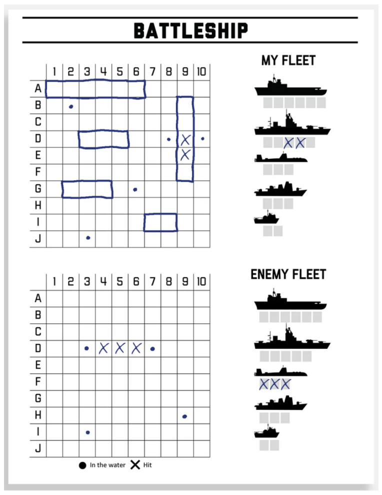
Amdur Final Paper: Project Choice One. Markov Decision Theory Applied to the Game of Battleship

Jonathan Amdur

5/12/2022

The game of battleship is a simplistic game that teaches children logic and strategy. At it's simplest form, it is a game of guess and check that allows even children to understand how to think critically about decision making in an unknown space and use their current knowledge to justify their next move. In this vein, by using current knowledge to support the next decision, battleship is a wonderful representation of a Markov Decision Process being implemented by the human mind. Every time a human is playing, their mind is focused on making the next hit and they are weighing their potential actions considering where the current board stands based on previous moves, the probability of getting a hit for the potential actions, and weighing the maximization of the rewards at each stage. As such, battleship is the perfect application to demonstrate the usefulness of Markov Decision Processes (MDP). By recreating the conditions the human mind is using to consider the players next move, we can utilize MDPtoolbox to program an automated player that will select the best move to take next. Thus, using Markov Decision Process we will create a playable game of battleship with an MDP based opponent and evaluate its effectiveness against a random chance player.

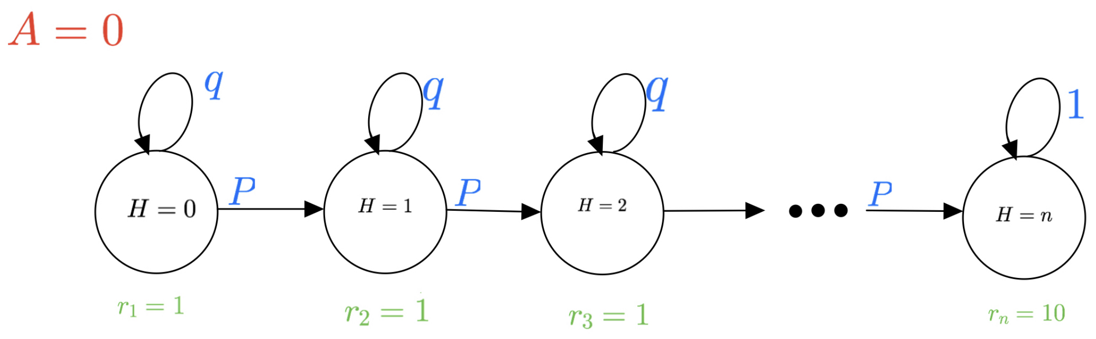
We take the basic game setup and rules from the Hasbro battleship instructions at https://www.hasbro.com/common/instruct/battleship.pdf. We simplify these slightly in order to allow for quicker evaluation of the mathematics instead of an undue focus on the game coding. We set the 10x10 grid to a 7x7 to half the number of positions. With the decreased board size, we also remove the large ship of size 4 from play in order to allow for more miss spaces on the board and not reward lazy learning. The current ships are of length 2, 3 (x2), and 5. These lengths will be relaxed later in our evaluation, but gameplay as a human will play with these sizing assumptions. When gameplay begins, the rules dictate that ships must fit entirely on the board, not overlap, and can only be placed vertically or horizontally. Further, during play players are able to see their ship positions, what ships opponents have hit or sunk, and what their firing choice resulted in. An example board is provided below. For our gameplay, we will follow all these rules and ensure the human and MDP player has the appropriate incite into the board play to make their decisions.



Now, we first must show we can formulate this gameplay as a Markov chain and it displays the Markov property for us to use MDP. By setting the number of hits as the states, we can create a chain where the only possible transitions are staying on the same number of hits by missing, and thus staying on the same state, or getting a hit and increasing the hit counter, and thus moving to the next state. There is no way to lose hits in battleship. In this scenario, all states except for the final state will move to state i+1 with probability p and stay on the same state with probability $q = 1 - p$. Further, since hitting the total possible hits means your opponents ships are all sunk and you win the game, the final state will be an absorbing state and probability of staying on it is 1. An example of this chain is provided below in the actions section.

Now, taking this we note that this chain displays the markov property and so we can apply markov decision theory to this problem. We implement this through the R package MDPtoolbox from the paper **MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problem** by calling on R in python. Calling R in python was done via the example on willfondrie.com in the bibliography. We take our set of actions to be the policies 0,1,2,3,4 where 0 corresponds to attacking one spot above last move, 1 to one spot below last move, 2 to one spot left of last move, 3 to one spot right of last move, and 4 to a random tile on the board. These basic actions were chosen to imitate the same choices a human would make given they know the last tile was a hit, sink, or a miss. If a hit and not a sink, they would know to search around the tile since it means there's another spot adjacent to the hit that must be a hit. If it's a sink, then they know little about the board and need to randomly select the next position. Finally, if it's a miss then they would know to go back and search the previous squares without this one. These actions allow for all of these scenarios. There are also a number of other possible actions discussed later in the paper.

With the actions and states decided, we need transition probabilities for the MDP learner. The only needed transition probabilities are the probabilities of a miss, staying on the current state, and a hit, moving to state $i + 1$. This was explained above and an example of how this would work in practice is provided below. Each action above has an associated transition matrix and the matrix of these matrices is the transition matrix utilized by the MDP. Each probability is determined according to basic logic. When the choice would place the player on a square already hit or missed, then the probability of a new hit is 0 and so the player would have $p(S_{t+1}|S_t, A_i) = 0$ and the probability of staying as 1. If the move would place the player on an unknown cell and the number of unknown cells equals the number of possible hits to win, then the probability of a hit is 1 and $p(S_{t+1}|S_t, A_i) = 1$. If the last move was a hit, a few scenarios arrise. When the opposite action is a known hit, thus indicating a chain, the probability the chain continues with this action is a 50% chance since the chain has to continue in one direction or the other. Thus, $p(S_{t+1}|S_t, A_i) = .5$. When it doesn't have another hit, then the probability is just a 1 in y tiles where y represents the number of tiles that are unknown within 1 of the last move. Thus, $p(S_{t+1}|S_t, A_i) = 1/y$ and it stays on the state with probability $1 - 1/y$. Finally, if the last one was a miss and none of the above is satisfied then no new information was gained and all tiles are equally possible with the probability being the number of hits needed to win out of the total spots unknown, since all spots are equally likely. Putting these in practice for the current state of the board ensure the probability only depends on the current state we're in and exhibits the Markov property. In the program, we complete this by setting all elements to 0, running the above logic to get p, then setting the diagonal equal to $q = 1 - p$ and the off diagonals to p thus representing each possible transition.



Finally, we must compute a reward matrix. A number of different possible combinations were utilized, but the best performer was one where having the reward be 1 at any state for taking action A_i with an extra award of 10 for getting to the final state and winning the game. When running with all 0's and a final reward, we found the MDP model did not learn as well, likely because there was no incentive to increase by one. With more time, a more robust reward system only rewarding transitions and not staying put could be developed as well.

Now, to implement the above the MDPtoolbox mdp_Q_learning module was utilized with the transition matrix above, reward vector, and a discount of .96. The discount originally parameterized according to the two papers from class, but experimentation found a maximum win rate at .96. Q learning was utilized based on the towardsdatascience article, **Reinforcement Learning for the Game of Battleship**, and **Hands-on Reinforcement Learning with Python**. These all explained that for the situation at hand, a more reinforcement learning approach was the best one to take and Q-learning was the first one to try. Understanding of this was enhanced through the videos provided at https://www.youtube.com/watch?v=iKdlKYG78j4.

Through Q-learning, we implemented the MDP based player. The code below was used to output likely policies, choose the appropriate policy for the scenario, and then get the firing choice.

```
# Run MDP - Q-learning
battle_ship_mdp = MDPtoolbox.mdp_Q_learning(T,R,0.96)

# policy choice
policy_choice = np.array(battle_ship_mdp.rx("policy"))[0][total_hits_made-1] - 1
fire_choice = action_choice[policy_choice]
```

Now, combining all these led to our MDP based player. In the next section, we will discuss the architecture of the program and how this fits into that.

When designing the program, the main considerations were integrating the MDP, ease of play, and potential user interface bugs. To begin, the game player can decide to play the game themself or allow a random number generator to play against the MDP. After this choice, the position of the boats on the board are set randomly for the RNG and MDP user, or manually for the human user. For the human input, error handling within a valid selection while loop and try except were put in place to ensure smooth use of the game. After this, game play for each player begins. Three separate functions were created for each user type to take their turn. The MDP user was described above and implemented the MDPtoolbox action choice. The only special case was the first choice since there was no previous choices made. In this case, the action is automatically random choice. For the RNG, a random square was chosen to fire on regardless of previous firing. Finally, the human user is asked their firing position, and will be repromted if invalid input is given. Once a choice is made, in all three functions the next step is to check whether that choice is a hit, sink, or miss. If it is a hit, both players boards are updated to show a hit on that location and the other users total hits left decreases. If it is a miss, just the players board is updated to reflect this. After each turn the current number of ships and hits still in play for each player is displayed. The game continues inside a while loop until the total ships for one of the player drops to 0 and the other player is declared the winner. For evaluation purposes, the function running this returns the winner and the decisions made over time by the MDP.

For a player to play this for real, you only need to call

```
python3 play_battleship.py
```

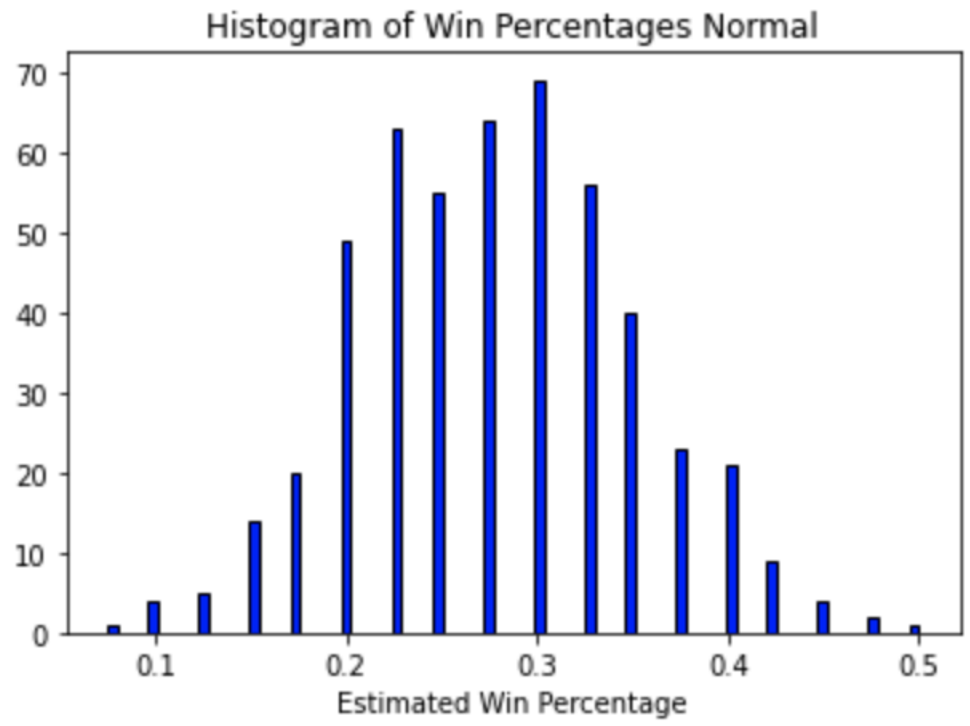
in terminal and follow the on screen prompts. It is suggested the user first uses the option of random to observe how the program works. After that, the user can play as the human player and try to beat the MDP. Code to run this program can be found at https://github.com/amdurj/stochastic_2_final_project_mdp_battleship and the particular code to run this is at https://github.com/amdurj/stochastic_2_final_project_mdp_battleship/blob/main/mdp_battleship/main_scripts/play_battleship.py. There is also evaluation code in the .ipynb.

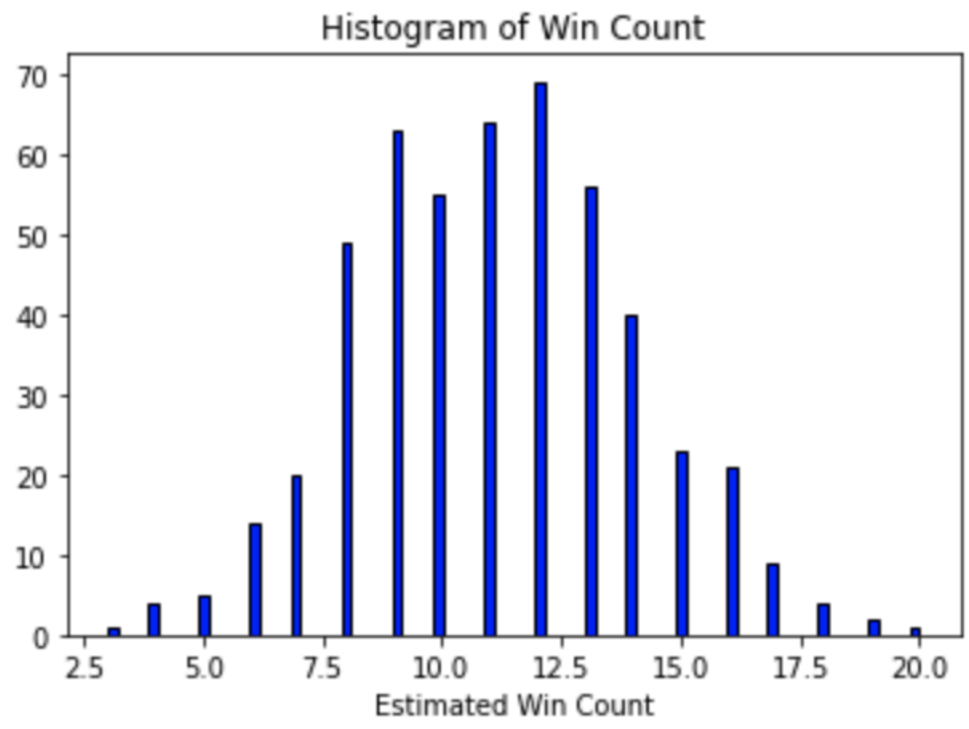
Now, with the code and methods in hand it is important to properly evaluate how the MDP program is doing as a player. To do this, we need to see if the MDP has any sort of improvement at the game over a standard random number generator. In order to evaluate this, we first simulate n = 40 different runs of the game between the MDP and the random player. We call the function as play_battleship(simulation = True) each time in order to suppress unnecessary prints and store the 40 simulations win/loss outcome for the MDP player in an array and the decision path array in another array. Now, using the win array, we want to find the distribution for the win count and percentage of wins that the MDP player has over the random player. By finding this, we can see if the MDP player will consistently outperform the random player or not. To continue our evaluation, we will look at the distributions of the decisions and sample paths chosen by the MDP.

In order to evaluate the distribution of the win percentage and win count for the MDP, we run a bootstrap procedure to estimate the distribution. To do this, code was based on professor Botts Monte Carlo Methods coursework, module 10. The below code was ran over 500 simulations. Each simulation, we sample with replacement from our 40 game simulations to produce an array of 40 possible win/loss combinations and calculate the number of wins and win percentage for that simulation. At the end, our function will produce two arrays of size B, 500 here, with one capturing win percentages for each simulation and the other the win counts.

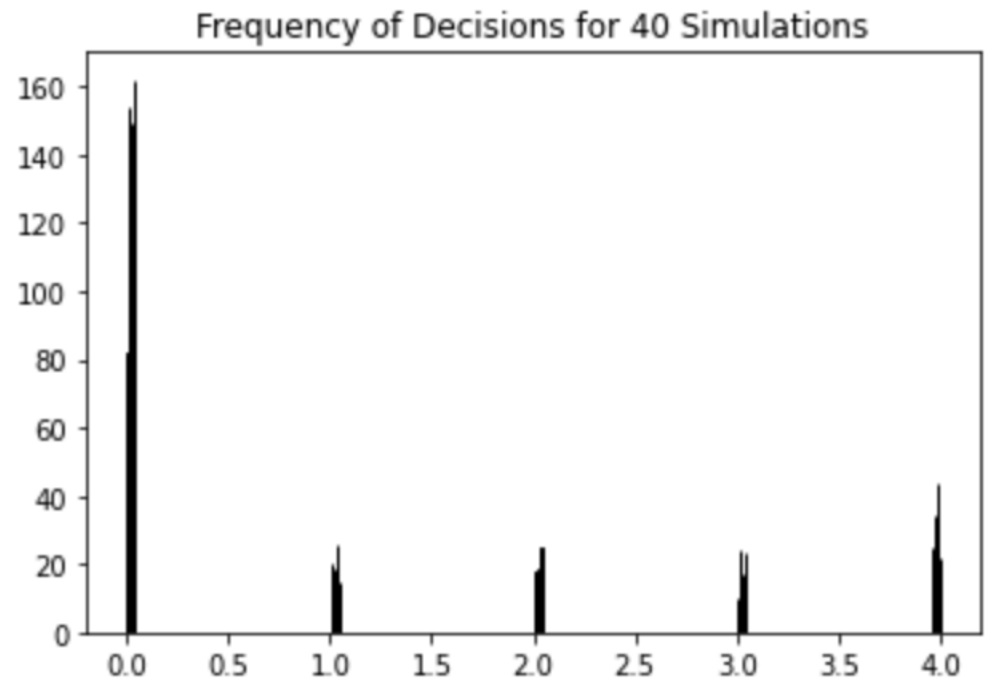
```python
# Bootstrap function
# Resamples from
def boot(data, B):
    win_perc_est = []
    win_count_est = []

    for i in range(B):
        # Sample with replacement
        boot_sample = random.choices(data, k = len(data))

        # Compute percentage win
        perc = sum(boot_sample)/len(boot_sample)
        # Average wins
        count = sum(boot_sample)

        # Append results
        win_perc_est.append(perc)
        win_count_est.append(count)

    return win_perc_est, win_count_est
```

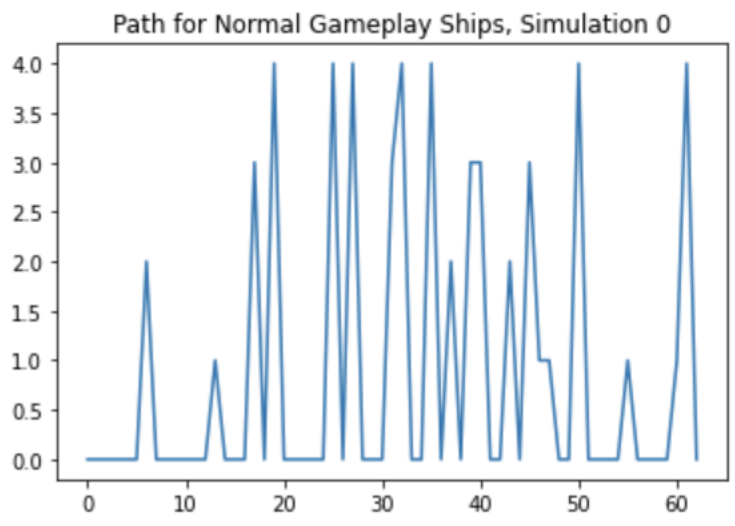
From this, we are able to find the mean, variance, and distributions for the win percent and counts. After the runs, it was found that the mean win percentage is 28% with variance .5% and the mean win count is 11 out of 40 with variance 8.25. The below histogram shows the distributions for each, which are both distributed normally around the mean.

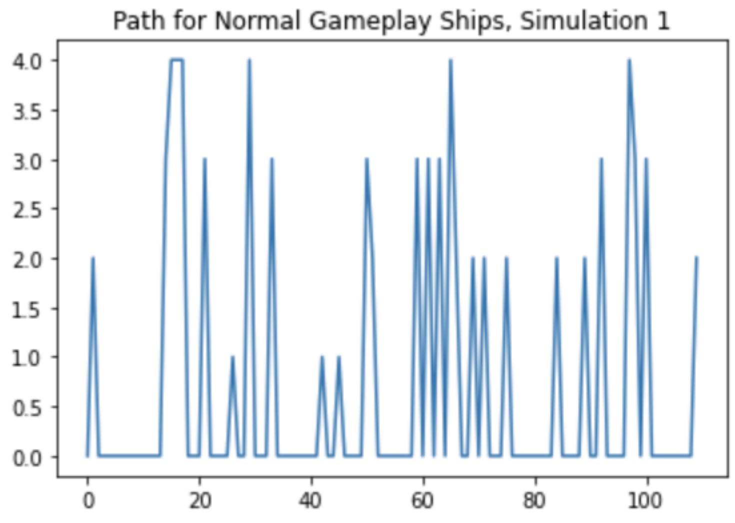


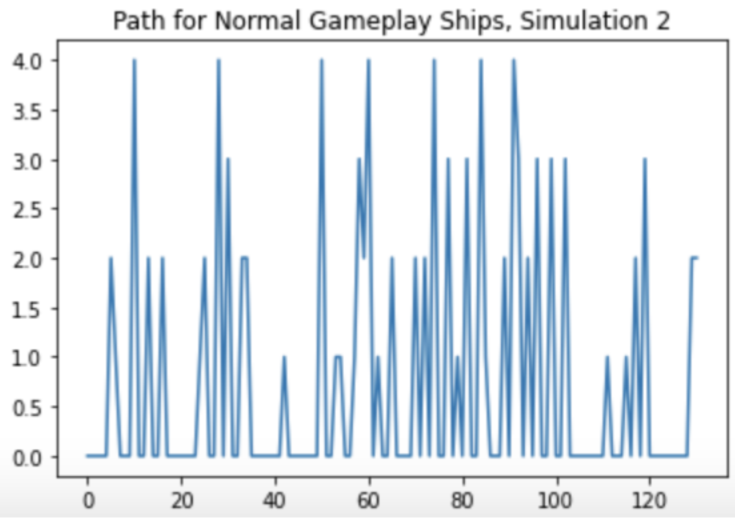


These results and figures point to the fact that the MDP learner is not performing great. Likely, this is because it is simply crawling along the board and not taking into account the last known hit location. A slight edit to the code to look at the last hit could be done in order to improve this. We now look at the distribution and some example paths to confirm this.



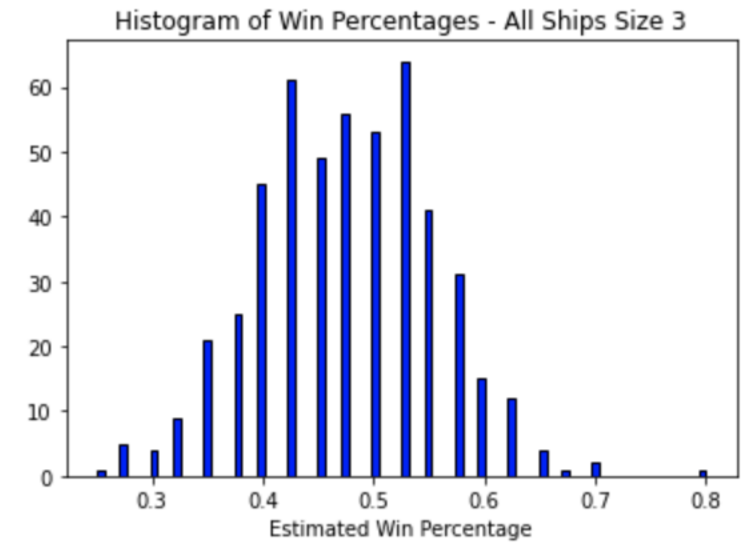


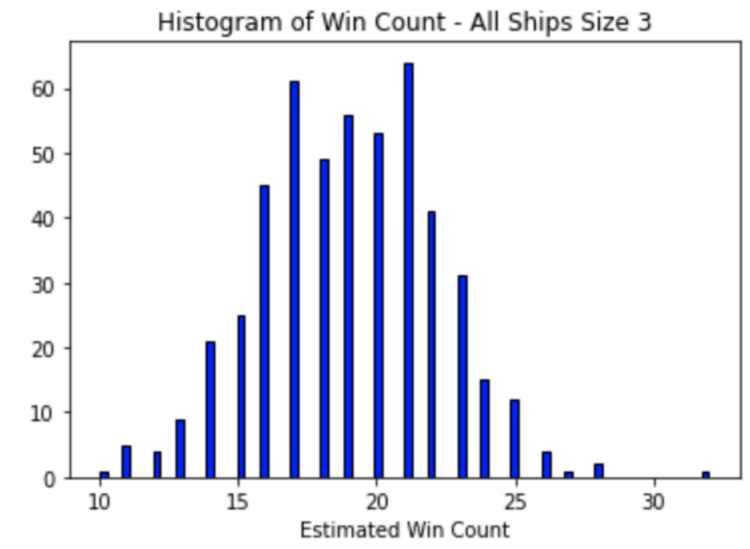


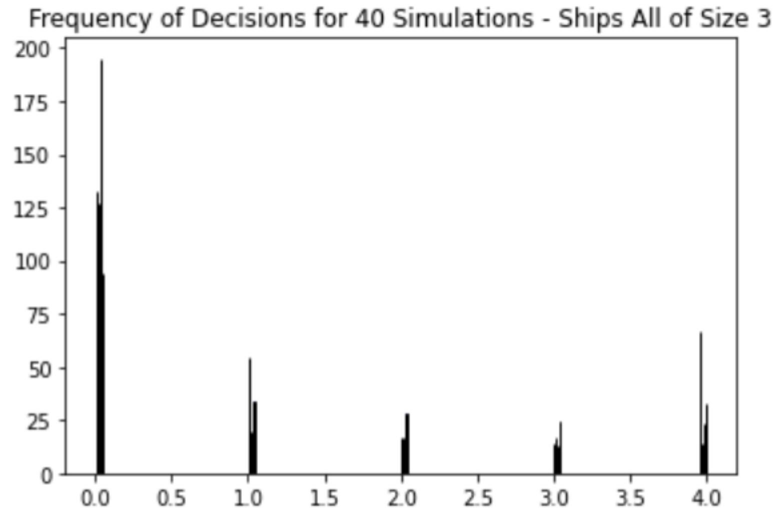


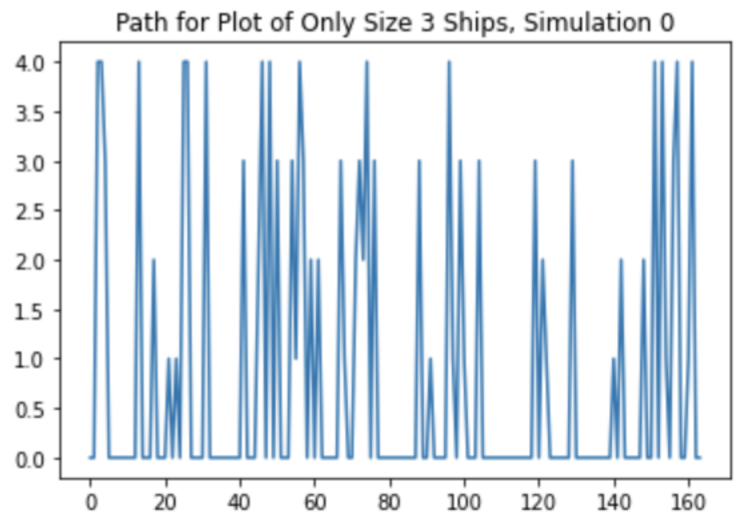
Looking at the above figures, we note that quite clearly the MDP player is overwhelmingly choosing to move up from the last position. As the sims suggest, we tend to stay at 0 before jumping up to random choice for a majority of the game. Likely it moves up until it checks the full column then moves over one or chooses randomly to repeat. This is not a bad strategy, but it is not optimal.

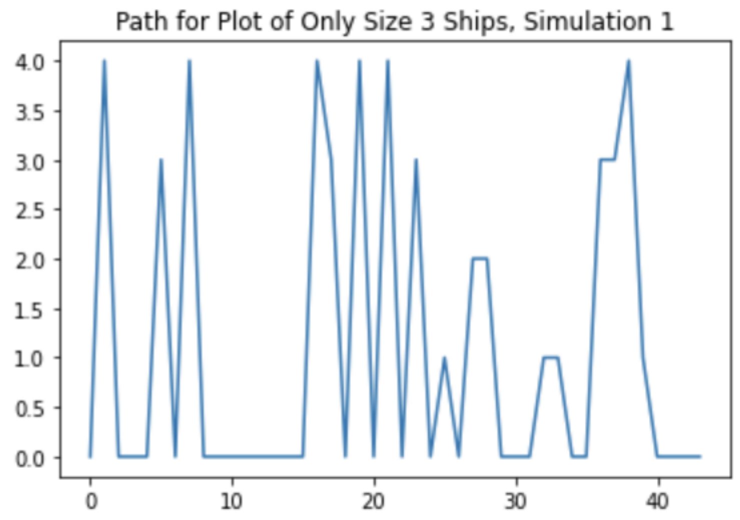
Now, we can see if this strategy is effective in a specific scenario. The above experiments were re-done where all ships had size 3. Logically, the strategies employed in the model are more effective and tailored to ships of size 3. Repeating the bootstrap we find the mean win percentage is now 47.5% with a varince of .64%. The mean win count is now 19 out of 40 with variance 10.2. The below distributions also confirm normality, as before. Looking at this, it seems the strategy does do well for ships of this size. We have nearly an equivalent win rate between the random and MDP players and further tweaks could lead to the MDP having the edge. Likely the large size ships were causing the low win rate for the previous simulations so updating our strategy, which will be discussed below, we may be able to match this win rate for the ships of varying size. Further, it seems the path taken for ships of size 3 is significantly more varied, but often after choosing to move up we see it again choose to go random then back to up here and there. Improvements can be made to prevent this strategy.

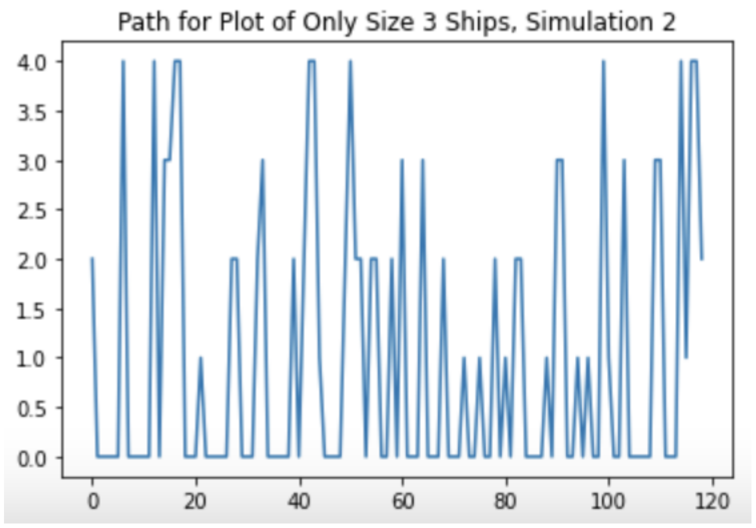





Frequency of Decisions for 40 Simulations - Ships All of Size 3
0
25
50
75
100
125
150
175
200
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0







Thus, as we can see by our evaluation, the current MDP can come close to the random player in the right situation. However, in general game play the current setup is only effective about 30% of the time. Thus, changes to the program can be made to improve this and make the MDP a formidable opponent when playing against a human.

While the MDP based player was not able to beat random chance, it was not far off from winning more than 50% of the time. There are a number of enhancements that can be made with proper time given. The simplest update that could be made is to take the actions based on the last hit square rather than the last known square. Applying the actions around the last hit square that didn't sink a boat is more likely to yield a potential hit than working off the last square. The second is updating the number of possible actions. In this project, it was assumed that the MDP would take the actions of moving left, right, up, or down from the last known move or randomly choose a square. Human action is more likely to search around a single square either directly or within a certain distance. Searching directly around yields a higher probability of a hit, but if only ships of a certain size are left a wide reach might be a better strategy. Increasing the actions to allow for movement up to the largest ship size away from the hit would help add more strategy to the MDP based player. This should be doable by having the transition probabilities equivalent to 1/(number of unknown squares y distance from the last hit) times the number of ships at least of size y+1/total number of ships in play. This probability would represent the probability of having a ship large enough to be on that square and that it is on the square chosen. It is important to note that according to **MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problem** page 919, "Because of the inherent computational complexity, solving large state or action space MDP can be prohibitive for modern computers." Thus, we would need to be conscious of the trade off of adding complex actions to improve the player and ensure we aren't rendering the game slow or unplayable. A third enhancement would be to add in logic to change the transition probabilties for actions dealing with chains of hits. If there is a chain of hits, a human would likely search on either end of that chain. The logic currently allows for some of this, but is not as robust as it should be. Further, this could be updated to handle the case there are all unknown tiles between 2 hits and we could increase the probability of those tiles being a hit. If we can make these updates, our MDP players gameplay will rival that of a humans and allow for greater wins.

As we've seen in this application of MDPtoolbox, the use of even the simplest of MDP is complex to set up but very robust and easy to apply once the initial problem is well stated. The most basic of rules was able to get very close to beating a basic random number generator based player and was able to do damage to human player ships in trials. Through proper evaluation, the current model could be tweaked in order to produce a truly intelligent opponent. While the win rate was not perfect, through the work around MDP a fun and intelligent program has been produced that will only improve with time. Thinking like a human is complex, but with Markov decision theory it is an achievable task to unravel in the computing and math worlds. There are numerous applications of the work done here in solving problems of landmine deactivation on a beach, fighting in true war, finding and removing the full margins of a tumor, and other spacial based hunting problems. All these are possible from this work, though, only by the theories interaction with a simple childrens game. This shows that adding a little bit of fun to mathematics and finding even the most basic of applications can still drive creativity and innovation in a powerful tool like decision theory.

Bibliography

1. Robert Gallager Stochastic Processes Theory for Applications
2. Chadès, Iadine, Guillaume Chapron, Marie-Josée Cros, Frédérick Garcia, and Régis Sabbadin. "MDP-toolbox: a multi-platform toolbox to solve stochastic dynamic programming problems." Ecography 37, no. 9 (2014): 916-920.
3. Denton, Brian T. "Optimization of sequential decision making for chronic diseases: From data to decisions." In Recent Advances in Optimization and Modeling of Contemporary Problems, pp. 316-348. INFORMS, 2018.
4. Nicolas Privault Understanding Markov Chains
5. Tomas Kancko Reinforcement Learning for the Game of Battleship https://is.muni.cz/th/oupp1/Reinforcement_Learning_for_the_Game_of_Battleship.pdf
6. Sudharsan Ravichandiran Hands-On Reinforcement Learning with Python
7. An Artificial Intelligence Learns to Play Battleship https://towardsdatascience.com/an-artificial-intelligence-learns-to-play-battleship-ebd2cf9adb01
8. Deep Reinforcement Learning-of how to win at Battleship https://www.ga-ccri.com/deep-reinforcement-learning-win-battleship
9. https://willfondrie.com/2022/01/how-to-use-r-packages-in-python/
10. https://www.hasbro.com/common/instruct/battleship.pdf
11. https://www.youtube.com/watch?v=iKdlKYG78j4