

For our final project, we created an agent that can beat out a fully random opponent most of the time when playing the classic children's game Battleship. It uses a spiral based search pattern, checking all sections of the board for opponent ships, and when it finds them, it switches to a targeting mode where it determines the direction of the ship within a few guesses and quickly sinks it. The libraries used were random (useful throughout), time (used for testing), argparse (used for testing), and copy (used for testing, only needed deepcopy for boards). Other than that, the battleship logic was coded to match the official rules as found online from Hasbro, and our agent was entirely original code. For reference, a quick summary of the Battleship rules is below.

The goal is to sink your opponent's fleet of four ships before they sink yours. The game is set up with each player secretly placing ships on a 9x9 grid horizontally or vertically, with no part of the ship hanging off the grid. Then turns commence with the active player calling out a coordinate pair, to which their opponent must truthfully answer whether it was a miss or a hit, and if it was a hit, which type of ship was hit. The active player marks down the hit or miss on their target board, while the opponent only needs to mark if their ship was hit (*not* opponent misses). Then the other player gets a chance to do this, repeating until all four of one player's ships are sunk. The player who sunk the ships wins, and the player whose ships were sunk loses.

We tested our agent by having it play against a random opponent, and by comparing the empirical average clear times for a board. Our agent wins against the random agent over 99% of the time, taking about 30 fewer guesses to clear the board on average (over 40% more efficient!) The random agent takes upwards of 70 turns to clear a random board, while our agent clears the same board in under 45. We developed a robust test program that allows our agent and a fully randomized cpu to clear as many boards as are provided as args, as well as play as many matches against one another. This was helpful not only to demonstrate the behavior we expected from our agent, but also as a metric for how much better our agent was than simply guessing at random.

Our goal was to create an agent that could beat a random opponent, and play a full game against a human, and that has been accomplished. Our agent is definitely a challenge to beat in testing, as its efficient sweep of the board is almost sure to uncover our ships faster than we can compete. This was created through a checkerboard pattern, as the smallest ship is 2 spaces long, so we only need to check every other square on the grid, and a spiral pattern where it stores the board as a list of the squares needed to check (A1,A3,A5,A7,A9,C9,E9,...) and then iteratively selects from certain indices within the array. This allows us to have an efficient sweep, while still having randomness from game to game, preventing the human player from figuring out the CPU's pattern and placing their ships in the last section to be guessed.

We learned a lot about efficiency and algorithm creation throughout this project. We ran into a couple of major roadblocks when trying to refine the logic of the agent that we were able to brainstorm our way out of. For example, we consistently had issues with getting the targeting mode to work as intended. Initially we had all targeting in a single array, but we needed it to instead be by ship. We also had to eventually modify the targeting mode again so that if a second ship was hit while targeting, instead of resuming spiral consuming behavior upon sinking the first ship, it would properly begin targeting that ship. We also learned a lot about testing the efficiency of our agent in various ways, and this in the end was helpful in determining we had issues in our logic, since we expected the agent to complete the board in under 50 moves if it was properly navigating the checkerboard.

Here's a quick explanation of all functions created for the game, agent, and testing:

`spiral_in(board_size)`: creates a list of coordinates alternating (checkerboard pattern) while also spiraling towards the center of the board.

`create_board()`: returns a blank board object to be filled with ships.

`setup_player(name)`: runs through the logic for prompting user input to place all of their ships, asking for coordinates and orientations for each ship, and validating input.

`get_coords(prompt)`: prints a user prompt and takes user terminal input in the form of coordinates (A1, b2, C3, d4, etc.), cleans them up, and validates the input. Returns (row, col) value pair.

`place_ship(board, name, size)`: Presents the user with their current board, and asks them where they would like to place their ship **name** of size **size**. Uses `get_coords(prompt)` and gives users the option of horizontal or vertical ship placement, validates their choice, and if it is a valid spot to place the ship, it updates their board with the new ship.

`print_board(board, showShips = False)`: prints a board, with the option for it to show all ships or hide them (your board or your target board where you don't know where opponents ships are).

`all_ships_sunk(board)`: checks for unsunk ships on board, returns True/False whether they are all sunk, used for game end logic.

`take_turn(player, target, own, opp, health, cpu=None)`: runs through the logic of presenting a target board and personal board, asking where the user would like to shoot, and updating all necessary boards with the results. Also handles CPU turns, it does not display their information, simply tells you where they shot and the results of their attack. **player** is the name, **target** is their own target board, **own** is their board of ships, **opp** is the opponent's board (used to update

when they are hit), **health** is an array of health values for each ship used to check when a ship is sunk and display a message, and **cpu** is where you pass the cpu you would like to use (random or trained).

`clone(board)`: used for testing, is simply a shortcut for `deepcopy`.

`progress(i, total, label)`: prints progress updates to terminal for long testing experiments.

`mark_attack(board, r, c)`: used for testing, as we need slightly different logic to mark attacks on single boards to determine average attacks used per agent.

`simulate_clear(attacker, opponent_board)`: runs through an entire modified game of Battleship, with our agent shooting on a board with no back and forth, so that we can see the average number of guesses needed to clear a board.

`experiment_trained_vs_boards(trials=1000, show_progress=True)`: runs through however many iterations of `simulate_clear` as specified (1000 by default) on random boards to take the average guesses for our trained agent. Can show progress updates or not.

`experiment_random_vs_boards(trials=1000, show_progress=True)`: same exact function, just for a fully random agent, not our trained one.

`resolve_attack(attacker, board)`: actual logic for our CPU to decide where to attack, mark it on board, and process the results. Returns True when all ships are sunk.

`experiment_head_to_head(games=100, show_progress=True)`: Pits our trained agent against a random agent for however many games as specified (100 by default) with the option to show progress updates. It returns trained wins, random wins, and ties.