Battleship: A Hit or Miss Affair

Christopher Compton
CS UML Undergraduate
Christopher_compton@student.uml.edu

Nicolas Stanzione
CS UML Undergraduate
Nicolas_stanzione@student.uml.edu

Jianing Liu
CS UML Graduate
axelning@gmail.com

ABSTRACT
In this paper we describe an interesting AI development on a classic game. Our team has created three algorithms that play the game Battleship. Each algorithm attacks a board with ships hidden and tries to find them all scoring the least amount of hits. The first algorithm takes random shots, the next is an optimal algorithm called Hunt/Target and the third is an attempt at traditional Q-learning. We use these three algorithms to analyze the abilities of Q-learning as well as the success and drawbacks of the way in which it was implemented. The goal was to use the Hunt/Target algorithm to compare to the Q-learning algorithm to determine if it is optimal. Unfortunately, our Q-learning algorithm was not complete and does not find an optimal solution.

Author Keywords
Battleship, Q-learning, MDP, Hunt/Target, Artificial Intelligence, Python, Pygame.

INTRODUCTION
The problem we set out to solve was to create an AI that could learn and play the game of Battleship optimally through Q-learning techniques. In order to achieve this goal there are three pieces that must be developed. The first and easiest is a player that takes shots randomly. This will give us an example and benchmark of a non-optimal way to play the game. Next we need an optimal algorithm that plays the game so that we can compare it to the Q-learning method to see if it is optimal. For this we implemented a Hunt/Target algorithm that smartly attacks ships when they are found. The final piece is the actual AI Q-learning agent that will learn how to play battleship optimally. What's so interesting about this program is that by the end of the Q-learning trials we will have the ultimate battleship player. It will be very interesting to watch how the agent learns how to look for the best shots to take and to see how it reacts to each situation it gets itself in. This project will be a fun way to demonstrate different types of Q-learning.

PROJECT DESCRIPTION

Problem Analysis
The problem we are trying to solve is easy to put into words; build an AI that can play battleship. Yet the actually implementation was not so simple. We obtained an open source python program that creates a battleship game using the Pygame module set where people can use the mouse to attack. We had to modify this code to allow for our algorithms and AI to play instead of a human. For this problem, we intend to use reinforced learning by implementing Q-learning and resulting Markov Decision Processes (MDPs). Although the traditional Battleship board is static, the locations of the ships are unknown. This makes the problem implicitly probabilistic and ideal for Q-learning and MDPs. The ideal way to solve this problem is to fire at squares based on the probability of a ship being in each square. Once a ship is hit the probabilities of the four or less squares around it should increase since it is very likely that another piece of the ship is there. Through trials the agent should learn these probabilities. Unfortunately it takes a while for these probabilities to be updated. The agent could fire randomly for a while and not hit any boats, which would cause the probabilities to only change slightly. The goal for the Q-learning agent is for it to play optimally like the Hunt/Target algorithm. The Hunt/Target algorithm starts by taking random shots. Once it hits a boat, it pushes all the surrounding squares of the hit square onto a stack. It then pops of each square, attacking them with each pop. If it gets another hit it pushes that square's surrounding squares onto the stack. It keeps attacking the squares on the stack until the stack is empty. It resume firing randomly until it finds another ship.

Data Set
The data used for this project is a 10x10 grid of squares each of which could be hiding a piece of a ship. The boats will be a Carrier represented by 5 squares, a Battleship represented by 4 squares, a Cruiser and a Submarine represented by 3 squares each, and a PT Cruiser represented by 2 squares. In total there will be 17 squares that contain a part of a boat. This means that at the start of the game every square will have a probability of containing a piece of the boat which is the number of boat squares divided by the

total board size. This comes out to be 17/100 so each square has a 17% chance of hiding a boat at the start of the game. The probabilities for each square will change when the agent takes a shot. If it hits a ship then all the squares surrounding that one gain a higher probability of being a ship while all the other squares decrease slightly because there are less ship squares left to hit. Once the agent hits another piece of the ship attached to the first one it hit then the next one in that direction greatly increases while once again all the others decrease.

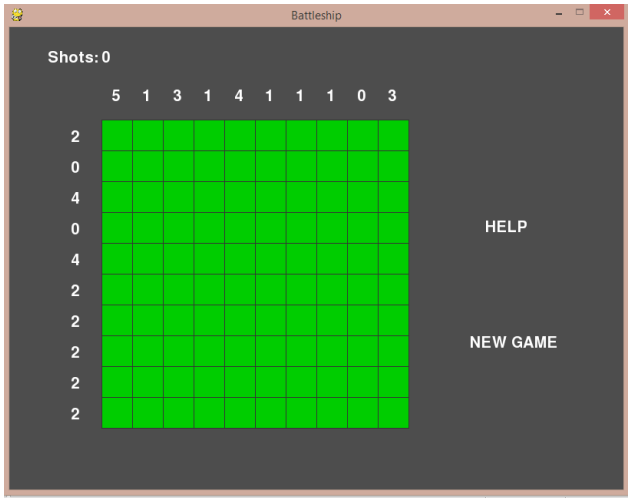

Figure 1. The battleship board at the beginning of the game. Green squares represent squares not attacked yet.

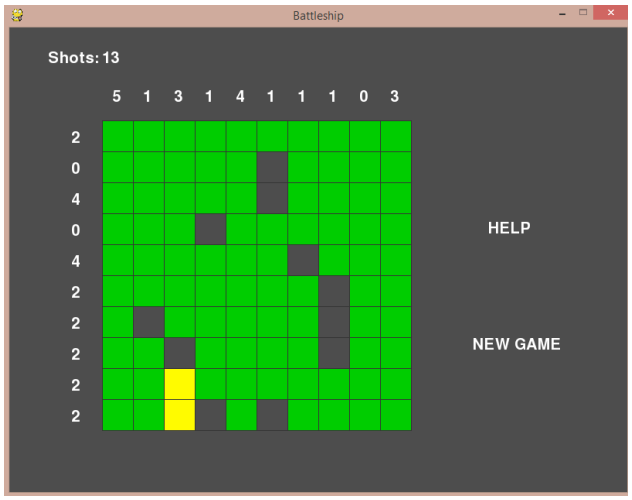

Figure 2. The battleship board after some random attacks. The black squares represent misses and the yellow represent hits on boats.

ANALYSIS OF RESULTS

In order to gauge the success of our algorithms, we planned on the approach of pitting them against each other. Each algorithm was run through fifty (50) iterations with the resulting total shots fired recorded. These results can be seen in Figure 3, below. The Random algorithm did noticeably worse than the Hunter/Target algorithm. However, our Q-Learning algorithm did not, on average, perform any better than the Random algorithm. This is, we think, due to the incredibly large state space which we failed to take into account when we chose to implement Q-Learning and MDPs.

Shots Taken to Win				
Algorithm	Min	Max	Mean	Median
Random	76	100	93.5	93
Hunt/Target	53	79	69.18	70
Q-Learning	53	100	91.56	94

Figure 3. The results of 50 iterations of each algorithm

Our game board consists of a 10 by 10 grid of squares, totaling 100 grid squares. The number of squares that can be occupied by our ships is 17. The size of our state space consists of all possible arrangements of ships on the game board. The calculation of this permutation is approximately 100 choose 17. This calculates to the following:

$$\binom{100}{17} = \frac{100!}{17!83!} = 6.65013 \cdot 10^{18}$$

The state space is actually smaller than this, due to the restriction that the occupation of one grid square by a ship piece means that at least one adjacent square must be occupied. However, we do not feel that the difference is significant in that the state space will still be much too large for a Q-Learning algorithm. Because this state space is so high, the number of iterations it would take for a Q-Learning algorithm to develop a MDP is almost countless. This directly explains our results above.

A more applicable approach would have been to use feature extraction. Using this approach, the size of the state space would shrink significantly. Each grid square would have just one feature to take into account, whether it contains a ship piece or not. This shrinks our state space from the above to a measly 2^{100}. Additionally, if a grid square is found to contain a ship and because all ships occupy 2 to 5 adjacent squares, we know that of the 4 adjacent squares to our target square, at least 1 must contain another ship piece. If our target square is found not to contain any ship pieces, then the adjacent squares are a mystery. They may or may not contain any ship pieces.

CONCLUSIONS

Q-Learning with Markov Decision Processes is not an ideal solution for an agent to learn how to play and win at the game of Battlehsip. The state space is just too large. Q-Learning with feature extraction would have been the ideal solution due to its significantly smaller state space.

The features that would give a feature extraction agent the fastest track to an optimal solution are:

1. Boolean : Hit/Miss

2. If a Hit, how many adjacent grid squares will also register as a hit.

ACKNOWLEDGMENTS

The work described in this paper was conducted as part of a Fall 2014 Artificial Intelligence course, taught in the Computer Science department of the University of Massachusetts Lowell by Prof. Fred Martin. Chris worked on the battleship board and allowing our agents and algorithms to manipulate it. Jianing worked on the Q-learning agent. Nicolas worked on the Hunt/Target Algorithm.

REFERENCES

1. Pandey, D., & Pandey, P. (2010, February). Approximate Q-learning: An introduction. In Machine Learning and Computing (ICMLC), 2010 Second International Conference on (pp. 317-320). IEEE.

2. PygameProjects/battleship https://github.com/PygameProjects/battleship