Masaryk University
Faculty of Informatics

Reinforcement Learning for the Game of Battleship

Bachelor's Thesis

Tomáš Kancko

Brno, Spring 2020

Masaryk University
Faculty of Informatics

Reinforcement Learning for the Game of Battleship

Bachelor's Thesis

Tomáš Kancko

Brno, Spring 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Kancko

Advisor: RNDr. Petr Novotný, Ph.D.

Acknowledgements

I would like to thank my advisor RNDr. Petr Novotný, Ph.D., for the patient guidance, encouragement, feedbacks, and the topic of this thesis.

Abstract

The thesis explores the usability of general heuristic search algorithms in the Battleship domain. The first part of the thesis is an overview of the Battleship game. The second part analyses different designs, patterns and implementations of several heuristics and compares them. The third part is an overview of the MCTS algorithm and its parameters. The thesis also contains an outline of the formulation of the (PO)MDP model based on the battleship game, as well as a brief outline of existing approaches towards the Battleship game. The last part of the thesis introduces and evaluates all methods including the domain-dependant MCTS heuristics.

Keywords

Battleship, Monte Carlo tree search, Markov decision process, heuristics, neural network, NEAT

Contents

Introduction

In some games, we are able to calculate every move to be perfect. However, some of them have too many possible moves to do so. That is when Monte Carlo tree search(MCTS) comes handy. It is a method to forecast the most promising path that should be taken by the policy to reach the solution.

This thesis uses the MCTS to optimize and calculate the best move for the Battleship game's current state. The strategy game of two players taking their turns, shooting at the enemy board, until they successfully uncover all their hidden ships. The first to uncover all of them win the game. Our main goal will be to minimize the number of moves needed to uncover all the ships completely.

In the first part, the thesis focuses on creating a grid-like board with randomly spread ships. Then we establish a general number of moves needed for a player that decides based on choosing completely random moves.

In the second part, we create a neural network without any heuristics in order to decrease the number of moves needed as much as possible. We also create a few rule-based solving heuristics and collect data from them to compare later with MCTS solver. Moreover, we discuss why solutions based on rewarding every move may fail and how MCTS improves them. Finally, we decide which heuristics work fast and efficiently on a small board and therefore are favorable candidates for the original, larger board.

Finally, we implement our own MCTS heuristic, investigate the impact of changing its parameters, and compare the results with previous rule-based algorithms. We also outline a partially observable Markov decision process (POMDP) needed for solving this game.

Lastly, we enlarge the board to the original size and compare algorithms once again on a real problem, rather than the prototype one.

1 Battleship

1.1 Introduction to Battleship game

Battleship, also known as a **Sea Battle**, is a strategy type guessing game for two players. Both players have their own grid-like board and a predefined number of ships with various lengths. At the start of the game, each of them distributes all the ships across the board, concealing them from the other player. The tiles that do not contain a part of the ship are considered the ocean. There are a few rules about ships placement that need to be adhered:

- **1.** – The ships can be placed either vertically or horizontally, however not diagonally.
- **2.** – The ships can not overlap.
- **3.** – There also needs to be at least one free space between any two ships. In other words, ships can not touch vertically, horizontally, or diagonally.
- **4.** – The ships can not move during the game.

After each player being satisfied with their layout, the game starts. The players take their turns, guessing a single tile in each round. The other player is obligated to inform the guessing player, whether the chosen tile contains a part of the ship or the ocean. If all of a player's ships have been sunk, the game is over, and their opponent wins. If both players lost all their ships in a single round, the game results in a draw.

1.2 Board

The board in our battleship game consists of a two-dimensional matrix of tiles. The original game is played on the 10×10 board with five ships located on the board. The sizes of these ships are 5, 4, 3, 3, 2. Since the players can not move their ships (thus they can not influence their enemy's selection), we can reduce the game from two players

into trying to minimize the number of moves needed to uncover all the ships of one player. In that case, we can set the end of the game to all ships being uncovered or every tile being opened. For the experimental and testing purposes, we will use a 5×5 grid with only two ships of length 3 and 2 blocks.

1.2.1 Structure of the board

To represent the battleship board, we use three types of tiles:

» *Ocean* - A tile that represents the ocean; hitting this type of tile results in a miss. Illustrated by space.

» *Boat* - A tile that represents a part of the ship; hitting this type of tile results in a hit. Illustrated by an asterisk.

» *Unknown* - A tile that represents an uncovered tile with no deeper information. Illustrated by a question mark.

1.3 Actions

If we play on the 5x5 board with no tiles opened yet, obviously, there are 25 moves that can be performed, each tile representing a one. However, before actually making a move, we can observe two types of outcomes from it. Either we miss or hit the ship. That is the reason the number of actions is twice the moves available.

$$moves * 2 = 50$$

Therefore, we have 50 possible actions to perform. With each move being made, the number of actions reduces by two.

1.4 States

There is a finite number of states the board can get into. Considering we do not know anything about the board unless we provide a move, at the very beginning, every layout of ships has the same probability. As the game goes on and actions are being taken, some become impossible or less probable.

1.5 Reward

As there is no reasonable reward for every move (even missing might be the right move, since there is usually no 100% certainty of hitting), we reward at the end of the game. Ending the game with the least possible moves is rewarded with 1.0 and is considered as a win. The maximum number of acceptable moves will be set as $allTiles - shipTiles$. Therefore, there is $allTiles - 2 * shipTiles$ spare moves. For every extra (spare) move, compared to the least ones needed, the reward is uniformly decreasing. When all the spare moves or more were needed to end the game, the reward is 0.0 and is considered a loss. More specifically, if we play on the 5×5 board with five blocks of the ship, there are 15 spare moves, with five being the least to complete the game.

$$1 - (m - s) * \frac{1}{e} \tag{1.1}$$

Where:

m	stands for the number of moves that were needed
s	stands for the number of ship tiles
e	stands for the total number of the extra (spare) moves

1.6 Heuristics and solving approaches

We will solve the problem of reducing the moves needed to uncover the board entirely with many approaches. The best way would be to set some basic number of moves with a straightforward algorithm and improve.

- Random player – The first algorithm will be a player deciding randomly. Making moves without any heuristic is the starting point of every other algorithm; we will be using. Therefore, we can observe the improvement of other algorithms compared to this one.

- NEAT neural newtork – Secondly, we will implement a neural network, which is closely related to the random player. Every neural network starts with a random structure. That is why the improvement will be readily observable compared to the random player. Training the neural network is a complicated and tedious process. We want to determine on a smaller board, whether the neural network has a potential.
- Human-like player – Then, we will create an algorithm, whose decisions will be alike with a human.
- Even player – Enhance the Human-like player a little bit with essential knowledgeable accessions. We will definitely aim to outperform a human with MCTS.
- Heat map player – Furthermore, we want to create a precise algorithm, which will choose the perfect move in the current moment, to see how close we can get with the MCTS.
- MCTS – Lastly, we implement the MCTS algorithm with a measurement of its speed and efficiency relative to others.

2 Implementation of various heuristics

2.1 Random player and Neural network

For the implementation of these two heuristics, we will use a Java-based programming language software called Processing. Processing's main advantage is the automatic connectivity of different classes, graphical interface, and two Processing methods useful with the training of the neural network. The neural network consists of a set of inputs, a middle layer, and a set of outputs. The inputs are information about the current state. We need to set up an evaluation of all three types of tile, which will be the inputs. The outputs consist of every tile, getting assigned a value. The tile with the highest value is usually the one we are aiming for. The middle layer is the most important one. It usually comprises of many nodes with plenty of connections, changing its weights randomly with each generation. The weight of the connection tells us in what manner does the value change, transiting through it. The main idea is that we can solve any problem by adjusting the weights of connections between nodes, given a sufficient number of nodes. The real challenge is to find the correct number and efficient rewarding to do so in a reasonable time. To do so, we will use a particular type of neural network called **NEAT**.

2.1.1 Processing 3 and GUI

Processing is a flexible software based on Java programming language oriented to visualizing the data in 2D, 3D, PDF, or SVG output. It consists of more than 100 libraries that extend the core software to do so. Considering neural networks usually contain many classes for the representation of its parts, we will greatly benefit from Processings automatic include of the classes system. [1]

For this thesis's purpose, we will only use a fragment of its functionality (lines and squares to represent the board, X and Y coordinates of the mouse for user interface and movement). With that, we create a graphical user interface to be able to observe what is happening on the board quickly. Another convenience comprise in its `draw()` and `setup()` functions.

- `setup()` – This function is called at the start of every run of the program. It is usually used to set up the environment, e.g., the window's size, basic rendering of the board, and set up the variables. We will use this function to create a new generation of the neural network every time; the last has served its purpose.

- `draw()` – This function is responsible for rendering the window. More precisely, at the end of this function, the result is shown to the user. Unless otherwise stated, draw() is called each frame in an infinite loop. We will use this function to make the calculations for each of the current generation agents until all of them finished its task.

2.1.2 Basic implementation

In the Processing implementation, we will set some basic number of moves, which then we will try to improve by neural network and elaborate if and how they succeeded. For better and faster results, we will solve a reduced version of a problem that consists of a 5×5 board with two ships of length 3 and 2 randomly placed in it. The implementation is optimized in a way that if the reduced problem has been solved successfully, only a slight change in parameters given is needed to try the original game. There are three options for playing:

» *GUI* – This option enables the user to try the game and evaluate some basic number of moves a human requires to finish it. After compiling the program, a window with the board pops up and is controlled by a user's mouse. It can be turn on by setting the `isPlaying` variable to 0.

In the (2.1) picture, we can see that white squares represent uncovered tiles; Blue represent the ocean and Red ones ships. The game continues in an infinite loop; the user can stop playing by closing the board window.

» *Random player* – This option enables the user to spectate a player that decides entirely randomly. This player's purpose is to set some essential number of moves needed to finish the game. After each game, a final number of moves is being displayed. After each one hundred games, the average number of moves is also

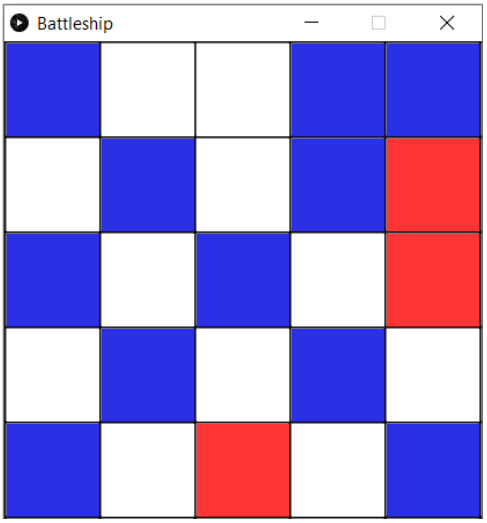

Figure 2.1: A sample of a game with GUI implemented in Processing

displayed. It can be turned on by setting the `isPlaying` variable to 1.

» *Neural network* – This option enables the NEAT neural network to solve this problem. It consists of 100 agents learning and mutating with each other. As with the Random player, the average number of moves is being displayed after each one hundred generations. It can be turned on by setting the `isPlaying` variable to 2.

2.2 Neural Networks for Battleship

Evolving neural networks through Augmenting Topologies (NEAT) is a relatively new approach to training the neural networks. The inspiration to use NEAT comes from a programmer under a pseudonym of Code-Bullet. The implementation of the NEAT neural network is slightly adjusted NEAT template made by him [2]. Let us analyze the advantages of NEAT compared to a regular neural network.

In this type of problem, where we do not see the full board, it is easy to identify the inputs – every tile on the board will be one. Some critical information like if we hit or sunk the ship in the last

move might be too. The same goes for the outputs – every tile gets assigned a number, and we simply try to hit the one with the highest value. However, it is too complicated to specify how many neurons are needed for the middle (hidden) layer to obtain the requested results.

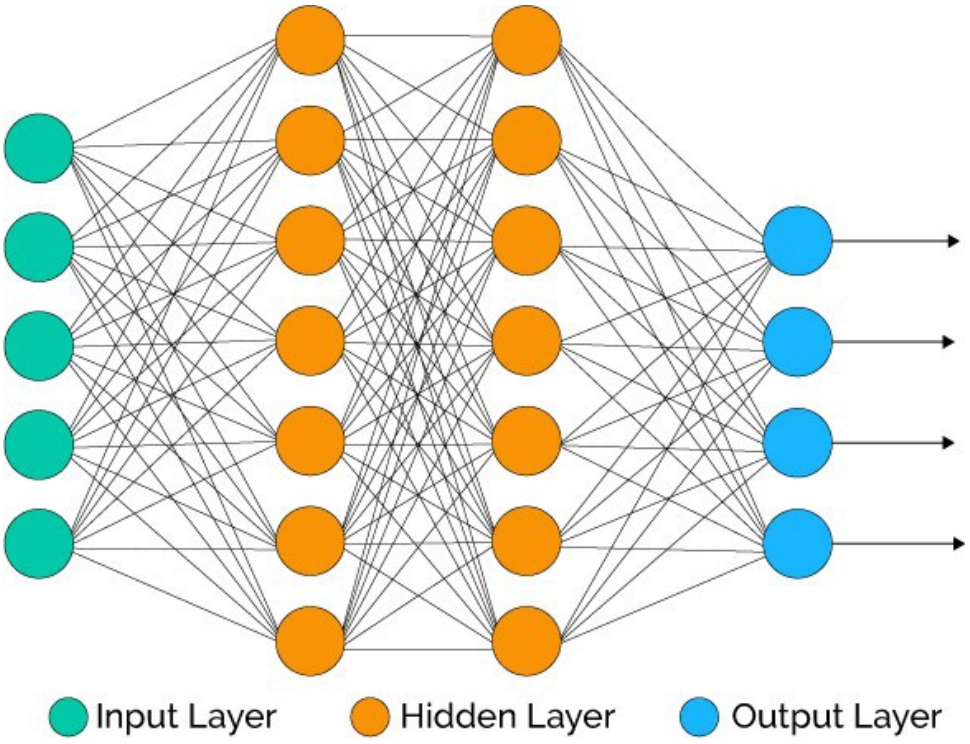

Figure 2.2: Basic neural network, adopted from [3]

Considering the computational power which might be needed and plenty of time it might consume to actually bring some results, instead of trying many variations of regular neural networks with a fixed size of the middle layer, we use an alternative approach. A neural network that starts with a small number of nodes in the middle layer, which slowly grows and connects randomly. Especially in an environment that does not offer computers with specialized graphic cards for maximizing the calculation power, NEAT was the best solution available.

2.2.1 Foundations

The training of the NEAT neural network starts with an elementary topology. At the very beginning, it consists of only input, output, and bias (usually one node) nodes. The reason is to ensure that when a solution is found, there is no additional work needed to minimize our final product. Instead, we found the lowest-dimensional weight space possible throughout all generations. Not only that we find a possibly best solution; moreover, we also gain in terms of performance [4].

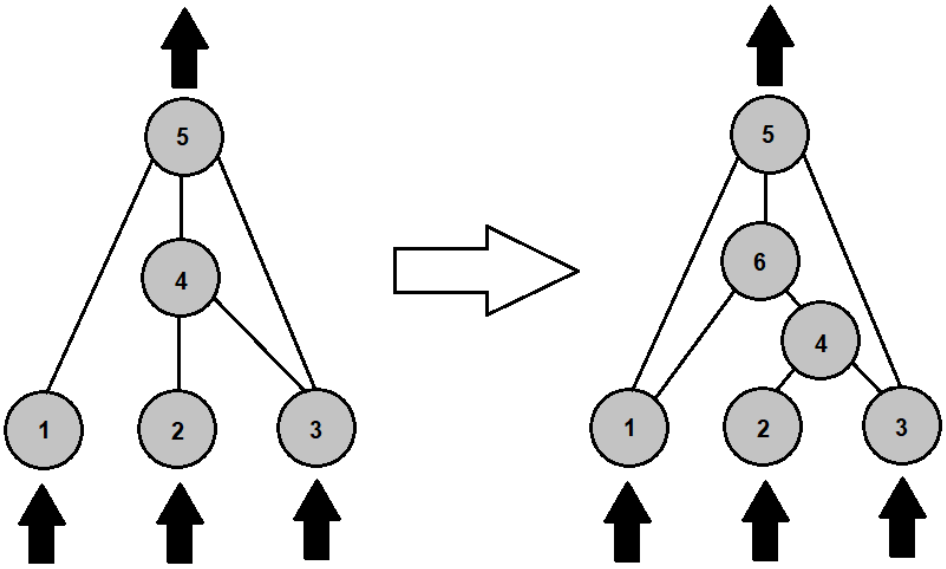

Figure 2.3: An example of adding a single node into NEAT

The encoding of our neural network is pretty simple as well. The population consists of an array of agents (players). Every one of them has its own brain called **Genome**, which represents the agent's neural network. The Genome carries two important attributes:

» *Nodes* – Which is basically an array of all nodes, the neural network consists of.

» *Connections* – The connections between the nodes. A single connection comprises of a few essential attributes:

– `Node from` – The node the connection comes from.

- `Node to` – The node connection goes to.
- `float weight` - The weight of a connection describes the way in which the value changes when propagating from one node to another.
- `bool enabled` – Considering NEAT starts with a small number of nodes and gradually adds them, if a new node is placed between 2 existing, the old connection gets disabled, and two new connections are created. On the contrary, if we are deleting a particular node, we need to enable a connection that has been there previously.
- `int InnovationNumber` – In the process of adding new nodes, some Genomes will be more successful than the others, that is when mutating from the most promising ones comes in. Usually, a new node gets assigned the next free not already used number. Nevertheless, in the process of mutating two Genomes together, which represent parents, innovation number is used as a historical marker, from which Genome the origin comes from.

2.2.2 Activation function and representation of layers

The neural network's input layer consists of a one-dimensional array of nodes for each tile with one extra node for a consecutive shot. Each node representing a tile gets assigned a value based on its type.

» *Ocean* – A tile representing the ocean gets assigned -1.

» *Ship* – A tile representing the ship gets assigned 1.

» *Unknown* – A tile that has not been revealed yet gets assigned 0,01. The reason behind such a decision is a form of a feed-forward algorithm. In the process of propagating values across the neural network, a value of 0 would automatically mean zero addition of this node, no matter the weights or the output of activation function. This way, it can still influence a little bit.

The last node gets assigned 1 if we successfully hit in the last round, -1 if we did not. Considering we would like to see each tile's probability

in the one-dimensional output array, we will be using the sigmoid activation function. The main reason is that it exists between $(0;1)$, which is ideal since we want to predict the probability as an output.

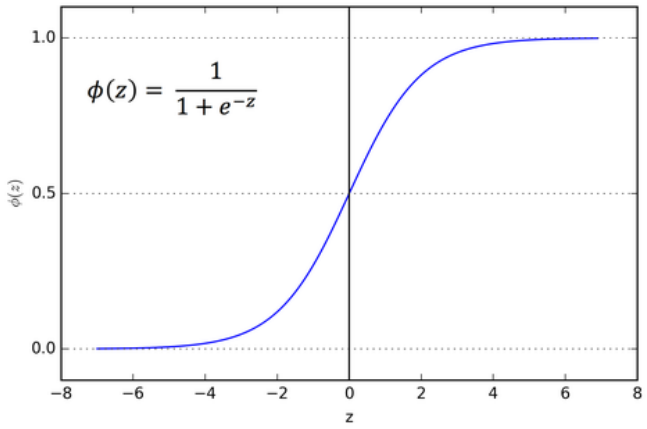

Figure 2.4: Sigmoid activation function, adopted from [5]

2.2.3 The process of learning

The process of learning contains the most fundamental part of every neural network – changing its weights, but also many nontrivial ones.

- There is an 80% chance for each connection to change its weight in between generations. In 90% cases, the weight will be changed slightly. However, there is a 10% chance for the weight to be chosen as a completely random one.

- Furthermore, we will utilize the fact; we are working with a unique type of neural network and add a 5% chance to add a connection between two nodes that are not connected yet.

- Moreover, there will be a 1% chance of adding a new node into the existing neural network. The addition works in a manner of picking a random existing connection between two nodes, disabling it. The new node is added between the two disconnected nodes, replacing the connection between them.

- Lastly, if two agents have similar enough neural networks, it is beneficial to make a crossover of their genes (connections), while keeping the number of their nodes. The crossover process is simple: every time we have to copy a gene that both "parents" possess, we randomly take it from one of them. If only one parent has such a gene, we take it with a 25% chance; otherwise, this gene will remain disabled (no connection between the two nodes).

- On the other hand, we also need to remove the neural network that does not function well. At the end of each generation, based on each agent's fitness, the bottom half is removed. We replace them with mutations of the more favorable ones. The only exception is neural networks that are not similar to any other, as described in the Speciation part.

2.2.4 Speciation

There has been a fascinating observation that when we add a new connection with random weight before any optimization, it often leads to lowering the neural network's performance. The same applies to just blindly mutating two neural networks together. There is a possibility they have different approaches and therefore have different characteristics. Although they might both work well alone, their combination is inconvenient in most cases. This is a significant problem in NEAT, considering we do not want to sustain Genomes that does not function well. Genomes that appear in the bottom half when sorted by their fitness, we usually cull completely. Henceforward, we need to protect new structures and allow them to optimize themselves before we eliminate them from the population entirely [6].

The process to solve this problem is called **Speciation**. To put it simply, we just split the population into several species based on the similarity of their topologies and differentiation of the weight between their matching genes (connections). Considering every agent involves the same number of nodes, Genomes can only differ in their connections. The compatibility formula [7]:

$$\frac{excessAndDisjoint}{largeGenomeNormaliser} + 0.5 * averageWeightDiff$$

- *excessAndDisjoint* – This variable is calculated as the number of excess and disjoint genes between the two compared Genomes. Basically, it is just a sum of genes that do not match (connection exist in the first Genome; however, it does not in the second).

- *largeGenomeNormaliser* – This variable works as a normalizer when we take into account large genomes. Predictably, more connections result in a higher chance of some genes not matching, but still, they might be similar enough to be considered the same species. This variable is set to be:

  $$Connections - 20$$

  However, if the result is less than one, this variable is set to 1.

- *averageWeightDiff* - This variable is calculated as an average weight difference between matching genes (connections) of the two compared Genomes. The formula for calculating the sum is the following [2]:

  $$\sum_{i=1}^{n}(|y|) = \begin{cases} genes[i].w - genes2[i].w & \text{if } genes[i] = genes2[i], \\ 0 & \text{otherwise.} \end{cases}$$

  Connections are being compared by their `innovationNumber`, which tells us the origin of it. Once again, we took an example from nature. People are also considered the same blood if they share their ancestors. The same principle is applied here. Not only the connection needs to exist. Moreover, we want to ensure its origin comes from the same Species. Then, the absolute difference in its weight is calculated. Every time the genes are equal, we increment an individual variable to store the final number of them being equal. The result is a division by the sum and the

incremented variable. This way, we know an average difference between a **single matching** gene.

Finally, the result of our compatibility formula needs to be compared to a preset threshold. For this thesis's purposes, the Genome is considered the same if a result is less or equal to 3.

2.2.5 Possible setbacks and chosen rewarding system

It needs to be stated that problems like Battleship, where we need to provide moves to get information, are not ideal for neural networks. What is the biggest problem in training them?

Neural networks, in general, are mostly improving based on step rewards. If we can specify the right moves and reward them and prevent wrong moves and decrease the score, the neural network can adjust its weights easier. Unfortunately, rewarding in such types of games is much harder. Let us say we get into a situation like this:

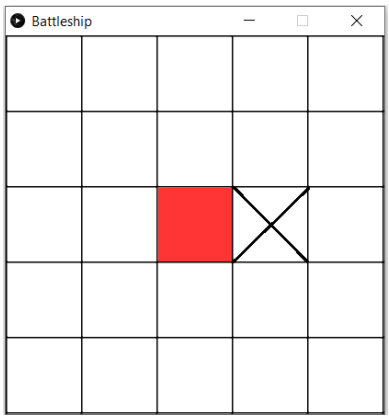

Figure 2.5: An illustration of rewarding problem

After successfully hitting a ship, we aim to the right of it (illustrated by X). However, we miss and find out there is an ocean. We cannot really reward this type of play, but on the other hand, we made the right move. We could implement such heuristics to reward trying the tiles around the ship, even tiles, prevent starting with the side tiles, but we would completely vanish the neural network's point to find its own heuristic. The most reasonable rewarding is for the total moves needed at the end of the game. Nevertheless, if we want to reward in

the middle of the game, there are two specific types of rewarding to choose from:

- *Early hits* – If we were to reward just by hitting, each game would end up with the same reward. A little adjusting needs to be done. The move scoring an early hit needs to have a higher value than the other. The reason this thesis do not incline to such an approach is the irregularity. Hitting in the first 2-3 moves is pure luck, and no heuristic can ensure it with 100% certainty. Therefore we chose the second approach.

- *Consecutive hits* – The goal is straightforward, we want to encourage trying the adjacent tiles after hitting a ship, considering they obtain higher chances of involving the part of the ship. Rewarding consecutive hits might help with it. Even though there would usually be 3-4 adjacent tiles to try, doing it just in 25% situations might work. There can still appear situations like this, but they are not as frequent.

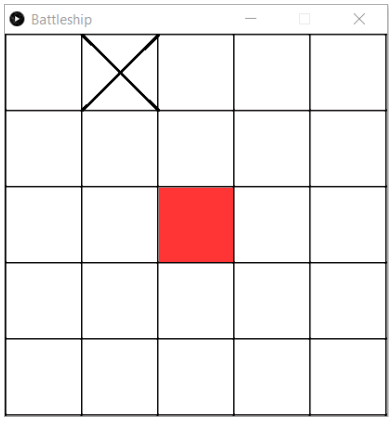

Figure 2.6: Another illustration of rewarding problem

If we successfully hit the enemy ship and then aim somewhere and luckily hit another ship (illustrated by X) or even the same ship but further away, again, we would end up rewarding pure luck instead of certain plays.

2.3 Human-like player

As it is the primary goal of most innovations to outperform a human, we will try to create a Human-like algorithm to measure it with achieved results, but also later with MCTS. While spectating several humans play the game, intending to minimize the number of moves needed, all of them shared a common technique. In the first place, they aimed to find a part of the ship by randomly guessing a tile. After they succeeded and hit a ship, they started trying the adjacent tiles to reveal its parts. This process repeatedly continued until all ships got revealed, and the game ended.

2.3.1 Implementation

In the implementation of a Human-like player, we need to define a **State** and a set of **Modes**. The Player's behavior will be depending on the State and, therefore, the Mode he is in.

Let us start with the explanation of modes. At each point of the game, there will be performed exactly one of them:

- *Hunt* – This action represents a human randomly guessing the tiles until they find a part of the ship. There is no deeper heuristic behind, only a random one from unopened tiles is chosen. The player stays in this mode until a prerequisite of hitting is accomplished. After successfully hitting, the player gets into the **Target** mode and all the unopened adjacent tiles are added as potential targets. After the number of targets gets back to 0, the player gets into the **Hunt** mode again.

- *Target* – Target mode is there to represent a human trying to find out the orientation of the ship - either vertical or horizontal. Each turn, a tile is popped out of a stack of targets, until another hit is executed. After that, we know with certainty the orientation of the ship. Consequently, the stack of targets is being emptied, and two new targets as illustrated on (2.7) picture are added as potential parts of the ship. Moreover, the mode is changed to **Destroy**.

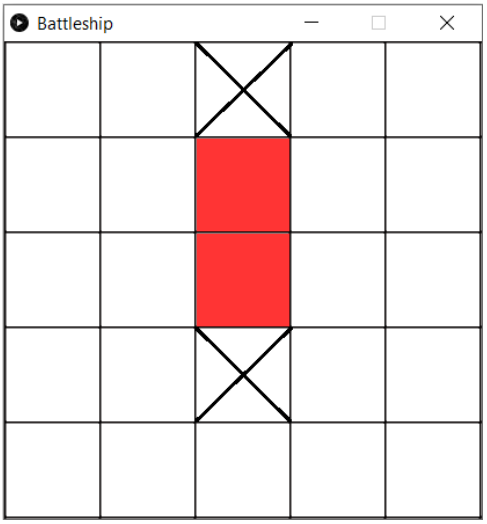

Figure 2.7: An illustration of only possible targets(X) after the Target mode

- *Destroy* – A Destroy mode secures the clearance of the ship. Until an ocean or the edge of the board is being found, the process of uncovering the sides is being held. After the ship being completely uncovered with 100% certainty, the mode is changed back to **Hunt**, and the process repeats.

State s - consists of basic information:

- *s.board* – The current content of the board.
- *s.moves* – The array of coordinates of every unopened tile. Considering most of them are chosen randomly, storage of the unopened tiles speeds up the process of choosing.
- *s.mode* – The mode a player is currently in, one of the three enum values.
- *s.targets* – A stack of possible targets.
- *s.lastHit* – The coordinates of the tile that led to the Target mode. Their usage is to calculate the next possible targets.

2.4 Even player

Even player's behavior will be principally the same as the **Human-like player's** with one major exception. In case he has no targets in his stack, instead of going for a completely random tile, he would aim only for the even (every second) ones, as we can see in (2.8) image.

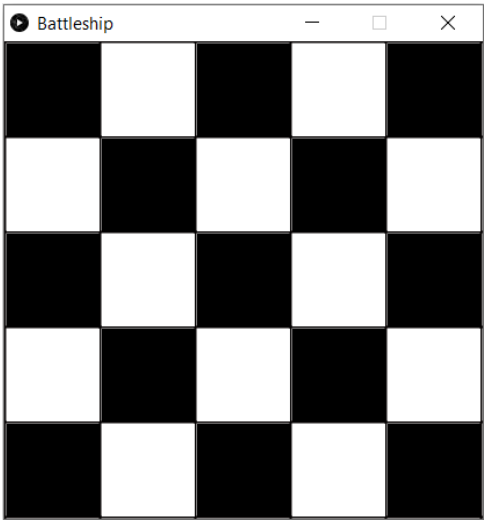

Figure 2.8: An illustration of Even player tiles to aim, represented by black color

The idea behind this strategy consists of ships being oriented vertically and horizontally, not diagonally. Therefore, every ship with the length equal to or higher than two will contain at least one of these tiles. Accordingly, if we uncover all the even tiles, we are bound to find a part of every ship on the board.

2.5 Heat-map player

All the previous implementations had a common future in the form of random, uncertain, or unpredictable moves. To compare their speed and accuracy, we will implement a mathematically based method, calculating every move. The method is called Heat-map biased choosing. It consists of calculating the probability the tile has to contain a part of the ship. We will basically compute every possible layout of the ships

from the current situation and number up how many of them contain a part of the ship at a particular tile. Then, we transition into a coloring of the 2D matrix with different color intensities, representing a higher chance (dark color) or lower chance (light color) to contain a part of the ship. This solution will always execute perfect moves, based on the probabilities, for the current state. The only solution that might work better and consider more paths would be a tree of all possible options, which is unfortunately too hard to compute as can be seen in (2.1):

Table 2.1: A comparison of total solutions of all possible branches (Tree) and all possible final states (Heat-map) on a 5×5 board

Player	Size	Ship tiles	Moves	Total solutions
Tree	5x5	5	5	6,375,600
Tree	5x5	5	6	637,560,000
Tree	5x5	5	7	36,340,920,000
Tree	5x5	5	8	1,526,318,640,000
Tree	5x5	5	9	51,894,833,760,000
Heat-map	5x5	5	5-25	524
Heat-map	9x9	17	5-81	482,547,096

When calculating for the Tree structure on a 5x5 board, the total number of branches for a certain number of moves that finish the game is:

$$\frac{25!}{(25-m+1)!} * \binom{m-1}{m-s}$$

Where:

m	stands for the number of moves
s	stands for the number of ship tiles

For example, to calculate the branches for a 7 move game with 5 tiles of the ship being hit, we need to choose 7 tiles from 25, which got

opened. Considering their order also matters ([0,0] and then [0,1] is a different sequence than [0,1] then [0,0]) – resulting in $25 * 24 * 23 * 22 * 21 * 20 * 19$. After that, knowing there are obligated to appear two misses, we need to choose two tiles from the first six (the last one can not result in a miss, because the game would not end). Considering their order does not matter this time (miss in the first and then the second move is the same as second and then choosing first) – resulting in $\binom{6}{2}$.

Even if we successfully encoded each sequence into two bytes, considering we are calculating for the 5×5 board and nine moves needed, we would still end up using around **96661.66 Gigabytes** in case of a Tree, which exceeds an average memory size by quite a huge margin. On the contrast, there are only **524 possibilities** of final boards. Even if every tile has been encoded into a true/false value (1 byte), it would consume less than **13 Kilobytes** in case of Heat-map player. Unfortunately, even the Heat-map method would fizzle on the original 10×10 game with 17 tiles of ships, considering even the reduced version of 9×9 has more than **480 million** possible final states. The encoding of it would consume more than **11 Gigabytes**. Therefore, the limit for a Heat-map player usage is 8×8 board.

2.5.1 Implementation

In the implementation of Heat-map player, we only need to define a **State s**, which will represent a current state of the problem:

- *s.board* – The current content of the board.
- *s.moves* – The coordinates of every unopened tile. We need to know for which tiles we need to calculate the chances.
- *s.possibleStates* – A vector of matrices that represent possible layouts. Every matrix consists of bit-wise values representing whether the current tile contains a ship (true value), or it is an ocean (false value). These need to be generated at the start. Then, all possible states that do not contain the last move need to be deleted after every move. In the end, when there is only one state left, we are certain, it is our final state. The speed improvement in comparison to the Tree structure of all possible moves lies in

reducing all possible permutations of a single state into which we can get by many paths. The Tree basically reorders the moves, but usually gets into the same final state. For example, a game with five blocks of the ship, finished in 10 moves representing only one possible state would occur in the Tree structure of all possible moves:

$$5 * 9! = 1,814,400$$

Where:

5	stands for a last move that need to be a hit
9!	stands for all possible orders of the remaining moves

- *s.lastHitShip* - We need to store if we successfully hit a ship, considering we are deleting states that do not contain our selection.

2.5.2 Heat-maps

In the Heat map player implementation, we can easily print the heat maps from a current state by calling – `void printHeatMap (std::ostream output)` method. This method prints to the output stream the board as a 2D matrix of tiles, each containing a number within the $[0; 1]$ interval, representing the chance of a particular tile to contain a ship part.

Let us look at the heat-maps and the percentage likelihood of a tile to contain a part of the ship. We are going to be working with numbers in $[0; 1]$ interval (for example, 0.65 represents a 65% chance of hitting). Our selection is represented as (X); our previous selections are displayed as (-). Uncovered parts of the ship are represented with the red color to be able to differentiate them.

Figure 2.9: Scale used for Heat-maps

Considering that more than one tile can store the maximal value, a random one is opened in that situation. If a particular tile has a chance of more than 70% to contain a part of the ship, we will display it with the darkest color. The reason is there is no possible layout, so more than one tile is going to have such chances. It also enables a user to observe minor changes, considering the contrast is higher on a smaller scale.

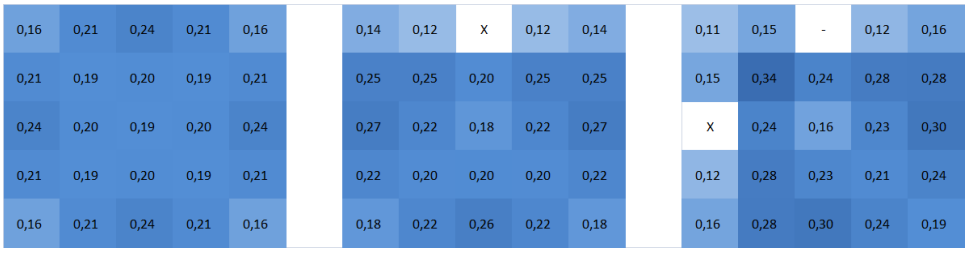

Figure 2.10: Heat map of 1.-3. moves

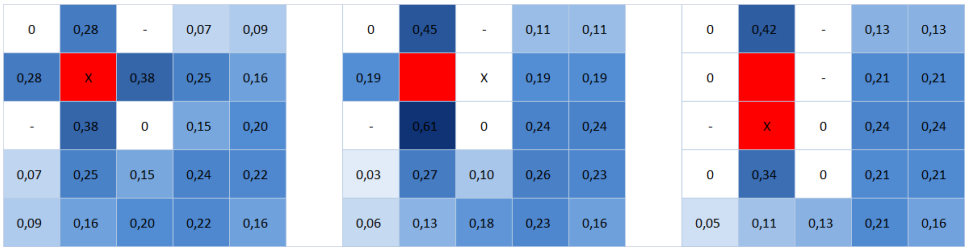

Figure 2.11: Heat map of 4.-6. moves

Figure 2.12: Heat map of 7.-9. moves

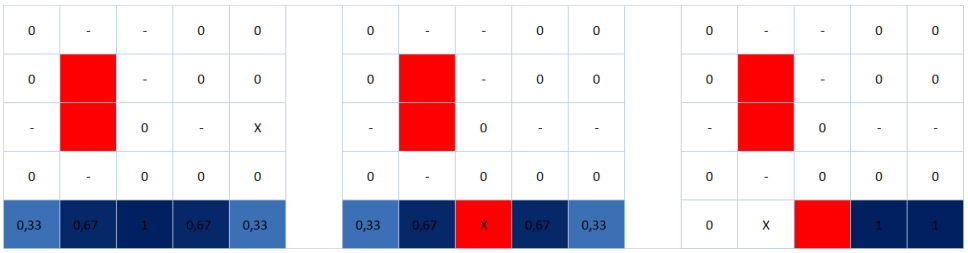

Figure 2.13: Heat map of 10.-12. moves

After the 12th move, we are certain and finish the game quickly in 2 more moves.

3 MCTS

3.1 Introduction

Monte Carlo Tree Search is a search method for finding the optimal decisions in a given domain. It combines the precision of a tree search with the generality of sampling. However, the huge performance gain compared to just blindly making a tree of all the possible consequences, this method continues and expands mostly the most promising nodes. As shown in the image below, the tree usually looks asymmetric due to the tree policy, not electing choices that do not seem convenient. [8]

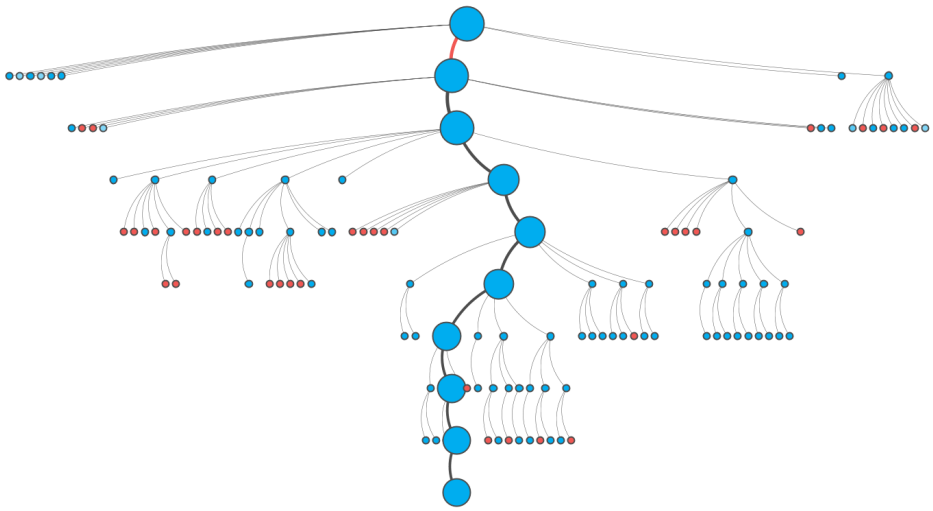

Figure 3.1: A sample of MCTS tree structure, adopted from [9]

3.2 Overview

Before we get to the MCTS overview, the state and nodes' representation needs to be defined.

State s – The class for the representation of current problem's state contains 3 fundamental attributes:

- *s.board* – The current content of the board.

- *s.moves* – A vector of coordinates with additional information representing hitting/missing. Storing the possible moves for the current state brings slight speed improvement at the cost of minimal space requirement. All the possible moves are then available to the nodes and do not need to be generated several times. Another convenient part is that generating possible moves at the start is an easy task. It consists of iterating through all the tiles and saving them twice into a vector. Moreover, after making a specific move, the vector needs to be adjusted only with a simple deletion of 2 moves(first for hitting, second for missing with the same coordinates).

- *s.engine* – An std::mt19937_64 engine we will be using for generating random numbers.

Node n – The class that represents a single node in our tree structure has 7 fundamental attributes:

- *n.moves* – The vector of untried moves, that still can be tried from this node. At the creation of the node, moves are being copied from the state into the node. Then, they are being adjusted as some of them are being tried. If the node has reached the maximum capacity of children (all possible moves from this node has been tried), we can easily detect it by this vector being empty.

- *n.payoff* – A float number that sums the payoff that this node, but usually its ancestors have acquired. Considering we are mapping the moves needed to complete the game into an $[0; 1]$ interval, implying how much of a "win" we acquired in the particular game, it is not bound to be an integer value.

- *n.score* - An estimated value of the action this node represents.

- *n.visits* - The number of visits of this node.

- *n.move* - The move that led to the creation of this node. In the process of sorting the children by their fitness (n.score), we can easily know which move they actually represent.

- *n.parent* – A pointer to the parent node. This attribute's only notable feature is to back-propagate results into ancestor nodes, considering the selection of them led into it.
- *n.children* – A vector of pointers to the children nodes, each representing the next move being done from the current node.

The MCTS model is conceptually simple. In every step, we are aiming to find the most urgent node of the current tree. The tree policy is built to balance considerations of exploration and exploitation. After the urgent move being done, simulation is being run from the selected node, and its result is updating the tree. In the beginning, this usually includes creating a child node representing the chosen move. After we reach a specific state, that can be described as final (no more moves can be made), we update the statistics of all the ancestor nodes until reaching our urgent node with the result we got. The selection of nodes during the simulation process is handled by the default policy, which in our (but also the majority) case is making uniform random moves. As the method of choosing random moves may sound uncertain, not calculating the chances at intermediate states, but only once at the final state stands behind the great performance improvement. Nevertheless, it requires a wise choice of the evaluation function and many iterations, so the right move shows itself being the best [8].

3.3 Markov Decision Processes

A Markov decision process (MDP) models sequential decision problems in fully observable environments. It consists of 4 main components that need to be defined:

- S – A set of states. Basically, all the possible positions we can get into.
- A – A set of actions. This component comprises of all the possible actions in a given state.
- $T(s, a, s')$ – A transition model that determines the probability of reaching state **s'** if action **a** is applied to state **s**.

- $R(s)$ – A reward function – which is called in the final state and evaluates the state in the $[0;1]$ interval, representing how much of a payoff we achieved from the final state. The payoff is adjusted according to the number of moves we needed to complete the game.

Unfortunately, this approach will only work with problems where we know the consequence of our actions. For example, when a piece is moved in a game of chess, we know exactly in which state we end up. However, when we guess a tile in the Battleship game, there is no information until we actually make that move. Therefore, our simulation needs a different approach - using Partially Observable Markov Decision Process (POMDP).

3.3.1 Partially Observable Markov Decision Processes

POMDP works pretty much as MDP, but we need to take into account that our decisions do not only have one outcome. That is why we need to specify also an observation model with the probabilities.

- $O(s, o)$ - An observation model that specifies the probability of perceiving observation **o** in state **s**. Considering, we want to work with a deterministic policy rather than a probability distribution over actions, we will extend the number of actions so that every move will have a missing but also hitting outcome.

3.4 Algorithm

In this part, we are going to elaborate in detail the four key steps of the MCTS algorithm, which are being repeatedly run until a certain *computational budget* – usually a time, memory, or an iteration count is reached. Then, the search process is halted, and the most promising node returned.

- *Selection* – The initial part of MCTS, we need to find the most urgent expandable node. A node is expandable if it satisfies two conditions:

  - **non-terminal state** – The node needs to represent a state, which is not marked as terminating, so it is not an end of the game when we choose it. Otherwise, we would not be able to expand this node anymore.
  - **expandable** – The node needs to have an unvisited child, which can be expanded. The unvisited children will have elevated values of their evaluation thanks to the **exploration term** in the choosing policy, which will be mentioned later. It is crucial in a manner of not omitting any possible solution. That is the reason why we are recursively applying a child selection until we actually find a node with an unexplored move (child). It can be easily imagined as traversing the tree, choosing the best moves at each state if we already know all of them. However, taking into account that nodes visited very few times can still involve the right solution (basically just applying Upper Confidence Bounds for Trees (UCT), which will be mentioned later). We repeat the selection phase until we find a node, where we do not know anything about a particular move, indicating this node can be expanded more.

- *Expansion* – If we reached an expandable node, we need to expand our tree with a new child of this node and continue the process. Considering there might be more than one path that we have not seen yet, we need to define a policy for choosing one of them. In our case, we will choose one of them randomly as we do not have any information about moves not done up to now, so they have the same evaluation by UCT.

- *Simulation* – After we created a new node in the tree in the expansion phase, we can finally simulate and collect some data. We continue choosing nodes until we reach the terminating state, adhering to the default policy, which in this thesis will be randomly choosing moves until we end in the final state. Two essential elements may be unclear and may need a more in-depth explanation:

  - **Moves leading to invalid states** – Two approaches can resolve this aspect. We can either analyze every move that

we randomly choose and actually perform it, only after validating, it leads to a possible state. However, we would end up reducing the speed of the simulation process and thus, the whole algorithm. We would have to choose the moves multiple times at each step of the simulation, and even though our tree would only contain valid moves, the decrement of speed would be enormous. That is why, in this thesis, we chose the second approach – making completely random moves without checking its validity. After the game is finished, we need to check if we got a correct layout. This will be the purpose of a **validation function**. Nevertheless, there would be a problem with invalidating every incorrect layout the same, as invalid. Let us say we randomly chose five consequent hits and ended in a state like this (3.2):

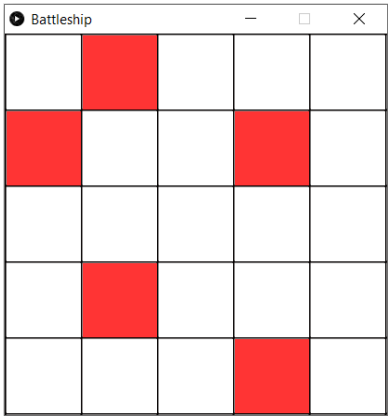

Figure 3.2: An illustration of invalid state

Although we achieved an end of the game in five moves, this path of simulation does not bring any logical results and does not tell us anything about the ships' position. If we look at another example of an invalid state (3.3):

Even though again, we ended up in an invalid state, we were able to predict a possible layout of 1 of the ships correctly. We need to take into account such discovery. Predicting the full layout of ships may have worked with smaller boards (5×5). However, if we try it with larger ones, it is relatively

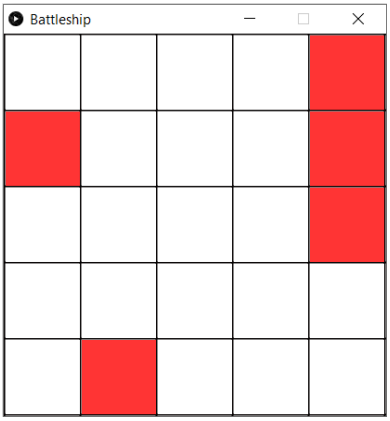

Figure 3.3: Second illustration of "less" invalid state

impossible to predict all ships' correct layout with no moves done yet. The algorithm would be stuck at choosing random moves as none of them would be promising due to the invalid states.

3.5 Validation function

We need to have a fast but effective function to know how many valid ships we guessed in the simulation process.

* **1.** The algorithm starts with the left upper corner and continues to the right, repeating at each row, until we find an unvisited part of the ship.
* **2.** We mark this part of the ship as visited and suppose it is a correct layout of the ship.
* **3.** After that we will look right and down, in order to find a continuation of it, until we know a full length of the ship, every tile being a ship marking visited.
* **4.** If the ship's length equals 1, we continue with seeking, as this does not represent a correct layout of the ship.
* **5.** Otherwise, we check that every tile around the ship is either an unopened tile or an ocean.

* **6.** Lastly, we only need to check there is a ship with found length remaining. If so, we erase it from the vector and record how many correct tiles the found ship consists of.

After calling this function, we know exactly how many tiles got marked as correct layouts of ships. Then, a little adjustment of the reward function needs to be done.

$$Reward = \frac{v}{s} * m \tag{3.1}$$

Where:

v	the number of valid tiles representing a ship (output of validation function)
s	the total number of ship tiles on the board
m	the moves mapped into an $[0;1]$ as stated in (1.1)

– **Choosing random moves** – By the default, if we do any activity randomly, there is never 100% certainty we get the expected outcome. The massive reduction of time needed for traversing the tree and making reasonable moves has its cost. So if someone were to ask a question if this method may fail, the answer would be yes. It could happen that in the best move for the given state, the simulation process will produce worse outcomes than it will be with other moves. However, the chances of this outcome are small. On the other hand, the huge impact of this uncertainty comes with the speed improvement. By general, if we do a step in the right direction, instead of going backward, we are bound to see better outcomes several more times if the number of iterations is high enough, which is the key to doing activities randomly.

- *Backpropagation* – This is the final step of the MCTS algorithm. We got into a state that can be marked as terminating. In this thesis,

the game ended if we hit all and therefore uncovered all the remaining ships or we uncovered all the tiles (in the simulation phase we are assuming which tile is an ocean/boat; therefore, we can end up with fewer ship tiles than there should correctly be). There can still remain covered tiles that represent water. We can map this state into a number in the $[0; 1]$ interval and update the payoff of the new child node that has been created, but also all its ancestors until we reach the initial one from which we are calculating.

3.6 Upper Confidence Bounds for Trees(UCT)

The goal of MCTS is to approximate the value of the actions that can be made in a current state. Obviously, it is most efficient to choose the action with the highest value. In an environment with many possible moves, the primary goal is to reduce the nodes' estimation error in the tree as soon as possible. There is no doubt that if we have enough time, we will get a relatively precise result (either a certain number the value is converging to or an oscillation). However, we need to spend as much time as possible, exploiting the promising nodes. To calculate the right balance between exploiting the most promising node and exploring the alternatives, which may later turn out to be superior, we have a very simple but efficient algorithm UCT [10]. It consists of an **exploitation term** and an **exploration term**. The first one – exploitation term – will always be in the interval of $[0; 1]$ and calculates the likelihood of a child to be the winning node. The second one – exploration term – stands behind the balancing part for not omitting other children, even though they might not seem promising yet.

$$UCT = \frac{w_j}{n_j} + \sqrt{\frac{2 * \ln n}{n_j}} \tag{3.2}$$

Where:

n	the number of times the current (parent) node has been visited
n_j	the number of times child j has been visited
w_j	the number of wins child j has achieved

The child with the highest value of (3.2) equation is chosen as a candidate for deeper exploitation in the current iteration. If more than one child acquires the highest value, a random one from them is chosen. It is generally understood that a value of $n_j = 0$ yields a UCT value of ∞ so that every previously unvisited child is assigned the highest possible value, to ensure every child is considered at least once before any of them is explored more in-depth. To avoid doing undefined calculations, e.g., dividing by 0, this thesis simply takes a random unvisited child until all of them have $1 \leq n_j$. Then, we calculate a child with the highest UCT value and continue with it.

The point of exploitation term is pretty straight forward. Let us explain the second term more in-depth as it may seem less obvious. As each node is being visited, the denominator (n_j) of the exploration term increases, which decreases the overall addition. Simply said, the more we explore one specific node, the less value we will be getting from it. On the other hand, if another child of the same parent node has been visited, the numerator (n) increases, and hence all the other sibling nodes will have its value elevated. The reason is to ensure; even unprofitable nodes have a non-zero chance to be chosen eventually (given a sufficient number of iterations). [8]

4 Results

4.1 Comparisons on a 5×5 prototype board

Until we get to the original problem situated on the 10×10 board, we want to determine which heuristics have a potential solving such a complicated task and what are the suitable parameters for them. Especially in the case of the neural network, training might take weeks (even in the games on smaller boards with full information like chess). The prototype board contains two ships of lengths 3 and 2, randomly hidden on the board.

4.1.1 Basic number of moves – Random player

First of all, we will set up a basic number of moves, without any heuristics at all. The average number of moves of the **Random player** will be the starting point, which then, we will improve. We will simulate 1000 games, average the number of moves after every 20 of them.

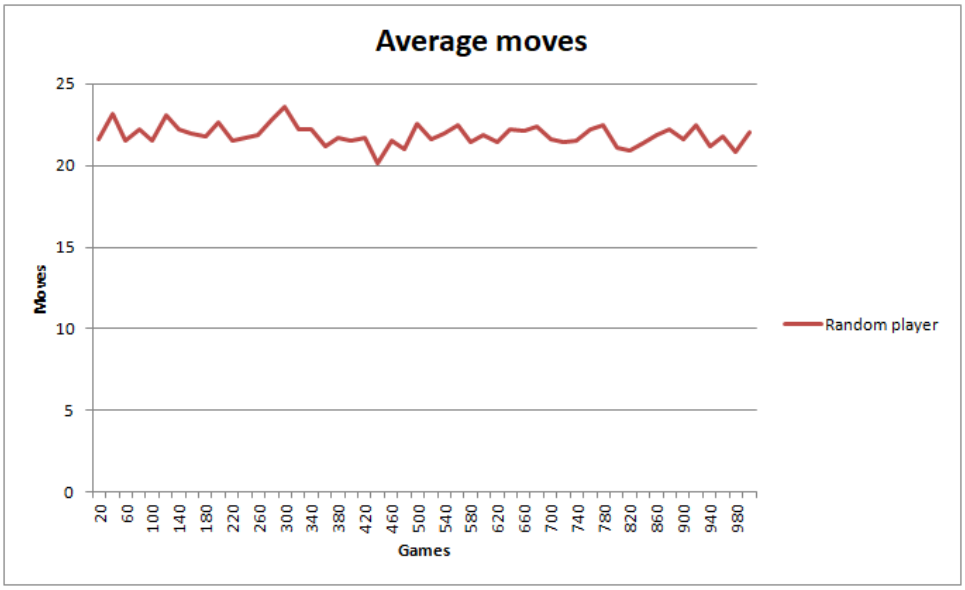

Figure 4.1: Average number of moves of the Random player

We can see in the (4.1) graph, that the average number of moves settled down to be around **21,8**. We will also take that number of moves as a starting point of the neural network, considering its weights are chosen randomly at the beginning.

4.1.2 NEAT neural network

The neural network aimed to find its own heuristic, adjusted by the number of moves needed to finish the game and consecutive hits. The objective was to improve the **21,8** move standard.

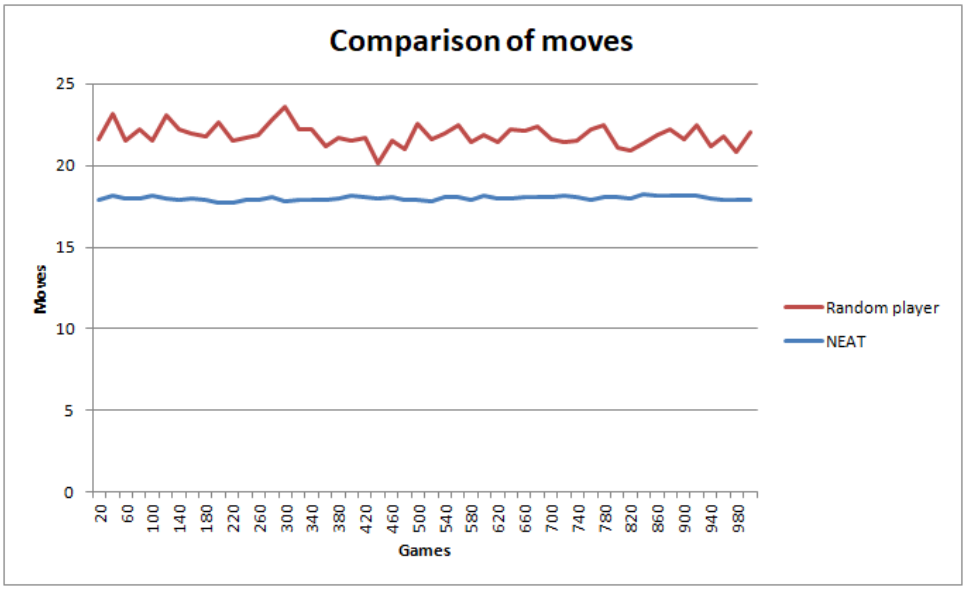

Figure 4.2: Average moves comparison

Despite all the setbacks, in the (4.2) graph, we can see that NEAT is able to stabilize average moves slightly below **18** in only a few generations as we expected using it. Unfortunately, after that, the improvement slows down, and no significant transition could be seen in hours of training. Let us dip into the trained neural network and analyze the reasons NEAT failed.

The neural network starts with a large number of species (40-60), basically meaning it tries many different approaches to solve the problem. However, it eliminates most of them in one or two generations and settles down to around 15. As the number of species goes down,

we can immediately see the reduction of moves, meaning only a few approaches are working. On the other hand, if the neural network tries to increase the species again, the number of moves raises.

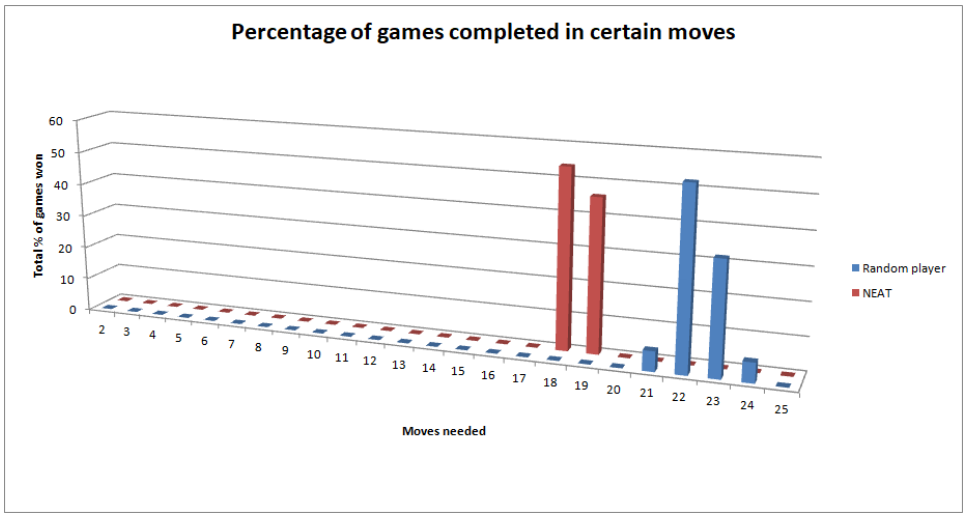

Figure 4.3: Percentage of games completed in a certain number of moves

There are being done many mutations, and the number of nodes continually grows, indicating a lot of them will be needed. The most significant progress of NEAT implementation that can be seen in (4.3) consists of its ability to finish games in an average of 17-19 moves with certainty. It seems the neural network quickly finds an algorithm to secure the number of moves under 19; however, the next improvement is much more complicated. There are a few possible problems causing that:

- *Complexity* – The complexity of solving efficiently the Battleship problem is higher than expected. NEAT is an improper candidate because it has to come through the whole process of adding nodes until we get to plausible results.
- *Speciation* – The speciation could be considered the main problem of NEAT in reducing the moves needed. Although it is an incredible improvement to other games, there are not many successful approaches in Battleship. However, the fundamental

principle of speciation is to find more of them. As the neural network continuously tries to add more of them, in most cases, it leads to deleting them again, wasting time.

- *Input and Output* – There are problems that contain many pieces of information, and therefore they have various types of inputs. The same goes for the outputs; they can represent many different types of actions. NEAT is incredibly successful with these. Unfortunately, in Battleship, most inputs and outputs represent the same piece of information – the type of tile, the chances of being a ship. We can not really observe much more. It would definitely be better to stick with problems like Battleship to a standard type of neural network.

4.1.3 Comparison of Human-like player

The algorithm's expectations were high, as most humans would not be able to score higher than it. The neural network aimed to get at least to the human level, in order to have a potential of improving on the 10×10 board.

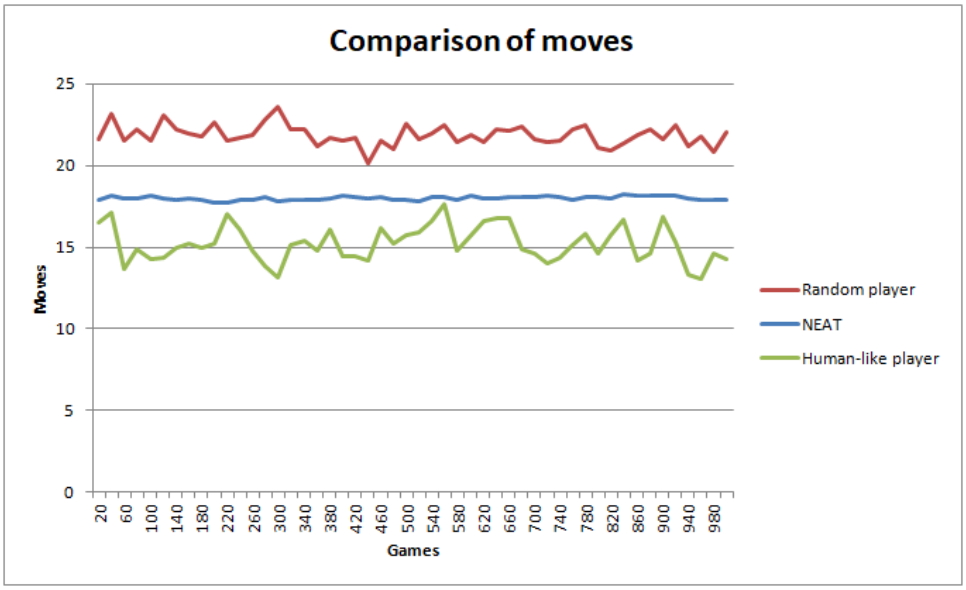

Figure 4.4: Comparison of Human-like player average moves

However, in the (4.4) graph we can see, the average Human-like player moves stabilized slightly above **15** moves, which vanquished the NEAT by quite a large margin. Therefore, NEAT will not be an adequate candidate for more extensive board training.

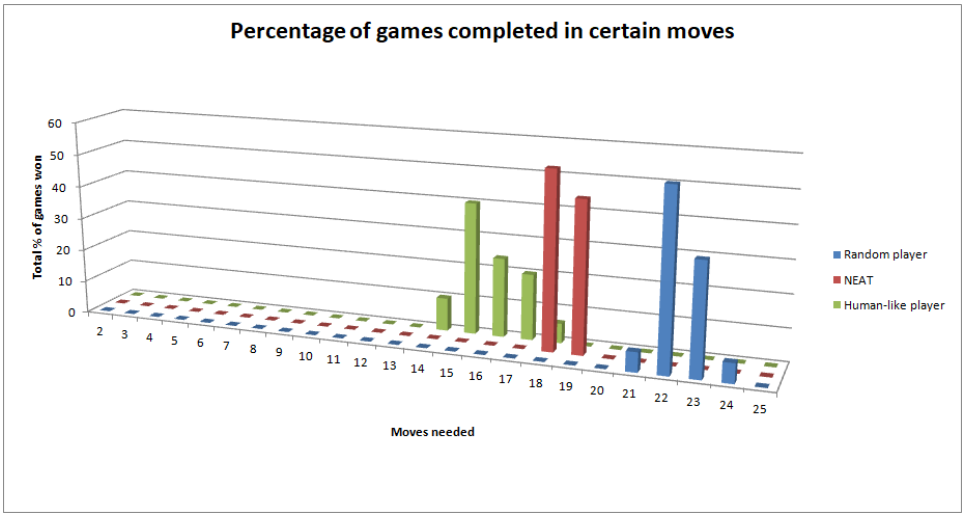

Figure 4.5: Percentage comparison of Human-like player games completed in a certain number of moves

From the second (4.5) graph, we can observe the randomness of Human-like player (the range of moves needed for the completion of the game is wide). It is comparable to the random player, however, with a decreased amount of moves. On the other hand, NEAT shows its steadiness and certainty, although an immense amount of training will be needed to bring the number of moves down to the Human-like player level.

4.1.4 Comparison of Even player to Human-like

The whole purpose of Even player is to extend the Human-like player into the full capability, considering the human-like player's strategy had one significant drawback of choosing completely random moves in the Hunt mode. That is the reason we will be comparing these two to see the improvement. We expect to see already a solid strategy with a decent number of average moves.

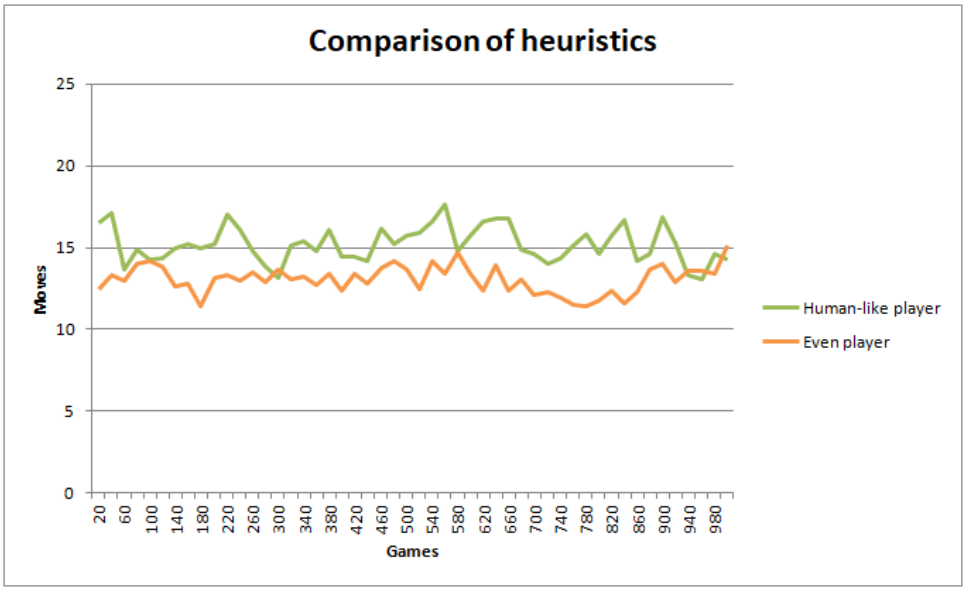

Figure 4.6: Comparison of Even and Human-like player

The (4.6) graph clearly shows the improvement that brings Even player's strategy.

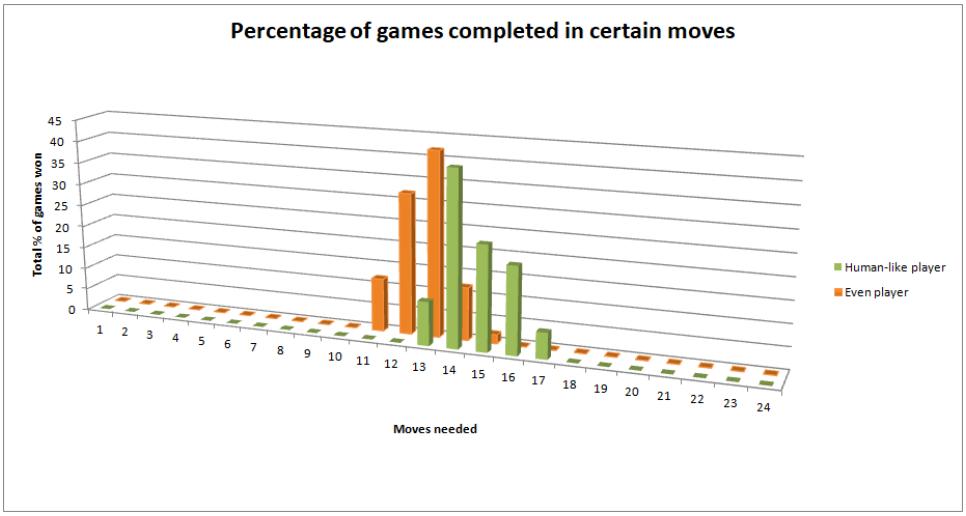

Figure 4.7: Percentage comparison of Even player games completed in certain number of moves

Reducing the number of tiles needed to be guessed by half brings a reduction of another **2,2 move** (4.7 graph) in comparison to **Human-like player**, resulting in **13** average moves needed for the completion of the game. Without doing any proper calculations, this strategy could be considered the best out of strategies manageable by a human brain.

4.1.5 Comparison of Heat-map player

Finally, we are getting to very accurate methods using calculations and precise probabilities.

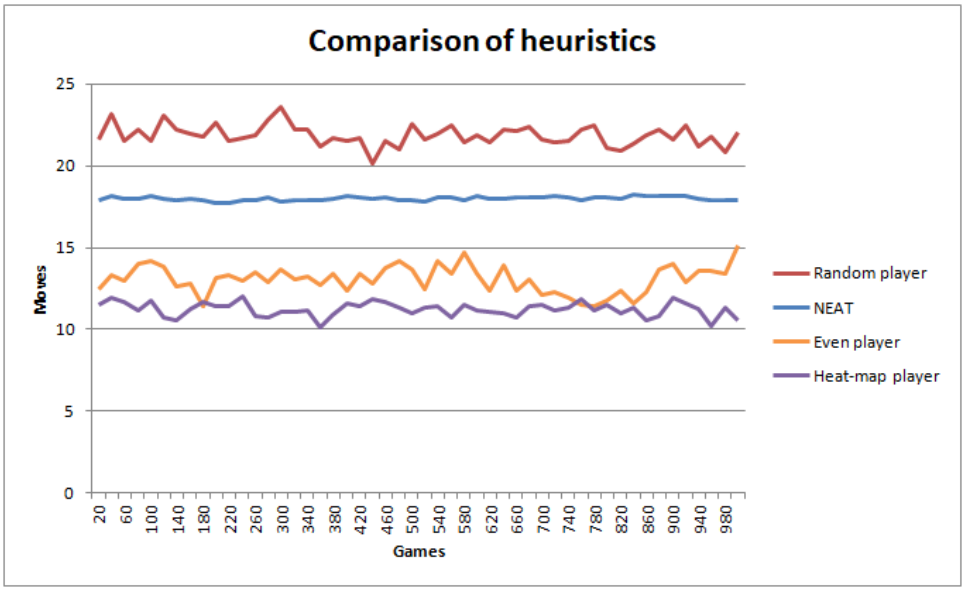

Figure 4.8: Comparison of Heat-map player with other heuristics so far

From the (4.8) graph we can see how highly accurate a Heat-map strategy is. With only **11.2** moves needed in an average it outperforms every strategy so far. Moreover, we can finally see the massive gap between the neural network and the highly precise strategy.

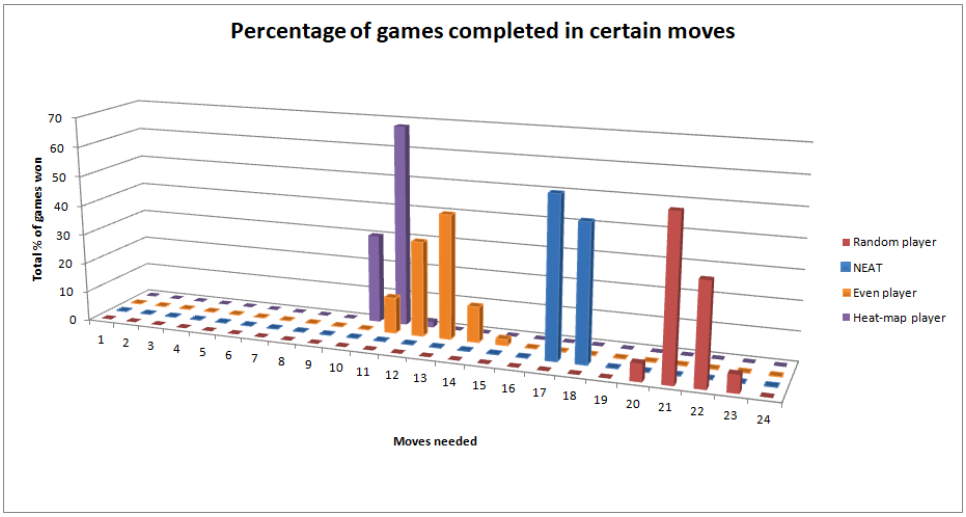

Figure 4.9: Percentage comparison of Heat-map player games completed in certain number of moves

From the second (4.9) graph we can see the certainty of almost 70% to finish the game in 11-12 moves. On the other hand, there need to be a consideration we are still researching the reduced version of a problem, therefore we should look at the speed and take in an account also that parameter.

Table 4.1: A comparison based on 1,000 simulations done by each of the heuristic

Player	Size	Ships	Average moves	Total time[ms]
Random	5x5	2	21.86	10
NEAT	5x5	2	17.99	8
Human-like	5x5	2	15.22	10
Even	5x5	2	13.07	10
Heat-map	5x5	2	11.23	2040

The Random player, NEAT and Human-like heuristics all solved 1,000 simulations in average of 10 milliseconds, while the Heat-map needed an average of 2040ms which is 204x slower than other algorithms. Even though Human-like algorithm is much more complex

than Random-player, the reduction of moves needed, speeds up the process to be as fast. On the other hand, in the NEAT case, we need to consider the huge amount of time needed in the training phase, so it is not really comparable to others, although it is faster than all of them after that.

Considering the amount of memory used when generating possible states for the Heat-map player, we can only use it on the prototype 5×5 version to compare with MCTS. Therefore, Even player seems like the most appropriate opponent for the MCTS in the original 10×10 version of the game.

4.1.6 Comparison of MCTS and finding suitable parameters

First of all, we will declare some essential variables we are going to be using. As we are going to calculate every move, we need to set up a number of iterations for each of the move as:

$$m1 * 500$$
$$m2 * 1000$$

Where:

$m1$	is the number of moves a current state has
$m2$	second try with doubled iterations

In other words, in 500 iteration scenario, at the start of the game with 25 moves available, resulting in a total of 50 (2 for each coordinate), we are going to do 25000 iterations (simulated games) and gather information about the best move. In the second move, there will be only 24000 iterations and so on. Then, we will be able to observe the impact of raising them. The total number of iterations will be:

$$\sum_{n=25-m+1}^{25} n * 2 * i$$

Where:

m	is the number of moves needed to complete the game
i	is the number of iterations

Table 4.2: A comparison of a single game guided with MCTS with different number of iterations (games simulated from current state)

Iterations per move	Average moves	Total iterations	Total time of 1 game[s]	Time of 1 move[s]
500	12.38	234 380	1.61	0.13
1000	12.23	468 240	3.18	0.26

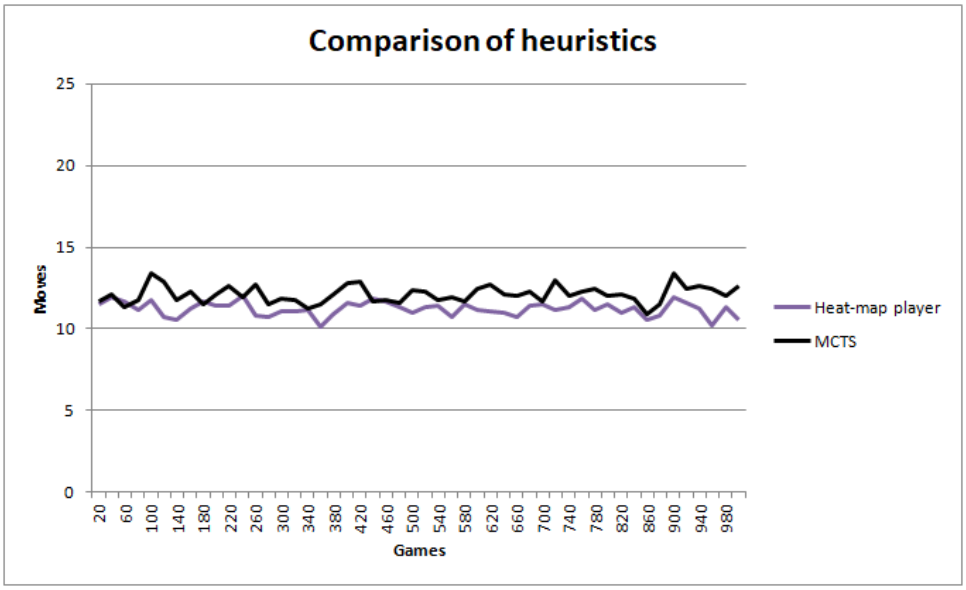

Figure 4.10: Comparison of MCTS with 1000 iterations per move

Even the 500 iterations done for each available move brings us a decent number of moves, resulting in only a **0,13s** needed for a single

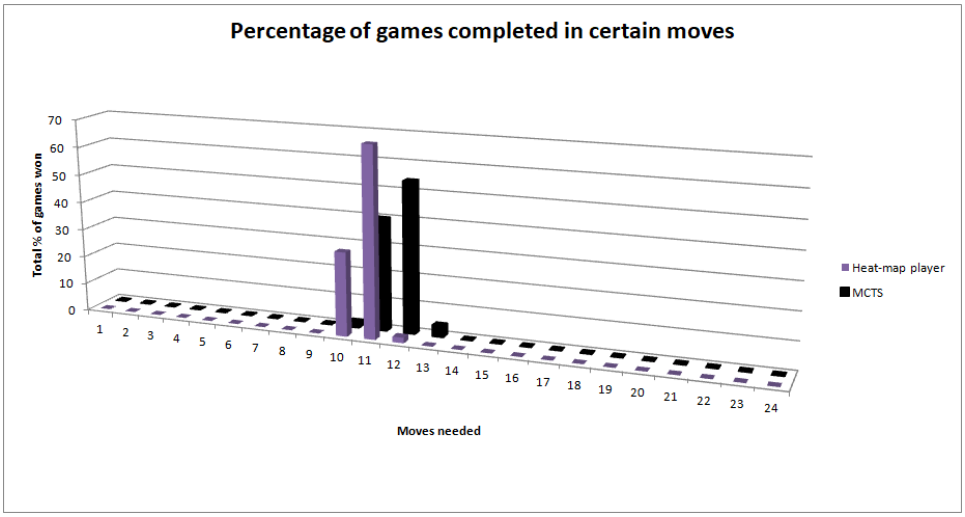

Figure 4.11: Percentage comparison of MCTS with 1000 iterations per move completed in certain number of moves

move. From the (4.10) and (4.11) graphs, we can see, the MCTS is about one move behind the almost ideal algorithm. However, the doubling of iterations do not bring much difference. The reason may be, we are still operating on a small board, where we can get to an optimal strategy with fewer iterations needed. We also acquired a speed of 1 iteration, representing a full simulation of 1 game, to be around **0.0136ms**, which is comparable to the speed of the Random player.

Considering we are still working with a small board, and more importantly, a small number of ship tiles, even a single ship's discovery brings us a satisfactory evaluation. Let us try to decrease the addition of an incomplete discovery of the ships in the reward function (3.1), by powering the proportion of ships found:

$$Reward = \left(\frac{v}{s}\right)^2 * m \tag{4.1}$$

The expectation of this change is to prioritize the full ship finding since revealing all of them results in $1^2 = 1$ while revealing only half of them results in $0,5^2 = 0,25$. We will also try a reward function accept-

ing only fully valid layouts (all ships need to be correctly predicted or the payoff from the simulation is 0).

Table 4.3: A comparison of a single game guided with MCTS with different number of iterations (games simulated from the current state), with emphasis on revealing more and all ships in the simulation phase

Player	Iterations per move	Average moves	% improvement
Ships-squared	500	12.105	3,7%
Ships-squared	1000	12.08	2%
Full-ships	500	11.97	5,5%
Full-ships	1000	11.755	6,5%

With the emphasis on revealing more ships, we were able to get closer to the **Heat-map player**. In the (4.3) table, we can see that with the emphasis on only correct layouts, we got under **12 moves**. Moreover, we reached **6,5%** improvement compared to the scenario with the original reward function. The adjustment of rewarding function has a potential and will be tried on the original 10×10 board.

Table 4.4: A comparison of a single game guided with MCTS with the different number of iterations, reward functions, with emphasis on choosing the hitting outcomes.

Player	Iterations per move	Average moves	% improvement
MCTS	500	12.18	2.7%
MCTS	1000	12.08	2%
Ships-squared	500	11.75	8,5%
Ships-squared	1000	11.70	7,3%
Full-ships	500	11.95	5,8%
Full-ships	1000	11.69	7,4%

The last parameter we are going to elaborate on is the choosing function of the best move. So far, we have been choosing the move

with the best payoff, discounting the move's result. What could have happened, is that simulating a move resulting in missing could have had the best payoff, and we tried it. Now, we will only choose moves based on their payoff, if they got into the simulation as hits.

In the (4.4) table, we can see that all three types of rewarding experienced an improvement. In the case of squaring the proportion of ships found, we have reached a **8,5%** betterment with only a **0,5 moves** behind the ideal level.

4.2 Original game with 10x10 board

So far, we have performed a comparison of heuristics on a reduced version of a game on a 5×5 board, where many solutions worked fastly and efficiently. However, the original game is situated on a 10×10 board with many more ships. We will simulate, average and collect data from 200 games for each of the following heuristics:

- **Random player** – To set up the basic number of moves.
- **Human-like player** – To see what an average human would be able to score like.
- **Even player** – This will be the heuristic, that got the closest to the MCTS on 5×5 and is still manageable on a larger board.

Table 4.5: A comparison of basic heuristics on a 10x10 board with ships of lengths [5,4,3,3,2] with total of 17 tiles being ships

Player	Average moves	Average time for a game[ms]	Average time for a move[ms]
Random	93.8	0.03	0.0003
HumanLike	61.3	0.03	0.0005
Even	58.35	0.03	0.0005

- **MCTS-classic** – The MCTS with the original (3.1) function.
- **MCTS-classic only hits** – We will enhance the MCTS with the original reward function with a heuristic of choosing only hitting moves. Considering, we are calculating the payoff of every

possible move (even a miss), it might be illogical to perform moves that were calculated as misses.

Table 4.6: A comparison of MCTS with the original reward function on a 10x10 board with ships of lengths [5,4,3,3,2] with total of 17 tiles being ships

Player	Iterations	Average moves
Classic	500	51.95
Classic	1000	51.0
Classing only hits	500	51.8
Classing only hits	1000	49.45

- **MCTS-ships-squared** – The MCTS with an adjusted version of the reward (4.1) function, putting emphasis on revealing as many ships as possible.

- **MCTS-ships-squared only hits** – We will do the same as the previous heuristic, choosing the hitting moves with the best payoff.

Table 4.7: A comparison of MCTS with the advanced reward function on a 10x10 board with ships of lengths [5,4,3,3,2] with total of 17 tiles being ships

Player	Iterations	Average moves
Ships-squared	500	56.1
Ships-squared	1000	55.55
Ships-squared only hits	500	53.05
Ships-squared only hits	1000	48.1

- **MCTS-full-ships** – The MCTS with an even more strict version of the reward function. If the simulated layout is not entirely valid, we will not get any score. The assumption is that this strategy will only be highly successful on smaller boards.

- **MCTS-full-ships only hits** – Once again, with only the hitting moves selected.

Table 4.8: A comparison of MCTS with the strict reward function on a 10x10 board with ships of lengths [5,4,3,3,2] with total of 17 tiles being ships

Player	Iterations	Average moves
Full-ships	500	82.55
Full-ships	1000	80.4
Full-ships only hits	500	82.6
Full-ships only hits	1000	80.15

From the (4.8) table, we can see, it is impossible to guess complete layouts of the ships on large boards. We were only able to guess it in the late stages of the game (usually after the 60th move). Therefore, this method is only suitable for the 5x5 board.

On the other hand, the original rewarding function works well, as shown in the (4.6) table. The number of moves got under the **50**, resulting in more than a half board remaining covered at the end of the average game.

Finally, in the (4.7) table, we can observe the enormous impact of choosing just hitting simulations, resulting in more than **7 moves** reduction with 1000 iterations being done for each move. We found the right balance in prioritizing layouts with many ships found, but also taking into account we will not be able to find all of them in the early stages.

The analyzes of players:

- **Random** – A random player's biggest problem is a **17% chance** to end the game with **100 moves**. In the last (4.12) graph we can see the vast gap between every other strategy and this player.

- **Human-Like** – The average amount of moves needed for an average human ended up being slightly above **61 moves**. The most significant disadvantage of this player is the uncertainty

of hitting a part of the ship, that is the main reason we can see many peaks in the graph.

- **Even** – The strategy of opening even tiles works perfectly, maintaining the speed but also improving the accuracy of Human-like player scored an average number of **58 moves**. Reduction of every move relative to Even player maintaining suitable speed and memory budget requires a smart strategy.

- **HeatMap** – Even though the Heat-map strategy exponentially reduces the memory needed in comparison to the tree of every possible move, it is impossible to generate possible states for a board larger than 8×8 in a reasonable time.

- **MCTS** – The most successful strategy turned out to be MCTS with (4.1) reward function, emphasizing layouts with as many ships found possible, however not strictly all of them, during the simulation process. The selection of only hitting moves also worked reasonably well, resulting in final **48,1** moves needed, outperforming the **Even player** by 10 moves, which can be seen in the last (4.12) graph. The average time for the calculation of 1 move settled down to be around **4,8 seconds** for a single move.

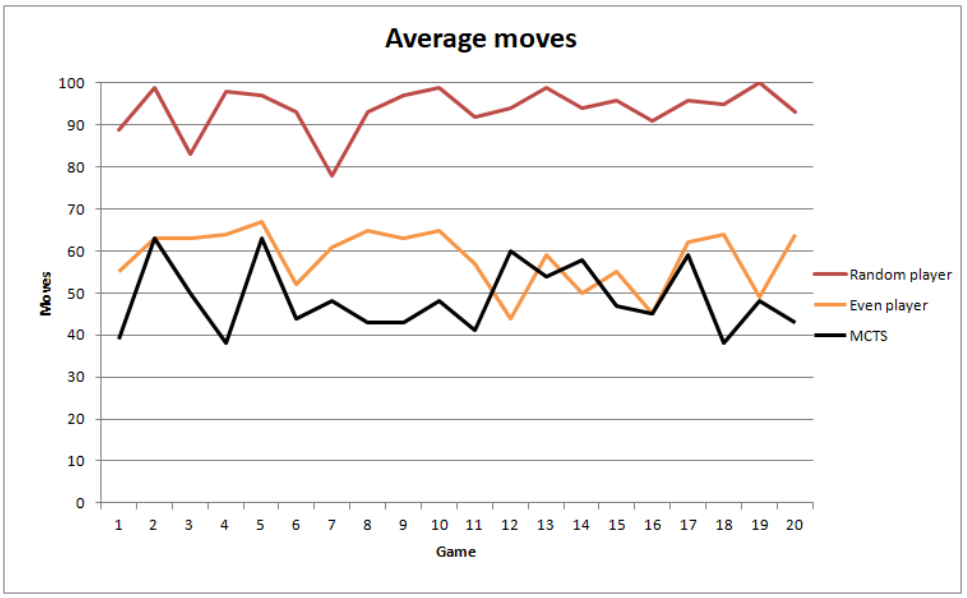

Figure 4.12: Final comparison of heuristics on a 10x10 board game with 5 ships

Bibliography

1. BEN, Fry; CASEY, Reas. *Processing* [online] [visited on 2020-02-07]. Available from: https://processing.org.
2. CODE-BULLET. *NEAT Template* [online] [visited on 2020-01-10]. Available from: https://github.com/Code-Bullet/NEAT_Template/tree/master/TemplateNeat.
3. CONOR, McDonald. Machine learning fundamentals (II): Neural networks [online] [visited on 2020-05-28]. Available from: https://towardsdatascience.com/machine-learning-fundamentals-ii-neural-networks-f1e7b2cb3eef.
4. IAROSLAV, Omelianenko. Neuroevolution — evolving Artificial Neural Networks topology from the scratch [online] [visited on 2020-01-28]. Available from: https://becominghuman.ai/neuroevolution-evolving-artificial-neural-networks-topology-from-the-scratch-d1ebc5540d84.
5. SAGAR, Sharma. Activation Functions in Neural Networks [online] [visited on 2020-06-29]. Available from: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.
6. HUNTER, Heidenreich. NEAT: An Awesome Approach to NeuroEvolution [online] [visited on 2020-02-02]. Available from: https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f.
7. YOSHUA, Bengio; IAN, Goodfellow; AARON, Courville. *Deep Learning*. The MIT Press, 2017. ISBN 978-0262035613.
8. CAMERON, Browne et al. A Survey of Monte Carlo Tree Search Methods. In: [online] [visited on 2020-07-13]. Available from: http://www.incompleteideas.net/609%20dropbox/other%20readings%20and%20resources/MCTS-survey.pdf.
9. GIJS, Jan. Walk the Line - MCTS Evaluation Functions [online] [visited on 2020-06-02]. Available from: https://codepoke.net/2015/03/03/walk-the-line-search-techniques-evaluation-functions/.

10. LEVENTE, Kocsis; SZEPESVARI, Csaba. Bandit based Monte-Carlo Planning. In: [online] [visited on 2020-07-13]. Available from: http://ggp.stanford.edu/readings/uct.pdf.

A Project structure

The project consists of the following two folders:

- Processing3 – This folder contains the source code for the NEAT, random player and GUI part.
- C++ – This folder contains the CMakeLists, which is needed for the compilation specification, but also the src folder, which contains the source code for C++ implementation.

A.1 Project compilation and execution

To compile the project, it is necessary to have installed the following dependencies:

- cmake $\geq$ 3.9
- g++-7 or higher to support C++17
- Processing3 and Java 9 or higher for the NEAT part

After installing the NEAT dependencies, the user only needs to open the necessary files in the Processing3 and click the play button.

In the C++ code case, after entering the C++ folder, a sequence of the following commands compiles the project:

```
mkdir build
cd build/
cmake ..
make
```

Once the project is compiled, it can be run using the following command:

```
./battleship
```

This command starts the program with prepared instructions. Following these, the user can specify which player will be his opponent.

Moreover, he can choose whether the game will be played on the 5×5 or the 10×10 board. Both players start with randomly distributed ships, choosing their moves until the game results in a win of one of them or the draw.