Analyzing the Efficiency of Algorithms for Battleship

John Ferstle, Michael Nguyen
ferst011@umn.edu, nguy3579@umn.edu

Spring 2021

Abstract

As algorithmic approaches to game domains and real world domains become more popular and advanced, evaluations of the various tactics that these algorithms utilize must be conducted in order to determine the optimal implementation for a given domain. The well-known strategy game, Battleship, presents a challenging two outcome, zero-sum, imperfect information application of an algorithmic approach. The problem addressed in this experiment is the application of different algorithms (Information Set Monte-Carlo Search Tree (MCTS), Greedy Human-Like Player Algorithm variants, Parity Human-Like Player Algorithm variants, and Greedy Probabilistic Algorithm) to the Battleship domain. Each algorithm was tested using a large sample of simulated games to compare average runtime, memory usage, and number of moves taken to win. The outcomes of these trials are discussed, and insights are made into potential improvements to our experimentation.

1 Introduction

As Artificial Intelligence becomes evermore prevalent in many facets of modern living, AI approaches can offer efficient solutions for a wider and more complex range of problems. Classic games of various types, such as board games, card games, and strategy games, represent one example of a field brimming with effective applications of Artificial Intelligence methods. There are several examples in recent decades of algorithmic tactics being used to defeat world class human players [1], such as AlphaGo, a program that plays the board game Go by utilizing approaches such Monte Carlo Tree Search (MCTS) and neural network training. Traditional games not only provide an environment with set rules, they have also been the focus of great research efforts for strategy development. The advances and research toward AI solutions for games are often useful well beyond the scope of the game itself. Real world scenarios can often be modeled effectively by games, as both can be classified by common features such as observability, (perfect information/observability versus imperfect information or single-agent versus multi-agent environments.

AI approaches to these games often focus on taking actions according to their evaluation of the current state of the game (known as game state)[1]. Many such approaches have utilized domain knowledge, represented by a game heuristic, in order to improve the performance of these evaluations. These approaches have been shown to perform extremely well in their respective games, however, they are limited severely by the use of heuristic domain knowledge. The task of implementing this knowledge requires domain-specific information that cannot be applied to other games, meaning that building such models becomes research intensive, complex, and difficult to generalize to other problems[1]. To address these difficulties, approaches that do not rely on domain-specific knowledge have been developed. General Game Playing (GGP) applications, which attempt to create systems that can be efficiently applied to any game that is represented using a formal language (such as GDL), have gained much traction among Artificial intelligence researchers [2]. These approaches are ideal for researchers because they can be defined explicitly in terms of legal moves, states, and rewards. The ability to take generalized approaches to game problems allowed for better solutions, along with solutions that were effective across several different problems [3].

Another issue that limits the applicability of a solution to wide range of problems is observability. Although several approaches have succeeded in domains with perfect information (where the precise state is known by all players) [4], domains with partial/imperfect information are much closer representations of real world scenarios [5]. These domains involve hidden information and uncertainly, which adds greatly to the complexity of the domain. Despite this added complexity, solutions to games of imperfect information are being increasingly researched for the purpose of more accurately addressing real world problems such as military combat, which inherently lacks perfect information [6]. Through research of game domains and similar classes of problems, several generalized algorithms (and their modifications) have gained popularity [7].

The advancements in algorithms such as MCTS have contributed to several game solutions, along with many real world application problems. MCTS solutions have far surpassed previous efforts in many games, such as the use of min-max alpha-beta pruning algorithms (MTD) and heuristic approaches used for GO prior to MCTS [8]. The success of MCTS in these applications led to breakthroughs in several other problems, notably ones that are solvable using GGP approaches. The insights from these successful algorithmic implementations are useful well beyond board/card games due to their domain-independent structure. MCTS, for example, has been used for the Capacitated Vehicle Routing Problem (CVRP), which involves planning and scheduling for the purpose of efficient supply chain functionality. It has also seen use in the field of chemistry with neural networks to plan chemical synthesis (using the 3N-MCTS approach) [8]. General approaches have taken other forms, including other greedy algorithms and probabilistic models. These approaches also exhibit vast problem-solving versatility, earning praise for their extremely fast runtimes and lightweight approaches [9]. Both MCTS and Greedy Probabilistic approaches have been modified to improve performance in increasingly complex domains, such as Battleship. The investigation in this paper seeks to provide further insights that can be beneficial for other complex domains.

---

2 Literature Review

2.1 Definitions and Terminology

2.1.1 Battleship

Battleship is a two agent, two outcome, partially observable game that places ships of set sizes on a 10x10 grid board and requires players to sink all of the opponent's ships without knowledge of their locations. The "fog of war" component of the game prevents players from viewing the position of the opponent's ships on the board, creating an incomplete information domain. Each player's fleet consists of 5 ships: A Carrier of length 5, a Battleship of length 4, a Submarine of length 3, a Cruiser of length 3, and a Destroyer of length 2. At the start of the game, both players position each of their ships in a vertical or horizontal (no diagonal or overlapping placement allowed) orientation on the board. Players then take turns shooting at the opponent's board in order to find and sink ships. Each move consists of a shot at a single grid sector, and a corresponding feedback of a HIT or MISS[10].

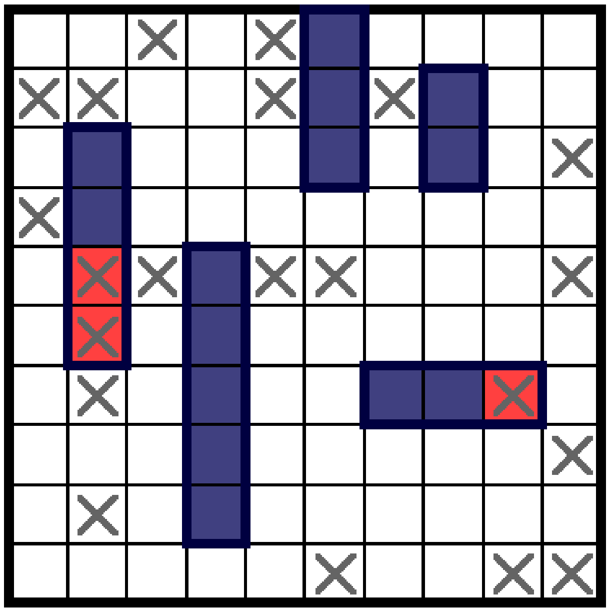

Figure 1: An example board for an in-progress game [11]

The above figure demonstrates a possible in-progress game board configuration between a hypothetical player P1 and player P2. White grid sectors represent Ocean spaces, blue grid sectors represent ships (specifically parts of ships that have not been hit by the opponent), the gray "X" spaces represent shots that resulted in MISS feedback, and red grid sectors a with gray "X" represent shots that resulted in HIT feedback. If the above board belongs to player P1, then both the blue and white grid sectors (all sectors that have not been shot) are indistinguishable to player P2 due to the fog of war component of the game. Player P2 only has knowledge of the positions marked with an "X" symbol (regardless of whether HIT or MISS feedback was received).

2.1.2 Perfect and Imperfect Information Games

Perfect information games have domains where all information about the current game state (what cards players have, what actions have been taken, the positions of pieces on the board, etc) is known to every player. There are several games that have this domain, including Chess, Go, and Checkers. These domains share the features of fully observable pieces, boards, and actions by players. Unfortunately, the perfect information game problem class is not well suited for real world situations (where hidden information and uncertainty are prominent factors for decision making) and their solutions are not often adaptable for games of imperfect information. These imperfect information game problems, such as Battleship, where hidden information prevents players from knowing what the outcome of their shots (actions) will be. Broadly, hidden information games contain information that is hidden to some or all players, such as a card game where only the owner of the hand knows what cards it contains, and no players know exactly what cards are in the rest of the deck or its order [12].

2.1.3 Zero-sum Games

Battleship is also an example of a two player, zero-sum game. In zero-sum games, there is no collaborative (common interest) aspect of gameplay, and players may not have identical information regarding the current state of the game [6]. When a two player game is classified as zero-sum, this means that each player attempts to maximize their own outcome for the purpose of winning the game. A consequence of this payoff structure is that if one player wins the game, the other must lose (no communal victory or loss). Minimax problems offer a simple tree representation of this approach for some games [12], as players attempt to maximize their outcome while minimizing the opponent's outcome. When playing Battleship, however, individual moves are not thought of as disadvantaging an opponent in the same way as losing a piece in Chess or Checkers does. Instead, the number of HITs or the number of ship sectors remaining vs the total number of ship sectors (17) is a stronger indication of which player is closer to winning the game. Although a MISS does not effect this ratio, it does provide improve the players knowledge of the game state by narrowing down the locations of the remaining ship sectors.

2.1.4 Game Notation and Representation

A game of Battleship, is represented for some of the approaches below (namely MCTS variants) as a directed graph. In this graph representation, the root node represents the initial state of a board with all ships placed (prior to the first shot by either player). All subsequent intermediate nodes represent the non-terminal states of the game, such as the non-terminal state shown in *Figure 1* above. The leaf nodes of the graph represent the terminal states of the game. A path from the root node (initial state) to a terminal state leaf node is thus a representation of a single game played to completion. Starting at the root node of the graph, each move increases the depth of the path by one, as the game state can change from the parent state to any legal child game state node (shots can not be repeated, which means that the branching factor of the graph decreases as less unexplored grid sectors become available for shooting) [12].

2.2 Related Work Analysis

The following sections consider the various previous experiments involving Battleship and various other partially observable board games such as Kriegspel [5] and Dou Di Zhu [4]. Each section outlines a set of approaches used in related works, along with their effectiveness toward the problem detailed above.

2.2.1 Naive and Human Player Approaches

The following approaches are detailed in [13], where they are discussed as benchmarking approaches for simplistic play. These approaches are useful as a baseline for comparing future algorithms, as well as an insight into common human approaches to the game of Battleship.

The most naive approach to the Battleship problem involves the use of random selection to pick sectors, with no heuristics, until the game is completed [13]. This method is discussed as a control algorithm that can be used to gauge the performance of the other algorithms in the experiment. This approach is not realistic of an average human player, who can instead be modeled below with simple heuristic models.

The first algorithm is representative of an average human player, whose behavior can be emulated by creating a "human-like player" algorithm [13]. The performance of this human-like player algorithm (HLPA) utilizs different modes in order to simulate the actions of an average human, using hit and miss responses as a simple, but effective heuristic for determining the next action. HLPA begins in and defaults to the Hunt mode, which selects grid sectors at random until a hit response is received. Upon hitting an opponent's ship, HLPA will enter Target mode, which targets all sectors surrounding the position of the hit and will select one at random until a second sector is hit. This prompts HLPA to enter the last remaining mode, Destroy. This mode will select sectors along the ship's orientation (horizontal or vertical) until it receives a sunk response. HLPA then returns to Hunt mode and repeats the above process for the remainder of the game. This simple heuristic provides a large improvement over pure random selection [13]. This algorithm can be improved upon using parity to lessen the subset of grid sectors available for random selection in Hunt mode (as seen below).

The Parity HLPA (PHLPA), is an adaptation of HLPA that exploits the strictly horizontal and vertical (non-diagonal) positions of ships [13] (*referred to as "Even Player"*). Using this strategy, PHLPA randomly selects only the even (arbitrarily selected parity) subset of unknown sectors in the grid. The subset then contains only sectors that are not directly next to each other (a "checkerboard" pattern). In *Figure 2*, parity would be used to shoot randomly at only the black (chosen aribitrarily) subset of grid sectors, rather than all sectors. This assures that the grid is searched more efficiently while still assuring all ships are discovered. The PHLPA is meant to represent a higher level human player, and can be utilized as a benchmark for determining how much other algorithmic methods can improve over high level human tactics[13].

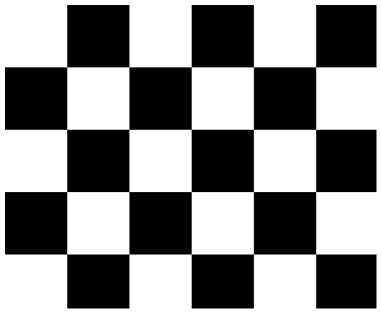

Figure 2: An example of parity (checkerboard)

The Optimized HLPLA (OHLPA) algorithm and Optimized PHLPA (OPHLPA) algorithm attempt to further improve these human-like player algorithms by implementing a simple adjustment to the target modes of the non-optimized HLPA variants. This adjustment is a human strategy of preference toward shots that are in the same direction (immediate left, right, up, down) of the successful hit. The intention of this added heuristic is to act optimally in regards to the orientation of the ship as horizontal or vertical. This optimization should lower the number of moves needed to win a game on average, as these algorithms will not continue to arbitrarily pick a search direction after making a series of hits on touching sectors.

2.2.2 Greedy Probabilistic Approaches

The greedy algorithm is an algorithm that is based on the principle of making the most optimal or "greedy" choice at each part. From that principle, we expect to obtain a solution that is approximately optimal in an amount of time that is practical. To implement the greedy algorithm, we need to incorporate a probability density matrix to allow us to make the greedy choice [14]. This probability density matrix contains probability values about where ships are likely stored on the grid. In order to create a successful probability density matrix for our greedy algorithm to work, we will be making assumptions based on ship sizes and positions on the board. For example, if we were to already have made a couple of shots on the board and those were all misses, we can lower the probability for some spaces based on how close those misses were next to each other. In the space between those misses, it could be impossible to fit a specific ship size. However, these probability values are calculated every turn and we would not have any reasonable probability values to make a greedy choice before the algorithm makes its first turn. Thus, we will include a "hunt" and "target" mode [14]. The hunt mode will specifically look for ships and the target mode will select parts of the board that contain a part of a ship that we have hunted based on our probability density matrix.

2.2.3 Traditional Monte Carlo Tree Search, UCT, and POMDP Variants

Monte Carlo Tree Search is a a heuristic tree search algorithm that is often used for board games. Monte Carlo Tree Search is a great method to employ when working with board games because of its ability to create a solution using very little knowledge [15]. It is effective for any finite length two player game, and it utilizes the explore-exploit paradigm in order to find optimal solutions [1]. To guide the algorithm, there may need to be some type of search-control knowledge that needs to be implemented for the algorithm to be effective [15]. Its goal is to approximate the game theoristic value of all possible next actions for the given current state by building a partial search tree iteratively [16]. There are several variations of MCTS that could be viable for the Battleship Problem, such as generalized, UCT, and POMDP [16], [13], [17]. The first set of these approaches are discussed below, along with the limitations that they have for applicability to the problem:

Monte-Carlo Tree Search: A New Framework for Game AI [1] discusses in general terms the benefits of using MCTS as an approach to two player games. MCTS is a viable algorithm for use in any terminating (finite length) games, and it utilizes the explore-exploit paradigm in order to find optimal solutions [1]. The article discusses the fundamental methods utilized by MCTS, and finds that these methods (selection, Expansion, Simulation, and Backpropagation)[1] are applicable to a range of multiplayer games with varying degrees of complexity. It also discusses the benefits of using heuristic knowledge to better weight actions in the simulation stage [1].

Nested Monte Carlo Search for Two-Player Games [2] examines the effectiveness of extending single-player NMCS to two-player games, as well as considerations of heuristics and pruning methods for these problems. It focuses on the use of NMCS to evaluate different successor states (Nested UCT) of the current state with minimax and applying methods such as Cut on Win (COW), Pruning on Depth (POD), and discounting heuristics in order to further improve their algorithm implementation [2]. The authors discuss the success of COW, despite concerns of unsafe pruning, and compared the results to standard MCTS (with random playouts). NMCS was found to perform better than the standard MCTS when applied with the discussed heuristics and pruning tactics over a sample of nine different two-player games.

Monte-Carlo Planning in Large POMDPs [17] offers the Partially Observable Monte-Carlo Planning (POMCP) algorithm that works well against more difficult POMDP problems such as Battleship. It uses the extension of UCT, Partially Observable UCT, in order to select actions based on the history search tree (formed from several iterations), and then prune accordingly and select again from the updated history[17]. This article addresses the problem directly, using Battleship in experimentation, and found that on average POMCP performed far better than human player strategies or random strategies [17].

2.2.4 Comparison Analysis and Applicability to Problem for Traditional, Standard UCT, and POMDP MCTS Variants

The three related works each offered a different level of analysis of the applicability of the MCTS algorithm and its variaties. The standard MTCS, although a contender for solving a variety of two-player problems, was a cause of concern for the Battleship problem due to the factors introduced in the problem description. Although the benefits of standard MCTS were discussed in [1], including its ability to overcome the hurdles of needing high quality domain knowledge and long runtimes, this alone did not provide a strong argument for its use in the problem. Guillaume Chaslot et al provided a generalized agrument for the usefulness of MCTS, but did not approach the discussion with the specificity or detail of the other two articles. Nested MCS with discounting heuristic and COW/POD pruning was found to be far faster than standard MCTS [2] as a result of exploring far less states. The use of unsafe pruning methods did not cause concern due to significant improvements in performance, although increased nesting (level 2 vs level 1) did not appear beneficial in all cases[2]. POMCP was found to be the strongest choice for a Battleship problem solution.

The article provided far stronger results (over 50 move improvement over random play, and over 25 move improvement from random preferred action selection, which most human players employ)[17]. Its use of PO-UCT provided low search times even on a full-game implementation, and far surpassed other full-width planning methods in efficiency[17]. The articles discussion of methods and evidence were well supported and the method fit the application setting the best of the three discussed methods.

2.2.5 Limitations of UCT MCTS Variants and Information Set Monte Carlo Tree Search Variants

Another approach to the Battleship problem uses Information Set Monte Carlo Tree Search (ISMCTS) and its variants. These approaches vary from traditional MCTS approaches in that they do not rely on perfect information [4] (much like the POMDP algorithm discussed in the first set of MCTS approaches). Before discussing the ISMCTS implementations, the shortcomings of the UCT variants must be addressed:

The limitation of the POMDP alrorithm, besides its complexity, is the use of the UCT variant PO-UCT. Other variants of the such as Determinization UCT, create several instances of the game as deterministic, perfect information game, so that the MCTS algorithm can be used applied to imperfect information (partially obeservable) games such as Battleship [4]. The effectiveness of the UCT variants for partially observable varies based on factors such as the amount of inference that can be made based on the use of "cheating" and refereeing [12], [4]. The Cheating UCT variant is given access to hidden information, and is able to therefore outperform Determinization UCT (expectedly), highlighting some of the issues that determiziation still faces when compared to UCT with the removal of hidden information. This includes (but is not limited to) strategy fusion, and non-locality [4], [7]. Additionally, the UCT variants suffer from duplicate state coverage and high CPU utilization [12].

ISMCTS uses information sets, which are a collection of states for a given player, in order to combat these issues [7]. It has been found to aforementioned UCT variant issues [12], as well as superior runtime in the cases such as SO-ISMCTS+POM [7]. This approach combines Single Observer ISMCTS with a Partially observable Marcov approach, providing superior performance in the scope of Battleship.

2.2.6 Other Approach Concepts

However, the Monte Carlo Tree Search may not always produce an optimal solution (due to it being an computationally dense anytime algorithm). In [18], a method using seeds is used for the Battleships problem. This method involved generating populations of a first and second player and eventually picking the player with the best performance. The two methods of Monte Carlo Tree Search and seeds method were tested against each other using a probability distribution with the initial position of the fleet. When comparing the results, it was found that the only the seed method could discover something significant with a 0.77 score against the baseline (winning rate.)

Other MCTS and non-MCTS variant General Game Playing approaches have been taken to several games with imperfect information. They generally rely on games being defined in Game Description Language - Imperfect Information (GDL-II), along with some of the advanced techniques mentioned above[3]. These algorithms often utilize approaches involving multi-agent game trees, acyclic graph representations, information sets, and planning (facilitated by Planning Domain Definition Language). These other GGP approaches are also not considered for the scope of this experiment.

---

3 Approach

For this experiment, we test the efficiency of Random Selection, HLPA, OHLPA, PHLPA, OPHLPA, Greedy Probability Search, and ISMCTS for the game problem Battleship. Note that two separate code implementations were used for this experiment (detailed below):

3.1 Python Approach

The majority of the code used for this section of the experiment was inspired by Eric H's *Battleship Probability Analysis* [19]. Slight modifications were made to this implementation for the purpose of data collection, but the code served as an effective representation for the majority of the experimentation without modification. The implementation begins with Board and Ship Generation, where ships are placed randomly in a 10x10 array board representation. Various checks are made to assure all placements are legal, and the simulation can then begin. Throughout the game, various algorithms check bounds and legality for shots to assure the game is emulated as realistically as possible. It should be noted that this implementation creates a single board for the algorithm to solve, rather than two algorithms or players competing against each other. It remains an effective representation of all the evaluated metrics, as none of the algorithms tested modify their behavior in the presence an opponent. This implementation provides several algorithm implementations that will be discussed in the following subsections.

3.1.1 Random Selection

The random selection approach is implemented by continuously generating random row and column coordinates using the random randint() function. As discussed in the *Related Works* section above, this approach is for demonstration/benchmarking purposes, as not even a human player would be likely to use this strategy. It is also used as a component for some of the algorithms discussed below.

3.1.2 Standard HLPA

This basic HLPA implementation begins in "Hunt" mode, utilizing Random Selection to search for ships until HIT feedback is received. It then switches to "Target" mode, considering the 4 directions from the hit sector and checking for legality before randomly selecting a direction for the next shot. The targeting behavior continues until the ship is sunk, returning the algorithm to "Hunt" mode to repeat the process.

3.1.3 Parity HLPA

The PHLPA implementation uses the same approach as standard HLPA, but with the added benefit of parity for the Random Selection component of the algorithm. This is implemented by generating a random row coordinate, followed by a corresponding even or odd column coordinate, based on the modulo 2 of the row coordinate. In all other aspects, this algorithms is implemented the same as the Standard HLPA implementation.

3.1.4 Optimal HLPA and Optimal PHLPA

These optimized variants add an improvement to the "Target" mode of HLPA/PHLPA, utilizing an additional human strategy to take advantage of the orientation of ships by taking follow-up shots in the same direction that yielded previous HIT feedback. This optimization aims to shave unnecessary moves off of the non-optimized approaches by limiting the randomness that occurs during the use of "Target" mode.

3.1.5 Greedy Probability Search

The Greedy Probability Search algorithm is a greedy algorithm that calculates the most probable location based on a superposition of all the possible locations for the opponent's ships. This implementation relies on probability density heatmapping to pick a grid sector with a high likelihood of containing a ship. It responds to information such as HIT and MISS feedback, along with ship dimensions and configurations in order to calculate the probability of a hit in any sector (in the context of a given ship or ships). For a given ship, the algorithm attempts to place it in all valid locations on the board, considering possible locations with a higher likelihood than surrounding, incompatible sectors may be. The heatmap representation is implemented using a numpy array of the board. All ships that have not yet been sunk are tried vertially and horizontally in their own heatmaps and values representative of their HIT probability are superimposed to calculate a total probability value for each grid sector grid. The implementation still utilizes its own version of the "hunt" and "target" modes, where hunting constitutes selecting high probability grid sectors, and targeting after an initial hit is represented by a sharp probability increase for the tiles immediately around the hit in the vertical and horizontal orientations. As the board becomes more populated with HIT feedback, MISS feedback, and sunken ships, the algorithm becomes more effective at narrowing down the locations where a ship could exist given its size and the current state of the board.

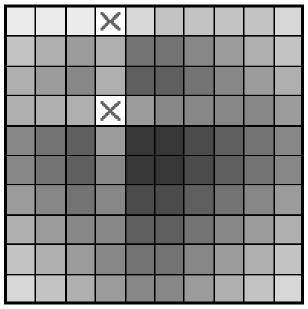

Figure 3: An example of a heatmap for the Carrier ship with two misses[11]

From *Figure 3*, we can observe the functionality of the probability heatmapping when there are a low number of shots (information available) for the current game state. The heatmap represents the probability of a given ship being placed in a grid sector, with darker colors representing higher probabilities (gradient coloring). It is clear that with very little information, the heatmapping does not differentiate greatly from sector to sector. It can be seen, even with only two shots, that the first three grid sectors in the top row are rated as having a lower probability due to the fact that a Carrier (length 5) would not be able to fit in a horizontal configuration, only vertically.

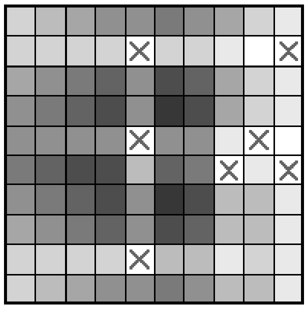

Figure 4: An example a heatmap for the Carrier ship with seven misses[11]

From *Figure 4*, We see that as the number of shots taken increases (All MISS feedback in this case), the heatmapping functions much more effectively to determine the most likely locations of the Carrier. The MISS grid sectors, along with any sectors that cannot possibly hold the ship in any configuration, are colored completely white to represent that there is zero probability of the given ship being placed in that sector. As the game progresses further, this becomes a highly effective method of hunting for ships when not in target mode.

3.2 Java Implementation

After attempting to implement the ISMCTS algorithm to our Python code implementation, we found that this was quite technically complex, and would not likely work without a custom implementation that would be better suited for ISMCTS. We then found Enes Kaya's *Battleship-MCST-Java* repository [20], which implemented ISMCTS to a simplified game of Battleship with a 5x5 grid and 3 ships of lengths 2, 2, and 3. In this implementation, the user also played against ISMCTS, rather than two algorithms competing against each other. Through modification of the codebase, a random Java implementation of the Random Selection Algorithm mentioned about was implemented in place of the user input, and the grid size was adjusted to 10x10 configuration. The correct ship count and lengths (2, 3, 3, 4, 5) were also implemented. Although this implementation was not directly comparable to the Python implementation, we assessed that it may still provide valuable insight into the effectiveness of ISMCTS in a Battleship Game domain.

3.2.1 Random Selection

3.2.2 ISMCTS

ISMCTS is a Monte Carlo Tree Search variant that uses information sets and determinization in order to estimate the best move from a current state. Like the standard MCTS, it executes several determinizations, which each emulate a game where hidden information is inferred randomly and the game is played to its terminal state, and averages them to find an ideal move for the next turn.

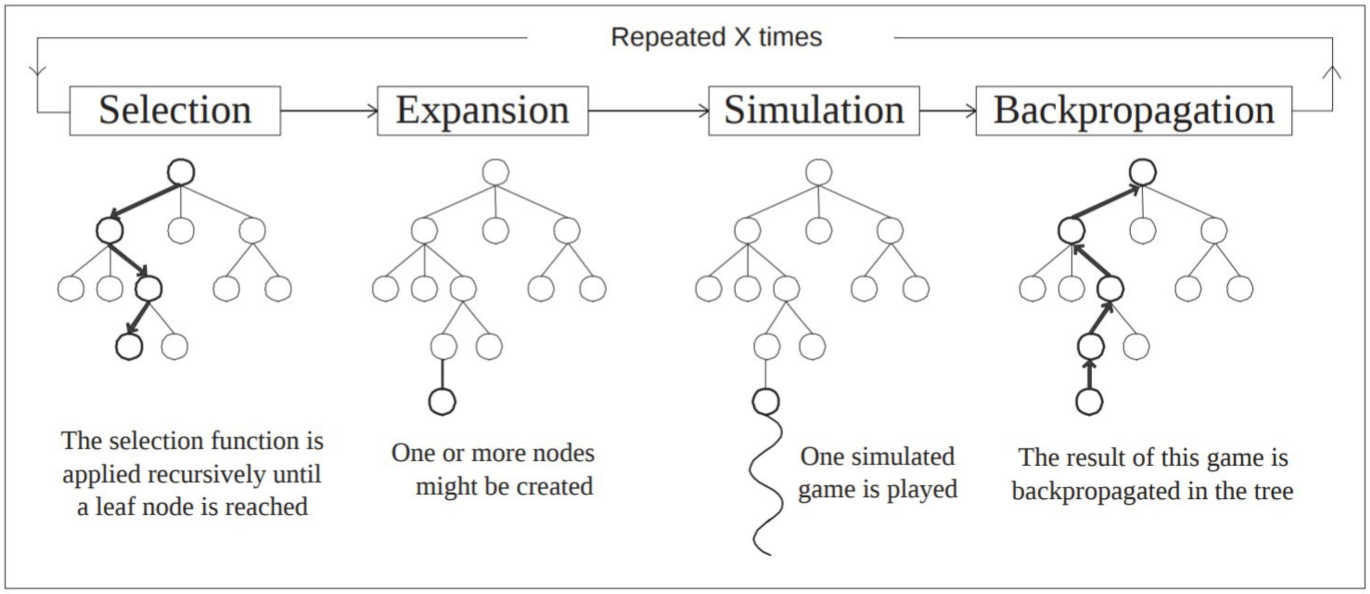
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes might be created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 5: The four primary steps of MCTS [21]

In the diagram above, the four main steps of MCTS are shown. MCTS builds a partial game tree, and using a multi-armed bandit approach, a leaf node is reached. From this leaf node, the tree is expanded to a randomly selected child node that has not yet been considered, and from this node, random nodes are selected until a terminal state of the game is reached. Then backpropagation is performed in order to update the tree with the result of the simulated game. The process is repeated for several iterations, often tuned to the problem to maximize a tradeoff of computational abilities and model accuracy [7]. These perfect state determinizations do not function well when applied to domains of imperfect information, as the simulation is played as if the locations of the opponent's ships (which have been randomly permutated for each simulated game) are known. This clearly disregards the prominent obstacle of the fog of war componenet of the Battleship domain, and will not serve as an accurate representation of the game states.

ISMCTS utilizes similar tactics, along with the idea of information sets. These sets represent a collection of states, cloned and randomized from the current node, which correspond with permutations of ship layouts of the opponent in this domain. Rather than averaging large numbers of determinizations, each is treated as its own iteration and is averaged continuously as it approaches its terminal state. This allows for much smaller branching factors, as illegal moves do not need consideration in this implementation [4], [7]. The ISMCTS Java implementation allowed for the tuning of determinizations per iteration, as well as the number of iterations completed for the calculation of each shot. In the majority of other aspects, this implementation was closely representative of the Battleship domain, and included the two player aspect desired for ISMCTS (that was not available in the Python implementation).

---

4 Experimental Design & Results

4.1 Experimental Setup

In order to compare each algorithm/approach against each other, we focused on certain metrics such as the amount of turns, runtime, and memory usage. For all approaches coded in our Python implementation, each approach had its own function that takes in a 10x10 board with randomly placed ships as input. Each approach was also run inside a for loop based on the requested amount of iterations to run. To determine the amount of turns taken to solve a board, the function takes in a variable that increments the amount of turns per move. The amount of turns per game would then be recorded into an Excel spreadsheet (using an xlsxwriter module) that records data based on how many successive iterations requested by the user. This spreadsheet would calculate other statistics such as average turns, minimum amount of turns, and maximum of turns.

To calculate the runtime of an approach of solving n-iterations, the start time at the beginning of the for loop and the end time at the end of the for loop was taken using the datetime module. We would then take the difference between the start time and the end time to obtain the runtime. Memory was calculated by calculating the different between the starting memory and the ending memory. This was done by taking the memory at the beginning of the loop and the memory at the end of the loop by using the psutil module. Both calculations were done for each approach found in the Python implementation for iterations of 250, 500, 1000, 1500, and 2500.

For the Java implementation of ISMCTS, we specifically chose to only focus on the amount of turns and successes. To accomplish this, we matched the ISMCTS algorithm against a random selection algorithm. We found this necessary due to ISMCTS' nature. ISMCTS makes a decision by using imperfect information which includes emulating the opponents board by taking in their move. Thus, we automated our testing by setting the opponent to be a random selection algorithm who always went first. A variable was used to increment the amount of turns for the sake of our results. Settings inside the implementation were also varied to test the effects of how many moves the ISMCTS algorithm would need to win a game. Settings such as number of determinizations, iteration count, and max amount of time to create a determinization for a player. This implementation was also tested on a 5x5 board with two size 2 ships and one size 3 ship along with a classic 10x10 board.

#	Avg Turns	Min Turns	Max Turns
Random	95.644	72	100
HLPA	60.731	26	99
OHLPA	60.188	21	98
PHLPA	54.042	28	72
OPHLPA	51.96	26	69
Greedy	46.723	25	72

Table 1: Amount of Turns Over 1000 Iterations

4.2 Results

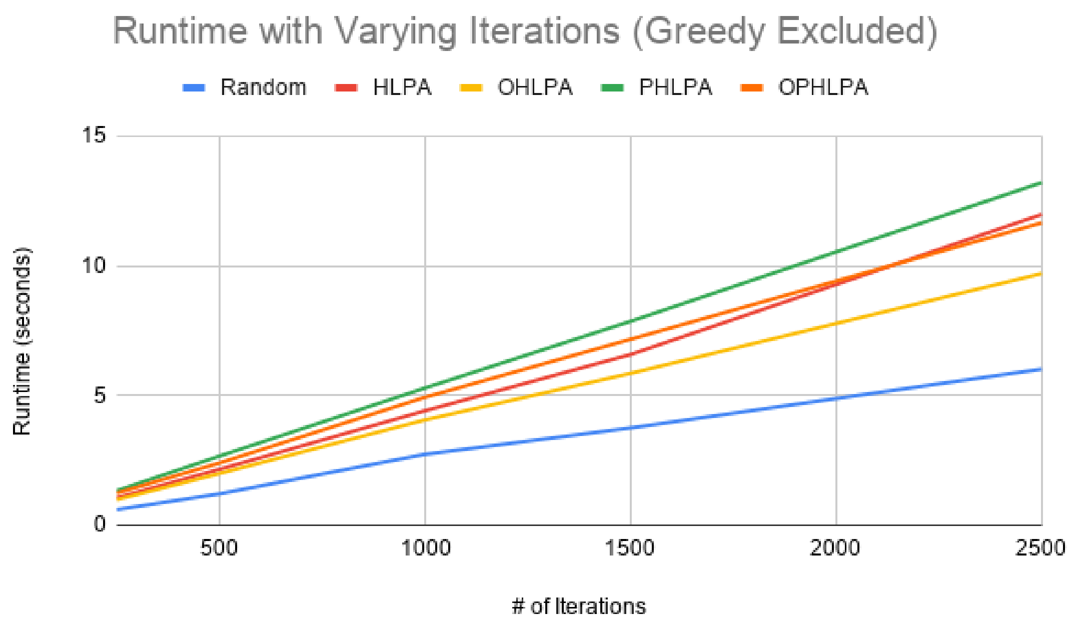

Figure 6:

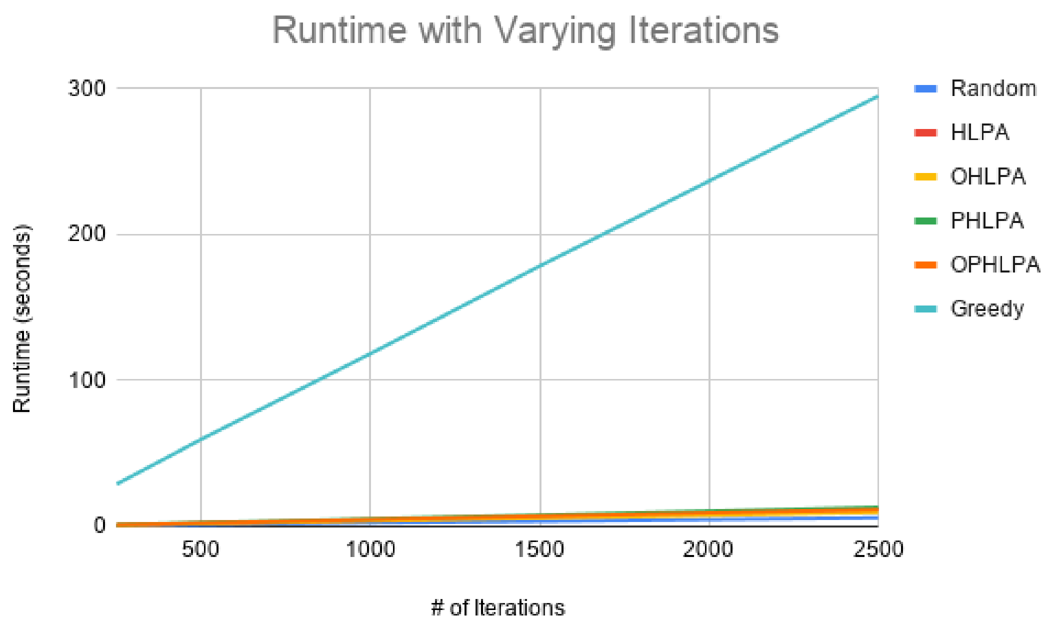

Figure 7:

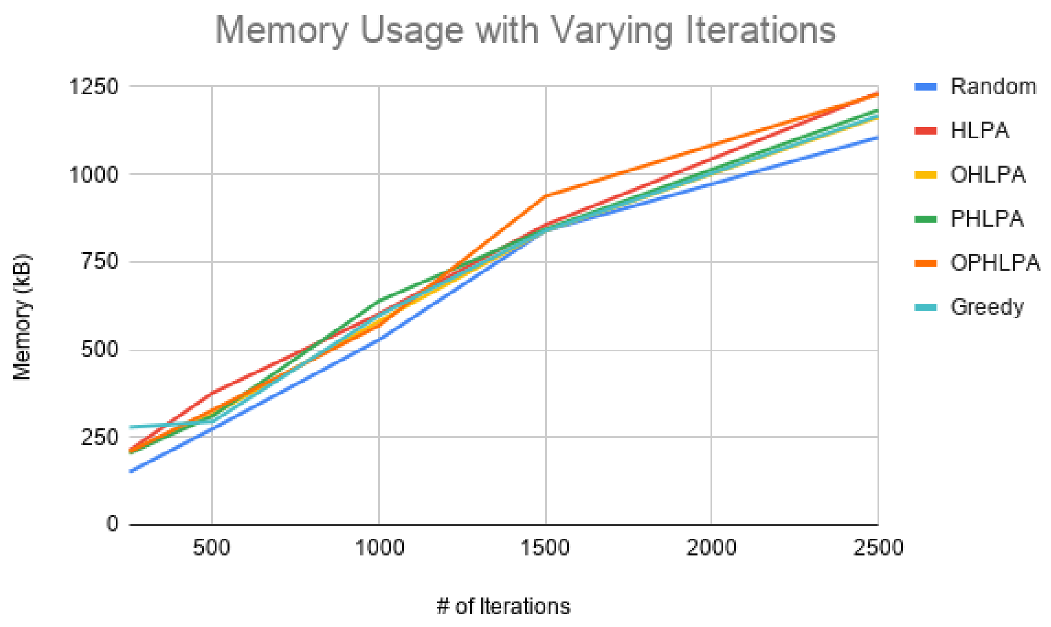

Figure 8:

5 Analysis of Results

Table 1 shows the results of the amount of turns for each game/iteration based on the approach used in the Python implementation. From our results, the Greedy Probability algorithm was shown to have the smallest amount of average turns. In contrast, the Random Selection algorithm was shown to have the worst performance amongst the approaches with the most amount of average, minimum, and maximum turns. With an average of 95 turns, the Random Selection would need to clear approximately a 10x10 board before finding all the ships. Comparing the HLPA and OHLPA approaches, the amount of average turns and maximum turns were approximately the same. However, the minimum amount of turns used by OHLPA was smaller by 5 turns, having the least amount of minimum turns. Now, comparing PHLPA and OPHLPA, the amount of average turns for OPHLPA was smaller by approximately 2 turns with a similar amount of minimum and maximum turns. We also see that the approaches that used parity had less average turns compared to their counterparts.

Examining Figures 6 and 7, the Greedy Algorithm stands out by having a significantly larger runtime compared to other approaches. Figure 6 shows the approaches being very similar in their runtimes, only within a couple seconds of another approach as it reaches 2500 iterations. However, had the a greater amount of iterations were tested, the difference in runtime between each approach would become much larger and more noticeable. From Figure 6, PHLPA has the greatest runtime and Random Selection has the lowest runtime. For each approach, the runtime remained linear with respect to the number of iterations. Similar to runtime, in Figure 8, the memory usage was approximately the same across all approaches showing a generally linear relationship with respect to the amount of iterations.

During the testing of the Java implementation of ISMCTS, the amount of determinizations, iterations, and max amount of time to create a determinization were changed frequently. Despite changing these settings, our implementation would rarely produce results due to stalling. We found that the algorithm primarily stalled when creating determinizations due to a ship being placed illegally on the board in one of its many emulations. This would cause the algorithm to continuously emulate the incomplete game states and restart the looping to create determinizations. In our testing, we found that lowering the amount of determinizations and iterations would significantly reduce the runtime of the algorithm. Although the runtime is lowered, the results and decisions of the ISMCTS are much weaker causing it to take more turns to successfully beat the Random Selection opponent. When the determinizations and iterations were set to much larger values, the algorithm takes much longer to choose a move. However, the algorithm makes a much lower amount if moves to win the game. Concerning the impact of the max amount of time to create a determinization, the runtime and results stayed quite similar. Lastly, we observed that the 5x5 board had a much higher success rate of not stalling and finishing a game versus a 10x10 board.

As explained above, the Random Selection algorithm had the weakest performance in terms of the amount of turns. However, the approach has the strongest performance in terms of runtime. In comparison to the Greedy Probability approach, this is the exact opposite. Upon examining the algorithms for each approach, it is clear that the Random Search has a very simple implementation with very few lines of code allowing it to solve a battleships board quite fast. The Greedy Probability algorithm spends a lot of time having to calculate the probabilities of potential ship locations based off previous turns. During this calculation, time is spent generating heatmaps dependent on what types of ships have sunk. This analysis of the amount of turns and runtimes using the two extremes in our findings lead us into a discussion of practicality. Given the nature of Battleship as a static game that is turn based, it makes sense to choose the Greedy Probability algorithm as an approach to play Battleship. However, if our goal was to play through many games as quick as possible, then the Random Selection algorithm is the best choice. Also, it could be the case that we want to have a relatively low amount of turns along with a relatively fast runtime, then we can choose a human player approach.

6 Conclusion and Future Work

After careful analysis of the various algorithms applied to the Battlehsip Domain in this experiment, our results found that the Greedy Probability Search Algorithm yielded the lowest average turns per game (46.723 turns). Despite its higher runtimes, we have determined it to be the most effective of the algorithms tested due to its superior turns per game performance. In a turn based/static game, we found that runtime was of lesser importance than minimizing the turns per game average.

In future experimentation, there are several considerations to be made based on the above findings. Most prominently, a functioning, custom testing codebase for ISMCTS should be developed and the experiment should be conducted in an environment where all approaches can be compared in their entirety. In addition, other MCTS variants, GGP approaches, and Greedy algorithms should be explored to determine if improvements to the current lowest average turns per game can be made. Other domain factors, such as human biases toward non-random ship placement, should be considered in future work to determine how our approaches are impacted by these changes to the domain.

---

7 Teamwork Contributions

The division of labor between contributors is detailed below:

John Ferstle

- Wrote multiple sections of the report, including: Abstract, Introduction, Related Works, Approach, and Conclusion
- Created and modified code for the Java ISMCTS Battleship implementation
- Researched and analyzed articles referenced in the report
- Revised report and reviewed peer work
- Ran tests and gathered data for use in the report

Michael Nguyen

- Wrote multiple sections of the report, including: Related Works, Experiment Design and Results, Analysis of Results, and Conclusion
- Created and modified code for the Python Battleship implementation
- Researched and analyzed articles referenced in the report
- Revised report and reviewed peer work
- Ran tests and gathered data for use in the report

References

[1] Guillaume Chaslot et al. “Monte-Carlo Tree Search: A New Framework for Game AI.” In: *AIIDE*. 2008.

[2] Tristan Cazenave et al. “Nested Monte Carlo search for two-player games”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1. 2016.

[3] Michael Schofield and Michael Thielscher. “General Game Playing with Imperfect Information”. In: *Journal of Artificial Intelligence Research* 66 (2019), pp. 901–935.

[4] Daniel Whitehouse, Edward J Powley, and Peter I Cowling. “Determinization and information set Monte Carlo tree search for the card game Dou Di Zhu”. In: *2011 IEEE Conference on Computational Intelligence and Games (CIG’11)*. IEEE. 2011, pp. 87–94.

[5] Paolo Ciancarini and Gian Piero Favini. “Monte Carlo tree search in Kriegspiel”. In: *Artificial Intelligence* 174.11 (2010), pp. 670–684.

[6] Fredrik A Dahl. “The lagging anchor algorithm: Reinforcement learning in two-player zero-sum games with imperfect information”. In: *Machine Learning* 49.1 (2002), pp. 5–37.

[7] Peter I Cowling, Edward J Powley, and Daniel Whitehouse. “Information set monte carlo tree search”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.2 (2012), pp. 120–143.

[8] Maciej Świechowski et al. “Monte Carlo Tree Search: A Review on Recent Modifications and Applications”. In: *arXiv preprint arXiv:2103.04931* (2021).

[9] Carmine Cerrone, Raffaele Cerulli, and Bruce Golden. “Carousel greedy: a generalized greedy algorithm with applications in optimization”. In: *Computers & Operations Research* 85 (2017), pp. 97–112.

[10] Loıc Crombez, Guilherme D da Fonseca, and Yan Gerard. “Efficient Algorithms for Battleship”. In: *arXiv preprint arXiv:2004.07354* (2020).

[11] Nick Berry. *Battleship Analysis (Image citation)*. `https://www.datagenetics.com/blog/december32011/`. 2019.

[12] Daniel Whitehouse. “Monte Carlo tree search for games with hidden information and uncertainty”. PhD thesis. University of York, 2014.

[13] Tomáš Kancko. “Reinforcement Learning for the Game of Battleship”. In: ().

[14] Faiz Ilham Muhammad. “Greedy Algorithm and String Matching in Battleship Game Strategy Using Probability Density Matrix”. In: (2013).

[15] Mark H. M. Winands. “Monte-Carlo Tree Search in Board Games”. In: (2016).

[16] Cameron B Browne et al. “A survey of monte carlo tree search methods”. In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[17] David Silver and Joel Veness. “Monte-Carlo planning in large POMDPs”. In: Neural Information Processing Systems. 2010.

[18] Olivier Teytaud Marie-Liesse Cauwet. “Surprising strategies obtained by stochastic optimization in partially observable games”. In: *IEEE Congress on Evolutionary Computation* (2018).

[19] Eric H. *Battleship Probability Analysis (Python Code Citation)*. `https://docs.google.com/document/d/1WyeNRwrXKOyGce1DBAulDZjLPJ79MYVDY3r81La7TA8/edit`. 2020.

[20] Enes Kaya. *Monte Carlo Tree Search for Battleship (Java Code Citation)*. `https://github.com/eneskaya/Battleship-MCTS-Java`. 2019.

[21] Michael Liu. *General Game Playing with Monte Carlo Tree Search (Image Citation)*. `https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238`. 2017.