Reinforcement Learning In Old School Runescape: Teaching an Old Dog New Tricks.

Shane Phillips, Declan Jeffrey, and Wren Maybury

***Abstract*— In our project we focused on exploring reinforcement learning within the domain of Old School Runescape (OSRS). We started by creating a simple agent that automated the collection of wood by continuously going to the closest tree and cutting it down. Then we improved upon our agent by implementing the reinforcement learning technique called q-learning. Q-learning agents start off knowing nothing about their environment and learn the optimal strategy by assigning q-values to different states and actions. The more q-learning agents experience their environment, the more they learn. This domain proved quite interesting yet very challenging when it came to training our agent. We were able to create an agent where obvious improvement was observed. However, multiple domain specific problems came along that made training our agent a quite tedious task. Regardless, OSRS is a unique domain that provides an ample environment for pushing the bounds of machine learning.**

I. Introduction

Throughout this paper we dive into how to create a reinforcement learning agent within OSRS that implements q-learning in order to learn about its environment. OSRS is an old school massively multiplayer online role player game where players can level up skills, go on quests, and trade items on the grand exchange. Since this domain processes a girded coordinate system and lists of simple actions it makes the perfect environment for q-learning. We will dive more into what exactly q-learning is and how we implemented it into our agent in sections III and IV. Nevertheless, these features and many more made working in OSRS a lot more manageable and made the whole process more streamline.

As mentioned before, OSRS is a unique domain with interesting opportunities but also some frustrating problems. Our main issue was with getting our agent to learn. Since OSRS is massively multiplayer it is not possible to speed up the learning episodes and reduce training times. In addition to this, the OSRS servers' are looking out for bot like behavior so our agent consistently got banned. We talk more about how we solved this issue in the methods section, however, it is worth mentioning because it had a big impact on our results. Essentially we just had to make sure we could save the q-table as well as epsilon in order to allow our agent to pick up where it left off when it gets banned. Unfortunately, we still had to create new accounts when this happened and go through the tutorial all over again. We managed to find a script that allowed us to roll through the tutorial, but could not find one to create new accounts. This slowed our progress down tremendously, but never killed our spirit. Although OSRS is riddled with small specific problems, each one we solved was more rewarding than the last. Once we got our agent learning, we realised how enjoyable and truly extensive OSRS is.

We believe that although the process was lengthy and time consuming, we were able to achieve our goal of creating a q-learning agent within OSRS. We manged to obtain sufficient evidence that this agent is able to start off knowing nothing about its environment and continuously learn the best way to accomplish its task. We will talk more about our results in sections VI VII but we manged to prove how truly exciting creating machine learning agents in OSRS can be.

II. Related Work

In order to create our agent we needed to gain more background knowledge on what exactly we were doing. Our main subjects of research were reinforcement learning and more specifically q-learning. When it comes to actual definitions and equations to implement our best resource was our textbook. [1] On the other hand, when it came to actually implementing code we mainly referenced our previously completed project over reinforcement learning. This project pulled most of its materials from the Berkeley's Artificial Intelligence project repository. [2] After strengthening our background knowledge of our task at hand , we dived into some interesting related work in reinforcement learning and q-learning. In addition, we found some convenient and resourceful previously existing scrips within the OSRS domain.

A. Reinforcement Learning

Stuart Russell does a great job going in depth on what exactly reinforcement learning is. Touching on techniques like temporal difference learning, q-learning, and SARSA. This mainly helped us get a general understanding of what reinforcement learning is before getting into the specifics of q-learning. [1]

Since OSRS is fairly unexplored when it comes to creating reinforcement learning agents, there were no previous examples that could help us. That being said, video games are a popular type of domain when creating any type of machine learning agent, so we were able to find some cool examples.

Our first example looked into how to create a human-like reinforcement agent in a 3d multiplayer game. This project took place in the domain of and old first person shooter called Quake III. Again, the domain is fairly different to ours however this article was riddled with some fascinating applications. Mainly we were focused on how they dealt with other online players, as that is a big part of the OSRS game as well. Within this paper a lot of higher level techniques were used to create better, more human-like agents that can better

respond to other players. One of the main techniques was a genetic algorithm of tournament selection that was used to essentially select only the best agents to keep learning. Although these technique were a lot more complex than what we were implementing, it gave us some useful insight into what kind of techniques could be used for future work. [3]

B. Q-Learning

When looking into implementing Q-learning we had to make sure that we understood the concept enough to implement it into our own agent. In order to improve our understanding there were several sources and related works we used. As mentioned, our main source of information came from our textbook and prior knowledge from the class. With this information we were able to gain the necessary understanding of what exactly q-learning is, and what is necessary for it. Unfortunately, we could not find any previously created q-learning agents within OSRS. However, we were able to find some interesting q-learning agents in different video game domains.

The first source we found dealt with enhancing a first person shooter bot by fine tuning it using q-learning. This paper's domain was a lot more complex than ours with exceptionally more game mechanics. Although the domain was different to ours, it helped us understand the process of starting with a baseline agent and using q-learning to enhance its performance. In addition to providing us with a different perceptive on how to implement q-learning. Essentially the authors dove into how bots are regularly perceived as static and forgetful. When the agent is taught to learn from its environment, it starts to develop more human-like tendencies and eventually out perform most players. [4]

Another source we found focused on how q-learning could be used in strategic video games. Now although OSRS would not be considered a traditional strategic video game , this source exposed us to expansive state spaces and action lists. Which was one of our problems when dealing with OSRS, understanding how to handle a large domain. [5]

C. Related Work Within Our Domain

It is worth mentioning some of the related work we found within our domain. Any sort of machine learning bot within OSRS is pretty scarce , however we did use a lot of previously made scripts pulled from the OSBot website for reference.

OSBot is the software that we used in order to create our agents within OSRS. On their website there is a plethora of different scripts made by their thousands of users, that handle a variety of different tasks. We would not of been able to overcome a lot of our initial obstacles if it was not for the help of referenced scripts or related forum posts. We mainly referenced some baseline woodcutter agents in order to get our baseline agent working. Then their forum was full of solutions to some of our problems within the domain like: finding the closest actual tree, dealing with other animating players, restricting domain bounds, and much more. [6]

It is also worth mentioning that a lot of training when it came to working in OSBot, also came from a variety of YouTube videos. Most of these YouTube video were similar to the scripts and were used as a way for us to get started working in the domain. However, we also ran into a lot of cool projects done within the OSRS domain that were not documented on the OSBot website. There was one video in particular we found very interesting, and that was a machine learning agent which learns all about the grand exchange. The grand exchange is OSRS's expansive market where players can buy and trade most of the in-game items for varying costs depending on the supply and demand. This agent was able to learn optimal peak selling and buying times for different items and even start predicting future prices. Although, this was not super useful in helping us implement our own project, it showed us how truly endless the possibilities are within this intriguing domain. [7]

III. Methods

In order to successfully achieve our goal of creating a q-learning agent within the OSRS domain, there were several crucial steps we took to get there. First of all, we started by getting some background information and example scripts off the OSBot website. We will touch more on what OSBot is in section IV , yet we were able to find example scripts within OSRS that performed simple tasks. These sample scripts were then used to help us in creating our own baseline agent for a simple bot that can cut down trees and collect wood. This baseline agent functioned by constantly: checking where the closest tree was , chopping it down, and repeating this until its inventory was full, then returning to the bank to deposit the materials. When developing our baseline agent we ran into several problems and developed some useful techniques that are worth mentioning.

Our first problem we encountered when developing our baseline agent was that the technique of finding the closest tree was not sufficient enough in getting the actual closest tree. Because the method for getting the closest entity did not take into account walls or other objects it would often chose trees that required the agent to walk a long way to get around a wall. To fix this we implemented a OSBot function called web walking, which allowed us to find the closest tree based on the actual path the agent would take to get to that tree. There were a lot of other small domain specific problems we ran into in our process. Another one being our agent would get attacked by enemies every now and then while trying to perform its task. Instead of having the agent waste time fighting these enemies, we programmed it so that once his health got too low he would run away back to the starting point and wait for its health to regenerate. Once we got our baseline agent working we logged the time it took for the agent to fill up its inventory and deposit it in a bank to set a goal for our q-learning agent.

After we were able to get a baseline agent working, the next step was to fine tune it using q-learning and evaluating states. When looking into creating a q-learning agent there are some important definitions to look into before

Fig. 1. Within this picture, the red grid represents the area being searched by the agent, we had to limit this due to the size of the game's map. The white tiles represents the agent's path to be taken to the closest tree. There is also some information in the top left corner with useful stuff like run time and player position.

jumping in. Q-learning is a form of reinforcement learning where the agent has no model of the environment and must learn the best action to take based on learned rewards and punishments. This approach differs from other forms of active reinforcement learning in that it uses an action-utility representation instead of just using utility. This means that there is a q-value assigned to each action possible from each individual state. In order for the agent to actually learn about its environment a q-table must be created which stores the individual q-values of taking different actions in different states. Yet, these q-values can only be learned once the agent performs those actions in those states and experiences a reward or punishment. When looking at how to evaluate different state action pairs, there are a couple important equations we used.

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q(s',a')$$

The equation above is what we used for when evaluating state action pairs. Essentially this states that the value of a state action pair is the reward gained from that action in addition to the discounted utility of the next state. The discount rate is used to tell the agent how much to consider future state values, or if it should prioritize more immediate rewards. Meaning if the discount is set to 1 then the rewards gained from future states are just as beneficial as immediate rewards that can be experienced in the current state. Yet, if the discount was set to 0 then future states utility values are not to be considered when determining the utility of a state action pair. Where the utility of a state is defined as being the best possible q-value for a state action pair from that state, as shown in the equation below.

$$U(s) = \max_{a} Q(s,a) \ .$$

Once a reward or punishment is experienced by the agent it can then change the value associated with that state action pair, and update it in the q-table. So the next time the agent is in that state it will know that the value of that action possesses a positive reward. These q-values get updated with a separate equation that takes into account the learning rate of the agent. Where the learning rate essentially determines how much new values should overwrite old values, and basically controls how fast the agent "learns". [1]

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma\, Q(s',a') - Q(s,a)) \ ,$$

This update function takes the original q-value and adds on the experience reward as well as the discounted value of the next state. It additionally takes away the q-value of the current state that has been multiplied by the learning rate to account for how much we want to overwrite that value. In doing this, the q-table gets updated every time the agent takes an action and experiences some form of reward of punishment.

With a solid background knowledge of what exactly q-learning is, we had to start applying it to our domain. Our starting point was to properly define a state space and legal action list for our agent. In order to do this we create a separate state class to define current states and store state information. Since OSRS's movement system is based off of a grid system (meaning that each position has a set x, y, and z value) we knew that each position the player was in would be a different state. Since OSRS's map is expansive we knew we had to restrict our state space to only include a section of the map. Without this, we would risk the possibility of creating an excessively large state space. OSBot has some good tools that allowed us to select certain areas of the map and reduce the environment the agent was working with. In addition to the agent's position, there was some extra information that was needed to give states to ensure the agent performs properly. Each state also has reference to a list of legal actions within that state. This was crucial to the process because it allowed us to differentiate between positions that are next to trees and positions where chopping the tree is available. And positions that are next to a recently cut down tree, where chop wood would no longer be a legal action. This leads us into how we defined our actions. Since OSRS uses cardinal directions (North, south, east, west) we were able to define actions for the agent moving in any of those directions. We of course had to take into account whether these actions were legal or not depending on if that movement would take the player into another existing entity or our of bounds of the limited environment. In addition to cardinal movement, we also had a chop wood action that was only legal if the player was next to a legal tree. This action was also associated with rewards. Once we defined our state and actions we hard to start reinforcing the agents behavior with rewards and punishments. When the agent cuts down a tree it gets a reward of +6. However, the agent is also continuously punished with a -1 reward for being alive. This means that there is a -1 punishment every time the on-loop command runs. This deters the player from repeatedly going back and forth to high valued states, and prioritize filling up it's inventory.

Once we defined the state space with a set of actions and rewards we were able to start implementing q-learning functions. Most of our referenced material and code came from our previous project in reinforcement learning. [2] The functions we needed to define were: Get action, get

q-value, compute value from q-values, compute action from q-value, and update. Get action just returns the action that the agent should take. When dealing with q-learning there is an additional component to take into account and that is exploration (or epsilon). Exploration refers to how often the agents decides to take a random action (not necessarily the optimal action) in order to better explore its environment and learn more state values. For our project we implemented an epsilon greedy approach. Epsilon greedy means that our agent will first check to see if it should move randomly depending on the defined exploration rate (or epsilon). If it failed to move randomly, then the optimal action is returned instead. In order to get the action the agent needs to use the get action from q-values function. Get action from q-values essentially just returns the best action in a state given the q-values associated to those actions within that state. This method uses the q-table to reference all the given q-values in that state and return the maximum one. In order to get the q-values, we also had a get q-value method which just returned the q-value from the table given a state and action. The compute value from q-values function takes in a state and determines its value by just taking the maximum q-value available in that state. Finally, our update function just updated q-values based on the equation given. Taking into account rewards and future utility values.

Although these functions are seemingly straight forward we had to adapt them a bit to work within our domain. One problem we ran into while trying to do this was that OSBot only allows the running of one script at a time. And that main script has access to certain in game functionality that the supporting files do not. This meant that we had to implement some of our functions within the main file in order to gain access to those functions. This just meant that if a function required access to entities on the map or other information about the environment it would either have to get sent that information or be performed in the main file. One specific example is our get reward function and get next state. In order to get the reward received by the agent after performing an action, the agent has to see what the next state would be by performing that action. This required the ability to get access to the individual x and y components of the agent and manipulate them separately, a function only possible in the main file.

As far as implementing the previously talked about q-learning functions, that was just a matter of sending over the right parameters. In our q-learning class, we had all the function previously mentioned with a couple specific domain changes. The project we referenced was done in python, so we had to implement it with java in mind. In order to create our q-table, we had to create a hash map of hash maps. Where the first map takes in the state as its key, and returns the hash map with action keys and different q-values. In doing this we were able to reference individual q-values by sending in a state as well as the action we are looking at. When doing this we realized that when our agent got banned, it would lose all its q-value information. And in order to combat this we needed to create a text file we could store and pull q-values from, so we could continue training even when we needed to create a new account. This not only allowed us to train longer, and get more meaningful results, it also allowed us to store our epsilon value and even implement a diminishing exploration rate approach. Diminishing exploration rate just refers to the idea of starting with a high chance of moving randomly and exploring the environment. Then slowly decreasing that epsilon value so that the longer the agent trains, the more it starts to take the optimal path. In doing this we were able to create an agent that starts off with a 50% (.5) chance of moving randomly then deceases this chance by .025 each episode until 0.05 was reached. This allowed our agent to spend the first couple episodes focused on exploration of the environment, and learning as many q-values as possible. The more the agent learned and experienced, the more q-values it updated and evaluated. In decreasing epsilon, we were able to get the agent to slowly start taking the learned optimal action more. If we went all the way down to zero, then the agent would only take the optimal action. However, we wanted to make sure that even if the agent has been learning for a long time, there was still a small chance of it going off and exploring more. If we were to allow it to go to zero that would mean the agent would be following its learned optimal policy. As soon as our agent was able to continue learning even with the given roadblocks in OSRS, we got to training it. We will look more into how we trained it, and what type of experiments we did with out agent in V

IV. Old School Runescape

For our project we are working in the domain of Old School Runescape or OSRS for short. OSRS is a remastering of the original video game Runescape developed back in 2001. OSRS is a massively multiplayer online role playing game that includes a huge open world environment with a variety of different biomes and npcs to run into. The different biomes include: Tundras, Deserts, Caves, Forests, Castles, Cities, and Graveyards. There are also a plethora of different skills to improve upon, and tasks to automate bots to complete in order to level up those skills.

Fig. 2. This is the map of OSRS. It is very expansive, with a lot of cool stuff to discover. The red section is where we limited out agent to in order to restrict the environment.

Within OSRS there are three main categories of skills: combat, gathering, and artisan. Combat skills aid the player in fighting other players and enemies and includes skills like strength, range, and magic. Gathering skills assist the player

in gathering resources like mining, fishing, and woodcutting. While artisan skills increase the players ability to craft items through cooking, crafting, and smithing. Overall, although it is a simplistic and old school domain it is surprisingly realistic and expansive. In order to limit the domain a little we focused our project on creating an agent to learn the best way to gather resources like wood, ore, and fish.

In order to create a q-learning agent within this domain we used a program called OSBot which allowed us to write scripts to interact with the OSRS domain using its application programming interface (API). In doing this we were allowed to create an automated bot for cutting down trees, and even implement our own q-learning agent. Although OSBots API uses Java as its default language, it comes with a lot of additional syntax that was created by OSBot to assist the programmer with gaining information and interacting with the environment. Some of these additional tools include: Getting information about surrounding entities like trees and players, controlling the agent and assigning actions to perform, and a very useful onPaint function that allows us to use a graphical user interface (GUI) to print important information to the screen.

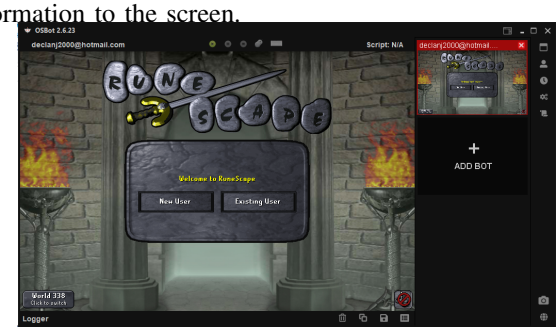

Fig. 3. This shows what the OSBot platform looks like. You can create bots over to the right, then control OSRS on the big screen. There is also a place to look for stored scripts to run on different bots.

Although OSBot provides a lot of useful resources when working in the domain of OSRS, we still ran into a lot of domain specific problems which are worth mentioning. Because OSRS is a MMORPG, that means there are tons of other online players interacting with the same environment. This proved to be a relatively difficult problem to deal with when trying to create an agent who's main purpose was to learn the best way to gather resources. We often found that our agent would try and interact with trees that other players were in the process of chopping down. Or other players would try to engage in combat with our agent. Implementing code to deal with this was surprisingly difficult, so we ended up restricting the agents domain to a small area away from other players and kept him within a server that had the fewest amount of players. With OSRS having 175 different world servers, it was easy enough to keep our agent in a low population world. This was not the only problem we encountered within this domain, another example problem was the way the OSRS servers deal with automated bots. Because there are plenty of other players creating bots to exploit the game's mechanics the servers would frequently ban our bots. We talked about how we solved this problem within the methods section, but it was very tedious and slowed down our progress.

Regardless of the challenges present, we chose this domain because it contains an easy to define state space, with a simplistic yet somewhat expansive set of actions. Now although this game is old school and fairly basic, it contains a lot of unique and quite complex elements. Previously mentioned were the expansive map space and abundance of skills; however the game also includes tons of different quests, other online players, and a grand exchange where players can trade in game items with varying prices depending on supply and demand. Not only is this game an interesting domain to work in, it proves to be an interesting stepping stone in the world of artificial intelligence (AI).

When creating an AI the power of the agent depends on the domain's definition of "intelligence". Often AI is thought to be intelligent when it can perform as well as any human could in the same situation. Since OSRS does not include too much complexity it is a lot more manageable when trying to create a self learning AI that can perform as well as a human on simplistic tasks. This means that it provides an unique opportunity to develop strong AI that can think and act like a human would within the domain. Using a domain like this opens up the possibility for creating low level AI in a restricted domain, and then transferring the same idea into more complex domains. Increasing the state space, amount of legal actions, and interesting reward/punishment scheme.

V. Experimental Procedure

Once we created our agent and got it to the point where it could start learning, it was time to start training. As we mentioned before, q-learning agents learn based off experience and can only evaluate state action pairs once they experience them. In order to train our agent we needed to make sure that we defined what each training episode would look like. How exactly we were going to measure the success of this agent.

Episodes in q-learning agents, just refers to a single run within the problem domain. Meaning that, it shows how the agent would perform and if it improves after several runs. We started by defining a starting point for our agent to begin each episode as well as a near by bank for depositing materials. We also implemented a run timer that started when the agent returned to the starting point and stopped once the agent is done banking. The agent would then work to fill up its inventory until full, and go to the bank to deposit the goods. Once the agent deposited the wood, the timer stopped and the episode time was recorded in our text file. This allowed us to keep training the agent and getting an idea for if it is improving or not, based on the time taken to collect wood. Since we were able to store our q-table, epsilon and episode times in a text file, this allowed us to continue to train that agent without having to start over.

Once we got an idea of how we could train our agent, we needed to add in some logging information to show us if the agent is really learning or not. As mentioned before, we had a stop watch that started when the player reached the starting

position, and ended when it banked the materials. Along with this we used the graphical user interface built in to OSBot to create colored squares based on values of states. This

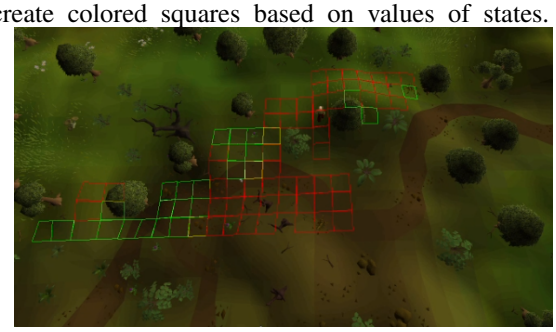

Fig. 4. This picture shows our GUI addition of colored tiles depending on state values. Green corresponds to states whose values are over 5, yellow refers to greater than 0 but less than 5, and red was for negative spaces.

graphical information was pivotal in our experimentation, because it allowed us to visualize how the state values were changing. Whether it was going from red to green, because the agent chopped a tree down in that spot. Or going from green to yellow, because the agent keeps walking in and out of that state. We started off with these 3 colors than ended up adding in extra colors to give us a better range of state values.

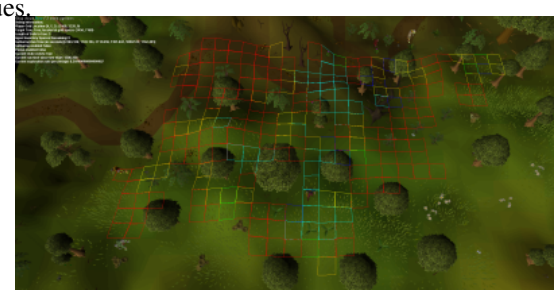

Fig. 5. This was our final colored grid system that helped us determine how well our agent was performing, what moves he was taking, and what values the state held. Red: value <0 ; Orange: $0<$value<5 ; Yellow: $5<$value <10 ; Green: $10<$value <25 ; Blue: $15<$value <20 ; Cyan: value >20 .

Once we were able to visualize how the agent was training, we could see what part of the map it had explored as well as what spaces it cut trees in. This allowed us to analyze its behavior and make tweaks to some values in order to better train our agent. For example, we found that with a fixed epsilon of .35 the agent explored quite well, but very seldom learned to take the optimal action. In order to fix this, we implemented our diminishing epsilon technique that was previously mentioned. In doing this we were able to get the agent to explore more in early episodes to expand its environment and then focus in on more optimal actions. In order to do this we tried several values. First we started with the original .35 and decreased by .05 each episode. That proved to be too much of a decrease, by the time the agent had done 6 episodes it was down to .05. So this agent did not explore nearly enough of its environment. We then proceeded to increase the starting epsilon to 0.5 to further encourage the agent to explore more states and experience more actions in the beginning. Then we only decreased by about .025 each episode, in order to slowly push the agent towards taking the optimal action. We talk more about how these different types of experiments related and which ones proved more beneficial in sections VI and VII

When experimenting in this domain, and letting our agent run there were some problems we ran into along the way. We have previously mentioned that the game is online, so OSBot had no way of speeding up our episodes. So when we started training our bot would often get around 30 minutes to complete an episode. Although that was to be expected since the exploration rate was high, it still meant that training the agent was a long and tedious process. In addition to this we had about 6 agents get banned in the process of trying to experiment with our agent. This slowed down our progress even more, and hindered the agent's ability to spend some time learning. We knew that we needed to train our agent for at least 20 runs to see some type of meaningful improvement in the data. However, we were only able to get around that for our final run with an agent starting at epsilon .5 and going down. Nevertheless, we believe that the results we gained skill show case the improvement possible within our agent.

VI. Results

Now it is time to dive into how well we were able to create our agent, and look into some of the results and what exactly they prove. We will talk more about what these results mean in section VII but for now we will just explain what each table is and how we got these results. In each of these tables we provided several important features: the episode number, the time taken in that episode, the epsilon value in that episode, and whether or not the agent got banned. As well as the average time and best time over all the episodes. In addition, we color coded the times to show which times were better than the average (green) and which were below (red).

Our first results collected were for our baseline agent (fig. 6) that simply goes to the closet tree and collects the wood. It performed well since it was programmed to be efficient, and has not form of learning. This was what we were striving to teach our agent to perform as well as. However, it is very bot-like and we will talk more about why that is a problem in the next section.

Episodes:	Times: (seconds)	Bans:
Episode 1	470.215	
Episode 2	360.002	
Episode 3	445.698	
Episode 4	425.741	
Episode 5	495.063	
Episode 6	380.511	Banned
Episode 7	448.467	
Episode 8	419.76	
Episode 9	473.636	
Episode 10	393.689	Banned
Episode 11	431.598	
Episode 12	502.356	
Average:	437.228	
Best Time:	360.002	

Fig. 6. Baseline agent, essentially fixed path.

This second table (Fig. 7) was our initial collection of data. Our first agent had a fixed epsilon value of 0.35 and would

very often move randomly instead of optimally. We figured that we would want our agent to move more optimally as it learned more about its environment. So we moved towards a diminishing epsilon technique.

Episodes:	Times: (seconds)	Epsilon:	Bans:
Episode 1	2154.013	0.35	
Episode 2	2049.365	0.35	
Episode 3	2005.061	0.35	
Episode 4	2068.574	0.35	
Episode 5	2170.316	0.35	
Episode 6	1567.171	0.35	
Episode 7	1721.097	0.35	
Episode 8	1819.401	0.35	
Episode 9	1530.078	0.35	
Episode 10	1691.928	0.35	Banned
Episode 11	1695.224	0.35	
Episode 12	1638.056	0.35	
Average:	1842.523667		
Best Time:	1530.78		

Fig. 7. Fixed epsilon of 0.35

Our next table (Fig. 8) contains the same type of data, however this agent possessed a diminishing epsilon that started at 0.35 and decremented by 0.05 each episode. We used this technique because it allowed our agent to start off taking random actions and exploring its environment. And move towards taking more optimal actions more often. We continued to decrease epsilon by 0.05 each episode but created a lower bound of 0.05. This was to ensure that the agent still had some level of exploration since it would take a lot more training to actually converge to an optimal policy. After working with this agent, and looking at the data we realized that the epsilon was decreasing too fast and the agent was not exploring enough of the environment. In order to improve upon this further, we decided to increase the initial epsilon and decrease how much we decremented epsilon over time.

Episodes:	Times: (seconds)	Epsilon:	Bans:
Episode 1	2288.21	0.35	
Episode 2	1348.398	0.3	
Episode 3	1487.605	0.25	
Episode 4	1265.431	0.2	
Episode 5	1813.251	0.15	
Episode 6	1724.432	0.1	
Episode 7	1744.087	0.05	
Episode 8	1798.756	0.05	
Episode 9	1593.417	0.05	
Episode 10	1755.788	0.05	
Episode 11	1552.214	0.05	
Episode 12	1560.57	0.05	Banned
Average:	1661.01325		
Best Time:	1265.431		

Fig. 8. Diminishing Epsilon Starting at 0.35

This final table (Fig. 9), was the last collection of data we were able to obtain within the time frame for this project. We started this agent at an epsilon of 0.5 and only decremented by 0.025 each time. In doing this we created longer starting times, but allowed our agent to explore more of its environment and learn more q-values. We knew this would make our agent learn at a bit of a slower pace, but should show more improvement over time. Unfortunately, our agent got halted by the bot detection software within OSRS and caused our text file of saved q-values to get erased. Another roadblock we encountered when it came to gathering results. With little to no time left, we were unable to train more values. However, we believe that this table is enough to prove our point.

Episodes:	Times: (seconds)	Epsilon:	Bans:
Episode 1	2484.338	0.5	
Episode 2	1749.194	0.475	
Episode 3	2113.838	0.45	
Episode 4	1391.042	0.425	
Episode 5	1890.512	0.4	
Episode 6	1762.493	0.375	
Episode 7	1769.616	0.35	
Episode 8	1586.324	0.325	
Episode 9	1556.458	0.3	
Episode 10	1611.124	0.275	
Episode 11	1502.036	0.25	
Episode 12	1735.698	0.225	
Episode 13	1369.875	0.2	
Episode 14		0.175	
Episode 15	Values	0.15	
Episode 16	got	0.125	
Episode 17	Wiped	0.1	
Episode 18		0.075	
Episode 19		0.05	
Average:	1732.503692		
Best Time:	1369.875		

Fig. 9. Diminishing Epsilon Starting at 0.5

VII. Discussion

Now that we have looked at some of the collected data, it is time to talk about what exactly all of it means. We will start out by discussing each table individually and dissecting the data.

Fig. 7 shows how a reinforcement learning agent would train with a fixed exploration rate of 0.35. Meaning the agent would consistently move randomly 35% of the time, regardless of the optimal action. This agent started off poorly and just thrashed around the environment trying to learn what to do. Once it got about 5 episodes in the agent seemed to be performing relatively better and seemingly learning some optimal actions. However, since the epsilon never changes the agent often got caught roaming away from optimal paths. Which was good for exploration, but bad for performance. This gave us the idea to diminish epsilon over time and allow for the agent to take optimal moves more frequently.

Which leads us into Fig. 8 which shows an agent whose epsilon started at the same 0.35 but decreased by 0.05 each episode. We hoped that this would allow the agent to slowly converge onto a more optimal path. And increase performance faster. Looking at the data we see that the agent does improve however it seems to converge to an optimal path too quickly. This agent yielded a better overall overage and best time, yet did not consistently improve. After testing this agent we realized that the agent needed more time to learn about its environment before trying to take the optimal actions. So we needed to increase the starting epsilon, and decrease how much it gets decremented each episode.

With our final agent we did just that. In Fig 9. we show our agent's results where we increased our starting epsilon to 0.5 and only decremented by 0.025 each episode. The data matches what we expected, giving us longer starting episode times because the agent is exploring more. While also slowly improving over time as it should theoretically take more optimal actions each episode. Unfortunately, we ran into another problem while trying to collect our final results. The OSRS server detected our bot but did not ban it straight away. Their detection software often messes with bots by forcing them to miss click to test their humanity. This caused our learned q-table text file to get wiped out and we lost all our data. With little to no time to train our agent back up again, we were forced to just settle with the results we gained. Now, we would predict that this trend should continue and although epsilon is decreasing the agent should continue to explore while converging on an optimal strategy.

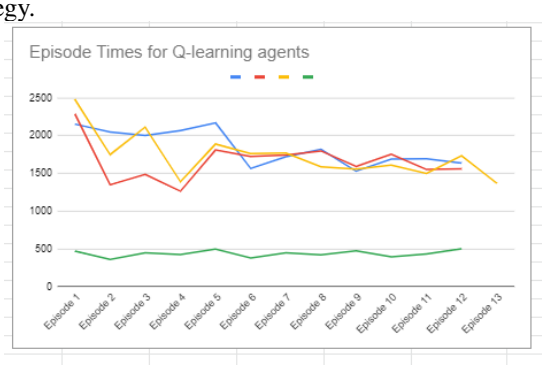

Fig. 10. Graph showing training times for the different agents over 13 episodes. Green: Baseline. Yellow: Fig 9. Red: Fig 8. Blue: Fig 7.

We went ahead and graphed all 4 of the data sets to visualize how exactly each agent compares. Obviously the baseline agent performs a lot better than the other agents, however it does not improve and constantly tries to take the same path each episode. Looking at the other 3 agents all of them start off quite sporadic yet seem to form a relatively clustered negative slope. We believe that with more time and training episodes we would be able to continue this trend to eventually converge on the baseline agent. We believe our main adversary when it came to accomplishing this goal was time, and the inability to speed up training. Possible additions to our final agent would be to add in specific features and how much they should be weighted. Features such as how far away the closest tree is from that state, or how close enemies are, maybe how far the agent is from the bank. These features could better teach the agent to properly evaluate each state action pair.

One thing that is worth mentioning, is that although our baseline agent performs well it gets banned a lot more often than our other agents. Because it performs the same path over and over, the bot recognition software can pick up on it faster. Our agents seemed to be a lot more "human-like" and took longer to be detected when in the environment.

We do also want to mention that we did plan on trying to apply this same strategy to different resources as well as an agent that can collect multiple resources and optimize experience. Since time did not permit, we were unable to try and tackle this task. Now that is not to say that we could not. In fact, the process of transforming our agent to perform new actions such as mining or fishing would only require some small additions. We would need to add in some different legal actions, choose a better part of the world to start in, and optimize different tasks. When it comes to trying to create one agent that can collect multiple different resources and maximize experience, that would be a bit more difficult. Nonetheless, extremely possible and would mainly require a different reward function, multiple new actions, and a much bigger domain. Not to mention all these extra agents would take weeks to fully train and test.

VIII. Conclusion

Throughout this project we learned a handful of important lessons, and came to a couple overarching conclusions. First of all, OSRS is a vast domain with unique opportunities for machine learning and artificial intelligence as a whole. That being said, we realized that it is riddled with some frustrating problems that make working within this domain arduous but rewarding. The tedious process of getting our agent to learn taught us that the most difficult part about reinforcement learning is getting it to experience its environment. Sometimes it took us hours to see any improvement, just to realize we should probably go back and alter a value. Nevertheless, it was all worth it to finally create an automated wood cutting agent that can learn about its environment and improve its strategy over time. We hope that although we ran out of time for this project, we can continue to explore this domain and all it has to offer. Overall, we concluded that it is indeed possible to teach an old dog like OSRS some interesting new tricks.

References

[1] S. J. Russell and P. Norvig, *Reinforcement Learning*. Pearson Education, Inc., 2010.

[2] U. B. C. S. Department, "Project 3: Learning Materials," 2014. [Online]. Available: http://ai.berkeley.edu/reinforcement.html

[3] M. Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, and A. Garcia, "Human-level performance in 3D multiplayer games with population-based reinforcement learning," *Science*, vol. 364, no. 6443, May 2019.

[4] P. G. Patel, N. Carver, and S. Rahimi, "Tuning Computer Agents Using Q-Learning," in *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2011, pp. 581–588.

[5] C. Amato and G. Shani, "High Level Reinforcement Learning in Strategy Games," in *9th International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, 2010.

[6] "OSBot Forums," 2021. [Online]. Available: https://osbot.org/forum/

[7] C. Coder, "Predicting Runescape Grand Exchange Prices with Machine Learning," 2019. [Online]. Available: https://www.youtube.com/watch?v=D5TmBcpgm7k