American University of Armenia

College of Science and Engineering

CS 246 Artificial Intelligence

Battleship Game

Group members:

Monika Ghavalyan

Adrine Avanesyan

Melik Tigranyan

Fall 2023

Abstract

This project aims to implement and analyze various AI algorithms, excluding machine learning approaches, for playing Battleship. The primary focus is on evaluating these algorithms' effectiveness in predicting opponent moves, optimizing ship placement, and adapting strategies based on game progression. The goal is to determine which algorithm achieves the highest efficiency and accuracy in gameplay within a practical time frame, thereby deepening our understanding of AI application in strategic gaming scenarios and its potential for advancing decision-making processes in uncertain environments.

Keywords: Battleship, Greedy Best First Search, BFS, DFS, Monte Carlo, Algorithm.

Contents

Chapter 1
Introduction

The naval strategy game Battleship, which has been around since the early 1900s, is not just for fun but also for thinking strategically. The rules of the game are very simple but quite challenging. The ships of different sizes are set across the grid, and two players compete to sink one another's ships before the other does, without knowing where the ships are located. Battleship is not like chess, which requires deep planning, but rather is based on probabilities and a little luck.

When we explore the realm of artificial intelligence, The goal is both simple and complex: to deduce and accurately pinpoint the coordinates of the hidden ships based on strategic guesses. Each move is a step closer to revealing the true layout of the enemy fleet. This adaptation of Battleship emphasizes elements of deduction, pattern recognition, and strategic planning, making it an ideal scenario for exploring and testing artificial intelligence algorithms. The game transforms into a test of wits against a static but cunningly designed adversary, providing a unique and challenging experience for AI enthusiasts and strategists alike.

1.1 The Structure of the Project Paper

This project consists of the following parts:

- Literature Review
- Method
- The Description of the Implemented Algorithms
- Results of the Experiments
- Further Work

The literature review section focuses on describing the game's rules and examining relevant existing studies, drawing connections between the concepts. This analysis will lay the foundation for our coding implementation. The method section will concentrate on detailing the techniques employed.

Chapter 2

Literature Review

In "Cracking the Battleship Code," Jane Smith (2020) breaks down how game theory can help us understand and win at Battleship. Picture Battleship as a back-and-forth game where you and your opponent take turns making moves that impact your future choices. Unlike some surprise-filled games, Battleship gives you all the info you need about what's happening on the board – it's like playing with all the cards face up. This transparency makes Battleship what experts call a "fully observable" game.

2.1 Problem Setting and Description

In this variation of Battleship, the game is transformed into a single-player challenge against a pre-designed map of ships on an 8x8 grid. The agent's mission is to uncover and sink a fleet of ships strategically placed by the opponent. This fleet consists of ships varying in size from 2 to 3 cells, arranged either horizontally or vertically across the grid, ships are allowed to be placed adjacently. The game progresses in turns, with the player calling out coordinates on the grid in an attempt to locate and hit the hidden ships. Each guess can result in either a hit or a miss, depending on whether it corresponds to the position of a ship. A ship is considered sunk when all of its cells have been hit. In this single-player version of Battleship, the emphasis is on deduction and strategic thinking. The player must use the pattern of their previous hits and misses to deduce the arrangement of the opponent's fleet. With each turn, the player gathers more information, which helps update their strategy and improve the accuracy of their guesses.

The game reaches its peak when the player has successfully sunk all the opponent's ships. This version of Battleship requires careful planning, analytical skills, and a bit of luck. Success is measured by how efficiently and quickly the player can sink all the ships on the 8x8 grid. Victory is achieved when the player has detected the entire layout of the opponent's fleet and has eliminated all the ships.

2.2 Heuristic Based Search Methods

In this adapted version of Battleship designed as a solo challenge against a pre-set 8x8 grid, strategic decision-making still relies on heuristic functions. This approach is appropriate due to the complexity caused from the predetermined ship placements by the opponent and the depth of strategy needed in this compact grid. In this single-player context, the strategies often involve methods such as probabilistic guessing and pattern recognition. The focus here is on making informed guesses based on the observed state of the grid after each turn. These methods help us decide where to target shots more effectively and efficiently, increasing chances of finding and sinking the hidden ships. Instead of random guessing, heuristic-based search methods take advantage of patterns and logical deductions to determine the most promising locations to target. These methods involve analyzing the available information, such as the results of previous shots and the potential ship placements, this analysis guides algorithms for making the next move.

2.2.1 Greedy Local Heuristic Search

This strategy involves using a series of heuristic functions to guide the player's moves against a static opponent's grid setup. For this search method we chose the heuristic based on probabilities. The algorithm uses a probability density heuristic to guide the selection of the next shot. Cells with higher probabilities are considered more promising, and the algorithm focuses on exploring these cells. At the beginning of the game, all cells on the opponent's grid are assigned equal probabilities, indicating the uncertainty about the ship placements. The algorithm selects the next shot based on the probabilities assigned to each unexplored cell. The cell with the highest probability is chosen as the next shot. After each shot, the probabilities are updated based on the result (hit or miss) of the shot. The probabilities are adjusted to reflect the likelihood of the presence of a ship part in each cell. Greedy local search can get stuck because of local optima and might not explore the entire solution space, this can cause a problem when a more global exploration will be needed. The algorithm is deterministic and might not perform well in situations with high uncertainty or randomness, which for our case is not beneficial. In each iteration, the dominant operation is the selection of the next shot O(N). Therefore, the overall time complexity for one iteration is O(N). The number of iterations needed to complete the game depends on factors such as the arrangement of ships and the efficiency of the algorithm. The overall space complexity is dominated by the grid representation and the heuristic values. Therefore, the overall space complexity is O(G+N). Where G is the number of cells on the grid, and N is the number of unexplored cells.

2.3 BFS and DFS Searches

2.3.1 BFS

In BFS, our agent starts the search from the cell 'a8', placing it in the frontier. The nature of BFS, being First-In-First-Out, guides the expansion of child nodes. Initially, 'a7' and 'b8' are expanded from 'a8' and added to the frontier alphabetically. As the search progresses, 'a7' is the next to be expanded since it was first in the frontier. This expansion adds 'a6' and 'b7' to the frontier, making the new order (b8, a6, b7). The process continues with 'b8' being next for expansion, as it was the earliest in the frontier. This pattern of expansion and exploration proceeds systematically, ensuring that the search evenly covers the grid, gradually locating the ships, which means BFS is complete for the Battleship game. BFS will not be so efficient if the ships are located in deeper layers of the grid. Time and space complexities are the same O(N).

2.3.2 DFS

On the other hand, DFS relies on a Last-In-First-Out mechanism. This method dives deeper into the grid, exploring each branch fully before moving to another. The search also begins at 'a8', but the expansion strategy diverges significantly from BFS. After the initial expansion of 'a8', adding 'a7' and 'b8' to the frontier, DFS selects 'b8' for the next expansion since it was the last added. This approach means that DFS explores one row at a time, fully expanding each row before moving to the next. Although this method guarantees finding all parts of the ships, it may also not be much different from randomly hitting spaces across the board in case the ships are spread across the grid. For instance, if the last piece of a ship is under 'h1', DFS will explore almost the entire grid before locating it. For DFS as well time and space complexities would be the same O(N).

2.4 And Or Trees

In the Battleship game played on an 8x8 grid, employing and/or trees is a strategic approach corresponding to the game's stochastic nature, where the agent is unaware of the ship placements. This method, similar to Depth-First Search (DFS) in its need for large space complexity, offers a guaranteed solution by considering all possible scenarios in the game.The process begins at cell 'a8', the upper-left corner of the grid. From this starting point, the only possible actions are moving right or down into the grid. As the agent explores each new cell, the and/or tree branches to accommodate two potential outcomes: the cell either contains a part of a ship or it does not. This division into 2 branches occurs at every step, leading to a comprehensive exploration of all possible states of the game. The ultimate goal of this method is to locate all the ships on the grid, specifically those of two and three lengths. The and/or tree systematically expands, covering every possible arrangement and position of these ships. Each move into a new cell split into two branches – one assuming the presence of a ship part in that cell, and the other assuming its absence. The and/or tree's approach is exhaustive, ensuring that no possibility is left unexplored. While this method ensures the identification of all ships, it does so at the cost of high space complexity, often exceeding that of traditional DFS. Each decision point in the tree represents a different hypothetical scenario of ship locations, and the tree grows exponentially with each move.

2.5 Backtracking Algorithm Tree Search Version

We have a 8x8 grid with two 3-holes ships and one 2-hole ship with predefined placements. The Backtracking search starts from the top-left corner of the grid and proceeds to search cell by cell. At each step we categorize the choice of the action as “hit” or “miss”, this means we can choose between 2 options for the current state: either hit or miss the cell of the grid. We move through the grid cell by cell, row by row. With each new cell, we update assumptions based on previous hits, misses, and the constraints of the ship sizes.

For each 'hit', we are allowed to continue the search considering possible orientations of the ship (horizontal or vertical). For each hit, we explore both horizontal and vertical placements of the ship.

From the hit cell, recursively we try placing the ship in both directions (if space allows) and see if it leads to a valid configuration. If we assume a ‘miss’, we simply move to the next cell in the grid. If a placement leads to a contradiction (e.g., a ship extending beyond the grid or overlapping another ship), backtrack and try a different placement. The process continues until all ships are logically deduced and fit within the constraints of the game. The process might require several iterations and backtracks, especially as the grid gets filled and the placement options become more limited.

A contradiction occurs if the assumed ship placement:

extends beyond the grid, interferes with a previous hit or miss assumption, leaves no room for other ships that still need to be placed. Backtracking occurs when we encounter a contradiction. We then can; undo the last assumption (turn a hit into a miss, or vice versa), revert to the previous state of the grid, and try a different assumption from that point.

2.6 Monte Carlo Algorithm

Named after the Monte Carlo Casino in Monaco, known for its games of chance, this method relies on randomness to simulate and solve problems. This relies on repeated random sampling – it gets general random numbers and looks for probability in order to provide results. It generates a large number of simulations where the ships according to the game’s rules will be placed on the grid. Generating random samples means simulating various sequences of shots. For each simulation, the positions of the ships must be recorded and in this way the algorithm collects data which will be analyzed statistically later. The algorithm assigns probabilities to different cells or actions based on the likelihood of a favorable outcome. It then uses these probabilities to make informed decisions. So, the heuristic is based on the probabilities. For a hit, future samples should account for the partial ship's position. For a miss, samples should avoid placing ships in that cell. If we hit a part of a ship, we stay focused on completing that ship before returning to random sampling. After each shot (hit or miss), the algorithm adjusts its sampling strategy. That is why Monte Carlo develops a dynamic and adaptable strategy, it updates probabilities, additionally, if certain regions of a search space are more likely to contain ship cells, it can allocate more computational resources to explore those areas. Time Complexity: O(M*N) for a single shot, where M is the number of simulations run per shot, and N is the number of cells in the grid. Space Complexity: O(N). Additionally, the Monte Carlo algorithm is complete, it will achieve its goal whenever it exists.

Chapter 3
Method

3.1 Formulation of The Problem as a Search Problem

- Initial State: The initial state is an empty 8x8 grid with no ships placed.
- Actions: The agent can take action to shoot at a specific cell on the opponent's grid. Shooting at a cell can result in a hit (if there's a ship part at that location) or a miss (if the cell is empty).
- Transition Model: The transition model describes the effect of each action on the current state. If the agent shoots at a cell, the state is updated based on whether it's a hit or a miss.
- Goal State: The goal state is reached when all the ship parts are successfully located. In this case, the opponent's grid is fully revealed.
- Cost Function: Is designed to be the shot count which the player will try to minimize. It is the total number of shots fired.
- State Space: We have overall 64 holes (cells) on the grid and 2 possible outcomes for each of them, so the total state space is 2^64.
- Action Cost: Each action costs 1.

3.2 Solution to The Problem

We have conducted a study to determine the most effective algorithm for the Battleship game on an 8x8 grid with two 3-hole ships and one 2-hole ship, we tested several algorithms, including

Greedy Local Search, Breadth-First Search (BFS), Depth-First Search (DFS) and Monte Carlo. Our comparisons and algorithm evaluations mainly were based on time and space complexities, also we took into accounts other important factors, as well as success rates of each algorithm.

We already have described the nature of BFS and DFS. Since we know that BFS will explore the grid level by level and DFS will explore the grid row by row, they both will require a large number of moves to achieve a goal. It can be concluded that the success rates will be approximately the same for then, and whenever they are being compared with other algorithms such as Greedy Local Search, Backtracking Tree Search or Monte Carlo we conclude to not choose them as an efficient algorithm for our battleship game.

For the Greedy Local Search algorithm, we noticed that the success rate depends on the quality of the heuristic and strategies it uses. In terms of computational resources and time it can be efficient. However, due to its local decision-making nature it may struggle and will waste time and number of moves for achieving a goal, so we can not expect a desirable success rate for our game. Despite its potential in certain scenarios, the Greedy Local Search algorithm did not provide the level of consistency and accuracy required for the complex and unpredictable nature of Battleship.

Similarly, we analyzed the Backtracking Tree Search algorithm, which systematically explores each level of the grid before moving to the next. They can backtrack when necessary. Backtracking algorithms can be computationally intensive, especially if the search space is large. As we studied this algorithm in general can achieve high success rates, however, randomness and uncertainty which are the main components of our game can be harmful for Backtracking making it very non effective.

At the end, we analyzed the Monte Carlo algorithm. We have described its nature earlier and due to its adaptable and dynamic nature and strategies we observed that this algorithm can provide very high success rates. The reason behind this high success rate is the probabilistic sampling (explained earlier in the paper) and the skill to adapt quickly in a new, modified environment.

After evaluating these methods, we found that the Monte Carlo algorithm was the most appropriate for our Battleship game. We chose this algorithm due to its adaptability and precision. It could, due to its unique way of decision-making, explore the grid and achieve a goal with an optimized number of moves. This was the only algorithm within the algorithms that we have described, analyzed that had the most important characteristics for achieving an effective solution, for exploring the random and uncertain nature of our Battleship game.

Chapter 4
The Results and Conclusion

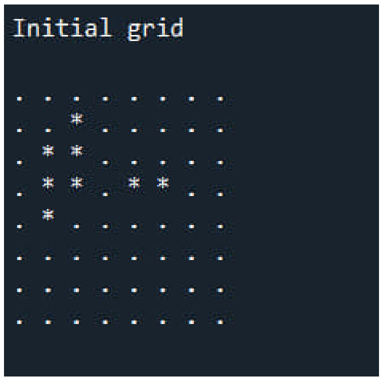
4.1 Results

Monte Carlo algorithms are a powerful tool for tackling complex problems like the Battleship game, where hidden ship placements and uncertainty play a significant role. In this guide, we will walk you through the implementation of a Monte Carlo algorithm for Battleship, starting with the initial grid containing ship placements and progressively improving accuracy through simulations. We have run simulations with variations of the sample numbers and the outputs will be represented below.

We begin with the initial grid, which represents the Battleship game board. This grid contains the positions of the two 3-cell ships and one 2-cell ship, placed randomly or according to specific rules. Here is the initial grid.

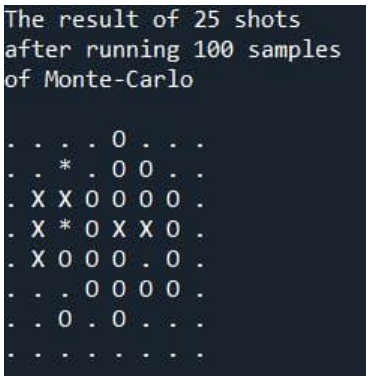
To estimate the optimal target for the next shot, there must be simulated multiple game scenarios using the Monte Carlo method. We started by setting the number of simulations to run. In our example, we simulated 100, 500, 1,000, 10,000, and 30,000 games.
In each simulation, the code randomly selects a target cell on the grid for the shot. Repeats the random shot and evaluation process until it completed a total of 25 shots in that simulation.

Keeps track of the number of hits and the states of the grid. Repeats the steps for the predetermined number of simulations (100, 500, 1,000, 10,000, and 30,000). After each simulation batch, it records the outcomes and grid states.



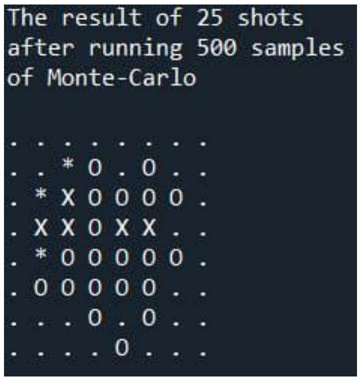
```
Initial grid

. . . . . . . .
. . * . . . . .
. * * . . . . .
. * * . * * . .
. * . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```



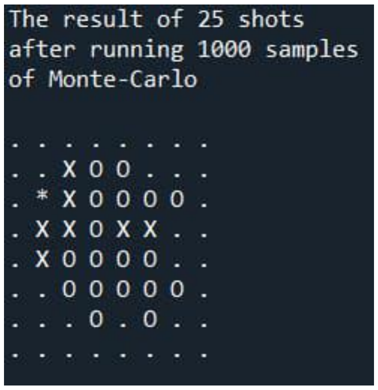
```
The result of 25 shots
after running 100 samples
of Monte-Carlo

. . . . O . . .
. . * . O O . .
. X X O O O O .
. X * O X X O .
. X O O O . O .
. . . O O O O .
. . O . O . . .
. . . . . . . .
```



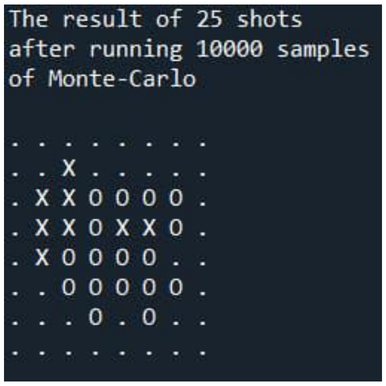
```
The result of 25 shots
after running 500 samples
of Monte-Carlo

. . . . . . . .
. . * O . O . .
. * X O O O O .
. X X O X X . .
. * O O O O O .
. O O O O O . .
. . . O . O . .
. . . . O . . .
```



```
The result of 25 shots
after running 1000 samples
of Monte-Carlo

. . . . . . . .
. . X O O . . .
. * X O O O O .
. X X O X X . .
. X O O O O . .
. . O O O O O .
. . . O . O . .
. . . . . . . .
```



```
The result of 25 shots
after running 10000 samples
of Monte-Carlo

. . . . . . . .
. . X . . . . .
. X X O O O O .
. X X O X X O .
. X O O O O . .
. . O O O O O .
. . . O . O . .
. . . . . . . .
```

Before implementing our own codes, we searched for the existing examples on the Internet and in one of the GitHub links we found codes that test random choosing and also Monte Carlo algorithm for battleship game with grid 10x10 and the results showed that 10 tests of random choosing run in 0.363s and 3 tests of Monte Carlo run in 0.006s. So basically Monte Carlo is 0.363 / (0.006 *3.(3)) = 18 times faster than random sampling. (The link to GitHub codes is provided in the references).

4.2 Conclusion

After evaluating various methods, we determined that the Monte Carlo algorithm was the most suitable for our Battleship game. We selected this algorithm due to its stochastic nature and ability to handle uncertainty effectively. It relies on random sampling to predict the most probable positions of ships on the grid, an essential aspect in a game where the opponent's layout is unknown. The Monte Carlo algorithm excels in balancing between random exploration and statistical inference. By generating a large number of random configurations and aggregating the outcomes, it identifies the most promising areas to target. This probabilistic approach is particularly advantageous in Battleship's environment of concealed information and dynamic play, making it an optimal choice for our game's unique requirements.

References

- Schwartz, A. (2022, April 20). *Coding an intelligent battleship agent*. Medium. https://towardsdatascience.com/coding-an-intelligent-battleship-agent-bf0064a4b319
- Backtracking - university of illinois urbana-champaign. (n.d.). http://jeffe.cs.illinois.edu/teaching/algorithms/book/02-backtracking.pdf
- *Artificial Intelligence: A modern approach, 4th us ed.* Artificial Intelligence: A Modern Approach, 4th US ed. (n.d.). http://aima.cs.berkeley.edu/
- Implementation of random choosing and Monte Carlo algorithm from github. https://github.com/jacebrowning/battleship/tree/master