**Strategy for Minimizing Shots in Battleship**

Nicholas Winter and John Paul Depew

1. INTRODUCTION

Battleship is a game that involves a great degree of chance. Regardless of the random nature of the game, patterns can be found to improve performance and increase the odds of winning a match. Ship placement can be tailored based on expected hit areas of the other player. Ship targeting can be influenced by the number of unexplored cells surrounding each cell, clustering detection, and strafing patterns.

For ship placement, a simple learning algorithm can be used to decrease the chances of an AI player’s ship being hit. At the end of every game, the AI records the locations of the cells that its enemy targeted on its board, and saves it to a frequency table. The table contains the number of times each cell has been hit. The AI uses this chart and places ships in the cells with the minimum values. When our Battleship program plays in AI vs AI mode, this tends to result in a ship clustering placement strategy. Clustering ships in areas that are targeted less frequently tends to have the highest success rate in surviving for a greater amount of moves. Our program does not use the Monte Carlo algorithm for ship placement utilized by the researchers at Pennsylvania State University, an adaptive strategy is utilized that learns from its opponent's moves [2]. The program starts with no knowledge of the opponent’s strategy and places ships in a random fashion. It then begins to adjust its placement based on the locations the enemy targets on its board. This strategy should result in the AI taking fewer hits over a period of time.

The main strategies used for targeting were heatmaps, strafing patterns, and clustering detection. The heatmap was simple, raising and decreasing cells’ values depending on how many open spots were surrounding each cell. To detect clustering, the algorithm would revisit areas where a ship should have sunk but didn’t, the ship’s coordinates were not aligned, or the length was greater than the largest possible length.

One strong strategy for targeting in Battleship is to use a strafing pattern based on a number of subgrids of the battleship board [1]. Within these subgrids, values can be assigned to each cell based on the frequency of a specific ship appearing there. These values are used to create a strafing pattern for searching for ships, and the pattern is specific to each size of the ship being looked for. Once completed, this pattern is guaranteed to find all the ships except the ship of length two, while only searching one-third of the board. In addition, the algorithm uses a cleanup pattern to find the smaller ship while minimizing the necessary number of cells to target.

1. OTHER CONSIDERATIONS

Minimax with Alpha Beta Pruning is commonly used in two player games and was considered for this project. However, this algorithm has certain limitations. It is dependent on knowing how one player’s actions will affect the other player’s actions, and how each player will respond to each other. In battleship, a player’s actions have absolutely no effect on the other player’s actions, and one cannot judge what the other player will do based on one’s own actions. Therefore, our project Minimax did not seem like the ideal algorithm for this project.

1. PREDICTIONS

The expected results of this project are that utilizing heatmaps, strafing, and clustering detection will lead to fewer average turns for each game compared to versions of the AI that does not use these features. Each of the above categories will be studied separately. In addition, the heatmap is expected to show something like quadratic as opposed to exponential growth when increasing the board size. It iterates through each cell (L \* L = number of cells) and for each cell, it looks the length and width of the board L + W = 2L. Since L = number of cells / L, the complexity is N \* N / L, or O(N2/L), N being the number of cells, and L being the length of the board.

1. MAIN RESULTS

The heatmap assigned each cell a value based on how many open spots surrounded that cell, and how many hits surrounded that cell. For example, a cell surrounded by four misses would receive a value of 0, but a cell in an open area or an area near a hit would receive a much higher value.

However, there were a few edge cases. For example, a cell surrounded by three misses, as well as a cell containing a sunk ship, could not be marked as zero. Even though the ship next to it was supposedly sunk, there could have been two ships together, and the sunk cell could actually be part of another ship.

The heatmap was surprisingly disappointing, with little to show regardless of whether or not it was activated. Figure 1 shows the heatmap used for all parts of the search, only for cleanup, only for the strafing, or not at all. The results are based on about 20000 plays of the game and ended up with about a 0.2 difference in results number of turns to finish a game.

Figure 1. Graph displaying the number of total turns for two AI agents comparing heatmap use versus random selection for strafing and cleanup, only cleanup, only strafing, and neither.

The heatmap did appear to match our complexity predictions. Figure 2 displays the result of running the heatmap on different board sizes, and the graph does appear to be quadratic as opposed to exponential or linear. Modifications to the program enabled taking into account that the maximum ship length was 5 changed the complexity from O(N2/L) to simply N \* 20, since each cell must explore five cells in each direction. This significantly reduced the growth factor.

Figure 2. Graph displaying the number of iterations to generate the heatmap based on the board size.

Clustering detection had much more noticeable results. The AI is programmed to revisit areas where it took more than 5 hits to sink a ship, and areas where the hit coordinates for a sunk ship were not aligned. It is also programmed to hop over a hit coordinate when searching for surrounding coordinates, considering certain edge cases with clustering. As shown in Figure 3, when the first check was turned off, games averaged about five turns longer. When the second feature (skipping over hit coordinates) was turned off, it took about fourteen turns longer to finish the games. These results lined up with our predictions.

Figure 3. Graph displaying the number of total turns for two AI agents comparing clustering detection, no clustering detection, and skipping previous hits.

The strafing pattern also yielded notable results, as shown in Figure 4. Utilizing the strafing pattern, which divided the initial number of shots by three, followed by random shots to find the remaining ship, was similar to the strafing pattern combined with a minimal cleanup pattern. However, they both outperformed random shots targeting even or odd squares by about five turns per game. They also outperformed completely random initial shots by about fifteen turns per game.

Figure 4. Graph displaying the total number of turns for two AI agents with strafing and cleanup, only strafing, random with parity, and purely random shots. Data for each column is based on 1000 games.

1. CONCLUSION

While Battleship is frequently seen as a game of chance, strategies can be used to increase the odds of winning a game. As shown in the paper, using a heatmap that uses surrounding hits and misses to generate values for each cell has very little effect. However, using logic to detect clustering as well as an initial strafing pattern to minimize shots significantly reduced the number of turns each game.

REFERENCES:

[1] E. Y. Rodin, J. Cowley, K. Huck, S. Payne, and D. Politte, “Developing a strategy for ‘battleship,’” *Mathematical and Computer Modelling*, vol. 10, no. 2, pp. 146–151, 1988.

[2] J. G. Bridon, Z. A. Correll, C. R. Dubler, Z. K. Gotsch, *“An Artifically Intelligent Battleship Player Utilizing Adaptive Firing and Placement Strategies”,* The Pennsylvania State University,