CISC 453 – Project

Group 13

Battleship with Reinforcement Learning

Team Members

Cian Allen – 20058113

Kai Laxdal – 20063565

Anthony Marsili – 20061996

Jared Taylor – 20075820

Introduction

Battleship is a very simple game where two players ‘engage in naval combat’ and try to sink each other's ships placed in unknown locations on the board. This game has started as a “pencil and paper” game, drawing inspiration from the popular french game *L’attaque* (Stratego) played by french soldiers. Part of the motivation behind this project was the simplicity of the game and its overall relevance to the content taught in class. Another motivating factor was the ability to create a unique AI agent to play a video game. We have all seen AI’s act as friend and foe in video games and it was especially interesting to see what happens behind the scenes when creating a competitive agent to play against. The complexity of a seemingly simple game makes this project an interesting choice for the scope of this project.

Problem Formulation

The main goal of our project is to make an agent learn to play the board game, *Battleship,* when they do not know where the opponent’s ships are. The main problem of our project (and battleship in general) is finding these enemy ships, and sinking them. In order to do this, we have decided on a State Space and Action Space. The State Space consists of the 1-based indexes of the player’s ships that are stored in a list. The states are represented as tuples. For example, the state (2,4) represents the state in which the only ships that have been sunk so far are the player’s 2nd and 4th ship have already been sunk. The state (5,) represents a state in which the player’s 5th ship is the only ship that has been sunk so far. The initial state is represented as (0,) and represents none of the player’s ships being sunk so far. The constraints of the state space revolve around the coordinates that make up the ships that the indexes in the tuples refer to. These constraints are: ships cannot be diagonal, ships cannot overlap, and ships cannot be placed outside of the grid. In addition to this, although it is not officially included in the State Space, the agent is aware of the actions that they have taken so far -- this is required for the agent to correctly choose which action to take next and helps in narrowing down the Action Space for each state. The Action Space consists of the different locations on the board that the agent is able to attack in each state. This is determined by subtracting the coordinates of the already sunken ships in that state by the list of all locations on the board.

In addition to our initial goal of having the agent learn to sink the opponent’s ships we also introduced the option for competitive play between a human and the computer. This allowed the player to attempt to sink the computer’s randomly placed ships and have the computer attempt to sink their ships, making a fun offshoot of the original RL problem.

Background

For our project we used Python as our primary coding language. Within Python, we used the numpy, tkinter, and itertools libraries/modules to assist with different features of our project. Numpy was essential in the implementation of the epsilon-greedy method, allowing for easy creation of random integers. Tkinter was used in order to produce the interactive GUI that is used to play the game. Although this GUI is not absolutely necessary for the core function of the RL problem, it is a very helpful tool to represent the results of the project in a fun, interactive way. Itertools gave us access to the Combinations library that was crucial in determining the list of tuples that make up the state space.

Design

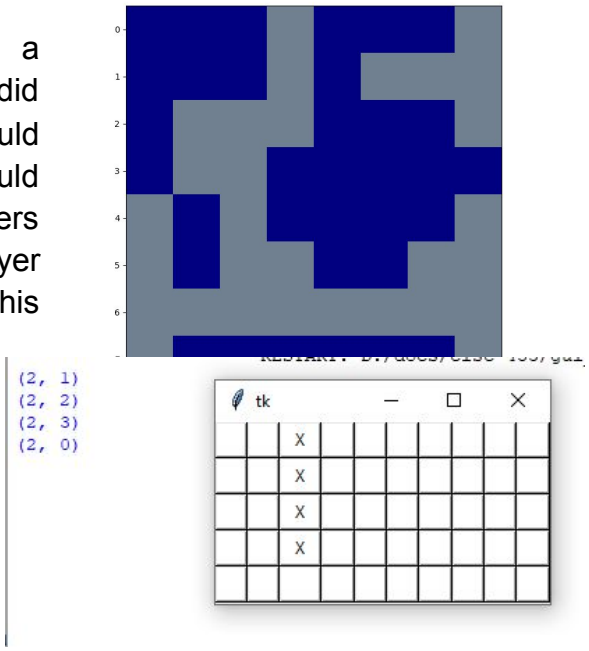
First we started with how the state space will be represented. It was decided that we would attempt to implement the entire game. This means that the state space would be based on an 8x8 grid that contains 10 vertically or horizontally placed ships: 1 carrier, 2 battleships, 3 cruisers, and 4 destroyers which are made up of 5, 4, 3, and 2 coordinates respectively. Next is deciding how the rewards will be calculated for each state. We chose a simple technique of making every miss return a reward of -1, every hit that does not sink a ship return a reward of 0, and every hit that does sink a ship return a reward of +1. This would encourage the agent to hit the correct locations and work toward sinking ships and therefore toward the goal state (i.e. all ships sunk). The agent's board, which was originally designed for testing our learning algorithm, is a randomly generated 2D array where each index represents a set of coordinates on the board. The board is built in board.py which creates a board object that initializes a 2D array of empty positions, and an array to hold the locations of all the place ships. Board.py then begins to fill the board by first choosing a random set of coordinates, checking the location by validating it with the state space's constraints, and then finally placing the ship iteratively in its chosen orientation. The constraints that the locations of the player's ship have are also: ships cannot be diagonal, ships cannot overlap, and ships cannot be placed outside of the grid. While all the ships are being placed, the object saves the location of each ship by appending a list of the ships coordinates to the ships array. This ships array is later used for building the agent's board for the player to attempt to sink the ships. As mentioned earlier this board object was originally used to train the agent, but as we shifted our aim to making a full game of battleship, we decided to use it to build the board that the player will play against while the agent will then use a board that the user creates to learn and choose actions.

Learning Algorithm

Early on in the design process, we decided on using the Q-learning algorithm. This algorithm involves storing and updating values for each state-action pair that is possible in the game. This value represents the quality of a given action in a given state. Depending on the algorithm's rate of exploration, the agent will usually lean towards choosing the action that has the highest q-value in a given state. By doing this, the agent is maximizing the reward in the next step; for our intention of the agent beating Battleships, this is perfect because the agent will attempt to either hit a ship or sink a ship every time that it is told to choose an action. Eventually, after the q-values have been updated over a certain number of episodes, the agent will be able to determine a set of actions that will maximize the reward for the whole game and lead itself all the way to the goal state.

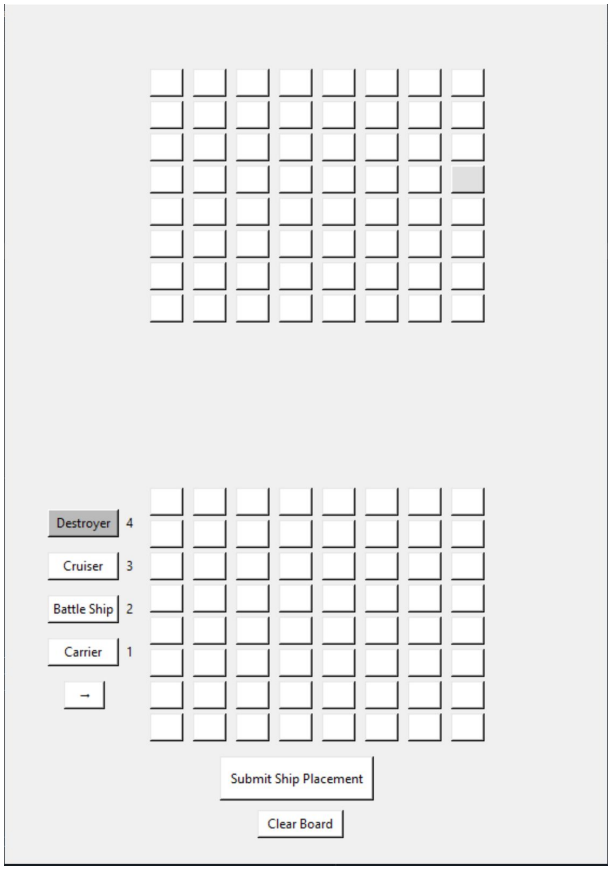
GUI V.1

The first version of the GUI was a simple representation of the game which did not allow for traditional "play", the player could simply place their ships and the CPU could perform the algorithm on the players distribution of ships with little to no player interaction besides the initial placement. This acted as a building block for the later, more powerful GUI.



GUI V.2

The second version of the GUI was more complex, allowing the representation of 2 boards (player and CPU). This is a more accurate representation of the game than GUI V.1, allowing for real-time updates to the board and for different ship types to be chosen and placed accordingly, this version added collision detection for the ships.



GUI V.3

The third version of the GUI was complete with a fitting background image as well as difficulties buttons which change the epsilon value for the agent's algorithm. The harder the difficulty, the smaller the epsilon value will be since the epsilon value is used as the percentage that the agent will explore instead of just constantly attacking the ships. We also added labels for the number of agent's ships that are left over. This was added to help give the player a better sense of what ships can still possibly appear on the board. It also replicates the idea of a player calling 'you sunk my cruiser' when their ship is sunk.

Results

Expected results as stated in our proposal:

"If our implementation is correct, our program will produce an optimal policy for the agent that will sink the opponent's ships. For every given game state, the policy will produce the best action for the agent to take. If the action the policy returns produces a tile that does not result in a "Hit", the agent's reward will be reduced. When that tile does produce a "Hit", the agent's rewards will be increased. This policy will be generated until the terminal game-state is reached (when all of the opponent's battleships are sunk). We would also like to be able to demonstrate that the policy our agent follows is optimal."

Actual results after implementation:

Upon implementation, we realized that the agent was able to converge to an optimal policy rather quickly and became virtually impossible for the player to beat. With a learning rate of 0.9 and an exploring rate of 0.001, the agent was able to sink all of the player's battleships in exactly 30 moves. This means that the player would need to have a perfect game in order to beat the agent. Therefore, we decided to implement four difficulties that the player can choose from when they start the game: Easy, Medium, Hard, Expert. The difference in each of these difficulties is rooted in the value of epsilon that the Q-learning algorithm uses. The chosen values are listed below:

```
Epislon Values for Each Difficulty:
    Easy: 0.8
    Medium: 0.5
    Hard: 0.3
    Expert: 0.05
```

Throughout the assignments that we have completed in this course, especially Assignment 3, we learned that the higher the epsilon value is, the higher the rate of exploration. This is perfect for our idea of different difficulties. A higher rate of exploration means that the agent will not always choose the best action and therefore will take a longer time to reach the goal state. Taking a longer time to reach the goal state means the player has more of a chance to win the game! With that being said, after some analysis of our software, we determined the average number of actions that the agent takes to reach the goals state in each difficulty.

```
Average Number of Actions Taken By Agent in Each Difficulty:
    Easy ~= 60
    Medium ~= 50
    Hard ~= 40
    Expert ~= 32
```

After implementing these epsilon values, we observed that the algorithm will not necessarily converge to an optimal policy -- a higher rate of exploration means that the resulting list of actions varies quite often. Therefore, we decided not to implement the idea of convergence in the sense of checking for the amount that old Q-values differ from the updated Q-values. Instead, we let the algorithm run for 100 episodes and use the list of actions that is found on the 100th episode. Through our testing process, we discovered that this list of actions will always give the agent the potential to reach the goal state (all player ships sunk); whether or not it reaches that state is dependent on how well the player performs against it.

Conclusion

A reinforcement learning based approach to the classic game “Battleship”, which has been played for over 100 years now, is an effective strategy in simulating the moves of a human player with varying degrees of difficulty. Our implementation used a standard Q-learning algorithm using an epsilon-greedy to tackle the RL problem and successfully give the agent the potential to reach the goal state every time. We learned through our study of the nature of the game and the algorithm used to play it that sometimes the AI can be ‘too’ good at the game, beating the player in ~100% of the games played. This fact pushed us to alter the epsilon values used in the Q-learning algorithm to allow for more competitive play. Competitive play is both more fun for the player and gives the player a greater sense that they are playing against a real human, not an omnipotent AI.

References

1. https://onq.queensu.ca/d2l/le/content/460719/Home
2. https://www.hasbro.com/common/instruct/Battleship.PDF
3. https://www.cs.nmsu.edu/~bdu/TA/487/brules.htm#:~:text=The%20object%20of%20Battleship%20is,the%20squares%20on%20the%20board.
4. https://paulvanderlaken.com/2019/01/21/beating-battleships-with-algorithms-and-ai/