A Parallelized Monte Carlo Battleship Implementation

Project by: Yu-Chun Hsiao and Ranjani Sundaram

Abstract

While the game theory field has existed for years, traditionally, it has focused most of its efforts on perfect information games like Chess and Alpha Go. Specifically, these perfect information games are all characterized and defined by rules such that at each turn, every player has complete information over all the events that have previously occurred, including the initialization of the event. From here, decisions need only be made based on the opponent's moves. However with imperfect information games, various factors of the game remain unknown, and each decision must factor in all possible variants of the unknown variables. Here, we implement a parallelized Monte Carlo Tree Search algorithm to play the imperfect information game Battleships.

Problem Description and Motivation

Battleship is a common household game played around the world. With a simple set of rules where players attempt to sink their opponents' ships by guessing spots on a grid, Battleship is truly a game for all ages. Yet being a deterministic game, researchers have also taken the challenge to try and develop algorithms that can play the game at a sufficient level. Such algorithms can not only act as a benchmark for algorithms research, but also serve as artificial intelligence bots for individual play.

In this project, we attempt to implement, reconstruct and parallelize select game theory algorithms that have traditionally been successful in perfect information games so that they can be applicable in an imperfect information setting as well. While previous studies have shown that algorithms such as the Monte Carlo Tree Search algorithm, the MiniMax algorithm, and the Double-Oracle sequence form algorithm are all applicable in game theory, we plan on focusing our efforts on the Monte Carlo Tree Search algorithm. In addition, we aim to show the time and efficiency improvements that parallelization has over serial algorithms. Specifically, we want to see the effects of parallelization when applied to a scenario where the problem is deterministic in nature, yet at each time step, information is imperfect. Outlined below is a brief description of the Battleship rules we will assume and implement.

- 2 players
- Deterministic
- Imperfect Information
- Played on a grid
- Zero sum

- Imperfect information
- 3 Ships of size 2x1
- Boards of size 5x5 and 10x10
- Ship locations can be placed or randomized

Initially, each player places their ships on different locations on the grid. However, these locations are not disclosed to the other player. Players alternatively call “shots” at the other players' ships. The objective of a player is to destroy all of the other player’s ships. At the end of each turn, a player’s shot is characterized as either a “hit” or a “miss”. Both players have the same collection of ships, and each ship is assumed to be a contiguous sequence of grid squares. Each grid square of the ship has to be hit in order to destroy it.

At any turn, there are two possible moves/strategies for a player:

1. Try to hit a new ship of the other player.
2. Destroy a pre-hit ship with a minimum number of misses.

After every turn, each player gets more information regarding where the opponents’ ships are placed. This information can be used to either further sink an already hit ship, or deduce where any remaining ships may reside. The ultimate goal and conclusion of the game occurs when a player no longer has any ships left.

Prior Work

There have been several search algorithms that are currently used for game-playing systems. Some examples include branch-and-bound techniques, alpha-beta tree pruning and the minimax algorithm. Most of these algorithms operate on a *Game Tree*, which is a tree with vertices denoting game states and edges denoting moves. Most tree search algorithms parse the game tree to identify desirable states. There has been extensive work on search-based algorithms for perfect information games- that is, games where a player can view all the other player’s moves. The most famous example is perhaps the Monte Carlo Tree Search algorithm for Alpha Go. However, Battleships is a deterministic, imperfect information game. Current literature on imperfect information games is much scarcer. [4] provide a parallel alpha-beta pruning algorithm for Tic-Tac-Toe. [1] presents and implements a parallel Double-oracle Sequence form algorithm for imperfect games. We plan to leverage some of the game playing methodologies from these two papers. [5] provide a parallel Monte Carlo Tree Search algorithm for imperfect games. We plan to adapt and implement this algorithm for Battleship.

Goals and Deliverables

Our goals for this project are to reconstruct algorithms that have shown success in playing perfect information games so that they can successfully play imperfect information games as well. In this case, our game of choice is Battleships. Before we begin with any of the actual game-playing algorithms, we must first implement a playable version of the Battleships board game, including proper initialization of the board, logic behind what constitutes a hit or miss, and tracking of the board on a per turn basis. In addition, the implementation should have some form of user interface so that users are able to interact with the game. The implementation should also have multiple settings where users can choose to either play against a random bot or an integration of the algorithm of choice. The underlying logic of the game should follow design specifications discussed above; There are only 3 ships of size 2x1, board size is 10x10, and location of the ships are initialized randomly.

Initial metrics for evaluation will be compared against a random-decision model. This is based on standard machine learning evaluation methodologies where if the model can make decisions superior to randomized guesses, then the model is considered intelligent. In other words, if our algorithm can win a game with on average fewer moves than a bot that chooses a random move at every turn, then we determine that the algorithm is successful. From there we can further improve the algorithm by optimizing parameters like the probability density functions. After successful implementation of the above mentioned quasi intelligent algorithm, our next objective is to study the effects of parallelization on the algorithm. We expect that parallelization will have improvements to runtime, but also hope to show that parallelization will have some effects on algorithm efficiency and decision making. By varying the number of threads, we can monitor exactly how parallelization optimizes the algorithm.

While our original goals were to provide implementations for multiple game playing algorithms, including the Monte Carlo Tree Search algorithm, the MiniMax Algorithm, and the Alpha-beta pruning algorithm, due to time constraints and unforeseen complexities with designing the algorithms, we decided to focus our efforts on the Monte Carlo Tree Search algorithm. Our updated goals are to analyze the time and efficiency of the Monte Carlo Tree Search algorithm in a serial and a parallelized setting. Specifically, we want to identify the advantages and disadvantages of applying parallelization algorithms to a deterministic yet imperfect information scenario such as in the case of Battleship. We would also like to study whether using parallel algorithms would produce different winning strategies compared to their serial counterparts.

Brief Overview

In order to set up our experiment in an environment where we can run and compare our serial and parallel algorithms effectively, we have to first build an implementation of the Battleships board game. This involves designing Class objects to encapsulate the game logic, the board, and proper management of players making moves. We also designed a Player class to represent Players in the game. The Player class can further be defined through inheritance as either a real player, a random bot, or the MCTS bot. Each type of player is defined in detail within their own class, and common functions are inherited through the Player class. Lastly, several helper and miscellaneous Classes are defined, such as the Ship class and utility functions. After building a working implementation of the game, we then test the parallelized MCTS by running it against its serial and random counterparts. We obtain results from varying the number of threads used during each run.

Description

In this section, we describe the properties of a game of Battleships under our current model, and explain how the Monte Carlo Tree Search algorithm can be adapted to it.

Theory behind MCTS and Battleships

In combinatorial game theory, a **game tree** is a graph that represents all possible states of a game. It is a good indicator of the complexity of a game, as the number of children per node, or the **branching factor,** determines the many ways a game can pan out. Algorithmic bots for games typically use the game tree partially or fully to determine a winning strategy. Monte Carlo Tree Search is one such algorithm.

Recall that Battleships is an imperfect information game. Monte Carlo Tree Search is easier in the case of perfect information games like Chess or Alpha Go. Let us first go through the high-level idea behind this algorithm by using Tic-Tac-Toe as an example.

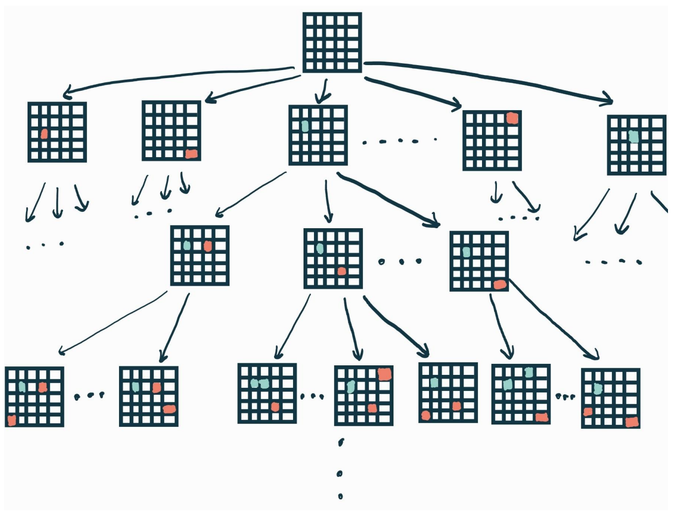


Fig: Game tree for Battleships

MCTS Algorithm

MCTS is a probabilistic search algorithm that parses parts of the game tree to evaluate moves with highest expected payoff. It randomly samples unknown states (children) at every node instead of brute forcing its way out of every possibility. Initially, the algorithm builds a game tree with a root node and then expands it iteratively using rollouts. For each node, the tree keeps track of the number of visits and wins/losses.

The MCTS algorithm primarily consists of four steps:

- Selection
- Expansion
- Simulation/Rollout
- Backpropagation

Selection: The objective of this phase is to pick a leaf node that yields maximum payoff. By a leaf node, we mean a state of the game tree that is one move away from the current state. We wish to make sure that every leaf node is given a fair chance. For Tic-Tac-Toe, this can be Upper Confidence Bound (UCB) value.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

Here, w_i is the number of wins, n_i is the number of simulations after the ith move, c is a constant, and t is the number of simulations of the parent node. This formula ensures that unexplored children get chosen with some probability while simultaneously favoring winning children.

Expansion: Unless a leaf node is a definite loser, the algorithm appends one or many possible children of the leaf node to the tree.

Rollout: After expansion, the algorithm completes one random playout from the child node until a win, lose or draw is achieved.

Backpropagation: Finally, the algorithm traverses all the way back to the leaf node in the selection step while updating values of every node in the path.

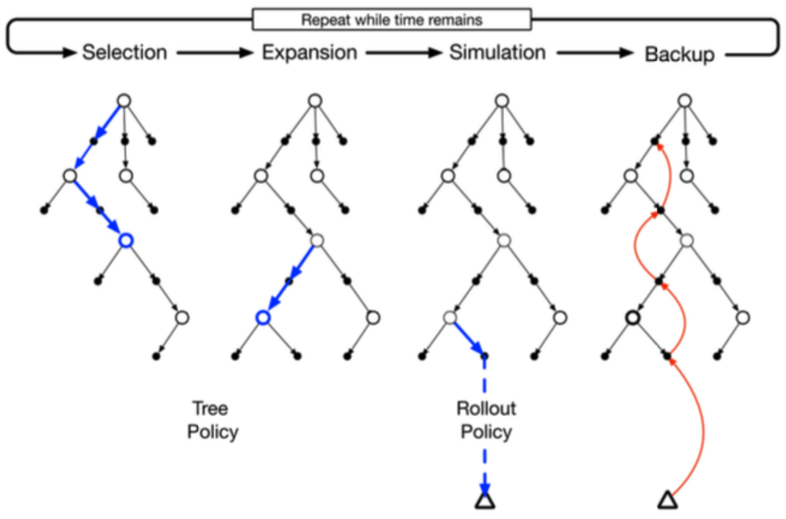


Fig: MCTS algorithm

Imperfect Information

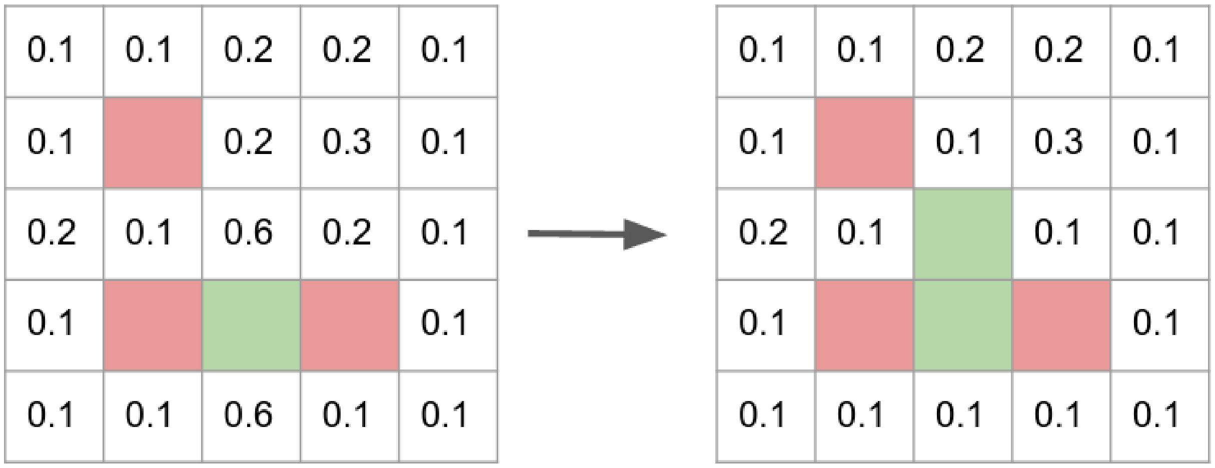
In the case of Chess or Tic-Tac-Toe, during the rollout phase, the MCTS algorithm can determine whether or not a state constitutes a win, lose or draw. However, with Battleships, this is unclear. Only when all slots are explored can we definitively say that all ships have been discovered.
In order to overcome this problem, we use a probability **distribution function** over the unexplored slots. Here, the value corresponding to each slot denotes the probability of receiving a hit when we play that slot. Note that this probability can be used in lieu of the UCB value described for Tic-Tac-Toe. This is because at every step of the selection and rollout, we re-compute these probabilities and update the probability of the parent during backpropagation.

Computation of Probability Function

For each slot on the board, we compute the estimated probability of a hit based on its neighboring slots. In other words, each slot (named current) may or may not contribute a small value to its neighbors (named neighbor). For simplicity, let 'numFinish' denote the number of ship positions that have not yet been discovered and ‘numEmpty’ denote the number of empty neighbors of the current node. We then determine this value in the following manner.

- If current is a hit, each empty neighbor gets a value of numFinish/numEmpty.
    - Exception: A neighbor is also a hit, in which case all neighbors get 0 value from current.
- If current is a miss, neighbors get 0 value from current.
- If current is empty, each neighbor gets a value of numFinish/(total number of empty nodes left)

Finally, we upper bound each probability by 1, and assign a probability of 0 to all explored slots.



0.1	0.1	0.2	0.2	0.1
0.1	🟥	0.2	0.3	0.1
0.2	0.1	0.6	0.2	0.1
0.1	🟥	🟩	🟥	0.1
0.1	0.1	0.6	0.1	0.1

0.1	0.1	0.2	0.2	0.1
0.1	🟥	0.1	0.3	0.1
0.2	0.1	🟩	0.1	0.1
0.1	🟥	🟩	🟥	0.1
0.1	0.1	0.1	0.1	0.1

Parallelization of MCTS

We now consider different ways of parallelizing our probabilistic Monte Carlo Tree Search algorithm. Parallelization of MCTS was first considered in [7]. They consider three kinds of parallelization:

- Leaf parallelization
- Root parallelization
- Tree parallelization

Leaf parallelization: Only one thread traverses the tree and adds one or more nodes to the tree when a leaf node is reached (Expansion phase). However, starting from the leaf node, independent simulations or rollouts are played for each available thread. When all rollouts are finished, the result of all these simulations is propagated backwards through the tree by one single thread.

Root parallelization: This consists of building multiple MCTS trees in parallel, with one thread per tree. Here, multiple leaves are chosen in the selection step and expansion is applied independently to each of them. Similar to leaf parallelization, the threads do not share information with each other. Finally, the best leaf is chosen for the next move.

Tree parallelization: This method uses one shared tree from which several simultaneous games are played. Each thread can modify the information contained in the tree, therefore, mutexes are used to lock from time to time certain parts of the tree to prevent data corruption.

In our implementation, we use a combination of root and leaf parallelization. However, as an extension of this project, we seek to implement tree parallelization and use modified UCT bounds for probability computation.

Implementation

We now go over some details of our implementation of Battleships and some of the algorithms we use. We use C++ for our entire codebase. We use cilk++ for implementing the parallelization.

Battleships:

The first step in the implementation of this project is to get a working version of the Battleship board game. The game is initialized from the command line, and a user interface is required in order for the player to interact with the game. In addition, the game should follow certain specifications, as well as follow all traditional rules of the original Battleship game. These specifications are listed here:

- 2 Player board game
- Board size default set to 10x10; customizations allowed
- Ship sizes are 2x1
- Default ship locations are randomly set; customizations allowed
- Game logic should alternate moves between the players
- Detect whether a move is within the bounds of the board
- Keep track of all previous moves and hits
- End game when player has won
- Set player names from command line input
- Select type of bot to play against; random, MCTS, other player

Implementation of the above mentioned rules provides us a robust environment and template to test our serial and parallelized MCTS algorithm. Execution of the Battleships program will ask users to provide names for the players, as well as the type of opponent to play. If the random bot is selected, moves will be selected uniformly at random during each turn. If the MCTS algorithm is selected, the game calls the MCTS function to make the move. Full implementation of the game involves the Board class, the Battle class, the Player class, the Ship class and the MCTS class. Specifications for each class are described in detail below.

Board

The Board class represents the board of the game. During initialization, the Board class creates a board of size BOARD_DIM. For this project, this is default set as 10x10; the BOARD_DIM variable can however be changed as necessary. The Board class also contains initialization and utility functions such as making a copy of the board, computing the expected probability of a hit on each position on the board, printing the current state of the board, and placing ships on the board. The Board class also includes functions that pertain to a Player's move, including updating the board, and getting the current state of a position on the board. We chose an object-oriented design for our board so that it can be instantiated as required by other Classes. Additionally, each player's board is set to private in order to prevent the other player from accessing it.

Battle

The Battle class represents the logic and management of the game itself. It includes functions that pertain to managing the overall integrity and logic of the game. These functions include checking win conditions, managing Players to make moves one after another, initializing the Board class, and starting the game. The Battle class ensures that at every turn, logic of the game is upheld, and Players are only making moves that are valid.

Player

The Player class represents a player in the game. When initialized, the Player is assigned a unique number that is used by the board to manage player identity and turns. The Player class also has a field named isAutomaticPlayer which is used to establish non-user Players. When a Player is designated as a non-user Player, they will automatically make moves, for example the Random bot and the MCTS bot. Lastly, the Player class includes various other helper functions, such as making a copy of the Player and setting/changing field variables. The role of the Player class is to hold information pertaining to what type of Player is playing the game.

Ship

The Ship class represents the ships on the board. It contains fields that hold the coordinates of the ship, the current state of the ship, the size of the ship, and a name that corresponds to a Player in the game. The Ship class also includes functions that operate on the ship's own fields, including various checks of the ship's current status, update functions, and return functions. The Ship class is initialized by the Board and Player class, and is initialized with a name that corresponds to the Player that owns the ship.

Execution

The overall execution of the game iteratively follows this sequence:

1. Print the menu and retrieve input values.
2. Call the Battle class constructor that initializes the Board and Ship values.

3. The playGame function in the Battle class assigns the players to the boards and starts the turn-based game.
4. If the player is a Random Bot, the corresponding function is called. If they are an MCTS bot, an instance of the MCTS class is created with a copy of the current board and the best next move is returned. If the player is human, a valid move is collected as input.
5. The moves are played and the active player/board are switched.
6. Winning condition is checked.

Random Bot

At all times, the current state of the board is saved by the game object. The random bot takes this board and selects a random empty spot during every turn.

From the perspective of a random bot, the Battleship problem is similar to the ‘urn problem’, where there are ‘n’ goods and ‘m’ bads in an urn, and items are drawn without replacement. Therefore, in k shots, the expected number of hits the random bot gets is upper bounded by

$$E[X]<=k*6/(N^2)$$

MCTS

The MCTS class contains the functions corresponding to Selection, Expansion and Rollout phases. Whenever it is the turn of the MCTS bot, the Battle instance creates an instance of MCTS with a copy of the board. The Selection function computes hit probabilities for all empty spots of the board, and selects top ‘r’ values from these. In our experiment section, we use the variable ‘Branch’ to denote the initial value of r.

The Selection function then calls Rollout, which recursively branches out, and assigns a value called ‘average depth’ to each node. ‘Average depth’ denotes the average number of recursive calls from a node until all the ships are discovered. At each iteration of Rollout, the number of branches, or ‘r’, reduces by a factor called ‘decay’. This is done to reduce the total number of nodes, and thus the overall running time of the MCTS algorithm. At each node, Rollout function also computes the hit probability ‘p’ of every slot. Then, whenever it simulates a move, it places a hit with probability ‘p’, and a miss with probability ‘1-p’.

Once Rollout returns the ‘average depth’ value for each of the ‘r’ initial leaf nodes, the move corresponding to the minimum depth node is returned to Battle as the next move.

In our implementation, we use a different thread for each leaf node in the Selection phase. Our algorithm is also amenable to using different threads for some branches in the rollout function.

Experimental Results

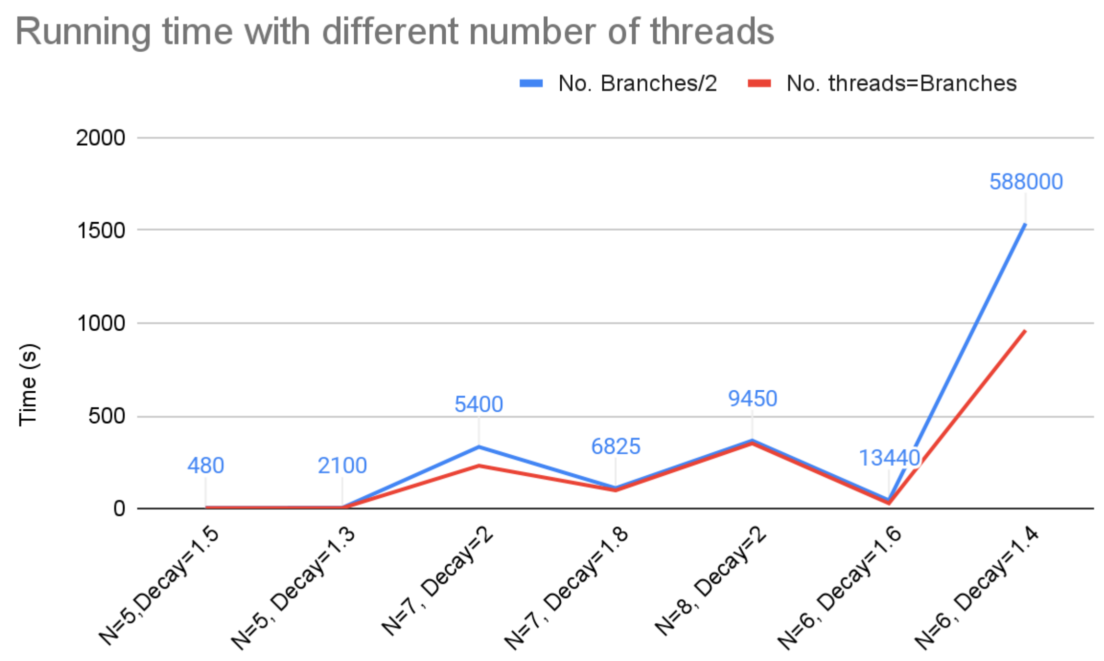
We ran our algorithms on a number of inputs with varying parameters. We ran 5 samples for each set of parameters and obtained the following results. The parameters are as follows:

- N = length of board
- Branch = Initial number of branches in Selection and Rollout
- Decay = The factor by which number or branches is decremented in every iteration of Rollout.

Threads		**1**					**Branch/2**					**Branch**				
N=5, Branch=10, Decay=1.5	Time (s)	7	5	7	6	7	6	4	4	5	6	5	3	4	6	5
	Won	0	1	1	1	1	1	1	1	1	1	1	1	0	0	1
	Moves	10	11	20	18	16	18	15	19	23	13	23	18	18	24	15
N=5, Branch=10, Decay=1.3	Time (s)	9	8	9	8	7	6	6	5	6	6	4	5	5	4	5
	Won	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1
	Moves	20	16	18	17	14	11	19	18	18	16	20	19	21	24	18
N=6, Branch=20, Decay=1.6	Time (s)	160	205	158	187	221	57	44	48	45	37	24	37	31	38s	28
	Won	1	1	0	1	1	1	1	0	1	1	1	1	1	0	1
	Moves	26	25	29	31	22	27	24	29	21	25	23	30	26	32	20
N=6, Branch=20, Decay=1.4	Time (s)	killed	killed	killed	killed	killed	1284	1163	1682	1143	2412	1019	751	1010	1206	823
	Won	killed	killed	killed	killed	killed	1	1	0	1	1	1	1	1	0	1
	Moves	killed	killed	killed	killed	killed	28	26	35	20	24	25	19	29	33	24
N=7, Branch=25, Decay=1.8	Time (s)	killed	killed	killed	killed	killed	128	120	95	113	103	100	108	93	100	97
	Won	killed	killed	killed	killed	killed	1	1	1	1	1	0	1	1	1	1
	Moves	killed	killed	killed	killed	killed	39	34	25	39	30	39	37	27	36	28
N=7, Branch=25, Decay=1.6	Time (s)	killed	killed	killed	killed	killed	338	258	357	287	429	116	160	225	290	371
	Won	killed	killed	killed	killed	killed	1	1	1	1	1	1	1	0	1	1
	Moves	killed	killed	killed	killed	killed	34	23	27	21	37	23	20	25	23	30
N=8, Branch=30, Decay=2	Time (s)	killed	killed	killed	killed	killed	204	363	475	393	404	254	350	379	399	382
	Won (s)	killed	killed	killed	killed	killed	1	1	0	1	1	1	0	1	1	1
	Moves	killed	killed	killed	killed	killed	55	34	60	47	42	35	49	39	46	42
N=8, Branch=30, Decay=1.8	Time (s)	killed	killed	killed	killed	killed	548	652	550	444	426	406	443	395		
	Won	killed	killed	killed	killed	killed	1	1	1	1	1	1	0	1		
	Moves	killed	killed	killed	killed	killed	58	39	48	28	49	38	62	43		

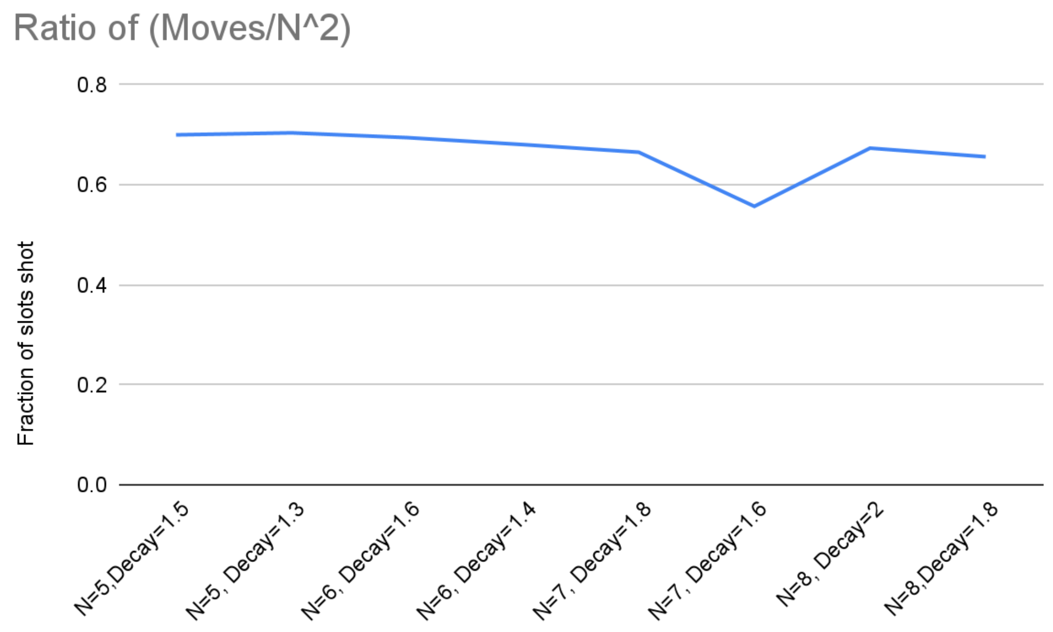
We plotted various portions of our results.

Plot 1: Running times with different numbers of threads against input size.



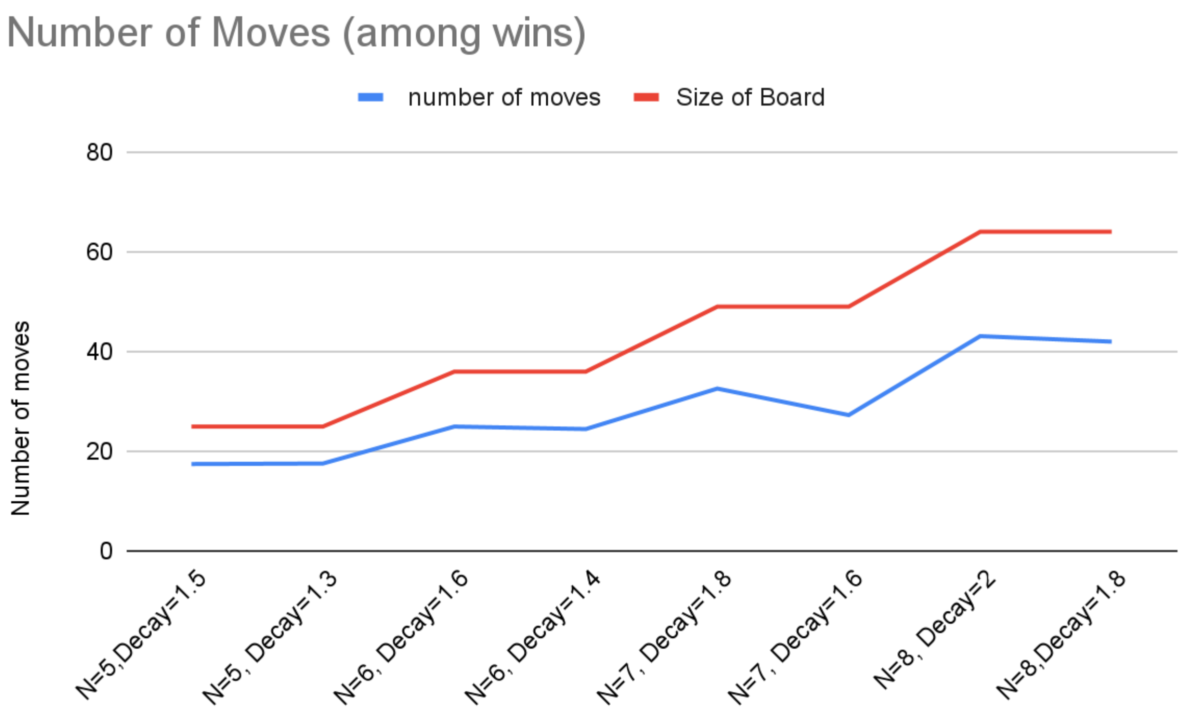
In the above graph, we plot the average running time of the game when we use Branch/2 and Branch number of threads. The numbers above each data point on the graph represents the total number of branches in the game tree.

Plot 2: Fraction of slots shot in a board



In the above graph, we plot the fraction of slots in the board that were used as moves among the games where MCTS won. We note that around 60-70% of the slots were shot. In the case N=6, Decay=1.6, we observe that less than 60% of the slots were shot. This indicates an improvement in the performance of our MCTS algorithm when the total number of branches is increased.

Plot 3: Number of Moves taken to win



Conclusion

After tabulating our results, we can clearly see that parallelization significantly helps with the run time of the Monte Carlo algorithm. Generally once we start implementing parallelization on our most basic settings, we see a speed increase from roughly 60 seconds to run times in the range of 5-10 seconds. However as the size of the board increases, the number of computations required scales exponentially. This is shown where as we increase the board size, many of the computations start timing out. Decreasing the decay rate also produces a similar effect. We conclude that using a 5x5 board size, a branch size of 10, and decay rate of 1.5 produces the most optimal results. However, in a realistic scenario, these parameters may not be practical. It is not feasible to restrain the Battleship game to a board size of 5x5. While parallelization within the Battleship setting has shown immense potential for success, more work is required in order to build a system that is optimal in a real world setting.

Future Work

1. Since the decision making of the Monte Carlo Tree Search algorithm relies heavily on the probability density function, optimizing it would provide significant improvements to the overall algorithm. In other words, the better the probability density function is at identifying sections of interest, the better the overall algorithm is. Given more time, this would be the most important part to improve, as it would lead to the most significant improvements of the algorithm.
2. Currently, we have all of our ships set to constant sizes of 2x1. An interesting topic to implement in the future would be to vary the type of ships on the map. This would introduce a whole new parameter for the algorithm to consider. At every step, the size of the ship can lead to further hits in the proximity.
3. Parallelization of the Monte Carlo Tree Search algorithm can be generally split into three categories; lead parallelization, root parallelization and tree parallelization. In this project, we implemented leaf and root parallelization. A future goal would be to implement tree parallelization. While it may not lead to improvements with the algorithm's decision making, this optimization most definitely can lead to improvements in run-time.
4. This project focuses on parallelization of the Monte Carlo Tree Search Algorithm. An extended goal would be to create implementations for the alpha-beta pruning algorithm and minimax algorithms. It would be interesting to see how these three algorithms compare with each other, as well as how parallelization affects each individually.

References

[1] Svatoš, M. (n.d.). *Parallelization of an algorithm for solving imperfect ... - dspace.cvut.cz*. Retrieved May 15, 2022, from https://dspace.cvut.cz/bitstream/handle/10467/24272/F3-BP-2014-Svatos-Martin-prace.pdf

[2] Crombez, L., da Fonseca, G. D., & Gerard, Y. (2020, April 15). *Efficient algorithms for battleship*. arXiv.org. Retrieved May 15, 2022, from https://arxiv.org/abs/2004.07354

[3] C. B. Browne et al., "A Survey of Monte Carlo Tree Search Methods," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1-43, March 2012, doi: 10.1109/TCIAIG.2012.2186810.

[4] Y. AlSallut , A. (n.d.). *A fast parallel alpha-beta algorithm for Tic Tac Toe Game*. Retrieved May 15, 2022, from https://www.researchgate.net/publication/336472403_A_Fast_Parallel_Alpha-Beta_Algorithm_for_Tic_TAC_Toe_Game

[5] J. Nishino and T. Nishino, "Parallel Monte Carlo Search for Imperfect Information Game Daihinmin," 2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming, 2012, pp. 3-6, doi: 10.1109/PAAP.2012.8.

[6] Silver, D., & Veness, J. (1970, January 1). *Monte-Carlo Planning in large POMDPs*. Advances in Neural Information Processing Systems. Retrieved May 15, 2022, from https://papers.nips.cc/paper/2010/hash/edfbe1afcf9246bb0d40eb4d8027d90f-Abstract.html

[7] *Parallel Monte-Carlo Tree Search - Maastricht University*. (n.d.). Retrieved May 15, 2022, from https://dke.maastrichtuniversity.nl/m.winands/documents/multithreadedMCTS2.pdf