Pathfinding AI

By: Casey Price

# Introduction

This project's purpose was to write a machine learning AI that could search a 20x20 grid world with obstacles and successfully find a given goal. The algorithm used was Sarsa Lambda, an eligibility trace method used to facilitate the AI’s learning. In our world only the player’s starting position is randomized. The obstacles and goal locations are always the same within the 20x20 grid world.

# Method

The AI starts off with no information about the world. It only knows where it’s current location, and that there is a goal somewhere in the world which will give it a reward value of 1. All other grids have no reward value. Walls or squares outside the map are simply marked as invalid moves, and will never be taken or rewarded.

When the goal is found, the path the AI took is recursively reward. Each step and action is then rewarded on a decreasing scale from the goal. The last step the AI took before finding the goal is given the most reward, and for every step back along the AI’s path, the reward is decreased.

Every state and action pair has two values, a Q value and an E value. Initially, the Q value is randomly populated with a small value, and the E value for each state action pair is set to 0. As the AI makes moves and finds correct paths, the Q value for state actions leading to the goal are increased. The E value is essentially the confidence in how correct a Q value is, and increases with each action taken from a specific state. As the AI randomly wanders throughout the world, eventually finding its goal, it obtains an increased confidence in what action it to take, as it takes that action and obtains its goal.

# Experiments

I first created my grid world using WPF forms, creating a grid User Control, and then a world control, which was essentially a 2D list of GridControls. This was then added to my main window with 2 buttons, one to play/pause, which told the AI to take an action every 15 milliseconds. This was initially set to 10, however, windows could not redraw the board fast enough to keep up with the AI’s movements. This caused weird effects, like the player eating the goal, or multiple players ending up on a single board. I think this is due to the lack of mutex locks since a new move command was being issue before the last one could complete. I fixed this by adding a mutex lock on the world before moving the player, and also reducing the play speed to every 15 milliseconds, which prevented the movement commands from getting backed up. I also had to make a one movement button to debug the Ai’s individual movements.

I found that sometimes I did not want to wait an hour for the AI to create a decent map, so I added manual movements using the arrow keys. This allowed me to move the player and simulate creating a board with Q values that helped inform the AI where to go. This was for debugging purposes only, and was not used to create the final map.

At first, I was confused about how to correctly implement the algorithm. Were the initial moves supposed to be completely random? How fast do I start moving from random to informed moves? Do you ever make informed moves or just when you think the map is done?

I started with moving randomly, and tried to build the map off these random moves, but this took an extremely long time and I figured there was no point a keeping Q and E values if the algorithm never used them. I added in decision making so the algorithm would randomly pick actions based on their Q and E values. This worked quite well, and while it still took the AI a long time to initially find the goal, after finding the goal, it was able to quickly use that path and find the goal again.

I then ran into issues where the AI would not explore enough. When it found one path, it would simply reuse that path to the goal whenever possible, no matter how inefficient that path was. This problem stemmed from my decision making algorithm being too influenced by the E value.

I also ran into issues with the AI making bad moves in states, and then making a good move in the state later, which rewarded the bad move since it was still in the path to the goal. I solved this by only rewarding the last move in each state, rather than every action in each state. This way only good actions were rewarded, and if a previously bad move was taken, it would be ignored.

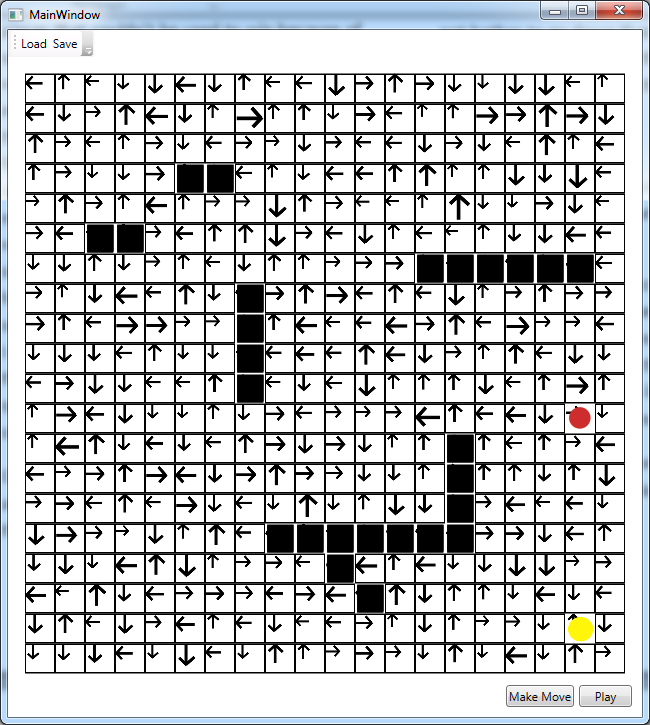
I finally decided on moving randomly at the start, and slowly working my way towards informed decisions once all moves were properly attempted. I forced the algorithm to try other moves if it hadn’t already tried them, and until it had tried all moves a certain number of times. Once that was accomplished, it used its Q values to make decisions. The number of moves I forced it to try until it was confident was 100. Essentially, it would nudge the probability of picking a move it hadn’t tried until it had tried all actions 100 times. Once all actions were done 100 times, it only used its Q value since it was assumed that it had found the goal enough times from this state to know which direction it needed to go.

This worked pretty well because the AI would explore all possible moves sufficiently before deciding on a path as being correct, while still being confident in a path when it did fine one.

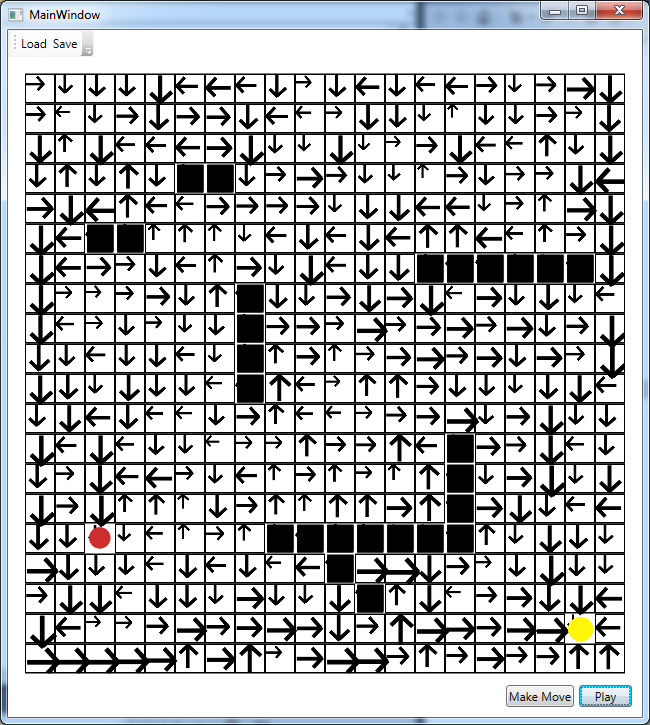
Finally, I used binary serialization to serialize the map of the grid world and each states Q and E values so runs could be saved and loaded later. This was simply this was simply enough since I just serialized a 2D list of grid states each containing a Q and E value for all available actions.

# Analysis

While my implementation of this algorithm was quite slow to create the map, possibly due to my slow simulation playback speed, it did eventually create a good mapping to find the end goal. Please note that the Arrow sizes are based off their confidence in relation to the other directions. A large arrow does not mean it knows that direction is right, but it does mean it knows that that direction is heavily favored over the other directions, hence why there are a lot of large arrows in the beginning and lots of small arrows as the algorithm continues.

Initial map: 

Map at 1 Hour:



As you can see the arrows slowly started pointing towards the goal as the AI was allowed to run for a longer period of time.

# Conclusion

This would be a very good algorithm to create a map for finding the best direction to a goal if given a large period of time to run simulations on the map. The map my algorithm created was very efficient; however it took an extremely long time to create it, and if the map changed at all the algorithm would have to restart from the beginning. So, it is only useful for maps that you have a long time to solve and know won’t change.

The machine learning portion of this is actually quite cool. The fact that my AI was able to learn how to navigate a map I gave it, with no information about where the goal was or what the map looked like, was amazing. The algorithm is also very greedy and fast because of it. Once you have the map done you don’t need to do any real calculations to find the goal later on, you just have to follow the map and you’re done. This is way faster than doing something like say A\* path planning since all the planning is done for you, and you can just make the correct decisions at runtime. This kind of algorithm would be very useful for making AI that needed to follow a level in a mobile or web game where you don’t have a lot of time, memory, or processing power to figure out where you need to go.