CS 4341: *INTRODUCTION TO ARTIFICIAL INTELLIGENCE*

Final Project Write-up

Yahel Nachum
Gregory Port
Duc Pham
Cuong Nguyen

Contents

Problem Description

The problem we decided to undertake was to get an agent (the Hero) to efficiently reach a goal position given some random world and combination of objects that had different behaviors. The objects that the agent could run into are walls and enemies. Walls are blocks that take up a position on the world and do not let any other object reside in that position. Enemies in the early iterations were static objects that would cause the Hero to die upon having the same location. In later iterations enemies were made to be dynamic so that the Hero would have to worry about running away from them rather than stepping around them.

Summary

The development of the Hero took on an iterative cycle where we would try a solution, gauge how well it was working, and try to add on to it to make the Hero perform better. This development naturally took on a step-by-step process.

The first step was developing the game engine to run the AI in. This included a GameManager to keep track of the game world and a GameWindow to draw out the results when asked to refresh itself.

The second step was developing the first AI for the agent. In this iteration, the agent chose completely random moves. This AI did not allow the Hero to perform very successfully, so the next step was to give the Hero a utility of its state. The utility given to the Hero helped the Hero improve its score (percentage of time reached goal out of all trials), but it hit a maximum threshold that still did not perform well enough.

The biggest step of the process was next. This was Temporal Difference Reinforcement Learning. This involved giving the Hero a reward function and, based on that function, it learned by itself the utilities of the states into which it got. The Hero performed significantly better than it had when it was merely given a utility function.

Setup: Getting the Game Engine Up and Running

The game engine is one of the most important parts of this project. Without it being completed in a timely fashion, the rest of the project would have been inconceivable. There would have been no world for the Hero to traverse and no framework to allow the game objects could interact with one another. Inspiration was taken from the Entity, Boundary, Controller (EBC) style of programming. The entity and boundary portions of EBC programming were utilized as to allow objects to exist within the world, managing those objects, and as a way to update and display the window. The game engine includes two major parts: the GameManager, and the GameWindow.

The GameManager is responsible for all the “behind the scenes” work. This includes: setting up the world boundaries, keeping track of all the game objects within the world, updating the game objects whenever the manager is asked to step, and providing collision handling for the objects of the world.

The GameWindow is responsible for the output of the GameManager’s result. It figures out the best way to represent the world based on how big the window is (at startup) and how big the game world is. Every time the GameWindow is told to refresh, it obtains the game objects from the GameManager and displays them on the window with their respective attributes (color, name, position in world).

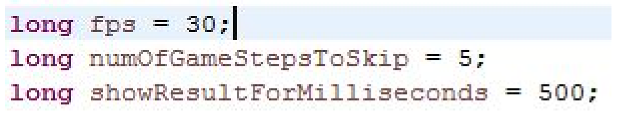
The resulting system gives a very good basis on which to build our AI. All that the program needed from this point of the development process was to follow a game driven process when testing the AI. This includes: initializing the game with objects, looping between telling the GameManager to perform one step and refreshing the GameWindow, ending the game on a win or lose condition, removing objects from the game, and repeating as many times as needed for the agent.

Game Stepping and Game Objects Details

Game Stepping

This section is provided to help anybody that will run the program change key variables for viewing purposes. This includes setting the desired frame rate, number of steps to skip in the game engine before refreshing, and the time to show the resulting Hero position of the trial after it has ended.

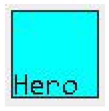
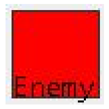
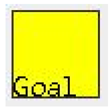
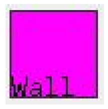
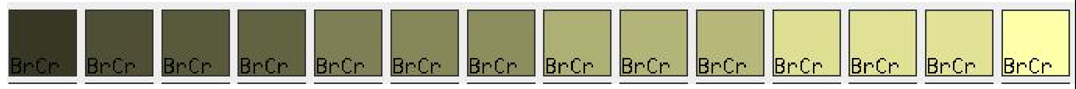
The figure below shows some of the most useful variables to change for observing how the Hero moves and learns. These variables can be found in the Main class file for the project. The fps variable is very simple and just tells the graphics window how many times to refresh the window per second. In the case shown below, it will refresh the window 30 times per second. The next variable tells the graphics window how many game steps to skip before refreshing. In this picture, it shows that the program will refresh every 5 game steps it performs. The last variable tells the program how long to display the result of the iteration on the window. This example shows that the Hero’s ending position will be shown for half a second before going on to the next trial.



```
long fps = 30;
long numOfGameStepsToSkip = 5;
long showResultForMilliseconds = 500;
```

Game Objects

This section is meant to give details about the graphical appearance of game objects. Learning what each object looks like will help the reader decipher the video analysis sections later since the video quality might be too low and fast to decipher which objects are which.

Hero	**Enemy**	**Goal**	**Wall**
			
Color: Aqua **Name:** Hero **Details:** is the primary agent of the project.	**Color:** Red **Name:** Enemy **Details:** Will kill Hero if Hero steps on it. Provides losing condition.	**Color:** Yellow **Name:** Goal **Details:** Is the goal position for Hero. Provides winning condition.	**Color:** Magenta **Name:** Wall **Details:** Is a solid object. Hero cannot be in same square as Wall.
BreadCrumb			
			
Color: Whitish Yellow (more recent BreadCrumb) -> Black (older BreadCrumb) **Name:** BrCr **Details:** Allows the viewer to see where the Hero has been recently and in what order based on the brightness of the BreadCrumb.			

First Iteration: Complete Randomness

At this stage, we set up a Hero that merely randomly moved about the world. Even though this agent was only randomly moving, it did have some positive attributes such as moving around the world and "exploring" it. Even with these positives, many negatives were present that weighed on the agent. Some of these included: the agent did not know which moves were good / bad, the agent did not know how to get to the goal, and the agent was almost never successful.

Other than trying to solve the problem, this iteration helped prove that the game engine was working as it should have. From this simple AI, we could see that the game was being initialized correctly, the game was stepping the given amount of times and then drawing the results after those amount of steps, the game was ending whenever the agent got to the goal or died from an enemy, and the game reset after a win or lose condition was met.

Video Analysis Part 1 Part 2:

As seen from the first and second parts of the video, the Hero moves about the world by only making random decisions. The Hero has no method of determining whether the goal is close by or what stepping on an enemy will do. This random behavior is good for exploring the world, but the Hero does nothing with this information. During part 2, the Hero can be seen traversing the world after many iterations.

Second Iteration: Half Random, Half Utility

The previous agent did not have any sense of what moves were considered good and which ones were considered bad. This iteration of the agent helps address this problem with the use of a utility function.

The second iteration of the agent includes half of the first iteration and a new half that would be able to tell the agent how good / bad of a state the agent is in. This state value was determined by a utility function that we constructed. The function is made up of the following variables: number of objects near the agent, number of enemies, if the agent is closer to the goal than it previously was, and if there is a enemy or goal where the agent is. Tweaks to the multipliers were made until the agent was performing substantially better than the fully random agent.

The benefit of this utility is that it gives the agent a sense of what are good choices to make versus what are bad choices to make. The first drawback surrounding this function is that the multipliers were evaluated by us in an arbitrary fashion. The second negative impact of having a set multipliers is that the agent does not learn after making any moves and, therefore, needs to make random moves every once in awhile to get it unstuck from a position the utility function may have gotten it into. In particular, this can be seen from the agent getting stuck in a “peninsula” created from a set of walls.

Video Analysis Part 3 Part 4:

The third and fourth parts of the video also depict the Hero using random behavior in combination with a given utility function. However, the most notable difference between the previous iteration and this one is that the Hero now has some understanding of what states are better to be in than others. The Hero determines this by the utility function given to it with constant multipliers. This allows the Hero to tell that a state that brings it closer to the goal is better and that one that brings it closer to an enemy is bad. Since the multipliers are constant, the Hero cannot change the value of a state if it has observed that the value might be different than what was given. This Hero might be able to get to the general area of the goal, but it will be limited by its unchanging utility function.

Third Iteration: Epsilon Randomness, Temporal Difference Reinforcement Learning

The previous agent had problems with it not learning and taking the utility function we made as the truth in its entirety when we were producing the multipliers in an arbitrary fashion. This iteration solves this part of the problem by offloading the utility function to the agent to learn by itself based on a reward function we give it that is far less arbitrary.

The third iteration included epsilon randomness and temporal difference reinforcement learning (TD LR). The epsilon randomness would allow the agent to be more random at first to allow exploration of the utility of different states. As the agent learns and progresses through more iterations, it starts making less random moves since it knew which states had more utility than others.

The second part of this agent was the temporal difference reinforcement learning, which is where the agent actually does all the learning. It followed the standard method of calculating the value of states by using the function $S_{new} = S_{old} + 0.1(R() +(S'-S_{old}))$, where S_{new} is the new value of that previous state, S_{old} is the old value of the previous state, 0.1 is the change multiplier, R() is the rewards function (-1 for a step, 1000 for getting to the goal, -1000 for dying), and S' is the value of the current state. To simplify, this function basically means that the previous states value should change, with a positive correlation, based on the current value of the state that it got to from the previous state.

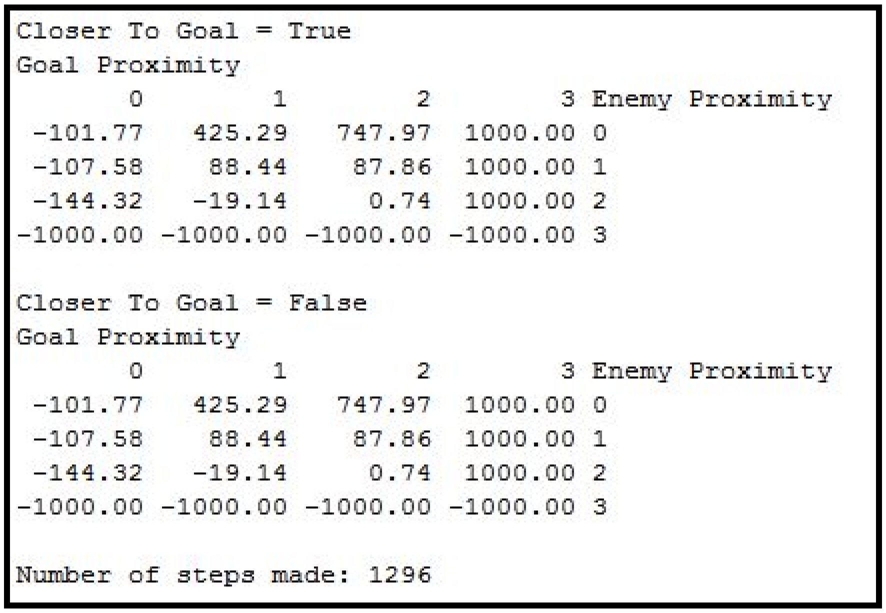
Video Analysis Part 5 Part 6:

Part 5 and 6 of the video show the final agent with temporal difference reinforcement learning. From the video, we can see that at first the Hero does not know what the utility of the states it gets into are. At first, the Hero will start exploring the world figuring out how good a certain state is and how bad it is as well compared to other states. After several iterations, the Hero has updated the table enough to get a general idea of how to reach the goal. After one hundred iterations, the randomness fades away and the Hero has to decide actions based on everything it has learned. The Hero at this state has a very high success rate to achieving the goal state.

Table Analysis:

After running this iteration of the agent over randomly generated worlds a number of times, we can see that the agent has learned some values for the utility of that current state. The values at the edges of the table are initialized through the rewards function while the values in the middle of the table are produced by the previously stated TD RL model. As seen below, the table has a gradient pattern where the states to the top right are generally higher than the states to the bottom left. This gradient makes sense because the closer the agent is to the goal the better chance that the agent will reach the goal, while the closer the agent gets to an enemy the greater change the agent could die from hitting that enemy.

Here is the TD RL table after 300 trials on randomly generated levels (Proximity is based on a block radius from 0 being nowhere within sight to 3 being at that current position)

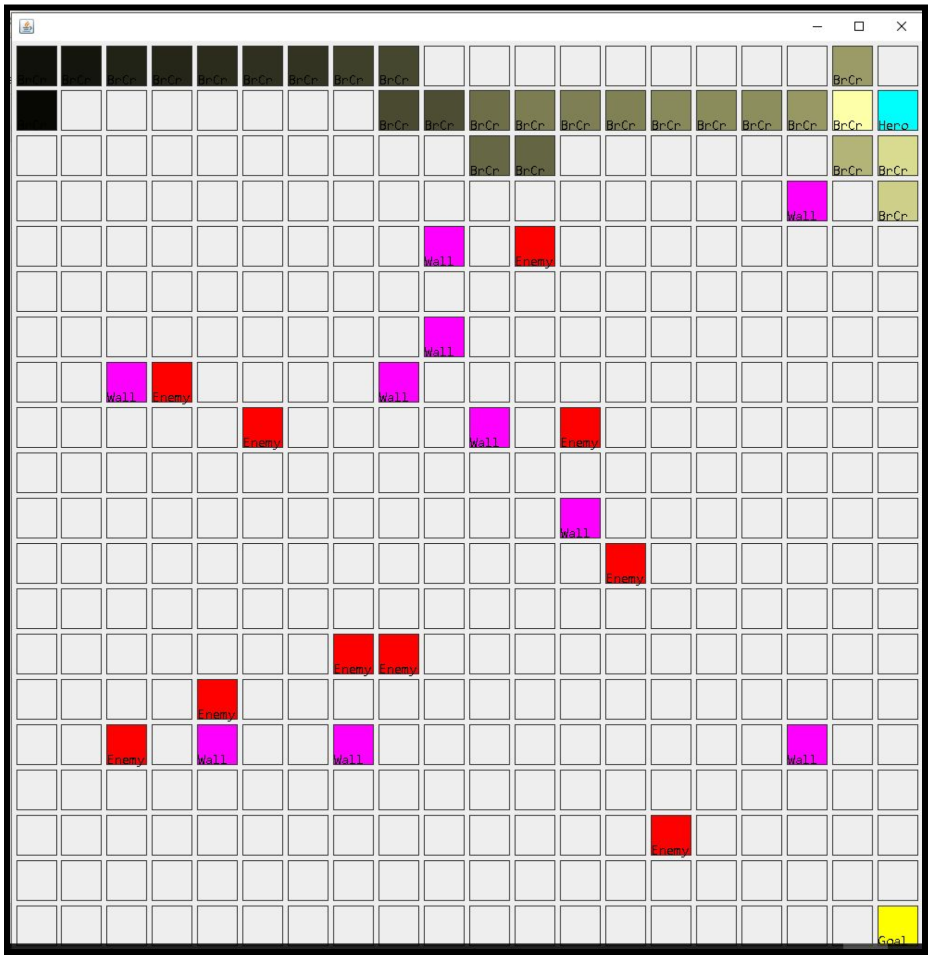


```
Closer To Goal = True
Goal Proximity
       0        1        2        3 Enemy Proximity
 -101.77   425.29   747.97  1000.00 0
 -107.58    88.44    87.86  1000.00 1
 -144.32   -19.14     0.74  1000.00 2
-1000.00 -1000.00 -1000.00 -1000.00 3

Closer To Goal = False
Goal Proximity
       0        1        2        3 Enemy Proximity
 -101.77   425.29   747.97  1000.00 0
 -107.58    88.44    87.86  1000.00 1
 -144.32   -19.14     0.74  1000.00 2
-1000.00 -1000.00 -1000.00 -1000.00 3

Number of steps made: 1296
```

Random level that table was produced from.



Gameboard used for project

Links To Demo Video

Main Class of Project:
https://github.com/yahelnachum/AI_Assignment_Final/blob/master/src/main/Main.java

Part 1 Completely Random slow steps: https://youtu.be/611906EFj30?t=1s

Part 2 Completely Random fast results: https://youtu.be/611906EFj30?t=17s

Part 3 Half Random, Half Utility slow steps: https://youtu.be/611906EFj30?t=38s

Part 4 Half Random, Half Utility fast results: https://youtu.be/611906EFj30?t=1m

Part 5: Epsilon Random, TD RL slow steps: https://youtu.be/611906EFj30?t=1m23s

Part 6: Epsilon Random, TD RL fast results: https://youtu.be/611906EFj30?t=1m45s