École polytechnique de Louvain

Conception of an AI for the game Stratego

Author: **Thomas Robert**
Supervisor: **Pierre Schaus**
Readers: **Alexandre Dubray, Harold Kiossou**

Academic year 2022–2023
Master [120] in Computer Science

Contents

Acknowledgments

Before beginning this thesis, I would like to thank a few peoples that helped me go through this work. Completing at the same time the period of five years spent at the Catholic University of Louvain (UCL).

First, the person who mentored me, followed my advancement throughout this year, and gave me advice during weekly meetings, Mr. Alexandre Dubray.

I also thank the thesis committee members for their interest and time invested in the review.

And finally, my brother that reviewed it multiple times, saving me a lot of time in the writing process.

Chapter 1

Introduction

Artificial Intelligence (AI) has become increasingly prominent in today's world, spreading in various domains. Its role lies in providing automated solutions to a wide range of problems. Initially, AI focused on addressing simple tasks through scripts, gradually progressing toward more challenging ones such as classification problems. Over time, AI has evolved into a domain that contains more creativity, with recent remarkable advancements in natural language processing (NLP), image processing, and game playing.

The focus of this thesis revolves around the third domain, game playing, and more especially developing an AI agent for the game of Stratego. Stratego is a strategic and challenging zero-sum game opposing two players on a board, this game is characterized by imperfect information and vast state space. To understand the level of complexity it has, consider that Stratego encompasses approximately 10^{535} distinct nodes in its *Tree of States*, surpassing the state space of other renowned board games such as Go, which has around 10^{361} nodes, and Chess, which has roughly 10^{64} nodes [2]. The hierarchical data structure *Tree of States* represents all the possible game states via its nodes and the transitions between these states based on the available actions via its links. This structure serves as the foundation for search algorithms like Alpha-Beta and Monte-Carlo Tree Search (MCTS), simplifying the exploration of potential future states stemming from the current one. Besides the exploration, it serves to backtrack information from the bottom or leaf nodes to the top nodes, near the current state in an efficient way, making the decision of these algorithms for the best move more informed.

One standout machine learning (ML) approach in the field of Stratego AI is called DeepNash [2]. This advanced method involves an autonomous agent that achieves human expert-level performance through self-play, learning towards and even reaching the Nash equilibrium. The Nash equilibrium is a state that is typically not exploitable by the opponent making it a significant achievement for this game. DeepNash uses a large neural network architecture that takes an impressive tensor as input. Previous ML agents attempted to use Convolutional Neural Networks but they did not reach the same complexity for the structure, nor the same input size, resulting in poor performance.

Other AI techniques have also been explored for Stratego. Some agents have utilized probabilities and scores to evaluate all encounters on the board, considering each ally piece and their ability to reach and attack any opponent pieces to choose their best option. Another well-known AI technique family for board games is Tree of States Search as previously mentioned, Demon Of Ignorance [1] is a great example of this, using an optimized Alpha-Beta algorithm designed for Stratego. However, most AI agents have only managed to surpass amateur human-level performance by a small margin, attesting the significant challenge that the game presents for AI.

As for my contribution, I opted for a Monte-Carlo Tree Search algorithm as I believe it is better suited for the exploration of the tree and the complexities of Stratego. This intuition is sustained by the fact that another board game AI champion AlphaGo Zero [20] also used it. Unlike the depth-first search approach used in the Alpha-Beta algorithm, MCTS implements a best-first search strategy, with multiple game simulation scenarios to obtain a more accurate representation of potential outcomes than exploring all possible actions in a specific order would give. This approach allows for a more informed and direct exploration of the enormous Tree of States of Stratego as it selects a better direction in the tree rather than a single node more valuable. To enhance its performance a little further, I implemented a custom evaluator tuned to consider various aspects of the game helping it to guide the search process towards moves that have a higher likelihood of leading to good outcomes. Additionally, I incorporated game modeling, probabilities, and other optimizations in the same idea of helping the decision-making process. Through extensive experimentation, I gained valuable insights into the strengths and weaknesses of employing these techniques. I also identified key factors that play a crucial role in enhancing the algorithm's overall performance and effectiveness.

The thesis is structured as follows: after the introduction of the subject, the second chapter provides an explanation of the game of Stratego, covering its rules, and strategies. This chapter ends with the presentation of the challenges associated with the development of such an AI, although the inherent characteristics of Stratego, already offered some insights into the difficulties of creating an effective AI agent. The third chapter presents the State-of-the-Art AI techniques applied to Stratego. It explains how these agents play and highlights their achievements in history. After presenting the State-of-the-Art, the fourth chapter focuses specifically on the implementation of my custom agent, going into detail about the different aspects of my agent. The results of the experimental evaluation are detailed in the fifth chapter, where all the agents compete against each other, my custom agent and the other AI agents in the bot pool. The performance of each agent is analyzed to determine which strategies prove most successful. Finally, the last chapter concludes the thesis with the potential future works possible to enhance the AI agent and further improve its gameplay.

Chapter 2

Stratego

Stratego Classic is a game where two players engage in a strategic battle on a 10×10 board. An example of a possible arrangement of pieces on such a board is represented in Figure 2.1[1]. Each player possesses a total of 40 pieces, which are distributed among 12 different ranks. Throughout the game, the rank of the opposing pieces remains hidden, only the position is known. Some pieces are more important than others thanks to their ranks. Some have special abilities like the scout that can move multiple cases at once while the others can only do one. Some others cannot move but capture most of the pieces that attack it like the bombs. Besides these special cases, they have a fight value, the higher pieces often present in fewer copies capture the lower pieces that are present in more copies.

The flow of the game is as follows: The two players must first fill their side of the board, the 40 first cases with their 40 pieces during the setup phase. Next, the game begins with the red side choosing a move, the player first chooses a moveable piece, in other words, not the flag nor a bomb, and then a case to move to, depending on the ability of the piece, the scout, for example, can move multiple cases in one turn. After that, the other player will have to do the same process. They both can move either on an empty case or on a case where an opponent's piece is. In the second scenario, an attack is initiated. In an attack, both players reveal their pieces and so do their ranks. This reveal is only available inside the attack process, after that, both pieces are returned in a hidden state for the opponent's as before. There are several rules over ranks of the pieces and their ability to capture each other or not, the result will either have one survivor or none if it they had the same rank. So players take turns like that till either one flag is captured, or a player doesn't have any move possible, that is to say when no remaining piece is moveable or lastly we reach the limit of moves.

A very interesting aspect of playing Stratego is that players are not aware of the rank of their opponent's pieces unless an attack occurs. However, when an attack takes place, either when initiating an attack or being attacked, the involved pieces are revealed to both players only at the time of the attacks, and after that, they return to their hidden state and the players have to remember them. Lots of strategies are based on maintaining an advantage in the information on the ranks that the opponent has, forcing him to take risks.

[1]Note that on this representation, we have full information for analysis purposes, compared to default games where the enemy pieces are hidden.

One popular platform for playing Stratego online is Gravon [3]. It serves as a hub for virtual gameplay, offering diverse variations of Stratego as well as other board games. The platform also hosts competitions where players can showcase their skills. Additionally, game data from these online matches can be obtained free of charge, providing a valuable resource for analysis and research purposes.

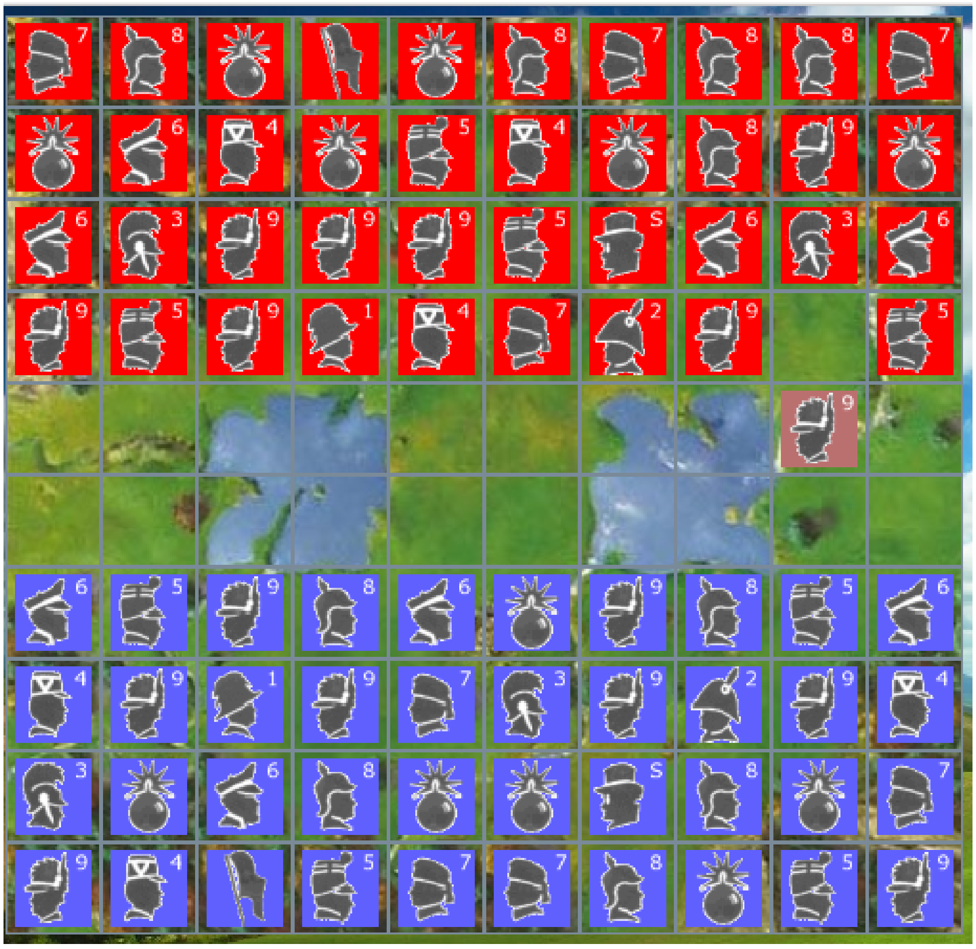

Figure 2.1: Representation of the board on the Demon Of Ignorance App available on GitHub [1].

2.1 The Game

Each piece has its own strength and some of them have special abilities. A higher value means that it can capture the lower-value pieces if there is no exception via the abilities. The number of units indicates how many of this piece you have at the start of the game and the mobility is the number of cases you can move a piece in one turn. Here is a summary of the different pieces:

Setup Phase The game begins with both players positioning their 40 pieces on their respective sides of the board. This initial setup phase greatly influences the overall strategy and approach of each player. For a deeper understanding of effective initial setups, please refer to Section 2.2.1 which provides insights into the composition of strong setups. This part of the game is mandatory but it can be automatised in most Stratego applications.

Name	Value	Number of units	Mobility	Ability
Flag	0	1	0	If captured, lose the game
Bomb	11	6	0	Only capturable by miners
Spy	1	1	1	Captures Marshal if it attacks
Scout	2	8	-	Moves of up to 10 cases in a single direction
Miner	3	5	1	Captures the bombs
Sergeant	4	4	1	
Lieutenant	4	4	1	
Captain	6	4	1	
Major	7	3	1	
Colonel	8	2	1	
General	9	1	1	
Marshal	10	1	1	Captures everything except the bombs

Table 2.1: Summary Table of the pieces for Stratego.

Filling thoughtfully the 40 squares at the start is an important part of the game since it's where you can establish a solid foundation for the subsequent gameplay.

Playing Phase During the gameplay phase, players take turns selecting a piece from their side and moving it according to its specific movement rules. When a player attempts to move their piece onto a square occupied by an enemy piece, it triggers the attack phase. At this point, the two involved pieces are revealed to both players. The outcome of the attack depends on the ranks of the two pieces. The winner will stay on the contested case while the loser is removed from the board. In the case of a tie, when the two pieces are of the same rank, they are both removed from the board.

Objectives There are three official ways to finish a game in Stratego. The first method involves a player capturing their opponent's flag, resulting in a decisive victory. However, another approach that is often used in defensive gameplay is to eliminate all of the opponent's movable pieces. If the opponent has no valid move remaining, it's automatically considered a surrender. While surrendering can be permitted in other game environments, it is not a valid outcome for my AI, so it doesn't count in the three ways of concluding a game. The third method consists in setting a limit on the total number of moves within a game. This limitation is implemented to prevent infinite computation and avoid scenarios where players engage in pursuits and evade endlessly. Typically, the maximum is often set between 1000 and 2000 moves, ensuring that only games where players are stuck are discarded. If the limit of moves is reached, nobody wins, it's a tie. In my experiment, this limit number has been set to 1500 moves.

The *Two-Square Rule* stipulates that no piece is allowed to repeatedly move back and forth between the same two spaces for more than three consecutive turns. This rule prevents players from engaging in repetitive and unproductive movements

As previously discussed, a limit on the number of moves can be imposed to ensure games do not extend indefinitely. While another approach, known as the *More Square Rule*, prohibits endless pursuits of unattainable captures, we will adhere to the limit of moves for our setup.

2.2 Strategy

The goal of this section is to provide insight into good strategies for playing Stratego efficiently. At first, we will see how to create a robust board setup that enables players to gain an early advantage in the game. Next, we will explore some mind games that are more commonly employed by human players rather than AI bots. We will also discuss a general approach to move selection or more precisely piece selection that is frequently adopted in high-level Stratego gameplay. These strategies are only ideas to guide a player and help him execute his game plan more easily.

2.2.1 Board Setup

To make a good board setup there are several aspects to take into account, and some of them are listed directly below. However, at higher game levels it is sometimes better to adopt an unpredictable behavior than trying to respect logic. Without further ado, here are these aspects I've been trying to respect in all my setups.

Defend the flag since it is the most valuable piece of your whole board and losing it means defeat instantly. It can, for instance, be placed on the first or second line of the board, surrounded by bombs or high pieces to create a protective barrier.

Similar patterns of bombs pseudo-surrounding something can be created as bait bombs to fake the location of the flag there while it is in another place completely, delaying the danger.

Covering the three *lanes* (corridors created by the board's limit and the lake in the center of the map) with different high pieces is important to keep a control already at the start of the game. For instance, you could place the general on the left, the marshal in the middle, and the spy on the right so that the opponent will have bigger difficulties making a breach in your defense early in the game. Keeping the same idea with all the pieces, dispatching them, and not stacking the same rank of pieces together is also a reasonable strategy. Lastly, setups often put some low pieces at least on the front line to be able to set up the *Lower-pieces reconnaissance* strategy (see below).

2.2.2 Common strategy

Bluffing is a strategic technique that involves creating a sense of uncertainty and deception in the mind of the opponent. It manipulates the enemy's perception and creates doubt about the true value and intentions of your pieces, so it works best if the opponent is deeply thinking about the moves. For example, by strategically deploying low-ranking pieces or concealing their true strength, you can slow the opponent's progress and force them to approach the game more cautiously. Bluffing relies on making the opponent believe that you are undertaking certain actions or executing specific strategies, even though you have no intention of following through with them. This psychological battle can be highly effective in disturbing the enemy's plans and gaining a strategic advantage.

Baiting is a strategic approach that capitalizes on the opponent's tendency to make superficial assessments of moves. Baiting is like fishing; you start by letting a piece known by the enemy that he could get in close range with before retreating to a more advantageous position. The aim is to lure the opponent's piece into a position where your own pieces can counter-attack or instant revenge efficiently. It exploits the opponent's greed and short-term thinking.

The acquisition of information and knowledge holds great importance for both players. Understanding the enemy's pieces without risking valuable high-ranking ones becomes a critical objective. As a strategic approach, players often employ their lower-ranked pieces to gather information on the opponent before committing their high-ranking pieces to capture them. This method which could be named *Lower-Pieces Reconnaissance* allows players to make well-informed decisions, minimize risks, and gain a strategic advantage over their opponents, making it important for playing Stratego effectively.

2.3 Why is it difficult

In order to understand the challenges involved in creating an AI for Stratego, let us examine the specific characteristics of the game. Subsequently, I will outline different issues encountered during my exploration of the subject.

Stratego's enormous state space poses a significant obstacle, making it challenging to conduct a search on it directly without employing effective pruning techniques and search guidance. Hence, traditional AI approaches often face limitations when it comes to probing long-scale plays, which results in their inability to match the strategic predictions that human players tend to do. Time constraints also limit the agent's ability to perceive the future course of the game accurately since the search is taking lots of time.

The imperfect information adds complexity to the evaluation of states or comparisons of the possible moves at a time. Additionally, it enables strategic elements such as Bluffing and Baiting. This mechanism of hidden pieces unless there is an attack, adds an element of uncertainty and strategic decision-making as players must carefully consider the potential risks and rewards associated with each move, weighing the hidden information against their own objectives.

Stratego lacks direct links between an individual move and the final outcome of the game or how much it contributes to it, like a reward. The nature of the game emphasizes the significance of sequences of moves or game plans that contribute to progress toward final victory. Resulting in having lots of states with a heuristic value really close to each other even if it is well thought out.

While these inherent challenges are substantial, my experimentation revealed additional obstacles. An example of it is the prediction of the opponent's moves. Accurately achieving this task proves difficult during the search for the best possible move. The non-obvious nature of the opponent's choices, coupled with interchangeable move orders, frequently leads to overlooking critical moves that the opponent may execute in subsequent turns and missing the real best move behind it. Furthermore, players or agents often have different game plans, and the bests even change and adapt them to the game, they can be bluffing, baiting, preferring to defend, or being aggressive. And that is making the process of estimating the opponent's move even more difficult.

Modifying parameters or assigning new weights to actions often yields minimal discernible results or rather they are difficult to understand at first hand as they need to be reviewed. The length of the games usually varies between 200 and 600 moves, even more sometimes, therefore to know the resulting impact of these modifications, there is the need to analyze thoughtfully every move of these long games to see if the behavior we wanted was respected throughout the entire game.

However, in my experiment, the real problem was due to the operation of the MCTS algorithm, first, the search it does is not fully deterministic since it relies on simulations of the game to decide a move. Moreover, the changes may not even affect the final decision if they are not great enough. That, in addition to the board setup being randomly picked inside a pool of setups each game to avoid overfitting. Evaluating the impact of such adjustments on performance becomes challenging, even when employing automated game and graph generation followed by analysis. The process is time-consuming, needs computational resources, and determining the efficacy of changes is still often a complex and uncertain task.

Overall, these difficulties highlight the limitations and struggles of basic AI techniques here. And they also underscore the complexities involved in developing an effective AI system for Stratego.

Chapter 3

State of The Art

This chapter will describe the state of the art of AI agents that have been done and released over the internet via site [11], open source project [1] or papers [2, 8, 15]. As well as giving a glimpse of how they work or what useful insight they provided me on the work of making myself an AI agent in the game of Stratego.

While we will first get into the other agents of the bot pool, the second section continues with the mention of great works and inspirations in the domain of AI for Stratego.

All the bots in the bot pool, together with my custom implementation and a vanilla implementation of MCTS, played in my experiment and thus will have their result against each discussed in Chapter 5.

3.1 Bot pool

The Bot pool in my experiment refers to the collection of AI agents with which I engaged during the course of my research. Each AI agent possesses its own distinct characteristics, strengths, and weaknesses. Here, I will provide you with information about some of the AI agents I encountered in my experiment, highlighting their notable attributes and limitations

Some of them were re-implemented by myself to be able to run it on my environment. The Bot pool represents a diverse range of AI agents, each with its own strengths and limitations, providing a valuable foundation for comparative analysis and evaluating the effectiveness of different approaches in the context of Stratego gameplay.

3.1.1 The UCC Programming Competition 2012

This competition [6] was held in 2012 with several agents written in different languages, I choose to re-implement Hunter and Asmodeus because these two Python implementations were more straightforward in their implementation than the others' while providing a reasonably good strategy via their tuning of parameters that gives scores the actions. Hence comparing my agent to these AI is as interesting as comparing it to more high-level ones to mitigate different levels of gameplay. As told before, I needed to refactor it to work on my environment. Both these AI evaluate all the moves available at their turn and select the best one. A move is valued by checking that the direction taken by the playable piece is toward a winnable opponent's piece or fleeing a dangerous one. They have a notion of danger, taking risks via tuned parameters, fleeing, or fighting. Their behavior is fully deterministic since they select the only best move via fixed weight. Moreover, they don't implement the Two-square rule or the More Square Rule.

Another participant of this competition was, for example, *Peternlewis* who won this cup by playing safely its lower-pieces first to discover before gaining material advantage and at the end playing its miner for pieces that didn't move before. *Celcius* was the runner-up in this competition. After that, there were also *Ramen* that due to an infinite chase behavior had some game discarded, *Vixen*, and finally some basic implementations. Most of these implementations at that time used move evaluation via score instead of search trees or more complex deep neural structures.

Asmodeus

Asmodeus is rather straightforward and focuses solely on evaluating attacks between its own pieces and the enemy's pieces. The evaluation function assigns a value to the different moves available at its turn. This value is based on various parameters that depend on the piece playing and other factors. The implementation has a list of risk scores, suicide scores and uses the probability of eliminating enemy pieces for each of its pieces.

During gameplay, Asmodeus prioritizes moving its pieces forward, selecting the move that offers the best opportunity to engage with an opponent's piece. However, it does not take into account the potential threats along the path to its target. As a result, there are instances where Asmodeus may unintentionally lead its own pieces to suicide by pursuing high-value targets without considering the overall strategic context.

The gameplay is aggressive and we see that the parameters are well-tuned because most of the time the assumption it makes are correct and it manages to fight pretty well.

Hunter

Hunter is an AI agent that generally exhibits more advanced decision-making capabilities compared to Asmodeus. However, during the transition to a different environment, a parameter has been fixed, which may have impacted its behavior in a way.

One key feature that sets Hunter apart from Asmodeus is its ability to assess the danger level of a situation and retreat. Hunter then takes fewer risks and opts to retreat even when it holds a positional advantage if one piece is highly menacing. As a consequence, there are instances where games end in a tie due to the limit of moves, as Hunter becomes stuck in a defensive mindset, fearing the opponent's potential actions.

Balancing risk and reward is an important aspect of future improvements to Hunter's gameplay strategy.

3.1.2 Demon Of Ignorance

Demon Of Ignorance is an open-source implementation [1] of Stratego with a bot that has 9 levels of difficulty which correspond basically to the time it has to think about a move. The formula used is $AiLvl^2/100$ seconds. A graphical interface is available and it is where I played my experiment against this bot. I had to link it directly on my configuration but

I only succeed halfway, so the graphical interface was serving to do the second part of the link.

This AI agent uses the Tree of States search algorithm named Alpha-beta, which is a pruning variant of the minimax algorithm. In addition, they tuned it greatly via several techniques such as iterative deepening, transposition table, heuristic history, and killer move in order to help the search explore the most promising moves before. This exploration increases the pruning success of the Alpha-beta. To solve issues related to shallow search depth part of the algorithm, they implemented Quiescent Search and Extended Search.

The Quiescent Search has been proven to work really well for Stratego [12]. That is a technique that extends the search of a Tree of States search algorithm for nodes that are unstable as it is. It serves to mitigate the horizon effect, an effect that happens in a conflicted area where lots of fights and different capture can happen, highly modifying the evaluation. Therefore if a node is in the situation of available captures nearby, the algorithm will extend it till it becomes less changing.

As for the other optimizations, they allow the use of an Alpha-Beta although some people [8] said early on that this algorithm won't work on Stratego.

Playing against it I've been impressed by how human the moves he makes feel, we can see real plans make their way via a play style always on the defensive instead of just single moves like most of the AIs. It can also use bluff, bait, and other mind games during a confrontation.

Besides making calculated moves, it takes into account hidden information and uses opponent modeling [10] to better predict the probabilities of the pieces on the board knowing when to take risks. The Two-Square and More-Squares rules are always respected via the move generation that forbids them.

3.1.3 R-NaD

The Regularised Nash Dynamics (R-NaD) algorithm is a method used to help converge the learning of DeepNash to a Nash Equilibrium, a well-known state in game theory that make it difficult to exploit any strategy against because no player can unilaterally improve their payoff. The three steps that loop in this algorithm are as follows:

1. **Reward transformation**: In this step, the base reward function, that is to say, the function that shows the reward that an action gives with a trajectory, is modified towards the Nash equilibrium. The modification involves adding a regularization term on the policy to encourage the agent to follow the right direction during learning. Resulting in a transformed game with a different policy.

2. **Dynamics step**: In this step, we will run the replicator dynamics until convergence to a fixed point. In other words, the agents interact with the environment with the new policy and reward and adapt till it obtains a new fixed point policy.

3. **Update step**: It updates the regularized policy for the next iteration with the newly found fixed point.

DeepNash

DeepNash [2] stands as the most recent and highly successful work in the realm of Stratego AI. This remarkable agent has achieved unprecedented performance by entirely learning how to play, make setups, and evaluate its own advantage in the game through self-play and deep learning.

Explaining in detail this AI's methods and gameplay is out of the scope of this work. In fact, DeepNash surpasses the level of most human players of Stratego since it demonstrated its mastery by doing an all-time 3rd place on the Gravon [3] platform. Gravon serves as a hub for elite Stratego players from around the world, hosting top-tier online competitions.

DeepNash combines the R-NaD algorithm with deep neural network architecture, jointly integrating both to learn and master effective strategies.

The training infrastructure behind DeepNash draws inspiration from the Impala [17] and Sebulba [16] structures, adapted to suit the requirements of the R-NaD algorithm. This architecture incorporates three primary components: *actors*, a *Replay Buffer*, and a *learner*. And each time the loop is done, there is an evaluation of the agent versus a bot pool.

The *actors* are responsible for the self-play part, generating complete game data. These game data are then stored in a *Replay Buffer*, functioning as a queue that temporarily holds games until the *learner* reads them into memory.

The *learner* retrieves a batch of full games from the *Replay Buffer* and segments them into smaller chunks. During the forward pass, these chunks are processed to recompute the precise network (LSTM) state for each timestep and trajectory statistic. In the backward pass, the chunks are reprocessed with the exact network state and complete trajectory statistics, such as the exact returns. Once this full process is done, it updates the parameters.

The actor is the part actually playing the game. Achieving such impressive results requires an enormous neural network architecture capable of processing a tensor observation of size $10 \times 10 \times 82$ as input. This structure is composed of several pyramid-like components, themselves being composed of a sequence of convolution residual blocks and deconvolution residual blocks.

The output of this architecture is more straightforward, comprising three policy heads in the form of probability distributions 10×10, and a value head representing the agent's value function as a scalar. The policy heads are used for different phases of the game: the deployment phase (first 40 steps), the piece selection phase, and the displacement phase.

Through extensive training, self-play, and dedicated efforts, DeepNash's final model achieves remarkable results, demonstrating superior performance against both human players and other state-of-the-art AI agents than anything done before.

Public Implementation

A public implementation of a smaller version of R-NaD was added to the Open_spiel repository [4]. It was mentioned that this implementation is designed to work only with smaller games. Despite this, I attempted to use it on Stratego but encountered several issues in addition to the structure not being able to manage such a great space.

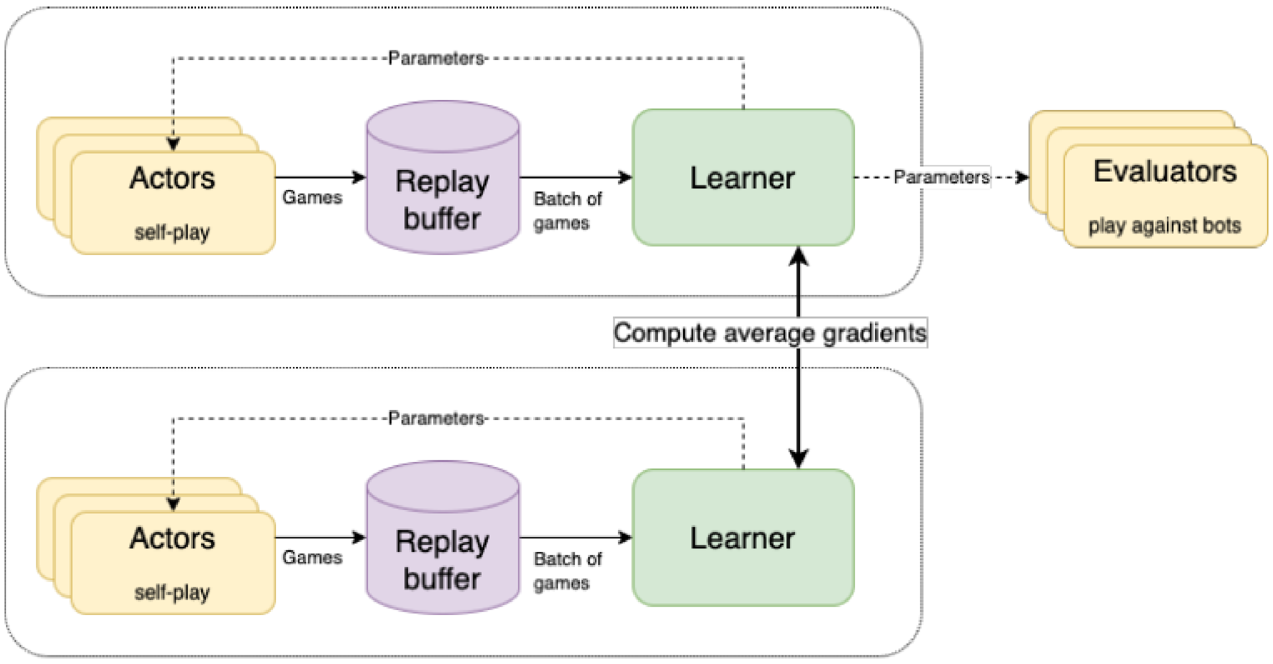

Figure 3.1: DeepNash Training component. From [2], p43

One limitation was the inability to include additional information beyond the state itself in the training or the playing part. This meant that valuable knowledge of the game, such as which pieces were discovered, could not be utilized by the agent during the learning process. This lack of contextual information makes learning even harder.

Moreover, due to hardware and time constraints, I was unable to train the agent for the 7.21M steps DeepNash did. Instead, I could only manage to train it for 100,000 iterations. Consequently, the agent's gameplay performance was greatly impacted and often exhibited random or bad actions.

3.2 Other mention

DeepNash not only evaluated itself against Asmodeus and Demon of Ignorance, but it also considered other Stratego AI agents to assess its performance [1]. While I won't go into the details for Celsius, Celsius1.1, PeternLewis, and Vixen from the 2012 Australian tournament as they share too many similarities with other agents, it is worth mentioning that they were part of DeepNash's evaluation process.

These various Stratego AI agents, presented in the previous section and here represent the state-of-the-art in the field. Each of these agents brings unique little ideas worth taking note of while developing our own Stratego agent. By exploring the advancements made by researchers over the past three decades, we can acquire a better comprehension of the techniques and approaches employed in creating successful AI agents for Stratego.

Probe It won three times the computer Stratego World Championship in 2007, 2008 & 2010 [7]. It also has different levels of expertise and is currently available on the Android app called "Heroic Battle".

Master of the Flag It won the same tournament as Probe but in the year of 2009 [7]. This AI is available to play against on this website [11].

Stratego Tiny In the article [14], the author worked on an AI agent that learned how to play via deep self-play reinforcement learning and search using the ReBeL algorithm. All this experiment was done on a variant of Stratego with a smaller board with fewer pieces obviously: The Stratego Tiny.

Learning Stratego via CNN In [15], the author first created a baseline AI using a Monte-Carlo algorithm, for each possible move in a turn, it chooses a random board state consistent with all known information about the opponent and simulates a number of games from that state for 10-50 turns. Then, it averages the value of the resulting states, where the value is determined to be large if the player wins, small if the player loses, and moderate otherwise, with small bonuses for having more pieces of value left than the opponent. This algorithm is far worse than a human player, as expected, but beats a random player 94% of the time in 100 trials.

After that he decided to break down the game into six layers of 10×10, to feed the CNN at the end: the first two are for the player's pieces, his immovable pieces (their bombs and flag), and movable ones (pieces ranks). While the four lasts are dedicated to the knowledge of the opponent. There is one for the opponent's known movable pieces (pieces ranks), one for his known bombs, one for his unknown movable pieces (which, notably, must not be bombs or the flag), and finally one for his unmoved pieces. Except for the pieces' ranks, all the other values are binary.

Then to create the real AI, a Convolutional Neural Network (CNN) is trained on the baseline AI data, which consists of 20,000 played games, capturing different strategies and their corresponding results.

Invincible Invincible [8] is another big name that no article on Stratego can miss. It has been the subject of a thesis so the number of ideas and data we can get from it is enormous. The following of this section will list some of these important thoughts.

This AI agent thinks about making plans via a black box rather than selecting a single move at a time, which is closer to human thinking and high-level play. It uses a matrix of guessing ranks of the opponent's pieces and each time a move happens, it actualizes and normalizes it again, to always have the right statistics.

Dijkstra is an algorithm for finding the shortest path between two cases, therefore it can solve problems like the detection of a path, the accessibility of the flag, and others. This algorithm is simple to apply on Stratego because the cost of a move is equal to one everywhere as we play on a board and the pieces move from cases.

It gives a heuristic for the value of a piece inside fights: It first gives a baseline value via the order of power, multiplying the value of the precedent one by a factor if the two players have a piece of that level. Then it rescales so that the highest piece has a value equal to one and the flag is given as value the number of pieces it has because it is much more important, besides that there is also: The mobility they have, the number of pieces they can capture, their special abilities, and some hardcoded value like 0.5 for bombs, the spy has half the marshal's value and the Marshal's value is reduced if the Spy is alive.

There is also a heuristic for the value of information in the game. In fact, revealing an opponent's pieces has a certain value. And there is a profit of losing a piece to a higher one if it discovers something new. This profit is obviously 0 if no new information was found. These values adapt for the endgame when the opponent has less than 15 moving pieces.

Lastly, the heuristic for the value of a state is calculated by taking the weighted sum of the values of all "Invincible"'s pieces and dividing that number by the weighted sum of his opponent's pieces. Making it so that less than 1 for this value means that the enemy is ahead. The weights are two for the highest piece each color has left after all the equal ranks are removed and one for all the other pieces. Resulting in an almost strict material advantage.

Chapter 4

Techniques

In this chapter, what will be discussed are the techniques utilized or experimented with, alongside the methodology applied during my experimentation, that leads to the resulting AI agent. Among the diverse mechanics explored, only a subset gave meaningful results or had a positive impact on the gameplay performance. To understand this progression, we will first focus on the techniques that were employed in the final version of my AI agent and subsequently discuss the others that did not prove successful.

Before delving into the details of the different techniques, it's interesting to present the methodology that guided my approach to the problem. After successfully creating a basic AI that consistently won against the random player, I decided to follow a well-defined workflow based on the valuable guidance of my advisor and insights from the training process of DeepNash, which is evaluating each iteration versus a sample of bots. The methodology that guided my agent's enhancement from its first milestone and throughout the majority of my experiment was structured as follows:

1. Play 10 games against various agents from the bot pool.
2. Employ an automated script that compiles and gathers data and observations about these games. This information includes the win rate of our AI against each opponent, the time taken, the number of moves needed, and the number of pieces left for each side. And additionally, it produces two types of graphs separated each time between the winning games and the losing games. First the evolution of the state's evaluation throughout the game length and then the accuracy of the generated states compared to the actual ones, also throughout the game.
3. Analyze the results of the script and derive several additional things: If the games were close or not depending on the number of pieces left for example.
4. Examine and replay each seemingly important game or at least a representative random sample. This analytical step allows a better comprehension of our agent's behavior, highlighting potential advantageous moves that it didn't take or disadvantageous moves it made. Alongside the replays, something implemented late in the experiment but that proved itself highly effective to understand the choices of our bot, is integrating a log file to store statistics for every move executed by the AI. Such statistics are for example the generated state, the actual state, the root's child nodes with their values, and the action chosen.

5. List potential changes that can enhance the agent's behavior based on the analysis and implement these in the code. When in doubt about the outcomes, conduct additional runs for validation purposes.

6. Reiterate the process from step 1.

By carefully following this methodology, I was able to enhance the performance and decision-making of my AI agent for the game of Stratego. However, sometimes assessing the impact of changing factors such as parameters or even introducing new ideas remains challenging. This complexity comes from the huge variability that can occur across 10 games, leading to possibly different outcomes even with the same parameters. While further examination of selected games by manual analysis helps in solving this issue, it demands a significant investment of time. Paradoxically, the intention to speed up the process by running fewer games is countered by the need for additional analysis. However, these analyses are indispensable, as relying only on global results isn't reliable enough to conclude the efficiency of a change.

4.1 Monte-Carlo Tree Search

The Monte-Carlo Tree Search (MCTS) algorithm is a type of best-first search using a tree structure with the nodes being a state and the links between them being the move chosen to go from one state to another. It will use simulations of the game at each node it selects to gain global insight into the direction the game takes via these nodes. These simulations are often played randomly whether it be the MCTS agent's turn or the opponent's one. While one random simulation doesn't give any knowledge of the chosen moves's worth, the principle of having enough simulations done, even if they are random, will retrieve a global good strategy.

This method builds and makes use of a new search tree of the game states each time the agent needs to select a move. The root of the tree is the actual state of the game and the best move search then consists of four steps, as shown in Figure 4.1 that will be repeated a number of times set by a parameter. After all these iterations, the move finally executed by the agent in the actual game, is the one corresponding to the most explored child.

1. The selection: From the root, the algorithm will choose the next leaf node to expand. This choice is done via a technique called the Upper Confidence Bounds applied to Trees (UCT) that balances the exploitation and exploration with a factor called the *Uct_c*.

2. The expansion: The node previously selected expands one of its children, creating a node and adding it to the tree structure.

3. The simulation: From this newly created node, the simulation of a game begins and continues till the end of it, whether it is a win, a loss, or a tie. Another method is to simulate the game till a predefined number of moves, which will be the case almost obligatory in Stratego as the length of a game is high. These simulations can be entirely random as stated before, but they also can use an heuristic to obtain a better level of play. We can choose the number of simulations we do there, we often call them Rollout.

4. The backpropagation: The value of the node is computed as the average of the

outcomes over the simulations done. While the outcome in case of win is 1, loss is -1, and tie is 0, if we didn't finish the simulation, we need to evaluate the resulting state obtained. To do so, the most basic way to do this is to count the material advantage and rescale it between -1 and 1 to keep the computation correct towards the classic outcomes. The value and the number of simulations done are backtracked to the parent's node till the root.

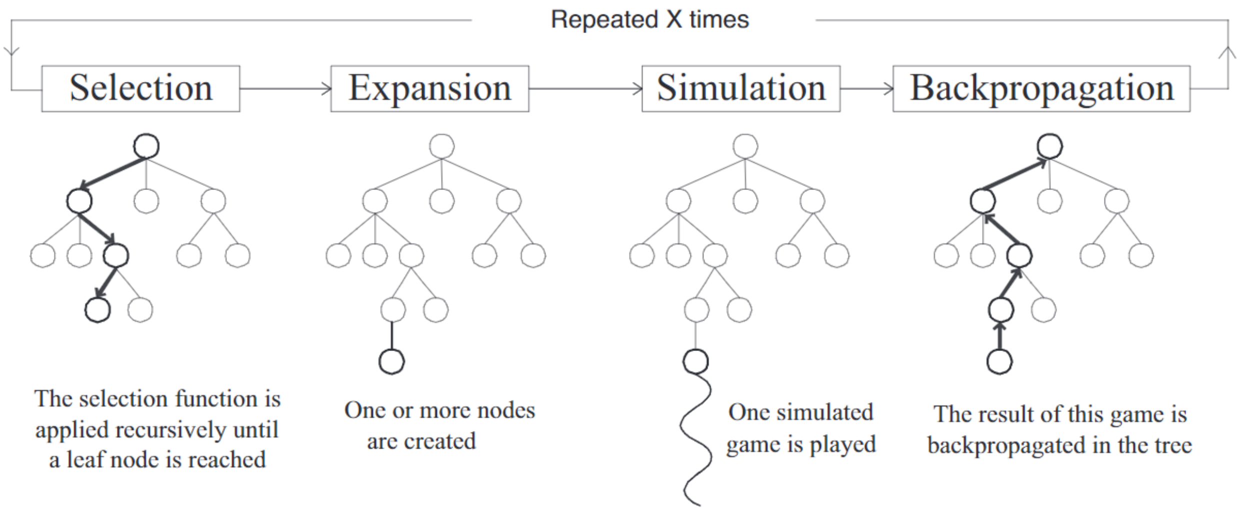
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 4.1: The four steps that describe the Monte Carlo Tree Search [19].

For the implementation, I took the one from the Open_spiel repository [4].

The general advantages of the MCTS algorithm lie in its straightforward implementation and its ability to come up with a solution without information beyond the set of moves it can do and the end-game conditions. But more importantly in our case, it offers a search of the state tree that is more vertical so theoretically a faster one to the solution. Because Stratego has this particularity that its state tree expands exponentially to lots of very similar states so a search like vanilla Minimax is highly not effective.

The disadvantages and limitations of using this approach are principally related to the great number of iterations and simulations needed to achieve efficient move selection and reduce randomness, resulting in speed issues. Also, the tree can grow significantly depending on the parameters set, leading to potential memory issues, even more, if combined with a system to save the nodes of the tree between the calls to the search. Selecting the most effective parameters for this technique is challenging since there is limited guidance on the impact of each change. Furthermore, due to the algorithm's randomness in rollouts and expansion choices, overfitting issues may occur. Even if we set the probability of certain moves to 0 via the prior, these undesirable moves might not be entirely eliminated from the pool of possibilities that the algorithm will expand and maybe select.

Some problems are more related to the nature of Stratego itself. Since the algorithm requires a possible state to play the simulations correctly, and it's a game of imperfect information, we first need to create a valid state before running the algorithm at each turn. Creating a valid state from the information we have is a realizable challenge thanks to some logic and opponent modeling. However, fixing the state suffers from inaccuracies, that is to say, the misses in the generation, especially when we increase the number of moves simulated in advance. Each move we simulate increases the uncertainty of what is the reality of the simulation in addition to the fact that we need to predict well our future behavior and the opponent's future behavior. Each simulated move adds uncertainty to the reality of the simulation, making it harder to predict both our future behavior and the opponent's future behavior accurately. But on the other hand, missing the prediction of the opponent's behavior also distances us from the true state of the game.

4.1.1 Parameters tune

Inside the agent directly, there are several parameters we can tune depending on what one wants to favor, these are typically:

- The *Uct_c*: A parameter that will balance the expansion choice between an evaluation factor and a discovery factor. It is balanced between 0 and a higher value, 0 cancels the discovery factor, hence, favoring only the weight of the wins to choose the next node to explore while otherwise will favor more and more the under-explored nodes. The theoretical value is $\sqrt{2}$ but in practice, it is found empirically.
- The *number of simulations*: This parameter will set the number of nodes to explore inside one step. Choose one move at a turn.
- The *number of rollouts*: Inside a node, the number of simulations of some moves in advance from a state. Each node will have this number of rollouts.
- The *number of moves in advance*: One simulation will simulate the game during a certain number of moves defined by this parameter. The evaluated state is the resulting state after this number of moves.

Surely with infinite computation power and time, the hardest constraints, these parameters could all go way up and the result should be more accurate. But since this scenario is practically impossible, we have to manage a reasonable time of thinking with good results it is to say, results that don't take random choice nor take the first good node it sees without considering other options.

4.1.2 Basic Monte Carlo Tree Search (MCTS)

This implementation is the vanilla version of the Monte-Carlo algorithm available on Open_spiel [4] used as a comparison with the custom version I made. It utilizes a default evaluator absolutely not tuned that has two needed functions: A prior function, assigning equal probabilities to all actions available at a given state. And a state evaluation function that involves conducting random simulations that extend to the end of the game to then backtrack the game outcome. Since these simulations involve multiple iterations, and each move within them is selected randomly, they are commonly known as random rollouts. We used the game modeling and state generation of my custom bot because MCTS needs a real state to be played in this configuration.

Therefore, it is important to acknowledge that this approach suffers from great inefficiencies. Having to repeatedly simulate the entire game just to make a single move highlights the computational burden and the suboptimal nature of this vanilla implementation. These inefficiencies become more apparent as the complexity of the game and the size of the state space increase, as is the case with Stratego. So I did only the minimal tuning to allow having a baseline MCTS in my experiment. It is to say, stopping the simulation after 10 moves chosen randomly and evaluating the resulting state via a basic material gain.

I selected this implementation to highlight the difference between the modifications I made to the MCTS algorithm.

4.2 Tuned MCTS

The Monte-Carlo Tree Search (MCTS) algorithm, as presented in Section 4.1, demands some requirements for its execution. To recapitulate, in order to traverse the *Tree of States* and make the simulations, the algorithm requires a possible totally discovered state as close as possible to the actual game state behind the hidden information. This state-generation process is initially characterized by lots of changes, but as it gains insights into the enemy pieces, the generated state becomes more consistent. This will be further discussed in Section 4.4.

Before continuing, here is a graphical representation of the components of my AI agent, for specific details about each component follow the sections indicated.

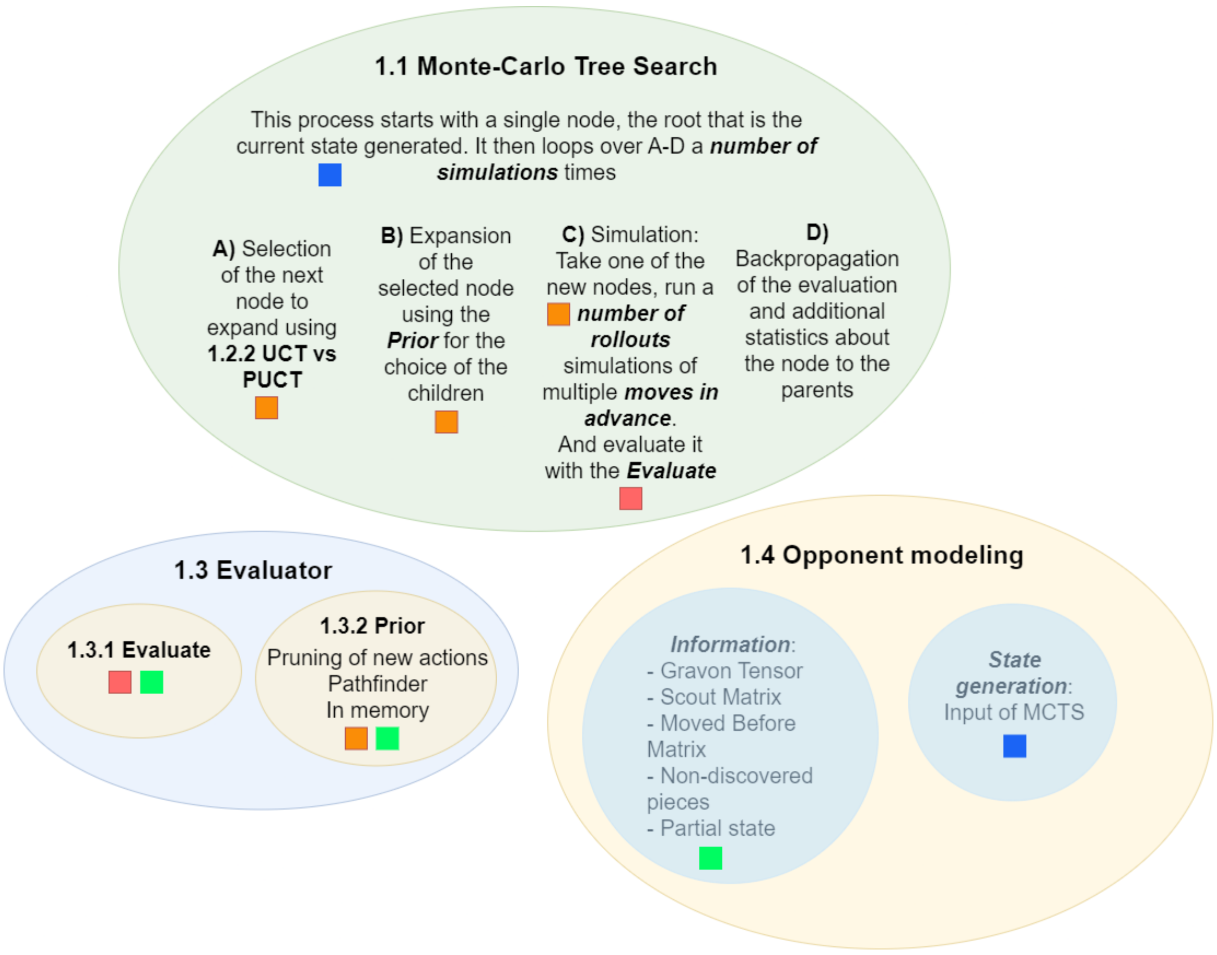
1.1 Monte-Carlo Tree Search
This process starts with a single node, the root that is the current state generated. It then loops over A-D a number of simulations times
A) Selection of the next node to expand using 1.2.2 UCT vs PUCT
B) Expansion of the selected node using the Prior for the choice of the children
C) Simulation: Take one of the new nodes, run a number of rollouts simulations of multiple moves in advance. And evaluate it with the Evaluate
D) Backpropagation of the evaluation and additional statistics about the node to the parents
1.3 Evaluator
1.3.1 Evaluate
1.3.2 Prior
Pruning of new actions
Pathfinder
In memory
1.4 Opponent modeling
Information:
- Gravon Tensor
- Scout Matrix
- Moved Before Matrix
- Non-discovered pieces
- Partial state
State generation: Input of MCTS

Figure 4.2: The components of my custom agent with sections.

Here, the potential problem that can occur is that the agent overly trusts the generated state it operates on as an absolute representation of reality. Leading to some risky behavior and audacious actions that need to be fine-tuned. This is particularly relevant in the context of MCTS, with the expansion phase that could explore nodes and perceive it is winning fights when it is not, thus rigging the evaluation of the next nodes. The pruning subsection 4.2.3 contains some solutions to this problem.

Same in the creation of heuristics, it is preferable to rely on statistics over the possible pieces at each square rather than assuming the guess is 100% correct. Because even if decent results have been reached in this area, thinking of achieving a perfect 40 correct guesses per game remains idealistic. To give an order of idea, if the 40 opponent's pieces are unknown, as it is at the beginning of the game, there are approximately 1.41×10^{33} possible configurations[1]. It is worth noting that although certain setups in this great number have a higher probability of being picked and some patterns may be more likely, humans often like to introduce some strategic mind games into their creation, making them even more uncertain and unpredictable. Indeed, if it was possible to know for sure the entire opponent's board from the start, the interest of the game would be lost, and a player in possession of such complete knowledge would invariably win all his games because Stratego is easily solved if you have this information.

The search for a single best move that needs a specific sequence is not a good idea, as the opponent can ruin such strategies during each turn by picking a move outside this sequence and there are lots of different possible moves. The approach of Mont-Carlo should help the search go in the right direction rather than the single best move with all the different simulations and explorations of the tree it does, backtracking outcomes to the root node.

This was one aspect of MCTS's tuning done to serve its efficiency in playing Stratego. In the following subsections, other dimensions will be explored, including the parameters with their resultant effects, a modification of the formula to elect nodes during the selection phase, and the utilization of a pruning process.

4.2.1 Parameters

The Monte-Carlo Tree Search algorithm has several adjustable parameters that can be tuned with respect to the desired duration allocated for the search process, as well as the way in which the tree exploration performs.

The parameters selected as the final result for this thesis are reported in Table 4.1. This configuration balances a reasonable time of thinking per move of 0,55s with some real solution to the problem of making an AI agent play the Stratego.

Parameter	Value
Uct_c	0.2
Number of simulations	50
Number of rollouts	9
Number of moves in advance	5

Table 4.1: Parameters of the Custom agent

Several alternative configurations were tried but the majority of them did not catch up and failed to show promising outcomes. Despite having multiple configurations that had

[1]This result is obtained directly using multinomial coefficients.

comparable results, I preferred lowering the exploration values to ensure a reasonable time for each move as the other AIs do the same.

Each parameter is interconnected with the others, increasing the difficulty of setting them efficiently. Consequently, modifying one parameter could lead to an increase or decrease in the others. For instance, increasing the *moves in advance* and more *rollouts* will be needed to certify the result inside a node, and not only be an overfitting delusive evaluation. On the other hand, these increases mean that less *simulations* are needed to observe approximately the same number of steps in the future. The trade-off here is between expanding more nodes, where each node offers a limited glimpse into the future, versus expanding fewer nodes, but with a more extended simulation within each node.

Putting a high value for the *moves in advance* parameter proves counterproductive, given that the board generation process is not consistent. Consequently, the computation spends time on things that are likely to never happen. In addition, nothing tells us that the prediction of the opponent's behavior is correct either. Augmenting the number of moves will increase the imprecision of our vision of the game, however, the computation time will also increase. It's important for the simulations to correctly operate, that the number of *moves in advance* is at least higher than two. While I initially experimented with three as a baseline, experiment results revealed to have better outcomes.

Another legitimate question one might ask is whether the parity (even or odd) of the parameter *moves in advance* could affect the algorithm's behavior. Indeed, it is very plausible that the player who makes the final move in the simulation slightly takes advantage, especially for low values of this parameter. This is exactly the horizon effect that is presented here, stopping the simulation just before seeing a decisive move. However, this issue is naturally smoothed by the simulation and evaluation process that is repeated for each node and backtracked to the parent.

The *Uct_c* parameter is set at 0.2 because the exploration has been pruned so that the remaining nodes are great and can all be explored. Consequently, the only factor that can really differentiate them and change the node selection is their evaluation.

Moreover, setting the parameter very low helps distinguish the evaluation of the states that may have their values close to each other. This chosen parameter's value put more weight on the exploitation of rewards rather than the exploration of new nodes.

4.2.2 UCT vs PUCT

The Upper Confidence Bounds applied to Trees (UCT) as presented before is a formula that is used to choose the next node to expand in the search process. It balances the exploitation and the exploration via a parameter named *Uct_c* that takes values greater than or equal to 0. The value 0 will favor the exploitation of already well-evaluated explored nodes and a higher one will favor the exploration of new under-visited nodes. The formula for UCT is given by

$$\frac{R_i}{n_i} + C\sqrt{\frac{\ln(N_i)}{n_i}},$$

with R_i the total reward of the node i, n_i the number of times it was visited, C the parameter *Uct_c* controlling the relative importance between the two terms and N_i the number of times its parent was visited. The agent finally selects the node that has the highest value regarding this formula in the selection phase of the algorithm.

But there exists a variation called Predictor Upper Confidence for Trees (PUCT) as used in Alpha-go [20] that adds the dimension of the prior function defined by heuristics. Therefore a relatively good prior function helps a lot in the exploration of the tree, at least that's what has been observed in my experiment and that's why this function was the chosen one. The formula for PUCT is

$$\frac{R_i}{n_i} + C\,\frac{P_i\sqrt{N_i}}{n_i+1},$$

with the same algebraic notation as before and P_i for the prior value of selecting this node from the parent node. The selection operates the same way by taking the node with the highest value.

4.2.3 Pruning

The algorithm has a chance to expand each node, every action that can be taken from a particular state. As highlighted earlier, the number of distinct actions feasible in every turn is part of what makes the *Tree of States* so large. The concept of pruning unpromising moves seems interesting for several purposes. Beyond enhancing search efficiency, it allows a better overview of the game faster. Pruning is the action of totally removing a child state from a specific state when checking all possible actions accessible to the agent.

While trying to reduce the presence of undesirable actions within the search, the introduction of a zero prior to such actions proved insufficient to prevent their selection by the algorithm. This is due to the underlying expansion formula, which does not always take into account the prior value, and even when it does, it may still opt to choose them anyway at some point. Therefore pruning these completely from the pool of choice from a state is the solution. These actions are recognized as blunders in the game so we avoid doing them, assuming the opponent will not either.

Furthermore, the pruning procedure is important on the evaluation side of the algorithm. MCTS will take for granted the generated state given, leading the AI to prioritize capturing an opponent's piece, even if it is unknown, if it perceives itself to win the encounter because it gains a material advantage and increases the evaluation value. To counter this behavior, a balanced approach was required, involving weights on statistics and a reduction in the value assigned to these high-risk actions. This adjustment enabled the pruning process to effectively remove such actions in unnecessary situations.

The suppression of unfavorable nodes reallocates the probability weight to the remaining actions, ensuring the sum of probabilities remains equal to 1. Consequently, this rescaling process will favor more promising or at the very least neutral actions. Even if sometimes there is no superior move, it happens that the remaining options are all of the same probability.

The difficulty lies in choosing the right balance in pruning to reach an equilibrium between its impact on the search process and the reliability of the different action evaluations. Over-pruning can potentially harm the algorithm's ability to find optimal solutions while keeping too many nodes will flood the exploration with lower-value moves. Thus, it is very important to find the optimal amount of pruning for reaching optimal results.

All this pruning is done in the prior function. The prior first gives a value to each available move and then the pruning eliminates all the entries under their computed mean. Hence, moves that have low value are removed directly but if, in the remaining actions, there were more than five actions, the fifth highest value is taken as a further threshold and all actions under this new threshold are removed too. Following this process, most of the time the number of possible moves is the same, five, but it can be more if multiple moves had the same value. It is better to know in advance the number of nodes each node expands so that the exploration of the tree stays the same as much as possible throughout the game.

4.3 Evaluator

The evaluator is a mandatory but external part of the MCTS algorithm. This component is composed of two methods, one is the *evaluate* that will give a value to a specific state, and the other is the *prior* that returns the possible actions associated with their probability to be chosen from an input state. It is used throughout the execution of MCTS when expanding new nodes and evaluating them.

The use of statistics, probabilities, and information outside the exact state is preferred over the generated state as it can contain small errors that we don't want to rely on. For instance, using the number of pieces remaining is the same for all possible states so the information is always good but rating the position of specific pieces can lead to misunderstanding the quality of a state during an evaluation.

That being said, the different topics and justifications about the *prior* discussed in the previous sections will not be repeated. The focus of this section is to describe what composes the *prior* together with the *evaluate* and their requirements.

4.3.1 Evaluate

The evaluation is divided into two principal parts. The first one is the node evaluation itself, a concept that is taken from the vanilla MCTS, but adapted to suit the requirements. The principle remains the same: the algorithm performs simulations of the game, named rollouts from a specific state. Each rollout ends on either a final state where a winner and a loser are determined or with an intermediate state after the predefined *number of moves in advance.*

In case we reached a final state, the rollout's value will be either 1, 0, or -1 based on the game's outcome. Otherwise, an additional evaluation of the resulting state is needed to assess its value in the form of a continuous reward ranging between -1 and 1. The next and last step in the *evaluate* function consists in aggregating all the rollouts' values and averaging them by the number of rollouts done to give a final evaluation value to the node. Inside the average computation, a weighting factor to the final state's outcome is introduced to favor the end of the game if a path to the win is identified.

The second part is the evaluation of the resulting state from each rollout. This evaluation resembles a reward function since simulations till the end of the game are not viable for larger games like Stratego, as MCTS usually does this on smaller games. This part is the most important since the majority of the simulations do not end in a terminal state. The remaining of this section will describe the techniques used to evaluate a distinct state.

Basically, individual scores are assigned to each player using some heuristics followed by the computation of the difference between these scores to acquire the evaluation of each player for this state. This difference is subsequently rescaled to a range of -1 to 1, with respect to the value of the outcome used earlier. Firstly there is the consideration of material advantage, but not a straightforward difference between the AI's piece count and that of its opponent. Instead, it takes the form of a weighted sum depending on the ranks of each piece.

These weights remain static as otherwise dynamic changes in their value can harm the decision-making process of the agent. Although the real value of each piece might change throughout the game with the remaining pieces, the agent tends to make weird choices if the weight of a piece is adapted. For instance, if the weight is modified based on the number of opponent's pieces that a piece can defeat, it leads to misleading evaluations. In this scenario, capturing a piece could reduce the weight of our own piece and consequently create a less valuable evaluation, despite successfully capturing a piece.

Therefore in the evaluation, the weights should remain constant throughout the entire game to preserve decision integrity. However, this technique could be used in the prior function to introduce a new dimension to the value of the fights, helping the comparison of different moves.

Table 4.2 presents the different ranks and their value in my implementation. The different pieces' weights are there to differentiate their importance for the outcome of the game. I decided that in the *prior*, there was no priority to one fight to another even if the importance of pieces is not similar, only its outcome is important. Hence, to make the AI choices better, I tuned them in the evaluation. The values were tested and thought carefully by myself with few inspirations.

Name	Value
Flag	2
Bomb	0.5
Spy	0.1
Scout	0.1
Miner	0.3
Sergeant	0.3
Lieutenant	0.3
Captain	0.5
Major	0.5
Colonel	0.8
General	1
Marshal	1

Table 4.2: Summary Table of the pieces' value for Stratego in the implementation

Secondly, it values the protection of the flag, that is to say, that each of the four possible squares around it must be occupied by an ally or at least not accessible.

Lastly, positioning pieces on the opponent's side of the board gives more value. The double effect of this factor is that our agent is encouraged to move forward on the board, and secondly, there is an incentive to defend its side of the board by preventing the coming of enemy pieces.

4.3.2 Prior

The primary objective is to cancel evidently disadvantageous moves and to achieve this, we assign very low values to such actions during the prior. Specifically, we minimize the use of moves that violate the two-square rule, go into dangerous places, engage in unfavorable fights regarding the probabilities, make early moves of more than one square with a scout to avoid giving up the information, and make use of the miners prematurely. Miners find their real value in the late game, where their role in disarming bombs to find the flag becomes decisive.

On the other hand, favorable values are given to moves with high possibilities of winning combat, actions that allow discovering new pieces when already in a dangerous position, actions that involve the scout's role in piece discovery, and moves that follow fights that are expected to result in victory.

In the fights, the priority is given to the chances of winning the fight via a weighted sum which is the result of multiplying rank probabilities by their potential outcomes. The importance of a piece isn't considered in this context but as talked in earlier about the material advantage, a dynamic value for the pieces could be used there. This could enhance the prior behavior, favoring the payoffs of fights, hence allowing a better comparison of different moves. For the moment the payoff of such actions is more shown in the evaluation function.

However, due to the high probability of winning as a high-value piece, this scenario needed a risk factor. Consequently, moves that try to capture a target that has not yet moved with a high-value piece are forbidden. The fights can be direct if the pieces involved are next to each other but there's also a balance of weight for the chase of such confrontations or the flee of them.

Keeping a sense of balance among the assigned values for all these moves is important, it allows us to ensure that they remain within a comparable range of magnitudes.

The purpose of the prior function is two-sided: it aims to anticipate both the opponent's actions and our own future moves. Predicting random moves for the opponent proved to be inconclusive, at least for so few rollouts, given that there exist obvious actions, such as engaging in winnable battles, that are considerably more likely to occur no matter the enemy's strategy. During simulations and priors, we adopt the position that the opponent has complete knowledge of all our pieces, thus preparing for a worst-case scenario.

4.4 Opponent modeling

Opponent modeling is the concept of modeling the opponent in every aspect. In the case of hidden information like in Stratego for instance, it helps model the opponent's part of the board, doing statistics over the possibilities or fixing the state. Additionally, it is also the process to understand the enemy's thoughts to model and predict his future behavior. The goal is to get as much information as possible on the opponent, on the game itself and how is it played generally to be able to predict the opponent's choices and future behavior the best possible.

The prediction of the opponent's moves is the same as our gameplay plan but applied to his side. It predicts that the enemy will play the same way as us. With the exception that we make the assumption that the opponent knows our board perfectly.

In the first subsection, all the different information, observations, and structures that gather them are discussed and then the second one presents the state generation, how it is done, and what it brought to the performance.

4.4.1 Information

There are pieces of information that the agent uses inside the modeling itself for the generation of the state, but also in different other parts of the MCTS algorithm to predict the opponent's behavior.

The first structure makes use of statistics from a dataset of over 31500 games played on the Gravon platform which is available on their website free of charge. The structure takes the form of a tensor with dimensions $10 \times 10 \times 12$, hence, describing the board as a 10×10 grid with each entry corresponding to the probability to be each of the 12 ranks.

These probabilities were derived from the initial placements of pieces in each game. To gain the data of both players for each game, I operated a central symmetry mirroring its setup, resulting in a compact tensor size of $10 \times 4 \times 12$. This tensor is applied to the right side of the board based on whether the player is positioned at the top or bottom. Entries not corresponding to an opponent's piece remain as zero arrays. As the game progresses, this initial positioning is dynamically updated to reflect player moves and captures. This whole process creates the tensor of dimension $10 \times 10 \times 12$.

In addition to this tensor, another interesting statistic came out of these games. The average number of moves needed to end a game was about 330 moves, which is close to my agent's which has an average of 350 moves when there is no tie caused by the limit of moves. Since the ties are declared at 1500 moves, it rigs the average.

Subsequently, the agent has some additional knowledge of the game than the tensor of statistics. It keeps the number of all non-discovered pieces alive in an array. Discovered ones are kept directly in the state but have no entry in the matrix of stats, therefore we know they are sure generation. Next, we have two binary matrices of dimension 10×10 with their values at one in case the piece at this place has already moved before in the game or in the second matrix in case the piece has already moved for more than one square at once, hence revealing its nature of scout.

The different aspects of the information are separated into modules instead of gathering all inside a big matrix that updates everything and rescales the probabilities directly. Therefore, for each move and when needed, that is to say, the discovery of a piece, its capture, or even just an opponent's move, the update of the knowledge must be done on all the individual structures. While this may seem not optimized, it allows for the other components of my AI program to use only the needed part of the information. The probabilities computation is done only when needed.

4.4.2 State generation

The knowledge of the game previously described allows us to generate a possible and complete state of the game as accurately as possible for each agent's turn.

In the context of this particular game implementation, employing a complete state is imperative when utilizing vanilla MCTS. Consequently, the generation of the state process was indispensable. Allowing to conduct simulations on a board with partial information would have led to a totally distinct experiment, although the knwoledge of the game would have remained unchanged.

The process of state generation is a combination of the use of various sources of information, including previous moves that led to this current state, the remaining pieces, and the game statistics from Gravon's games. The generation of new boards contains several steps, at first, it replicates the known squares based on our own pieces, the empty spaces, the rivers, and the revealed opponent's pieces. After, it begins by generating scouts and all the pieces that have previously been moved, applying probabilities, and finally filling the rest of the board with pieces that have never moved before, using different probabilities to include the flag and the bombs in the possibilities.

A particularly effective improvement lies in consistently placing the flag in a strategic and logical position. Given that the majority of players tend to position their flags defensively on the last row, our AI agent now always generates it there. This change helps a lot as it prevents the AI agent from mistakenly targeting the incorrect flag and, consequently, risking its pieces rather than playing safely as usual. While the impact of this generation might be less pronounced pieces other than the flag, the overall results are better.

The following Figure 4.3 shows the evolution of the total accuracy of the state's generation throughout the game. It is averaged on a sample of games, and we see the difference between when the agent wins or loses. It's caused by the fact that the faster the information is obtained, the faster and the better chances of winning. We also see the high variance due to the accuracy being so precise and without shade, either you hit the right rank or not. The important part is that the ranks correspond more or less to the actual ones but predicting a sergeant instead of a lieutenant is way less punishing than predicting a flag instead of a Marshal.

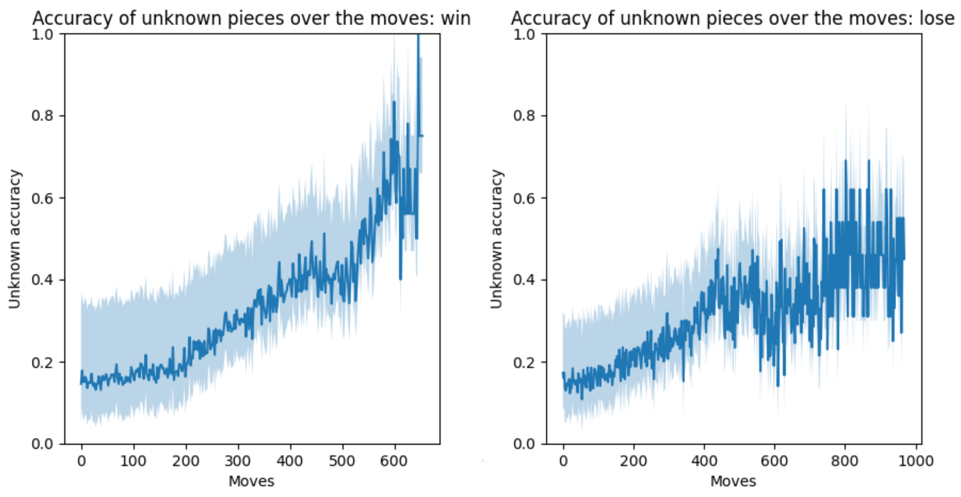

Figure 4.3: Comparison of the evolution of the state's generation's accuracy over the length of the game. The dark blue is the mean value over the selected games while the light blue is the upper and lower standard deviation.

4.5 Setup of board used

As it was not our main focus, we decided to use a setup pool, that is to say an outside selection of efficient boards instead of generating one each game. The creators of these setups are players on the Gravon platform. The starting boards were selected via multiple criteria: They were winning, respecting the prerequisites of a strong board of Section 2.2.1, having different game lengths from each other, and being made by different users to allow more variety. But it wasn't enough to ensure these boards had no big disadvantages when played at the same gameplay level. In addition, a human verification of the early result of each setup was made to remove the boards that appear to have bad positions and lose more than usual.

In my configuration, the program will select randomly from the selection two half-boards of forty pieces and create the initial state of the game. The half-board for player 1 (top) is kept intact while the one for player 2 (bottom) is the central symmetry of it with the center set at the middle of the whole board. Thus, even if the same half-board is selected, it's not a simple mirror where they both would have the same pieces for fights and the game could be boring.

If a problem can occur here, it would be that a specific state disadvantages one player but not in general. To detect it, it would have needed to run lots of games with every combination of half-boards.

Fixing the initial state didn't seem like a good approach to the problem in the testing phase because it is not the conditions in which they will later be playing

4.6 Pathfinder

As Invicible's thesis [8] mentioned, using search algorithms for path discovery on a board is fairly simple because we just put the cost of moving at one and the available moves are the four cardinal directions. In this case, the board is a matrix and most of the pieces can only move one square at a time in a cartesian way, respecting the predicate. By the cartesian way, I mean that the move can only change one coordinate at a time.

This functionality has its utility in the fight finder, if there is an opponent's piece nearby, the moves that go towards this piece in case of great possibility of capture must be of a higher priority. As much as the ones that want to flee a dangerous encounter. That is how it is being used in my agent, each of my moveable pieces searches for the first enemy in the range of five moves, in case it finds one, it computes its chances of winning or losing and divides it by the square-rooted distance of the path. Then it gives a higher value only to the moves that take the path toward a victory or anything other than the path if it's toward a loss. Otherwise, if it didn't find one, a default behavior is chosen.

The limitations are first that it stops searching at the first enemy instead of searching for all enemies and then takes the path toward the most promising fight. And secondly that it does not take into account the rank of the piece moving, scout has obviously different options than the others but so do the higher pieces compared to the lower ones. Lower pieces can take the first fight they see but the higher ones need to choose which is the most worth. These changes cost more computation time as this operation is done lots of time, typically at each evaluated state. That is the reason I did not explore this behavior.

The type of search used is a Breadth-first Search (BFS), which means that from the current position of the selected piece, the search operates by levels or number of moves needed to arrive at a position. Hence, first, it explores all the cases available within 1 move distance, then 2 move distance, and so on. If it finds one opponent's piece, it early stops the search and returns the fastest path between the two positions.

4.7 Prior in memory

A valuable optimization, deriving from the high probability of evaluating the same states multiple times during rollouts or the game in general, involves storing *prior* computations in memory. This event has a high probability of occurrence because of the rollout that starts at the same state each time. By reusing previously computed results as we encounter them again, this technique decreases the time needed per move in a game. Early on in my experiment, this approach led to a reduction of approximately 60% in computational time at equivalent parameters.

Logically this process is achievable on the two major methods, the *prior* and the *evaluate*. However, in the tests done, the percentage of *hit* of the *evaluate* was less than three percent. A hit is an event where you encounter a state that you already computed and stored. Additionally, the number of *prior* computations was generally fifty times the evaluations' unless the game was stuck between two states and our agent had a lot of information so that the generation wasn't changing so much. Therefore the memory usage of such state evaluations was not worth the time gained by the few hits over the game. Furthermore, the simulations are an important part of the MCTS algorithm, and reusing instead of evaluating them again may cause harm to the choice in the end.

The *prior* on the other hand had 70% of hits since the rollouts all begin from the same state each time. If the number of moves in advance increases this percentage decreases as the simulations go further and further off the actual state, reducing the likelihood of seeing these results again. For instance, with equivalent parameters, varying the parameter from three moves in advance to five decreased to 50 the percentage. Reducing the number of rollouts also decreases a little this percentage. Therefore in the final settings, this percentage was 45% most of the time.

Since the majority of the choices and rules are forced in the prior, at first, reusing these created a situation of bypassing the two-square rule, resulting in endless chase versus deterministic AI such as Asmodeus or Hunter. Hence it is important to add a condition to allow the hit to operate, the previous moves should not repeat themselves. This correction can be used as a general solution if any behavior mustn't repeat itself.

The implementation of this approach involves two components: the data structure for information storage and the way to distinguish entries within this structure. To ensure fast retrieval and efficient existence checks, I chose to use a Python dictionary. In this context, the key corresponds to the string representation of the state without the turn information, we keep only the complete board along with the player that must make a move. The associated value is the typical outcome of the prior function, that is to say, the different moves possible alongside their respective probabilities of selection. Notably, the implementation did not experience any major memory issues, even during games of maximal length. In the unlikely event of memory constraints, a solution could be to adopt an alternative key system, using a better hashing system.

Chapter 5

Results

This chapter is dedicated to the presentation of the result against the pool of bots. The custom AI won 70% of its encounters, lose a little more than 20% and let the rest be tied. It took in general 0,55 seconds of thought to make a move, evaluating 50 different nodes each time.

Throughout this section, I will present graphs but note that games have different lengths and that some graphs may also have fewer games for data representation even at the start. We average the data over a sample of selected games with respect to what we're analyzing. However, since the games have different lengths, only the start is a real average and it gradually loses stability with the fewer games in the computation. Most of the games have a length between 250 and 400 moves but it happens that they surpass this without getting to the limit of 1500. Those ties are not taken into account in the computation of the graphs, they are explicated by the fact that even if the two-square rule is respected, nothing stops them to chase themselves between multiple squares rather than only two.

The benchmark of over 1000 games distributed between 6 agents took three days and 9 hours to finish, excluding the games of *Demon Of Ignorance* that I had to half-play instead of having fully automated gameplay. This agent has only 10 games against my AI and 10 others against the second-best bot in the pool for comparison but without automating it wasn't possible to do better. All the others have 100 games against each other, 50 as the first player and 50 as the second one in case it was creating an advantage. The second-best one was chosen by the average percentage of wins only. Hunter had the best of all with 59% while Asmodeus and MCTS had respectively 51% and 49%.

The organization of this chapter is the following: first the global results, of my agent and the other agents in this whole experiment, then more specific analyses of the performance of the custom bot to finally go through the different sets of games versus each other AI.

5.1 Global results of bots

To read the Tables in the right way, take the row as the one we're looking at and the columns as the one we're comparing at. Each entry is composed of three numbers in the form "Wins/Ties/Loses".

RNaD as described before is suffering from several issues and is losing all its matches

...	Custom	Asmodeus	Hunter	RNaD	MCTS	Doi[1]
Custom		66/7/27	55/7/38	96/0/4	82/3/15	20/0/80
Asmodeus	27/7/66		42/21/37	99/0/1	38/6/56	
Hunter	38/7/55	37/21/42		97/0/3	64/9/27	0/40/60
RNaD	4/0/96	1/0/99	3/0/97		0/0/100	
MCTS	15/3/82	56/6/38	27/9/64	100/0/0		
Doi	80/0/20		60/40/0			

Table 5.1: Global results of bots

as the random player would do so it will not be discussed anymore.

5.2 Global result of my agent

Here are the graphs of evolution through the game with the whole data from all bots. Remember previously that dark blue is the mean and light blue is the standard deviation. And also the way the average is computed only in the games that lasted for the number of moves shown, therefore the first 400 moves are more accurate than the ones after.

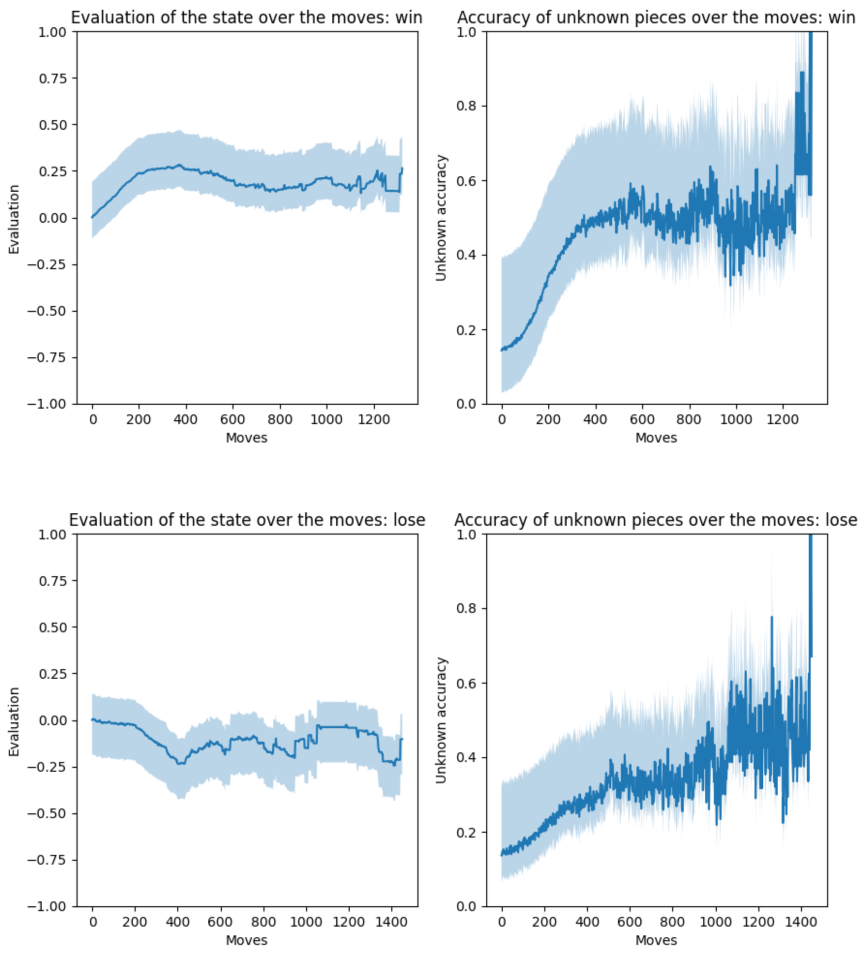

Figure 5.1: Whole data.

The global and final win rate that my custom achieved is 72% of wins, 21% of losses, and 4% of ties.

There are behavior that are less good like the chases that could occur in the game. The principal issues involve moving too much different pieces when all the moves are relatively similar in terms of weight, giving up lots of information away, and not committing to making real organized attacks. It is also inconsistent in its plan in certain situations like the flee of a piece, it can stop in the middle of it, losing its piece in this.

Against Demon of Ignorance, the late game was not well managed, often we would be in a situation where both of us can win and my custom AI will begin to lose slowly the advantage making wrong choice of fearing piece and then not advancing in the board.

For the mind game, bluffing against my AI is not good at all since it computes the statistics and takes the advantageous fights. Baiting on the other can prove efficient in ambiguous states.

The next graphs will each time be the games involving the custom agent and a specific bot only.

Custom VS Asmodeus

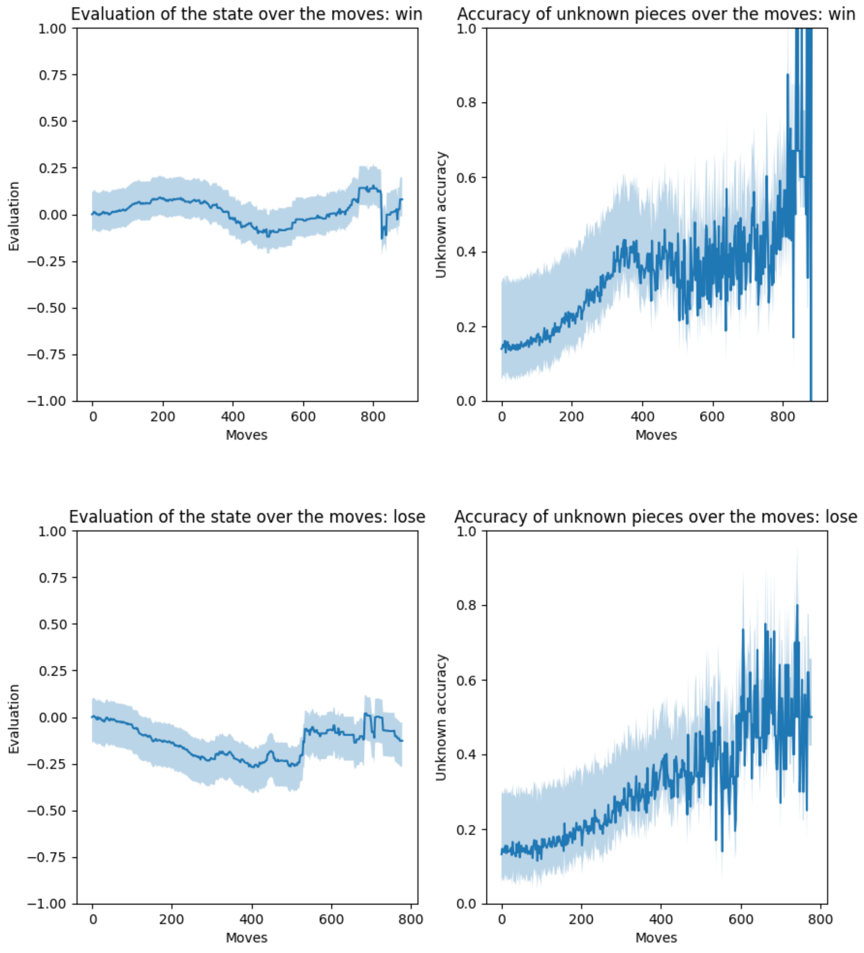

Figure 5.2: Asmodeus games data.

We see that Asmodeus has more difficulty when it’s not winning from the start, resulting in us taking advantage again later in the game. The accuracy is slightly faster when winning.

Custom VS Hunter

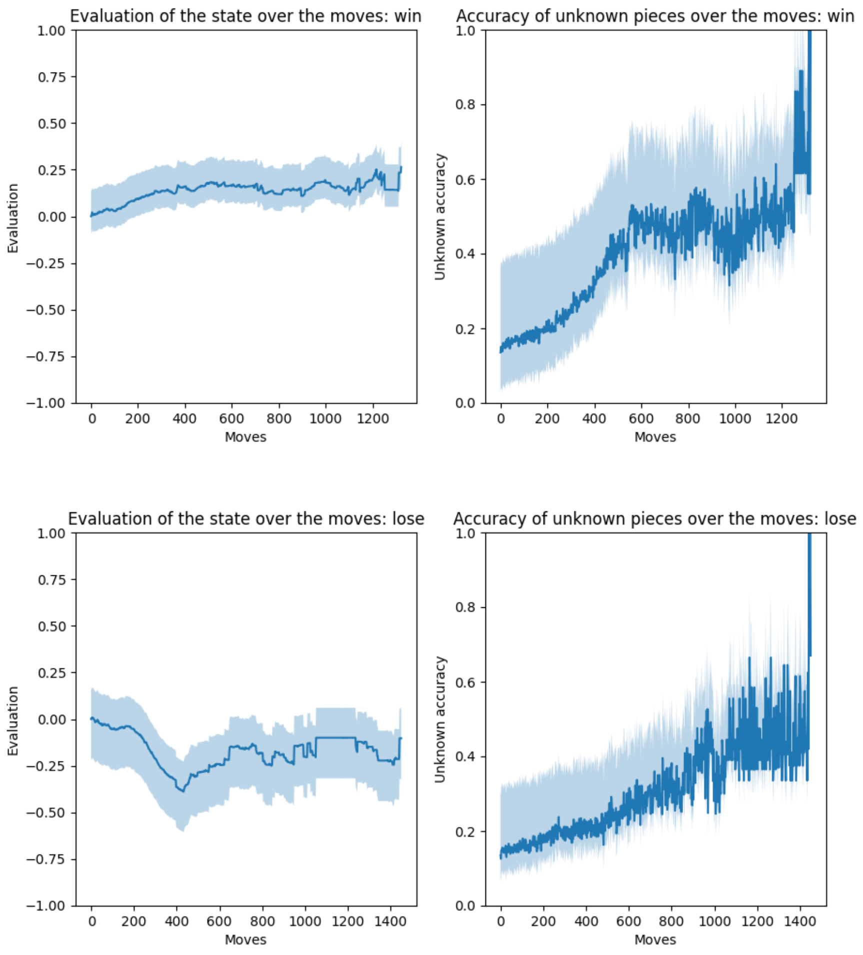

Figure 5.3: Hunter games data.

Overall we have the same result as for Asmodeus but here, when we manage to take advantage we usually keep it till the win.

Custom VS MCTS

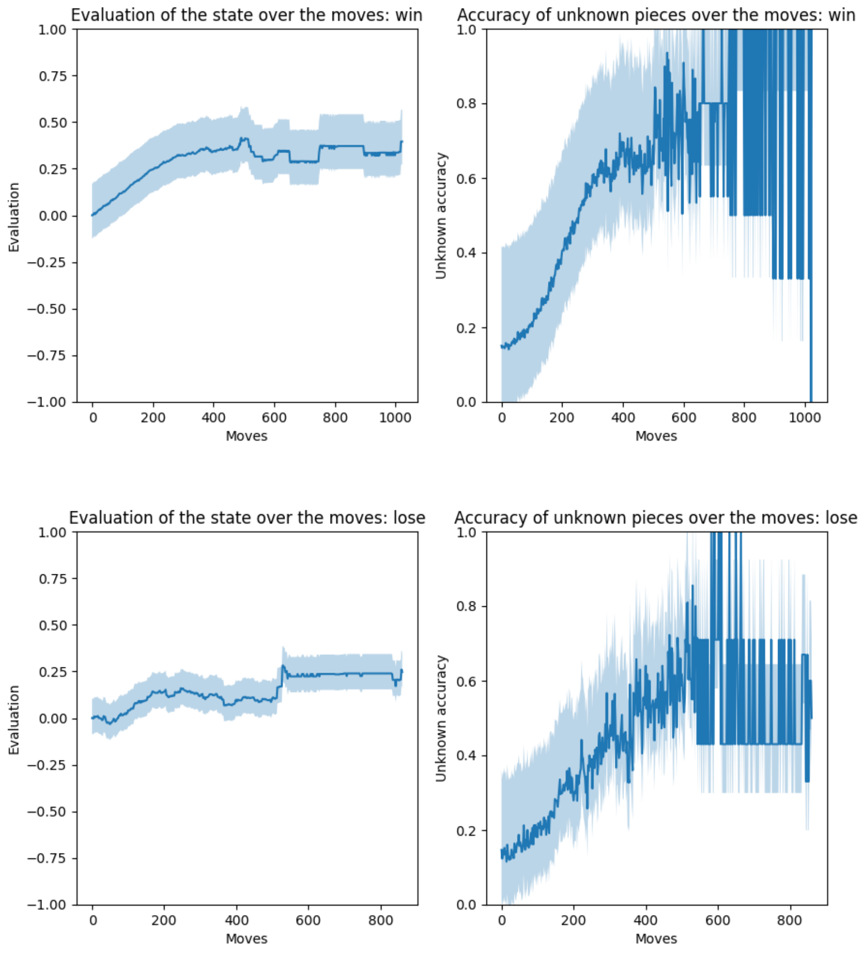

Figure 5.4: MCTS games data.

The basic form of MCTS has a really aggressive gameplay, taking every fight it thinks it can win. This can allow it to take material advantage in case of success but also give up lots of information and suicide pieces.

Custom VS Demon Of Ignorance

I had not enough samples in the graph of the wins, therefore it was not shown here. But the graph for the losses shows something interesting. Throughout the games, the evaluation remains really equivalent even if the accuracy is not good. So this AI manages to conserve the information of my agent and play defensively till the end of the game where it proves to be better.

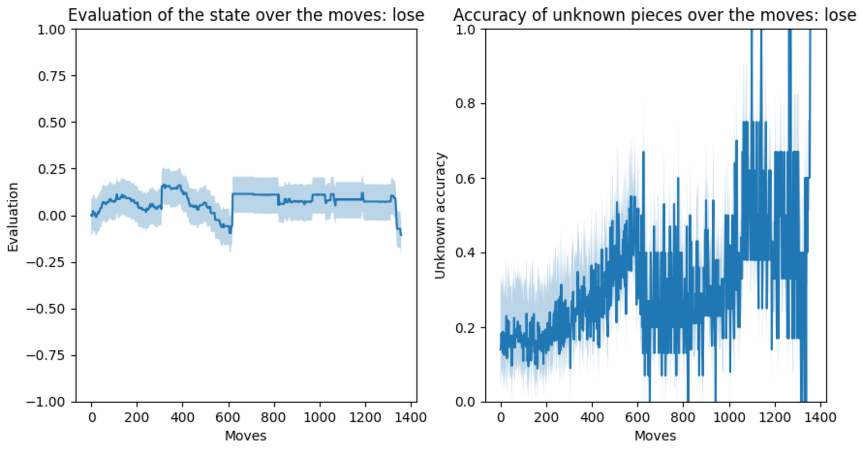

Figure 5.5: Demon of Ignorance games data.

Chapter 6

Conclusion

The goal of this thesis was to create an Artificial Intelligence agent for the complex game of Stratego. In order to achieve this, I chose the Monte-Carlo Tree Search algorithm, an option that offers an interesting exploration of the vast state space of this game. Therefore I examined the pros and cons of using MCTS and evaluated it against multiple other bots to ensure its performance. The imperfect information was partly solved using various techniques of the opponent's modeling but the future works go beyond these temporary solutions.

In this chapter, we will first summarize the results obtained by the agent before discussing its main advantages and issues. Consequently, present some possible continuation and future work that may improve the custom agent performance.

My implementation of MCTS achieved results against state-of-the-art agents from my pool of bots, a 70% of win rate on average in my experiment, proving its efficiency. However, I did not have the occasion to test it against humans or better AI. There are also lots of tuning possible already in the evaluation and prior of states. For instance, *Demon Of Ignorance* uses a similar approach that explores the *Tree of States* but has more complex evaluation and prior functions. Therefore the comparison purely on the algorithm and its efficiency does not stand, it must be taken as a whole.

In addition to its result, another good aspect of the MCTS exploration is that it focuses more on a global direction in the *Tree of States* rather than on a single node. The direction is encouraged by the simulations, the backtracking process, and the *UCT* formula that increases the chance of exploring promising nodes. Therefore, it relies less on the impact of one move and more on a move that takes the direction that will favor us the most in multiple cases.

The main issue that appeared recurrently is its lack of thinking. It relies too much on the state we fixed at the start of our turn which contains some errors almost surely. In my opinion, this algorithm needs something on top of it to reduce this behavior. In Section 6.1, two ideas that could help will be explained in detail. The counterpart is that they both add a layer of complexity and with it more computational power needs.

6.1 Future work

The conclusion of this thesis ends with thoughts on the possible improvements to the implementation of the current custom agent. Besides the optimization of the search on the hardware part or a possible multithreading process of the search, there is room for enhancement. The following subsection describes different aspects that can be improved.

There are potential ways to enhance the time taken for the algorithm to run. We could, for instance, use a single tree throughout the entire game, saving computations. This tree would be updated by removing and adding nodes adapting to the selected moves. Or a less costly method that aims at the same strategy, storing only the children of the selected move at the end of a turn to reuse it at the next one. The goal would be to keep the computation and the number of visits of the nodes to accumulate weights and keep coherence between turns. Both these methods could either improve the time of thinking and help our agent follow a plan via the accumulated weight. Or harm the process by not evaluating nodes with the latest information on the game it has. However, it could be interesting to see the effect of such a method on the choice of nodes over the different runs. And if we manage to have a consistent state as discussed in the previous paragraph, it could even more benefit from conserving previous computations.

Solving fixed state

This issue is the main challenge that remains besides behavior tuning of the heuristics. There are two beginnings of solution that will be presented here but remain to be tested. As said before they both come with a cost in computational power to ensure a more consistent state. In any case, having a consistent state which is not relying on generation each turn would help with the optimization techniques that aim to store nodes between runs of the algorithm.

The first idea is inspired by physics mechanics where you solve the uncertainty of states by applying the probabilities of the event to happens to add a factor to the different outcomes, basically doing a weighted sum. This is all hypothetical but in our configuration, the events would be multiple possible and likelier generated states. Each has a given probability to happen, and an outcome which is the resulting tree computed by the MCTS algorithm. Summing up all the trees together, we end up with a single tree. In this final tree, each child of the root would have the sum of all versions of itself summed up using the probability to happen and the actual number of times visited. Finally, the child of the root that is the most visited is chosen as the move for this turn. To summarize, we would run the MCTS search multiple times with several possible fixed states each turn, gathering their computations to hypothetically select an even better direction as it's the direction liked in general by most simulations across the future possibilities.

Another solution is to make an alteration of the game that allows running simulations on a state with imperfect information. A way this could work is that we run our simulation, moving pieces as usual and when it is needed, we generate locally an unknown piece to fix its behavior only. To resume it would be modifying the game implementation to allow part-unknown state.

Obviously, these are just potential solutions that I hadn't the time to test and their execution still is hypothetical.

Opponent's modeling

The opponent's modeling employed has already lots of advantages but it can be pushed further. Both in the state generation and in the prediction of the opponent's move.

Although the generation of state has already been shown to converge relatively quickly depending on the outcome via the graphs of the previous chapter, it doesn't take into account several details about the game. For instance, humans often repeat patterns of bombs around their flag or put at least one bomb around it, therefore it's possible to generate a pattern via the alteration of probabilities. Hence, generating the flag alongside known bombs. However, other patterns discussed in Chapter 2.2.1 can be applied too. Players often try to make setup boards that are well-spread, not placing the same piece around each other, separating the high pieces to control the board, and such.

The behavior should be deduced differently than using our priority function because nothing ensures that the opponent has the same thinking process as us. The goal of the simulation is to compute an overall gain if we take this direction and then, if we end up deleting nodes that the opponent selects, the simulation's purpose is gone.

It only has a few factors about what information someone has about the game at the moment, although the potential for this type of information is great. For instance, it could deduce that a piece that a piece that moved earlier is likely to move again in a sequence of moves, or a game plan instead of thinking about a single move. Moreover, it could also deduce that an opponent's piece that moves toward a piece of ours that is known is likely to be a piece higher than our known piece.

Besides that, there is knowledge about the game on our side and on the opponent's side that is rarely measured. The pieces' previous behavior could be stored directly or at least an explicit structure to store what pieces are known to the opponent. At the moment, the assumption that the opponent knows everything is taken.

Game plan

Something that performant AI agents, like *Demon Of Ignorance*, implemented and that is not observed all the time in my custom bot is game plans. Due to the inconsistent generation of state, the value and exploration of nodes can change even if very little change happens in the actual state. This sometimes causes the agent to change its play during a sequence of moves.

Therefore, a better generation of the state coupled with the different techniques talked about earlier could help. For instance, keeping in memory previous moves or previous pieces' behavior and taking them into account in our computations. Adding a factor about the information both players have about the other. Controlling the lanes, by moving high pieces to different places and waiting for the opponent to come.

The plans can be adapted to the game in itself, attack plans, more defensive plans, and playing with more risks if we're in a position of disadvantage for instance.

It's discussed above that the opponent is likely to play the same piece several times in a row and in fact, it works the same for us. We should favor playing a sequence of moves with the same pieces and make plans. To do so, we could keep in memory the last played pieces and give more weight to playing them again. This would help in escaping where it needs to play the same piece till it gets a potentially higher reward than just being captured.

Bibliography

[1] cjmalloy braathwaate. Demon of ignorance. `https://github.com/braathwaate/stratego`, 2020.

[2] Julien Perolat, Bart De Vylder, Daniel Hennes, Eugene Tarassov, Florian Strub, Vincent de Boer, Paul Muller, Jerome T Connor, Neil Burch, Thomas Anthony, et al. Mastering the game of stratego with model-free multiagent reinforcement learning. *Science*, 378(6623):990–996, dec 2022.

[3] Gravon platform. `https://www.gravon.de/gravon/`, 2022. [Online; accessed 25-February-2023].

[4] DeepMind. Open_spiel. `https://github.com/deepmind/open_spiel`, 2023.

[5] BluemlJ DeepMind. Add new game: Stratego/yorktown. `https://github.com/deepmind/open_spiel/pull/635`, 2021.

[6] Sam Moore. 2012 programming competition. `https://git.ucc.asn.au/?p=progcomp2012.git`, 2012. Also duplicated on Github: https://github.com/braathwaate/strategoevaluator.

[7] Stratego on wikipedia. `https://en.wikipedia.org/wiki/Stratego`. [Online; accessed 10-July-2023].

[8] Vincent de Boer, Léon JM Rothkrantz, and Pascal Wiggers. Invincible-a stratego bot. *International Journal of Intelligent Games & Simulation*, 5(1), 2008.

[9] Sergiu Redeca and Adrian Groza. Designing agents for the stratego game. In *2018 IEEE 14th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 97–104. IEEE, 2018.

[10] Jan A Stankiewicz and Maarten PD Schadd. Opponent modeling in stratego. *Natural Computing*, 2009.

[11] Master of the flag. `https://www.jayoogee.com/masteroftheflag/game.html`. [Online; accessed 12-July-2023].

[12] Maarten Schadd and Mark Winands. Quiescence search for stratego. In *Proceedings of the 21st Benelux Conference on Artificial Intelligence. Eindhoven, the Netherlands*, volume 102, 2009.

[13] Caspar Treijtel and Léon JM Rothkrantz. Stratego expert system shell. In *GAME-ON*, volume 2001, pages 17–21, 2001.

[14] Anton Falk. Stratego using deep reinforcement learning and search, 2021.

[15] Schuyler Smith. Learning to play stratego with convolutional neural networks.

[16] Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

[17] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

[18] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[19] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217, 2008.

[20] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.