Aditya Agashe (ava9)
Susan Li (szl5) **Due: December 15, 2016**

CS 4701: Battleship Against Intelligent Adversary Project Proposal

Introduction:

Battleship is a two-player, ***nondeterministic*** and ***partially observable*** game where the starting state is determined by each player:

1. Having 4 ships (our variation of the game uses 4 ships, some variations use 5 ships) - one 5 block ship, one 4 block ship, two 3 block ship, one 2 block ship
2. Having a 10x10 grid to place the ships (for our game, we experimented with an n x n grid where 5<=n<=15)
3. Must placed all 4 ships in n x n grid such that each ship can be placed in any horizontal or vertical position, but not diagonally
4. Must placed all 4 ships in n x n grid such that no part of any ship can overlap with any other ship or with the edge of the grid
5. Once all 4 ships have been placed, they cannot be moved

Battleship is played as follows:

1. Each player will alternate turns and call our one shot per turn to try and hit each other's ships
2. If a player calls out a shot location that is occupied by a ship on the opponent's ocean grid, then the shot is a hit - the opponent is required to tell you which ship you have hit
3. If a player calls out a shot location that is not occupied by a ship on the opponent's ocean grid, then the shot is a miss
4. Once all of the locations a ship occupies have been hit, the owner of the ship has to declare that the ship has been sunk
5. The player who sinks the opponent's fleet for 4 ships first wins the game

The game of battleship has a contingency problem, since the opponents actions are uncertain, and every move provides new information about the current state. This is why a heuristic based strategy helps when playing the game.

Project Description:

1. **Goal:** We are looking to build a battleship game in Python. We plan on developing a text based REPL interface through the terminal for a user to play the game against the computer. We will write an AI algorithm to intelligently play against the user and make the best possible combination of moves.

2. **Game Play:** The user will start the game by running our program. The user will be asked to place their battleships on the board. The user will have 4 ships - one 5 block ship, one 4 block ship, one 3 block ship, one 2 block ship. The computer AI program will automatically randomly place the ships on its board. The user and the computer will then take turns guessing coordinates on the n x n board for locations of the opponent's battleships. Once a guess is made, the board will get updated. The REPL will display both boards - the user's board and the computer's board with the updated "hits". Once a player wins, the REPL will display the winner and ask the user if he/she wants to play again.

3. **Architecture:** The game consists of 3 main components
    a. The user - The user interacts with the Game State module through the REPL text interface
    b. Game state - This handles all attributes of game play, from managing the two boards, ships of the user and AI, to getting moves from the AI and calculating hits and misses. It interacts with the AI using the AI interface.
    c. AI - The AI handles making moves in response to user moves, the information of which is passed to it by the game state. It returns its moves to the game state through the AI interface.

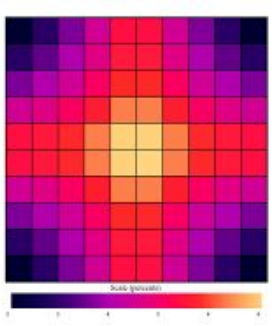
General Approach:

There are a few basic strategies to play battleship:

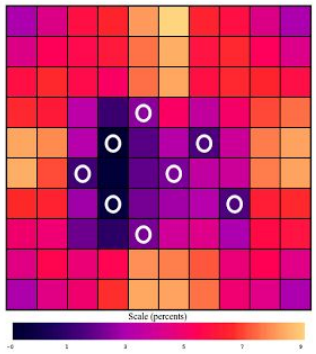
1. The most basic battleship strategy is to select a square in order (square 1, square 2, square 3).
2. The next most naive approach is to select a random square.
3. A heuristic that improves on this method is that you fire a shot at every nth square, where n is the length of the largest ship still alive.
4. We can improve the above heuristic by implementing a "hunt" function (once a hit has been detected, the AI will fire up, down, left, right to kill that ship before randomly selecting another square).
5. The final improvement is to implement a game tree search: Instead of simply picking the next n-parity square that is available, we will make our selection based on a game tree with one level. We will choose the square that if we miss, eliminates the maximum number of locations that the ship could still be.

Aditya Agashe (ava9)
Susan Li (szl5) **Due: December 15, 2016**

Illustration of the final "seek" functionality:



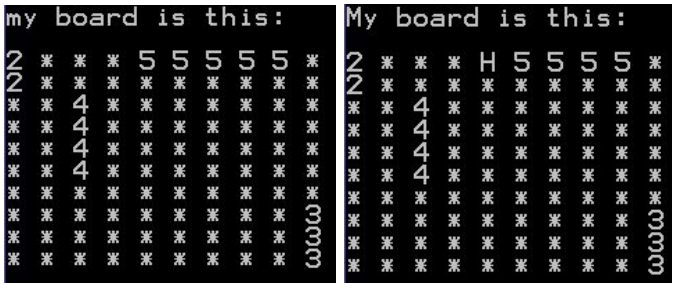
At the start of the game, when the AI is searching for our 5-ship, this is what the heat map looks like. It will strike one of the bright yellow spots in the center. We used a one-level game tree to determine for each spot how many possible length-5 intervals overlapped with that spot. Then we choose a spot with the highest value. In a way, this can be seen as choosing the spot that would eliminate the greatest amount of entropy (give us the greatest information gain).



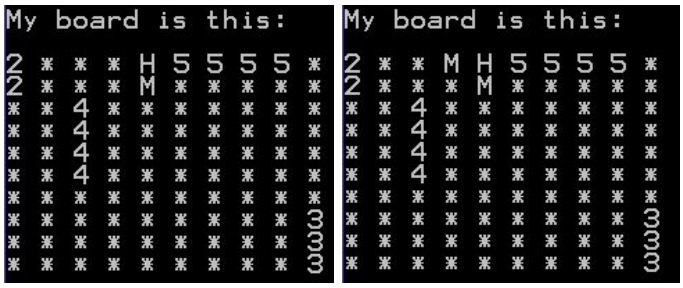
After a few misses in the center of the board, this is what the heat map for a 3-ship would look like. Now that the hot spots in the center have been eliminated, the next highest probability areas are the centers of each edge.

(Images taken from
http://thevirtuosi.blogspot.com/2011/10/linear-theory-of-battleship.html)

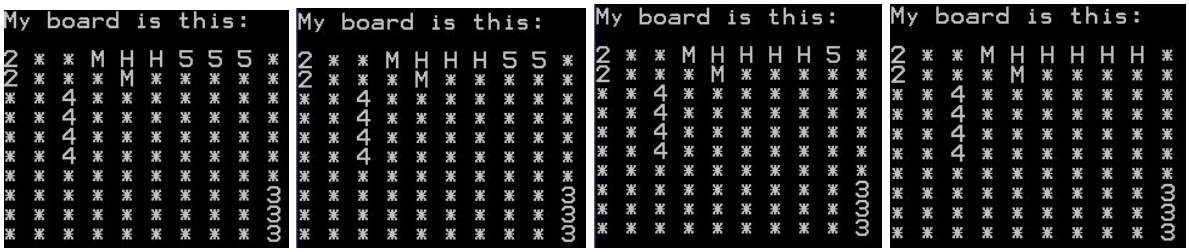
Illustration of the final "hunt" functionality:



The AI has found a ship



The AI is searching the neighborhood of the first hit



The AI successfully completes the kill

System Evaluation (Quantitative/Qualitative):

We tested the code that we wrote by breaking it up into the following the following testing plan:

A. Game Initialization
   a. The REPL is able to start the game
   b. When the game starts, the empty board is displayed correctly
   c. The REPL doesn't crash when unknown or invalid commands are inputted

B. Ship Placement
   a. User is able to place ships on board using REPL

i. Incorrect ship placement displays a text error message and does not update the board
ii. REPL is updated once the ships are added

b. The AI is able to randomly place ships on the board correctly
i. Ships should not overlap with one another
ii. REPL is updated once the ships are added
iii. Ships cannot be placed on invalid locations

C. Playing Game (Taking Turns)

a. User is able to choose a place to target and enter into terminal
i. If target is within the boundaries of the board, update user's board
ii. If target is out of bounds, display test based error in REPL and ask user to make another move
iii. REPL displays whether move was a hit or miss
1. Past moves are displayed on board, but user may still make the same move again
iv. REPL displays whether ship is sunk
v. REPL displays whether user won

b. AI will choose a location to target
i. Board is updated (because AI will always choose a target/move within bounds)
ii. AI will store past moves, so AI will never make the same move twice
iii. REPL displays whether move was a hit or miss
iv. REPL displays whether ship is sunk
v. REPL displays which player won when goal state is reached

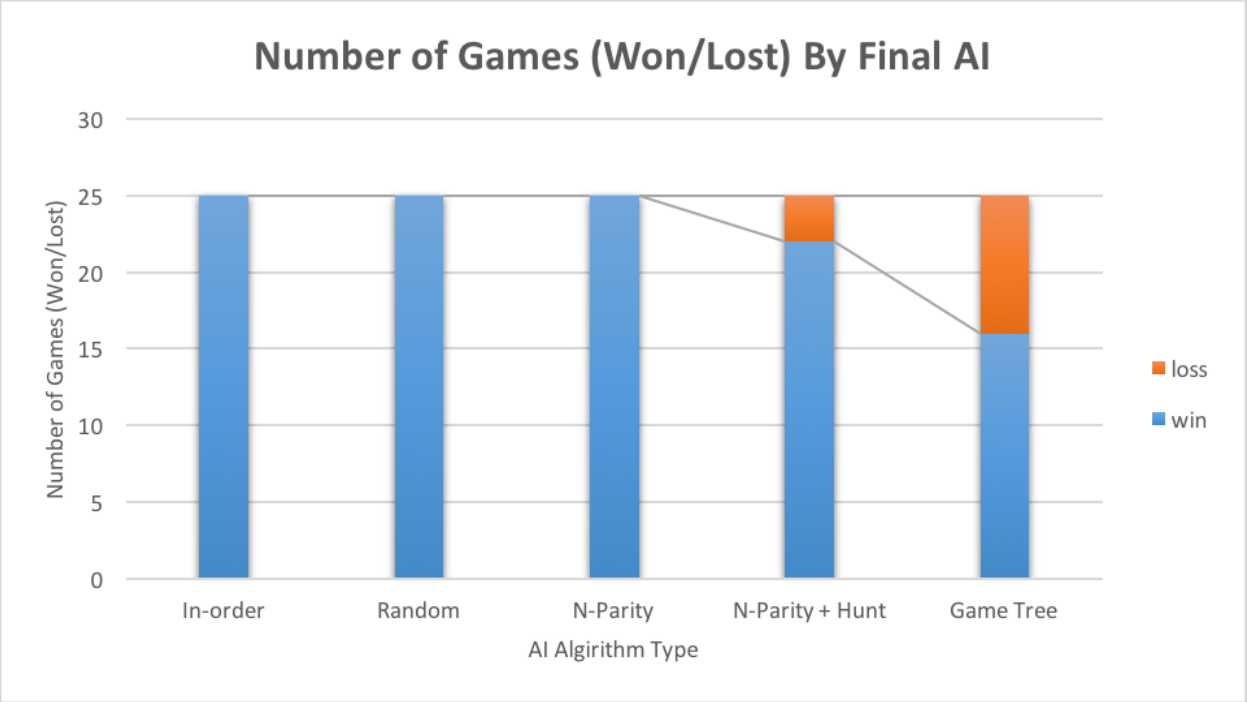
Qualitative Evaluation: To qualitatively evaluate our AI, we wanted to see how many times each iteration of the AI could beat a smart human player.

We tested the effectiveness of our Battleship AI iterations by pitting them against a human opponent (Adi and Susan) playing using an optimal/smart strategy. We tested the following iterations our Battleship AI:

1. Naive (chooses next empty spot: A1 then A2 then A3)
2. Random (chooses random empty spot)
3. N-Parity heuristic (chooses every second/third/fourth spot depending on which is the smallest ship still alive, chooses next available spot, e.g. A3, A6, A9, B2, B5….)
4. N-Parity heuristic + Hunt heuristic
5. Decision Tree (probabilities) + N-Parity heuristic + Hunt heuristic

The results for the probabilities of us winning against our Battleship AI iterations are shown below in ***Chart 1***.

Chart 1: Win:Loss Ratio vs. AI Algorithm



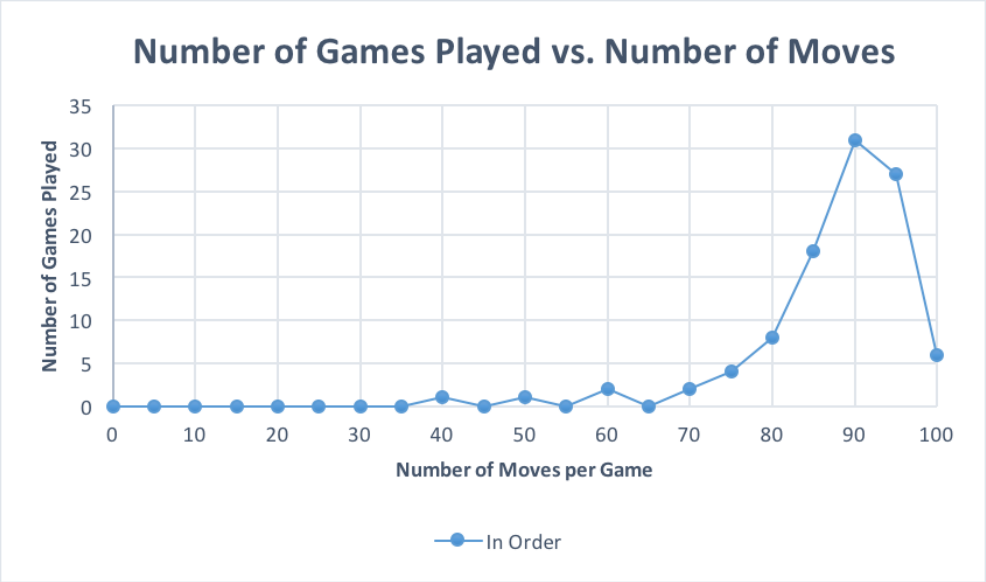
As the chart shows, our algorithm consistently won the majority of the time against all 5 adversaries - the human opponent win ratio was roughly 3:2 for our AI (we beat the human 3 times every 2 times we lost).

Quantitative Evaluation: Not only did we want to see what percentage of times our AI achieves a win against each level of adversary, but we also wanted to run our AI against 100 randomly generated moves and count the moves it took for the AI to hit all of the ships.

The chances of the AI using the random firing strategy to play a perfect game are roughly 1 out of 355,687,430,000,000. For each of the strategies that we considered for the AI, we ran the AI against 100 randomly generated moves and counted the moves it took for the AI to hit all of the ships. Below are the graphs for each of the strategies:

We ran the most basic battleship strategy is to select a square in order (square 1, square 2, square 3) against 100 randomly generated 10 x 10 battleship boards. The results are shown below in ***Chart 2***.

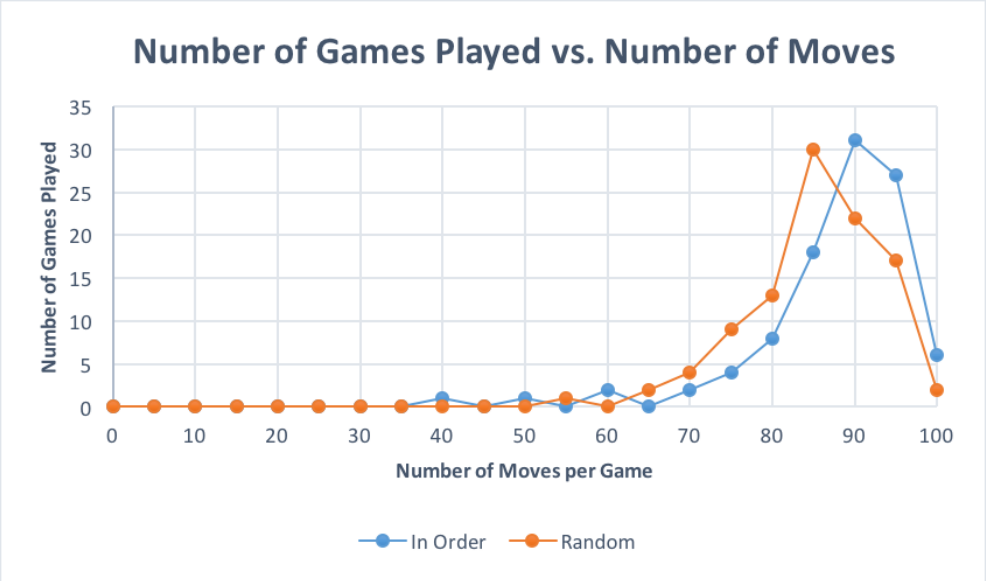
Chart 2: In-Order Firing Strategy



These results show that the peak in number of moves per game was ~90. There were a few outliers in earlier (around an average number of moves >=40 and <=65). These outliers can be attributed to the random placement of all ships in the top 50 squares (top half of the board). Since we randomly generated the ships on the board, this was one of the scenarios that presented itself in the ship positions.

Next, we ran the next naive battleship strategy to select a random square against 100 randomly generated 10 x 10 battleship boards. The results are shown below in ***Chart 3***.

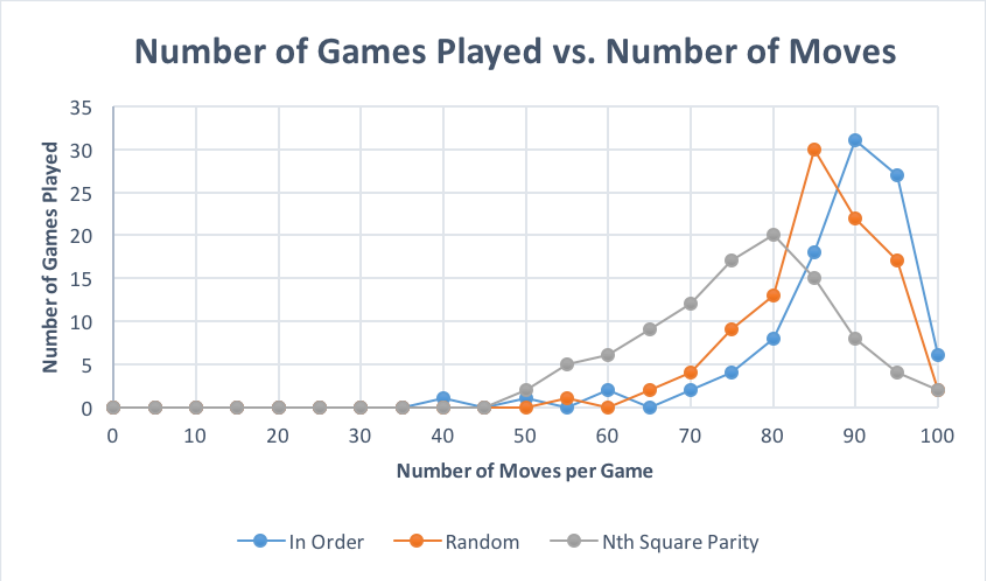
Chart 3: Random Firing Strategy



These results show that the peak in number of moves per game was ~85, lower than the peak from the in-order algorithm (~90). There were a few outliers in earlier (around an average number of moves >=40 and <=65). These outliers can be attributed to a few lucky games the AI played where the random hits actually sunk all 4 ships.

Next, we ran a heuristic that improves on the random method by firing a shot at every nth square, where n is the length of the smallest ship still alive. This strategy was run against 100 randomly generated 10 x 10 battleship boards. The results are shown below in ***Chart 4***.

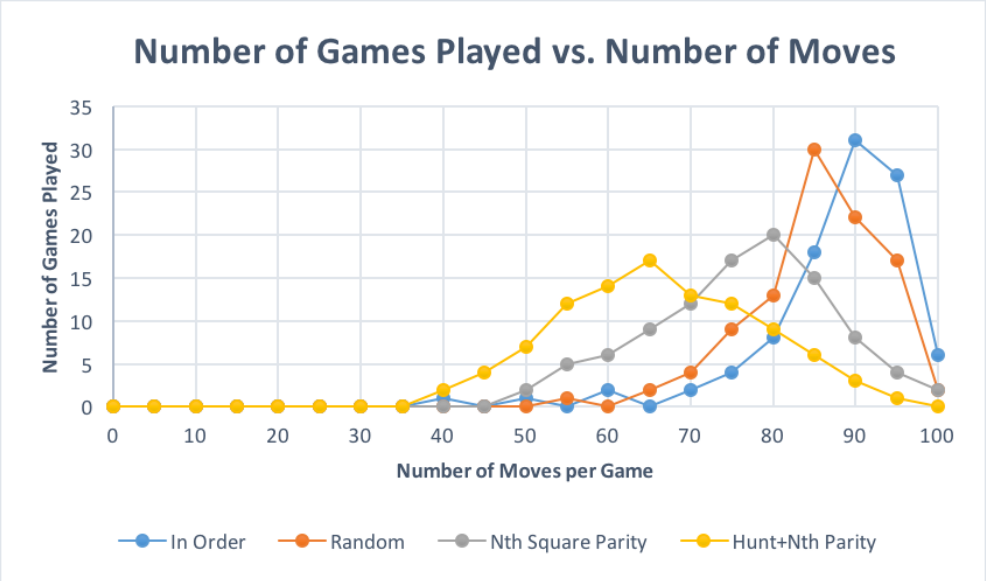
Chart 4: N-Parity Heuristic Firing Strategy



These results show that the peak in number of moves per game was between 75-80, lower than the peak from the random algorithm (~85). The shape of this curve is much wider than before, but shorter in height - this demonstrates that the average number of moves per game is spread out over a larger range (from roughly 60 to 90) rather than spiking to a peak in a shorter range.

Next, we ran a heuristic that improves on the n-parity method by hunting (once a hit has been detected, the AI will fire up, down, left, right to kill that ship before randomly selecting another square). This strategy was run against 100 randomly generated 10 x 10 battleship boards. The results are shown below in ***Chart 5***.

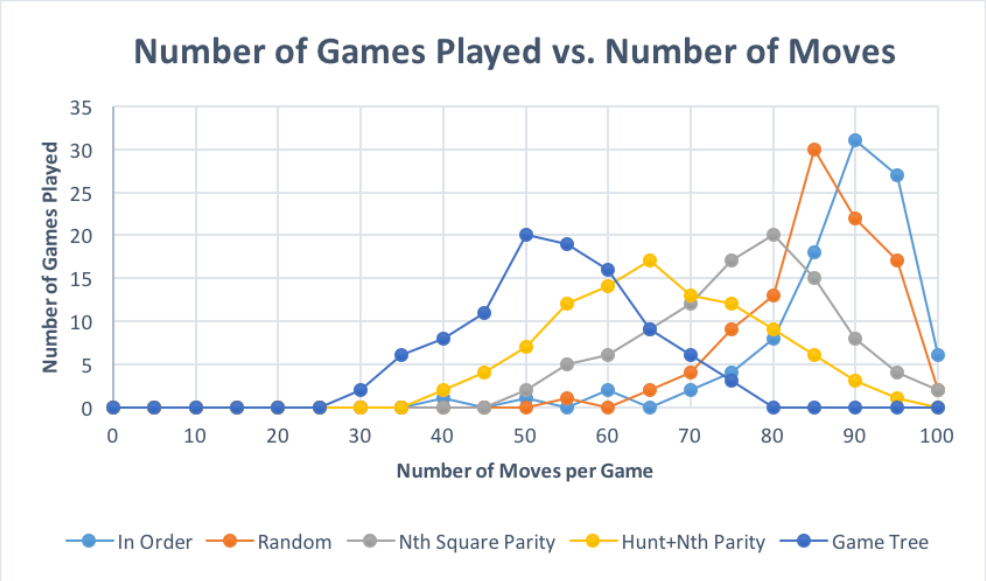
Chart 5: Hunt + N-Parity Firing Strategy



These results show that the peak in number of moves per game was between 55-75, lower than the peak from the random algorithm (~80). The shape of this curve is bit wider on both the left and right sides than before, but slightly shorter in height - this demonstrates that the average number of moves per game is spread out over a larger range (from roughly 45 to 85) rather than spiking to a peak in a shorter range.

The final improvement is to implement a game tree search: Instead of simply picking the next n-parity square that is available, we made our selection based on a game tree with one level. We choose the n-parity square that if we miss, eliminates the maximum number of locations that the ship could still be. This strategy was run against 100 randomly generated 10 x 10 battleship boards. The results are shown below in ***Chart 5***.

Chart 6: Game Tree Firing Strategy



These results show that the peak in number of moves per game was ~50, lower than the peak from the random algorithm (~65). The shape of this curve is bit steeper and sharper on the left side than before, and slightly taller in height - this demonstrates that the average number of moves per game is more concentrated over the peak range (from roughly 45 to 65) rather than being spread out over a large range.

Conclusion

Most humans, when they play battleship, easily intuit the heuristic of hunting down a ship once a hit has been placed. Advanced players can also conduct a game tree search in their mind of choosing a square with a highest probability mass of containing part of some ship. It is a testament to the power of the human brain that these decisions can happen naturally and in a split second.

We learned how challenging it is to even implement a heuristic that most humans are able to come up with naturally. So much data and so many rules had to be encoded for this AI to have a fraction of the knowledge that humans have. And still, it is difficult to integrate all the subtle nuances that people hold intuitive.

However, we were surprised to find just how effective the probability heat map ship-seeking function was. While playing against the AI, we were impressed by its ability to effectively locate my 5-ship within a small number of steps. In our mind, we would not

have been able to locate the 5-ship that effectively, and this speaks to the power that the one-layer game tree has.

We've thought about extensions that we could add to this project if we had more time. One would be the more nuanced heat map suggested in http://thevirtuosi.blogspot.com/2011/10/linear-theory-of-battleship.html where the probable locations of the smaller ships are affected by those of the larger ships. For example, since the middle squares have a much greater probability of overlapping with the 5-ship, the heat map for the 2-ship shows it having a greater affectation for squares near the edges, since the middle is taken by the bigger ships. So instead of considering the game tree for each ship individually, we could consider the probability of layouts for all the ships as a whole.

Another extension could be to try to find actual statistics about where people tend to place certain ships (e.g. maybe humans tends to place the 5-ship in the bottom right) as opposed to just probabilistically where they could be. We could use that data to give certain squares a greater weight for their corresponding ships, and that might help us better seek out those ships.

In addition to improving how we seek out ships, we could possibly improve how we execute a kill. Currently, our hunt heuristic looks up, then down, then left, then right; it's the same regardless of where the first hit is. Perhaps we could also use a probability heat map to determine which orientation the ship is most likely to have, given its size and location of first hit.

References

1. Alemi. "The Linear Theory of Battleship." *The Virtuosi*, 3 October 2011, http://thevirtuosi.blogspot.com/2011/10/linear-theory-of-battleship.html. Accessed Dec. 2017.
2. Russell, Stuart, and Peter Norvig. *Artificial Intelligence: A Modern Approach - Third Edition*. Prentice Hall, 2010.