agents. Sam Oh worked on the code for a deep Q-learning algorithm before the switch and then after the switch helped with the various coding aspects of the N-Chain-V0 game as well as a basic random algorithm for each of the games. Brandon Hills assisted with the coding for the value iteration on the Frozen Lake and Taxi games and also heavily contributed to the collection of data and analysis of the results. All three of us worked together for the poster as well as the write up.

References

[1] Arthur Juliani. Simple reinforcement learning with tensorflow part 8: Asynchronous actor-critic agents (a3c). https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2, 2016.

[2] Dan Klein. Uc berkeley cs188: Intro to ai. Problem Set 3: Reinforcement Learning. http://ai.berkeley.edu/reinforcement.html, 2015.

[3] Dan Klein. Uc berkeley cs188: Intro to ai. Problem Set 1: Search in Pacman. http://ai.berkeley.edu/search.html, 2015.

[4] Scott Kuindersma. cs 182 lecture slides. Lecture 3: Informed Search, Lecture 7: Local Search, Lecture 8/9: Markov Decision Processes, Lecture 10/11: Reinforcment Learning, 2017.