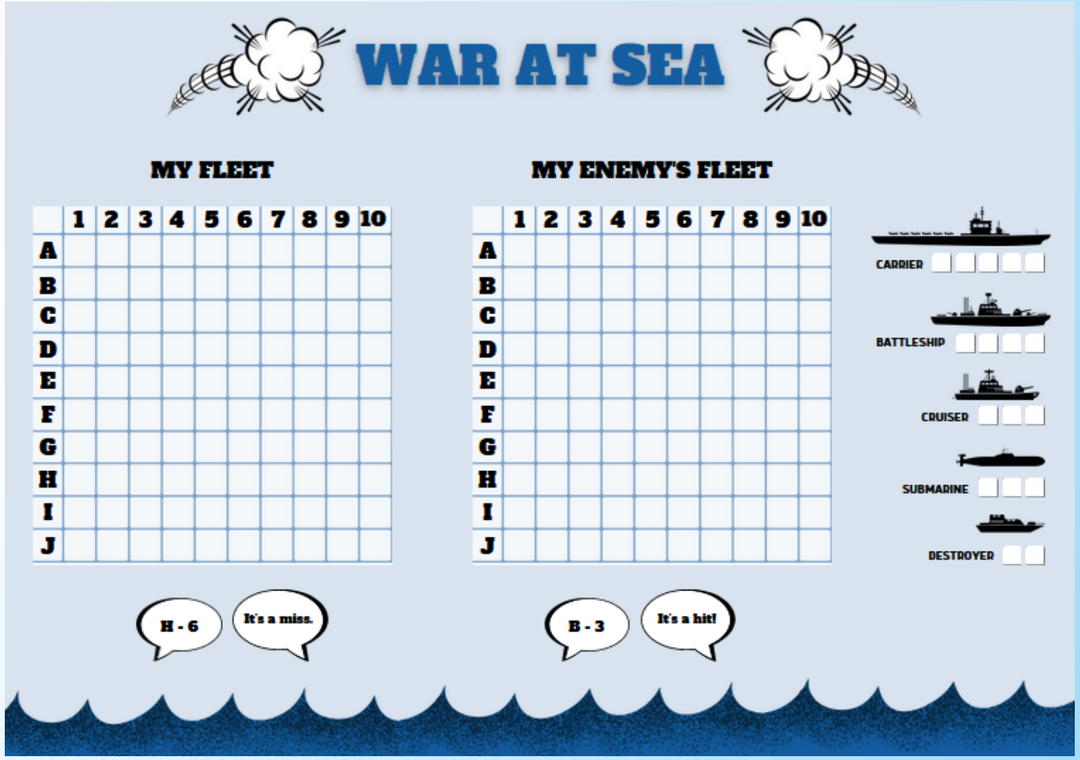
BATTLESHIP

Monika Ghavalyan
Adrine Avanesyan
Melik Tigranyan

CONTENT

- Introduction
- Problem Formulation
- Algorithms
- Comparison
- Code Results
- Conclusion



INTRODUCTION

- Battleship is a famous board game that has been around since the early 20th century. It started as a simple game called "Battleship" or "Sea Battle," where players used paper and pencils. Students and soldiers liked to play it a lot. This easy game was the beginning of what would later become a well-known naval war game.

- We have modified the original battleship game. In this 8x8 grid game, you’re up against an opponent who has already placed their fleet: two ships with three holes and one with two holes. Task is to find these ships by guessing grid coordinates. Each guess helps you figure out where the ships are hidden.

PROBLEM SETTINGS AND DESCRIPTION

In our simplified, single-player version of Battleship played on an 8x8 grid, the player competes against a pre-set arrangement of ships placed by an automated opponent. The objective is to locate and sink a variety of ships, ranging from 2 to 3 cells in size, through strategic guessing.

Each turn, the player calls out grid coordinates to locate the hidden ships. Hits and misses guide their strategy, helping to deduce the ships' placements. A ship is considered sunk when all its segments are hit.

This game version emphasizes strategic thinking and pattern recognition. As the game progresses, the player gathers information to refine their strategy and improve guess accuracy.

Victory is achieved by sinking all of the opponent's ships efficiently and swiftly, showcasing the player's planning, analytical skills, and a degree of luck. The ultimate goal is to decipher and eliminate the entire fleet of the automated opponent.

PROBLEM FORMULATION

- **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.

GREEDY LOCAL SEARCH

The Greedy Local Search strategy for the Battleship game is an approach that focuses on making the most promising move at each step based on current information.

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.

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. By continually updating these probabilities based on the outcomes of each shot, the algorithm becomes more efficient in targeting cells that are likely to contain ships.

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.
In the context of Battleship, a local optimum occurs when the algorithm focuses on a particular area of the grid based on initial hits or high probability estimations, while neglecting other areas where ships might be located.

BFS AND DFS

Unlike the Greedy Local Search, BFS doesn’t prioritize immediate gains but rather explores level by level, ensuring a more comprehensive search. At first we choose a starting cell. The nature of BFS, being First-In-First-Out, guides the expansion of child nodes.

While the queue is not empty, we will take the first cell and check it.
If it's a 'hit', we will consider surrounding cells (up, down, left, right) to be explored next. If it's a 'miss', we will proceed with the next cell in the queue.
BFS can be slower, as it might explore many empty cells before finding all parts of the ships.

DFS relies on a Last-In-First-Out mechanism. This method dives deeper into the grid, exploring each branch fully before moving to another. This can help us to discover the full extent of a ship once a part of it is hit. This approach means that DFS explores one row at a time, fully expanding each row before moving to the next.

DFS might miss shorter ships while deeply exploring after a hit, especially in sparse grids. Can potentially overlook certain areas of the grid, leading to a less thorough search compared to BFS.

MONTE CARLO

Monte Carlo algorithm relies on repeated random sampling – it gets general random numbers and looks for probability in order to provide results.

Initially, we have no information about the locations of the opponent's ships. Then we have to run a large number of simulations where the ships according to the game’s rules will be placed on the grid. For each simulation, we must record the positions of the ships.

From these samples, we select a cell to target that appears most frequently among the samples. After each shot (hit or miss), we adjust our sampling strategy. 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.
Requires computational effort to generate and evaluate new samples each turn.

This method dynamically adapts to the game's progress. It can quickly shift focus to areas of higher probability based on game developments. Focuses more on adaptability and dynamic decision-making based on the most current state of the game.

COMPARISON

Greedy Local Search

Time Complexity: O(N) for a single iteration, where N is the number of cells in the grid.

Space Complexity: The space complexity is O(1), as it does not require additional space proportional to the size of the input (the grid).

The Greedy Local Search is faster per iteration since it only considers immediate options. However, it will require more iterations to find all ships.

Greedy Local Search is not always complete in the context of Battleship. This is because it can get stuck in local optima.

BFS and DFS

Time complexities: BFS and DFS are the same O(N).

Space Complexities: are the same O(N).

BFS is complete for the Battleship game. It systematically explores all cells in the grid, ensuring that every possible ship location is checked.

DFS is generally complete in a finite space like the Battleship grid. It explores all possible paths by going as deep as possible in one direction before backtracking.

BFS/DFS: These methods are not adaptable. They follow a set path of exploration and do not dynamically change their strategy based on hits or misses. Might struggle with complex ship placements.

Monte Carlo

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).

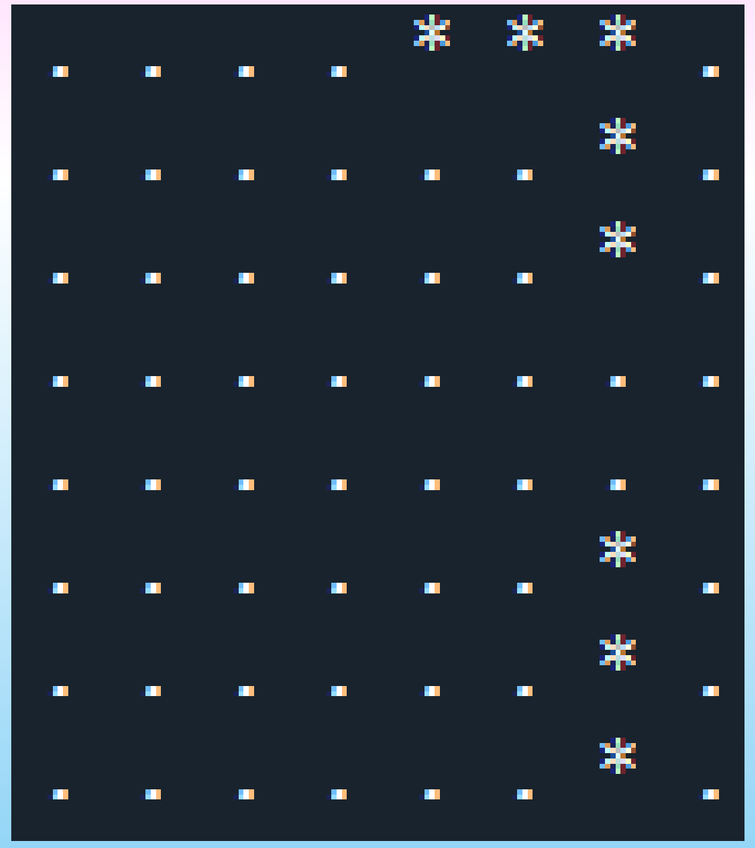
Highly adaptable. The Monte Carlo method updates its strategy based on the results of each shot, refining its approach as more information becomes available throughout the game.

The Monte Carlo method is considered complete in the context of Battleship, as it systematically covers the entire grid through its probabilistic sampling. This ensures that all possible ship locations are eventually considered.

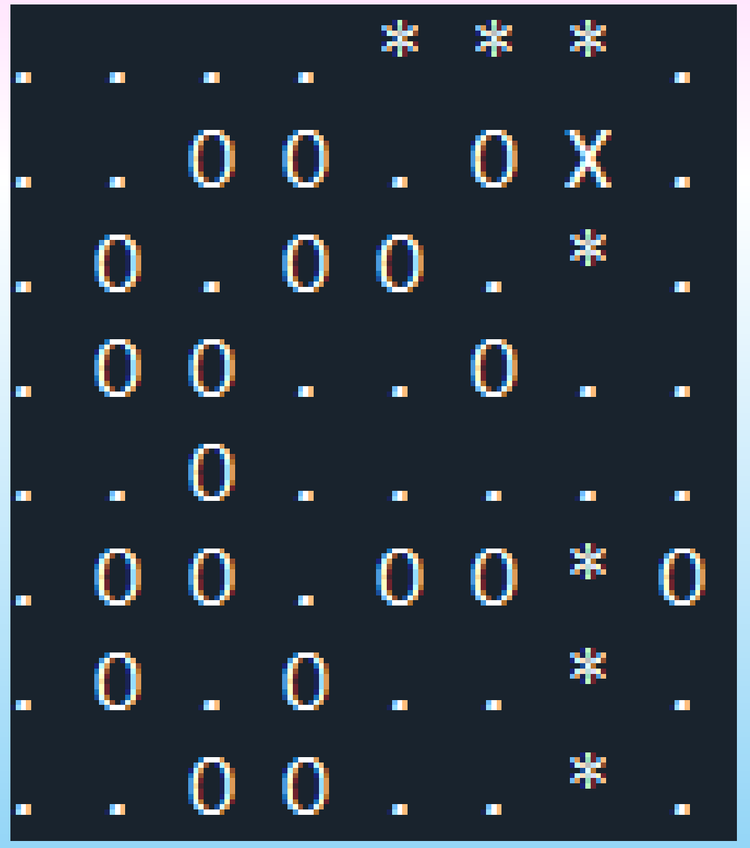
The accuracy of the method improves as the game progresses, becoming more precise with each shot based on the updated probabilities.

CODES RESULTS

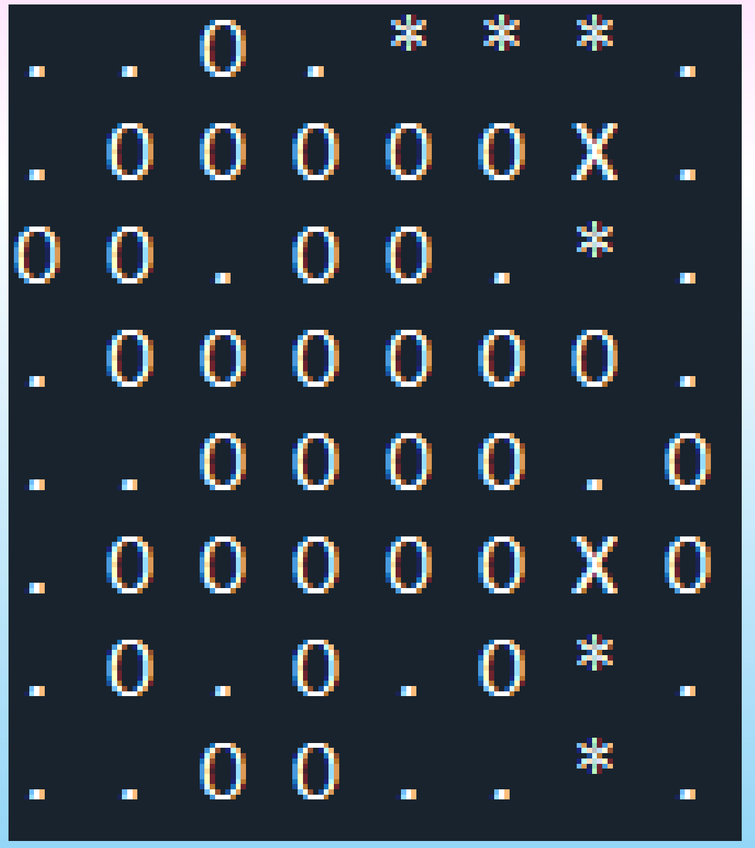
Initial grid



The result of 20 shots after running 100 samples of Monte-Carlo

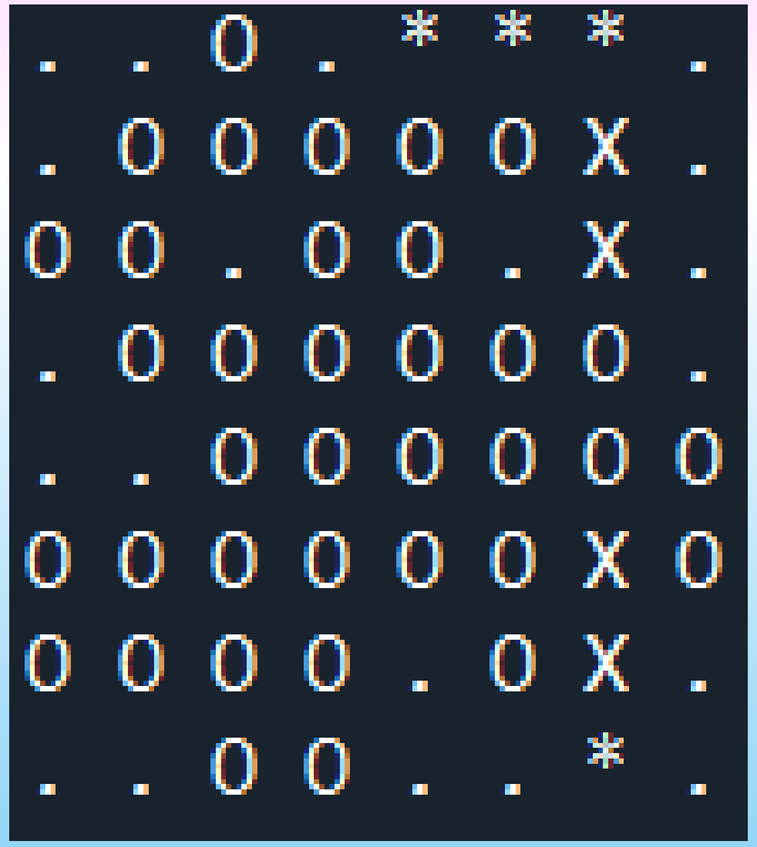


The result of 20 shots after running 500 samples of Monte-Carlo

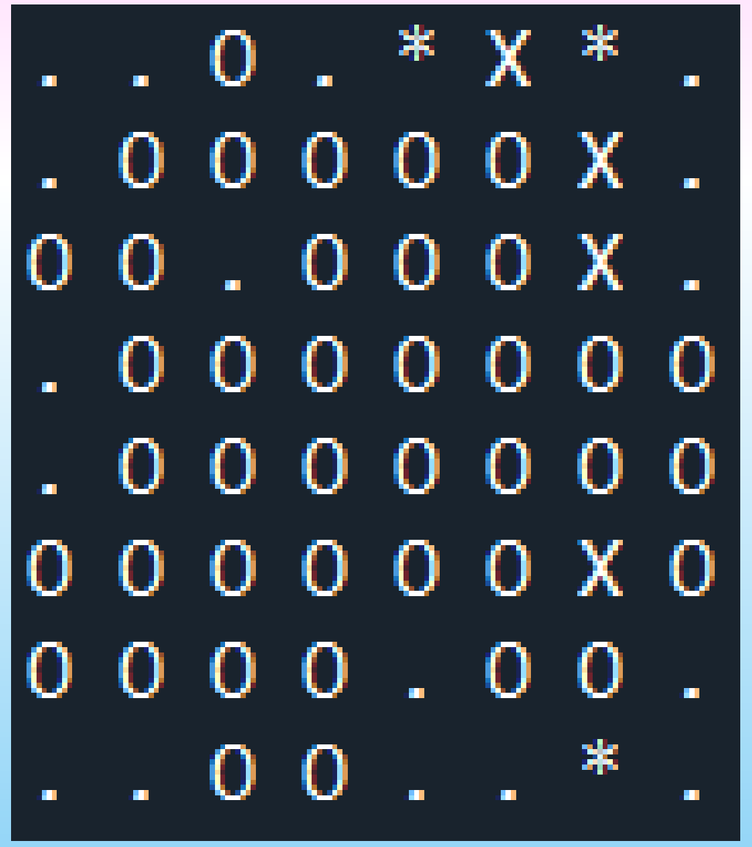


CODES RESULTS

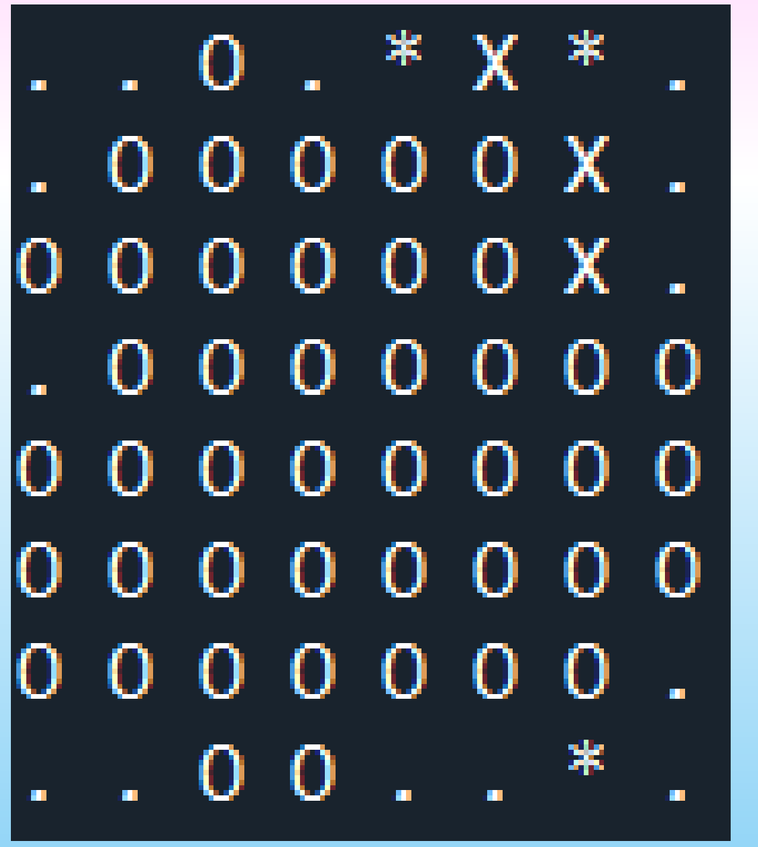
The result of 20 shots after running 1000 samples of Monte-Carlo



The result of 20 shots after running 10000 samples of Monte-Carlo



The result of 20 shots after running 20000 samples of Monte-Carlo



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/
- GitHub. (n.d.). https://github.com/jacebrowning/battleship/tree/master/battleship