Battleship AI: documentation

Vardhan Gupta

may-june 2017

Contents

1 Introduction

This project aims to make an AI which can play the Battleship game intelligently. the code for this is written in python and we aim to use the pygame library of python to implement the graphical part of the game. We also try to look into various algorithms to play the game and finish it efficiently.

2 the battleship game

Battleship is a board game played between two players. each player is supposed to place ships in two different oceans i.e 10x10 grids. the ships are of different sizes namely, aircraft carrier (5x1), battleship (4x1), destroyer (3x1), cruiser (3x1) and a patrol boat (2x1). they may be placed anywhere on the grids either horizontally or vertically. they may never overlap with each other but can be placed adjacent to each other.The players take turns alternatively. On each turn a player is supposed to fire (guess a point on opponent's grid) at the opponents ships which may result in either a 'miss' or a 'hit'. when all the points of a ship have been hit, it results in sinking of that ship. the first player to sink all of opponent's ships is declared the winner.

3 The project

3.1 algorithm 1

the algorithm used here is called hunt/target with parity. in this algorithm, it first randomly shoots in a checker board like pattern till a hit is found. once a hit has been found it tries to sink that ship on which the hit was found. if there are no hits left then and everthing is either in 'sink' state or 'miss' state or 'unguessed' state then it goes back to the random shooting.

Once a hit is found, the direction with most unguessed points is chosen to shoot at.

the functions made so far are:

- placeship, places ships on the board.
- checksink, checks the sinking status of any ship after a hit.
- shootingdirection, shoots in a given direction in order to sink a ship.
- targetmode, finds the best possible direction to shoot in and recalls itself if a hit is left on the board after sink a ship

such an algorithm gave an average of around 56 moves to beat a random board when tested for 10000 games.

3.2 algorithm 2

the next algorithm used calculates the probability of every unguessed point in the grid having a ship. the point with the best probability is targeted. if multiple points have the maximum probability then a *neighbour score* is calculated for each such point and the point with maximum neighbour score is shot at. if still multiple points qualify, then the point on the checker board like pattern is shot at. if still multiple points qualify, then any one of these points is chosen at random.

Also, after a hit is found, instead of shooting in the direction with most unguessed points, we add the no. of unguessed points of 'up' with 'down' and 'left with 'right' and compare them. the directions which add upto the bigger one of them is chosen and the best direction is now the direction with the most unguessed points of these two.

the functions made/changed for this algorithm are:

- calcprob, calculates the probability of having a ship of a particular size over the entire board.
- targetmode, changes were made as mentioned above for obtaining better shooting direction after a hit.
- nscore, calculates the neighbour score of a given point.

this approach was a significant improvement over the previous one as it resulted in an average of about 44 moves tested for 10000 games. This was the result with a random placement of opponent's ships on each game, subject to the constraints of the game and an intelligent placement might result in a worse average.

3.2.1 use of statistics of opponent's ship placement

A 2D matrix (oppstatscore) was created which gave scores to corresponding points where a ship was found on the opponent's boards. Also, scores were deducted if a miss occured at any point. points where no shots were fired in the duration of the game were given a "benefit of doubt" and partial scores were given to them. the scores were given as follows:

- +2 on a hit.
- -1 on a miss.
- +1 on unknown status of that point.

This was done to counter strategies which directly work against my own offensive strategy. The coefficient of variance of this matrix was used to see if a change in strategy was required or not. if so, the points with highest scores were hit-at first.

3.2.2 intelligent ship placement

So far, I had only been placing the ships randomly. A better way to place the ships would be to see the points where the opponent shoots early in the game and not place any of the ships at those points. This was done by using a 10x10 matrix with each point on the grid having an initial score of 0 and as the opponent shoots at the player board the corresponding point on this matrix get an incrementation in score. i.e. 100 for the 1^{st} point, 99 for the 2^{nd}, etc. based on this score, the ships will now be placed on the best possible blocks with lowest score. On competeting with the same algorithm with random placement of ships, It won about 9 out of every 10 games.

3.3 the gamemaster file

I have also made a file *gamemaster.py* which imports the code of the original file, *battleship.py* and uses it to actually play the game. what this file does is that it will ask a player the coordinates of the point where they want to play their next move. how they choose those particular coordinates is subject to the internal workings of their algorithm. then gamemaster "shoots" at the said coordinates of the opponent's board. the opponent now gives a feedback based on whether the player has missed or hit or sunk a ship. based on this feedback the player will now update their radar. the same process is then repeated with the roles reversed and this continues until someone wins. This file is also where the pygame module is used, to show the games played graphically.