Battleship AI Final Report

Arjun Agarwal, Patrick Shugerts, Anxhelo Kambo, Alan Morcus
CAP 4621.001S24 Intro to AI
Dr. Valentina Korzhova

***Abstract* – This report goes over the implementation of several different AI algorithms that play against a user in the popular board game, Battleship. Over the course of this report, a general overview of how the game is played and programmed will be discussed. Additionally, an in-depth understanding of the different algorithms will be provided along with their respective statistical and benchmark simulation data that details their performance and effectiveness.**

I. Basic Introduction to Battleship

Battleship is a two-player game where each player has a 10x10 board on which they can place ships ranging in sizes 5x1, 4x1, 3x1, 3x1, and 2x1 in a non-overlapping configuration of their choosing. In our program, the second player is the AI and it places ships in a random configuration.

Each player takes a turn to fire at their opponent by providing exact coordinates on the opponent's grid. Of course, the player cannot see the opponent grid so a portion of the game relies on random guessing to obtain a result such as a "Hit" or "Miss". A Hit happens when the coordinates provided conflict with one of the ships placed by the opponent – a Hit on all coordinates a ship occupies will result it to sink. A Miss, as you may have guessed, is a shot at a coordinate where an opponent ship does not occupy. To win, the player must sink all opponent ships.

A. Game Initialization

The AI Battleship program we created is in Python thanks to its functional programming abilities and simplicity. In the following subsections, the base game is explained:

1) Player boards: Each player is provided a 10x10 player board to place ships (as discussed earlier) however, another player board is also provided to mark what shots that were fired were Hits and Misses. This allows the player to make strategic inferences on what moves to play next based on what has been hit. This will be important moving forward when explaining the different AI algorithms.

2) Ship placement: There are 5 different ships of varying rectangular sizes that the AI places on the board using Python's random module. This places ships in random squares and orientations allowing for a different game every time. This is a very important move especially for simulation and testing purposes (explained in section I.B) where input data must account for all types of situations to ensure peak algorithm satisfiability no matter the state and accurate testing with results that hold weight.

Figure 1 below shows a mid-game position of the two different boards. The one on top displays the hit-misses board and the one on bottom displays the ship locations board.

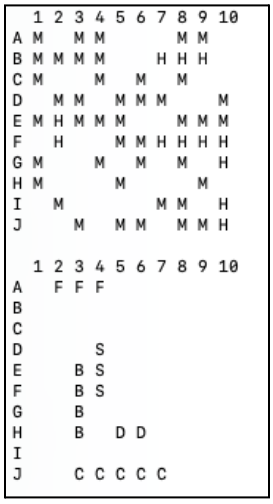

Fig 1: Player Boards

B. Simulation and Testing

The main control logic of our program sets up 2 loops that run the game. An inner loop that runs until 200 iterations (the max number of moves possible in a game). The inner loop defines one game. The outer loop can be set to any amount of iterations to simulate multiple games.

In order to run a simulation, both players must make moves automatically – so this involves choosing an AI algorithm for both players. This will allow us to test and benchmark these algorithms upon running a simulation to understand which is most effective in making the next best set of moves.

One of the simplest ways to make a move is for our program to just randomly guess. This would definitely be a very ineffective way to play Battleship as this involves no strategy however, running the algorithms we created against random guessing provides us a strong understanding of exactly how much more effective that algorithm exactly is by comparing it to the most elementary level of guessing. What combination of unique strategies and pattern matching can differ one algorithm from another and how it affects statistical data such as hit rate, win rate, time complexities, etc will be discussed further in Section II.

II. AI Algorithms that Beat Random Guessing

Before we talk about the AI algorithms, it's important to first understand random probability. It's safe to assume that running a simulation of 100 games with both players guessing randomly will result in roughly 50% win rate for both players. From our results, we obtained a 42% average win rate for the AI opponent after 3 trials of 100 games which lines up with that assumption. Our goal for this project is to beat random guessing and to accomplish this, we devised 3 different algorithms that are more than capable in doing so. If an algorithm can obtain an average win rate higher than random guessing after 3 trials, we conclude that this algorithm beats random guessing.

A. Description and Analysis of Algorithms

1) The Greedy Algorithm:

This algorithm is heuristic based and works by first random guessing until it identifies a hit. Once a hit is recognized, it assigns a higher heuristical value to all valid spaces around it and inserts these possible moves with higher priority into a min-heap based priority queue. It is essentially trying to pattern match by firing shots at squares near recent hits in attempts to sink a ship – just like how a real player would try to do.

As seen from table 1, the algorithm definitely beats random guessing with an average of 75.67% win rate after three 100 game trials. However, this algorithm isn't as effective as it could be due to it relying on random guessing for half of its shots. It also lacks more smarter implementations of assigning heuristics which could bring the win rate well into the 80-90% range. A smart Battleship player could win against the greedy algorithm most of the time.

	Greedy vs Random
Win rate % (Avg)	75.67%
# Hits (100 games)	1286.33
# Plays (100 games)	6373.67
Hit rate % (Avg)	20.18%
Time (sec)	0.374s

Table 1: Greedy vs Random benchmark data.

2) Custom Probability Algorithm:

The custom probability algorithm was the basis for the minimax algorithm. It uses two heuristics in combination to evaluate the current state space only. The first is a grouping evaluation. The algorithm assumes that a Hit registered on the board will increase the chance for an adjacent node to be Hit as well. The second is a rectangulation of the current game board for the highest existing size of ship remaining. It calculates where every possible ship remaining of the largest size could be in the state space and raises the probability of five most common intersection points among those possible ships.

This gave it the edge as a greedy algorithm is used to calculate the possible positions. This made it the fastest and least costly. However, by itself it is not a true AI and only reflexive in nature.

	CPA vs Random
Win rate %	99.67%
# Hits (100 games)	1694.33
# Plays (100 games)	6917
Hit rate %	24.38%
Time (sec)	0.393s

Table 2: Custom Probability Algorithm (CPA) vs Random benchmark data.

3) Minimax:

The algorithm works by passing the hits and misses of the ai to the minimax algorithm. The maximing player assumes a hit and the minimizing player a miss. This is carted out to the minimal depth of three for the search tree. The evaluation at the base nodes is the same Custom Probability Heuristic evaluation modified to fit the minimax algorithm.

This algorithm was extremely inefficient in large state spaces. The state space had to be reduced to gather metrics. The algorithm could only be used for the last 30 moves of play. The true time complexity occurs in limited factorial time.
O(n! / (n- d)!) Where n is the number of possible moves and d is the search depth. This means on an empty board the algorithm sorts through 100! / 97! moves each with their own heuristic calculation.

	Minimax vs Random
Win rate %	87.5%
# Hits (100 games)	1487.5
# Plays (100 games)	6402.5
Hit rate %	23.26%
Time (sec)	266.180s

Table 3: Minimax vs Random benchmark data.

II. Conclusion

With greedy not always able to obtain the next best move, custom probability not acting like a true AI algorithm in nature, and minimax being extremely inefficient in our implementation, we observed that each algorithm had one or more weaknesses however, our goal of beating random guessing was satisfied with each and proved with benchmarking data and statistics. A gathering of all this data (including trials) is presented in table 4 below. Note: The trials for minimax were before optimization.

Looking back, the ability for a smarter AI performance in Battleship can definitely be achieved. Hybrid approaches that combine the strengths of these algorithms could reduce weaknesses and make a more powerful AI opponent to really test the skills of real players on levels never seen before. Although our implementation for battleship was more focused towards simulation and benchmarking, creating a hybrid algorithm as well as a more finished game GUI could be the next big improvement to our game.

	trial #	win rate %	# hits (100 games)	# plays (100 games)	hit rate %	Time (sec)
Random vs Random	1	43%	731	3968	18.42%	0.065s
	2	38%	646	3509	18.41%	0.075s
	3	42%	714	3921	18.21%	0.068s
Average		42.00%	697.00	3799.33	18.35%	0.069s
Greedy vs Random	1	77%	1309	6506	20.12%	0.375s
	2	77%	1309	6449	20.30%	0.372s
	3	73%	1241	6166	20.13%	0.376s
Average		75.67%	1286.33	6373.67	20.18%	0.374s
Custom Probability vs Random	1	100%	1700	7016	24.23%	0.396s
	2	99%	1683	6916	24.33%	0.393s
	3	100%	1700	6819	24.57%	0.389s
Average		99.67%	1694.33	6917.00	24.38%	0.393s
MiniMax vs Random	1	62%	1054	5198	20.28%	362.495s
	2	66%	1122	5743	19.54%	364.155s
	3	61%	1037	5217	19.88%	364.390s
Average		63.00%	1071.00	5386.00	19.90%	363.680s

Table 4: All benchmark data and trials