UTRECHT UNIVERSITY

MASTER THESIS

Designing an AI agent for Stratego

Author:
Freek SCHOENMAKERS

Supervisor:
Dr. Tillmann MILTZOW
Dr. Hans L. BODLAENDER

A master thesis submitted in partial fulfilment of the requirements for the degree of Master of Computing Science

in the

Department of Information and Computing Sciences

August 6, 2021

UTRECHT UNIVERSITY

Abstract

Faculty Bèta Sciences
Department of Information and Computing Sciences

Master of Computing Science

Designing an AI agent for Stratego

by Freek SCHOENMAKERS

Stratego is a strategy board game that relies on incomplete information as an important gameplay element. This lack of complete information makes Stratego a challenging game for a computer player to play well. The game relies heavily on keeping information hidden from the opponent, allowing you to hide stronger pieces or to bluff with weaker pieces. Traditional AI methods such as Minimax that have been successfully applied to games like Chess are ill-equipped to deal with this hidden information, and AI agents based on these methods show poor results when applied to Stratego.

The goal of this thesis is to apply new methods, such as Monte Carlo Tree Search and Neural Networks, to Stratego AI agents in order to improve the quality of computer players.

Acknowledgements

Many thanks to Till for his excellent supervision, support and ideas throughout this Master's Thesis.

Special thanks to Wessel and Naomi for brainstorming with me during our weekly meetings, to Tom and Gijs for helping me test some AI agents and to Isabelle and my family for supporting me throughout my thesis.

Contents

Chapter 1

Introduction and background

1.1 Introduction

For over 5000 years people have played board games against one another. The ability of these games to entertain us has proven to be unwavering, as many ubiquitous board games such as Chess or Backgammon have existed for well over a thousand years. In more recent years, people have focused their efforts on making computers play board games. This has multiple reasons, the simplest being that having a computer player to play against makes it so that only one person ever needs to be in the mood for a game to play. They can usually also offer a variety of difficulty settings, so that players of any skill level can play a fun game at a challenging level.

A perhaps more important reason is that playing board games effectively is little more than efficient decision-making. Techniques that are applied to make decisions in board games can also be applied to real-life situations. This ranges from basic decision-making via MiniMax trees to Neural Network techniques being applied to Chess and Go but also to image recognition and more recently protein folding. As such, the creation of a computer player for a different board game may help in selecting and applying decision-making techniques to new problems.

For this thesis I will attempt to make a new computer player that can play Stratego at a better level than existing AI agents. Stratego is different from games like Chess and Go, in that it is an incomplete information game. This makes it a particularly challenging game for computer players, and techniques that have been successfully applied to other games do not readily apply to Stratego.

1.2 Stratego

Stratego is a strategy board game based on the French game L'Attaque. The game exists in its current form since 1961. It takes place on a 10x10 square board, with two 2x2 lakes in the middle, dividing the center of the board into three lanes. Each player has 40 pieces of one of 12 different ranks, one of which is the flag. The pieces face away from the opponent, so that they are unable to see which rank a piece has. The aim of the game is to either capture the opponent's flag, or to have the opponent be unable to make any move.

1.2.1 Rules for play

In the setup phase, each player places their 40 pieces in the four rows of the board closest to them. After all 40 pieces have been carefully placed on a square without revealing them to the opponent, the game begins by each player taking alternating turns to move one of their pieces. Each piece can only move one square orthogonally, with the exception of three ranks; the flag, the bombs and the scouts. The flag and

the bombs are unable to move, they must remain in the position they were initially placed in. The scouts can move orthogonally for more than one square; they can move until another piece is blocking its way. No piece can move into the two center lakes; these lakes block all movement.

If a piece tries to occupy the same square as an opponents piece, it attempts to capture it which removes it from the board. Whether or not it does so successfully depends on the rank of the attacking and defending pieces. Each rank has a number ranging from 1 (the spy) to 10 (the marshal), with the exception of the bomb and the flag. In general, if an attacking piece has a higher rank than a defending piece, it successfully captures the defending piece. If it is of a lower rank, it is removed instead and the defending piece stays on the board. If the ranks are equal, both pieces are removed from the board. There are exceptions to this rule: if a piece attacks a bomb, then the piece is always removed regardless of its rank. The only exception is the miner (rank 3), which can safely attack and capture bombs. If a spy (rank 1) attacks the marshal (rank 10), the marshal is captured. If any piece attacks the flag, the flag is captured and the game ends. The winner is the player who captured the opponent's flag. Note that with every attempted capture, both pieces must reveal their ranks to the opponent.

1.2.2 Two-squares rule

One special rule is the two-squares rule. A player may not repeatedly move a piece between the same two squares for more than three of their turns in a row. This is to prevent players from repeatedly moving back and forth, stalling the game. Additionally, it can be used to trap an opponent's piece, leading to a capture [Fed10].

1.2.3 Victory conditions

There are three possible ways for a player to achieve a victory. These are the following:

1. *Flag capture*: The primary way of achieving victory is capturing the opponent's flag with any friendly piece. Upon capture, the game immediately ends and victory is declared.

2. *No possible moves*: If a player is no longer able to make any moves, either because they have no movable pieces left or because the pieces that are still on the board can no longer move (either due to being blocked by friendly unmovable pieces like bombs and the flag or because the two-squares rule prohibits it), the game ends and the opponent is declared the victor.

3. *Resignation*: If a player resigns and the opponent does not agree to a tie, the game ends and the opponent is declared the victor.

Note that depending on how the game plays out, it is entirely possible that capturing the flag is no longer possible. An example would be a game in which one player has the flag protected by bombs and the other player has no more miners left. In such situations a player would be forced to eliminate all movement possibilities of the opponent to achieve victory.

1.3 Basic strategies

The nature of the game makes it so that there is no apparent dominant strategy that guarantees victory over the opponent. Certain defensive strategies may be able to counter certain other offensive strategies well, and may fail to defend adequately against certain others. Regardless, some strategies have proven to be consistently strong in many games and thus generalise well. This section will briefly touch on some of these.

1.3.1 Setup

For the setup phase, most players tend to place their flag either on the back row or next to the lakes (often known as the shoreline bluff), with bombs surrounding the flag. The back row is often chosen as it is furthest away from the opponents pieces and the player has more opportunity to defend with their pieces against enemies. To surprise and misdirect the opponent, some players like placing their flags next to the lakes, hoping that their opponent will assume the flag is in the back row and not search for it on the shoreline positions. The bombs surrounding the flag offer it more protection, as it will require the opponent to bring a miner over to capture the bombs. This may prove difficult, as the miner at rank 3 is lowly ranked and thus susceptible to capture by the enemy. It also makes it possible to eliminate all of the opponent's miners, making defeat by having the flag captured impossible. Note that in this scenario a player can still lose if they lose all of their other pieces, rendering them unable to make a move. The remaining bombs can be placed elsewhere on the board or protecting some other lowly ranked piece, as a bluff to misdirect the opponent as to where the flag is located.

Most players like to divide their scouts over board. The scouts in the front serve to scout ahead, finding high ranked pieces that the opponent may wish to attack with. The rest of the scouts can be held in reserve until the late game, when searching for bombs and the flag becomes more important. When there are fewer pieces on the board, the scouts will also be able to make better use of their enhanced mobility.

For a balanced strategy, it is advised to keep the high ranking pieces (e.g. ranks 7-10) somewhat equally distributed over the left and right side of the board. This prevents the opponent from bringing in a high ranking piece of their own on one side of the board and being able to capture many pieces there, without having the ability to mount a quick defense. Alternatively, one can focus their higher ranked pieces on one side for a very aggressive play over one flank, though this is considered much riskier.

Miners are valuable for disposing of the enemy's bombs, which usually only becomes a factor later in the game. Therefore, most players keep their miners near the back of the board. It can however pay off to have one or two near the front of the board, to deal with an opponent who has placed their bombs so that they completely block off one or multiple lanes.

One other quality that is good for any setup is to make it unpredictable. If the same setup is used often for example, an opponent can more accurately estimate the ranks of your positions, giving him an advantage.

1.3.2 During play

In general it is advised to postpone the revealing of the high ranked pieces as much as possible, and conversely it helps to find the opponent's high ranked pieces as

early as possible. This is because being able to trap or distract the opponents high ranked pieces may give your own high ranked pieces free reign. As an example, a game where the opponent has lost both their spy and their marshal renders the friendly marshal effectively invincible, unless it attacks a bomb. At this point, the marshal can attack any piece that the player knows has been moved at least once. Particularly in the late game, when many pieces have moved, this can be a very powerful position to find oneself in.

Bluffing with pieces is also advisable. For example, one can try to scare off an attacking marshal by moving a piece towards it, pretending that it is the spy. Another common bluff is moving the scout only one square per move, thereby keeping the rank of the piece hidden.

Once a piece has moved, a player knows that the piece can not be a bomb. This makes it much safer to attack it with a higher ranked piece. Pieces that have not been revealed yet or that could be bombs are best attacked with lowly ranked pieces, as they are not as good at capturing and therefore their possible loss is of minimal importance.

1.4 Difficulties for computer agents

While games like Chess have been considered 'solved' since the 90s, Stratego remains largely 'unsolved'. There have been numerous attempts at creating an AI agent for Stratego, but the previously applied techniques have considerable flaws, leading to agents that do not perform at a decently high level, but remain at the level of a beginning player. Often their strength also comes from applying a specific strategy that, once figured out by the opponent, can be easily countered [Boe07; Wol18].

This mostly stems from the fact that certain aspects of Stratego are particularly difficult to deal with for the most used AI strategies. Typically, an AI agent for some particular game will decide the next move or action based on a combination of *static* and *dynamic* lookahead analysis. For Stratego however, both of these types of lookahead analysis are particularly difficult [Wol18].

1.4.1 Difficulties with static lookahead analysis

Static lookahead analysis involves looking at the current gamestate, and analysing it to immediately provide the next action or some valuation of the gamestate. For example, a function that evaluates a gamestate and outputs a single value that is higher as the gamestate gets better, is a type of static lookahead analysis.

For Stratego, such an analysis based on the current gamestate is very complex. A good analysis would at least have to take the following into account:

- *Piece values*: On a Stratego board, pieces will have different values, with higher-ranked pieces typically having a higher value than lower-ranked pieces. Immediately however, a number of problems arise: whilst most ranks are ordered in a hierarchical way, there are exceptions: the spy, the scout, the miner and the bomb. The spy of course is the lowest-ranked piece, but it is also the only piece capable of capturing the enemy marshal in an attack. The scout has no extra offensive abilities, but is able to move more than one square at a time. The miner is the only piece capable of removing bombs. The bombs themselves do not neatly fit anywhere in the hierarchy, as their role and abilities are unique.

There is also the issue of the ranks being able to shift around in the hierarchy: once all bombs have been cleared out, the miners become less valuable. If the enemy marshal is captured, the spy becomes much less valuable. If the enemy marshal and spy are captured, there is no piece that can threaten the marshal anymore, allowing it to freely attack any piece on the board that has moved with no risk to itself. If the opponent for example has no pieces left above some rank, then all owned pieces above that rank have practically the same value, as they can all capture the remaining enemy pieces on the board.

Additionally, whilst in the early game it may be advantageous to sacrifice multiple lower-ranked pieces to capture a higher-ranked piece, this may be disadvantageous in the late game when there are few pieces left and a larger quantity of pieces may be able to outmanoeuvre a lower quantity of pieces with a higher quality.

- *Revealed and concealed information*: Stratego is a game without perfect information. This naturally applies some value to the information (or lack thereof) on the board. As a general rule, you want to reveal as much information about your opponent as possible whilst concealing as much information about yourself as possible. This allows for bluffing; the tactic of intentionally trying to deceive the opponent into thinking that a piece is of a different rank than it really is, in the hopes of tricking the opponent into making a wrong move.

  A naive approach would be to simply keep track of which ranks a piece could still have, and which ranks it can definitely not have. For example, an unknown piece that has moved can not have the ranks of either bomb or flag. Similarly, if all six miners are known, then an unknown piece can not have the rank of miner. However, it is easy to see that this approach does not take into account that information is not all valued equally: keeping the marshal (or higher-ranked pieces in general) secret for a longer time is desirable, as it gives you more time to discover the enemy higher-ranked pieces that can be captured with the marshal, as well as discover the enemy spy.

  It's also clear that the value of information degrades as time progresses. Near the end of the game, when there are few pieces on the board, it becomes much easier to guess the ranks of the opponent's pieces. At this point in the game, the most valuable information concerns the position of the opponent's flag.

- *Piece positions*: Perhaps the most complex part of a static analysis is the evaluation of the positions of the pieces. One can easily infer general rules, e.g.: more pieces on the opponent's side of the board is better, or not having pieces next to pieces that could capture it, but there are many nuances. There are many positions that allow you or the opponent to trap and capture a piece, yet it is not always immediately evident which positions those are.

  As an example, consider having a piece ranked higher than a miner in each of the three lanes between the lakes. This single piece is capable of evading capture by a single other piece, whilst simultaneously capable of preventing any miners from crossing over to the other side. It would require two pieces with an equal or a higher rank to capture this blocking piece. This position is very valuable, as long as there are no two pieces capable of capturing it moving into the lane. However, determining if this is the case is difficult if the pieces moving in are unknown; it could be a genuine threat or just a bluff. This makes most evaluations of the positions of the pieces an evaluation of risk, which is difficult to handle properly and consistently.

Even if one is able to do a proper, consistent static analysis of each of the above aspects, one still has to be able to weigh them properly when summing them together. These weights too could change depending on how the game has progressed. If the opponent appears to be using an unusual setup which is harder to predict, then the value of information may be greater than usual.

1.4.2 Difficulties with dynamic lookahead analysis

In theory, a perfect static lookahead analysis should be sufficient to develop a perfect agent. In reality, such a perfect static lookahead analysis method does not exist, or is not realistically attainable. To solve this issue, dynamic lookahead analysis is applied. The idea is that by effectively looking at future gamestates and evaluating those, it is possible to overcome these shortcomings. The most commonly used example is the Minimax method, a depth-first tree search algorithm that explores moves up until a certain depth, and evaluates only the gamestates at that maximum depth. By then continuously picking the best moves for each player (the minimum scoring move will be picked by the opponent, and the maximum scoring move will be picked by the player, hence the name 'Minimax'), the AI agent can determine the highest evaluated gamestate possible, where both the AI agent and its opponent play optimally. This overcomes shortcomings in the short term through the dynamic exploration of moves, and guides the AI agent towards a good final result through the static lookahead analysis.

There are a variety of different tree search methods, the most notable being variations of Minimax (such as Expectimax, which can deal with chance nodes) and Monte Carlo Tree Search. However, Stratego works in such a way that tree searches often end up being very inefficient. Vincent de Boer, three-time Stratego world champion and author of the 'Invincible' AI agent, has analysed the reasons why in his master thesis [Boe07]. It boils down to the following problems:

- *High branching factor*: On average, each player can take approximately 21.7 unique moves every turn [Art10]. This means that for a tree search after two moves there are approximately 470 unique move paths to explore, and after five moves we have over 4.8 million unique move paths. Note that this is five moves for both players combined, so just three moves for the AI agent and two moves for the opponent.

- *High required depth*: In Stratego, most pieces can only move a single square per turn. This means that before you see the effect of a certain series of moves, you need a considerable search depth. For example, a miner moving to capture a bomb 10 squares away requires a search depth of 19 (including the opponent's moves). Combined with the fairly high branching factor, means the tree that needs to be explored is incredibly big, and searching it becomes very difficult very quickly.

  This is different from a game like Chess, where despite the higher branching factor (approximately 31), most moves result in a large difference in the gamestate far earlier. This is because most pieces in Chess can cross greater distances, and also because the moves in Chess are often very strongly interrelated.

- *Low interrelation between moves*: Most moves in Stratego are not interrelated. Consider moving pieces in the left and the right lane. These moves will have no effective impact on one another, and can likely be executed in any order,

leading to no noticeably different gamestates. This has the unfortunate effect of inflating the branching factor unnecessarily.

Unfortunately it is not clear-cut when exactly this inflation is unnecessary. If a player finds a good move on the left flank by ignoring a real threat on the right flank, it risks missing opportunities or threats and thus making the wrong moves.

- *Long games*: The average game takes approximately 381 moves [Art10]. This is much longer than an average game of Chess, which would take roughly 40 moves, and often less when less experienced players play.

  This means that for most moves, the tree search will be unlikely to find a game state where the game has ended, making it more reliable on the static lookahead analysis. However, in general the total length of the games is much less of a problem for the dynamic lookahead analysis than the other listed points.

It becomes apparent that any dynamic lookahead analysis will require considerable work on narrowing the branching factor to increase the search depth. However, it seems that there may still be practical limits to the search depth that may be very hard to overcome.

It seems that for any AI agent doing proper lookahead analysis will require both a very strong static as well as a strong dynamic approach. If either is not good enough the play of the AI agent will likely suffer considerably as a result. Particularly against human players the AI agent will struggle, as humans typically have less issues with separating moves that have no relation to one another and typically also have an easier time looking multiple steps ahead when each of the steps is fairly simple (e.g. it is easy to imagine moving the miner towards a bomb ten squares away).

Chapter 2

Literature Study

2.1 A note on Stratego literature

Compared to other games like Chess and Go, not much work has been done on creating an AI agent for Stratego. As such, the available literature is far and few between, and mostly consists of bachelor's and master's theses. In fact, most agents created for Stratego are largely undocumented or closed-source, making it difficult to effectively ascertain exactly which particular methods and techniques have been applied and how effective they were. After a discussion with the supervisor, it has been agreed that the literature study therefore will also be somewhat limited.

The current holder of the title of 'Strongest Stratego AI' is Probe 2, the latest version of which has been incorporated in the 'Heroic Battle' mobile app. There used to be annual Stratego Computer World Championships, but this competition is now defunct. Other notable contenders from these contests include Master of the Flag 2, Probe 1 and Invincible.

With the exception of Invincible, the AI agents listed are not covered by available literature. It is therefore difficult to determine exactly how these agents work, although we can deduce the general approach. For the literature study, this is not too detrimental; the literature that does exist covers most modern techniques and approaches, and whichever techniques these agents may be using is likely a combination of several of these techniques or a refinement of some of them.

The literature study will therefore focus on the documented attempts at creating a Stratego AI. Note that for many of these, it is hard to establish exactly how effective the attempts were. With the exception of a few, most works tend to lack a proper evaluation against either different AI agents or against human players. Typically the evaluation does contain some description of certain strengths and weaknesses, from which we can very roughly estimate the effectiveness of the AI agent.

2.2 AI agents designed using expert domain knowledge

The existing literature can be broadly subdivided into two groups. The first group consists of people that designed their AI agents using expert domain knowledge. Here, people have tried their best to design, create and fine-tune their AI agent based on their own observations, knowledge and evaluations. The second group features people that have attempted to use some kind of learning algorithm to help create their AI agent instead. They may still be providing the basic framework, but for example the exact way of doing static lookahead analysis is largely learned instead of provided.

This first group has an initial advantage, in that they are able to identify shortcomings of their agents more quickly and are able to adjust accordingly. Their agents will therefore quickly reach a certain level of competence. They however have an unfortunate disadvantage in that their agents are also limited by the knowledge of their creators. The second group has the potential to overcome these limitations and thus the potential of the first group.

We will begin the literature study by starting with the first group.

2.2.1 Multi-Agent Stratego

The oldest literature concerning computers playing Stratego appears to be the master thesis by Caspar Treijtel, written in 2000 [Tre00]. Treijtel proposes that a complex system can often be simplified by dividing the complex system into multiple smaller, much simpler systems.

Treijtel suggests treating each piece on the Stratego board as a separate subsystem, with their own perceptions, objectives and variables. Each piece would then perform their own static lookahead analysis, and determine which action it would prefer to take, e.g. attack another piece, move a square, run away, etc..., and how much it would like to do these. The pieces then submit this information to the central decision maker, which simply picks the move that is preferred most.

While this AI agent is capable of making some decent decisions when it comes to small localised situations, it is largely incapable of any long-term planning. This is largely due to the lack of any dynamic lookahead analysis, and a fairly basic, uncoordinated static lookahead analysis. It may be able to defeat some weaker AI agents, but it does not stand a reasonable chance against a human player.

It should be noted that while Treijtel originally developed this agent, most literature appears to reference a bachelor's thesis by Ismail [Ism04], who has implemented this agent. It appears however that large portions of text have been directly copied from Treijtel's work into Ismail's. I will therefore not further reference this work.

2.2.2 Monte Carlo Stratego

Jeroen Mets wrote a paper in 2008 on applying the Monte Carlo Tree Search (MCTS) method to Stratego [Met08]. MCTS is a technique where the agent randomly selects moves in the gametree until it reaches a leaf node. At this point it does an evaluation of the node, either through an evaluation function or a random rollout. The obtained results are then averaged down the tree. To deal with uncertainty, Mets uses 'strooien', e.g. assigning random possible ranks to the pieces.

Mets has divided his paper in three parts. The first part seeks to simply apply the MCTS technique to Stratego. The evaluation function is defined as a random rollout up to a certain depth, e.g. X random moves are selected to 'simulate' a potential playthrough. At the end of the rollout, a valuation function is applied. The valuation function is based on the revealing of information, the capture of pieces and moving a previously unmoved piece. Mets assumes a direct inverse correlation between what is good for the agent and what is good for the opponent, e.g. ten points for the agent result in minus ten points for the opponent.

In his first test, he finds that MCTS is able to play the game somewhat decently. It manages to beat a random agent 96-97% of the time. Mets notes that if there are bombs around the flag, the MCTS agent is much less likely to capture it and may instead rely on capturing all moving enemy pieces to win. The average game length also goes up significantly. He also notes that the high branching factor, which leads

to many different random games, is a cause for lowered performance. Mets also uses a fairly low amount of rollouts (on average ~ 330).

In the second part of the paper Mets investigates the effect of various parameters related to MCTS. He specifically looks at the time given to find a move, the maximum number of moves used in rollouts and the method of 'strooing', the assignment of ranks to unknown pieces. Mets finds that the best 'strooing' method is to do it once at the start of every turn. He also concludes that the performance of the MCTS agent improves if he increases the amount of time given per turn and if the rollouts use more moves before evaluating. He does note that the amount of rollouts here matters most; more but shorter rollouts are superior to few but longer rollouts.

In the third part, Mets attempts to optimise the MCTS agent by using progressive pruning. His exact implementation does not quite become clear, but the idea is that during the exploration of the MCTS tree, moves that are found to be bad can be eliminated and no longer explored. By doing this, the MCTS tree should naturally begin exploring only the better moves. Unfortunately, the application provides limited results, showing an improvement only with specific parameters related to the number of moves per rollout, which if shortened eliminates the advantage progressive pruning provides. Mets also claims that if the number of rollouts increases, the advantage turns into a disadvantage, as progressive pruning can on accident prune away the best move.

Mets concludes by mentioning other methods to improve the MCTS agent's performance, though he has not implemented them himself. It appears that other move selections that are typical for MCTS, such as the Upper Confidence Bound method or the ϵ-Greedy method were not tried.

2.2.3 Invincible, A Stratego Bot

In 2004 three-time Stratego world champion Vincent de Boer developed a new AI agent for Stratego in his master thesis named Invincible. De Boer used his extensive and deep knowledge of the game to identify multiple weaknesses in the tree-search approach for Stratego agents, which was by then typically used in the more powerful agents such as Probe. Most of his findings can be found in section 1.4.2.

De Boer stipulates that humans, unlike computers, do not think of individual 'moves' when playing Stratego, but rather formulate plans. Each plan has a specified goal, a method to attain that goal and a measure of necessity to reach said goal. He gives an excellent example in figure 2.1.

In this example, all pieces have their ranks revealed. Red can win in this scenario, but it requires a staggering 200 moves before this happens. This may seem like an incredibly large amount, but a human player can quickly see how this will happen. There are three things that need to happen:

- The blue miner in the bottom right needs to be trapped and captured by the red marshal. This secures the red flag.
- The red miner in the bottom left needs to evade the blue general close by it. There is only one higher ranked unit, so the general is unable to trap and capture it provided the miner keeps moving.
- Once the marshal has captured the miner, it needs to chase the blue general away so that the red miner can capture the flag.

Each of these items is relatively easy to see and understand. The reason why it takes such a large amount of moves is because of the second item, the evasion of

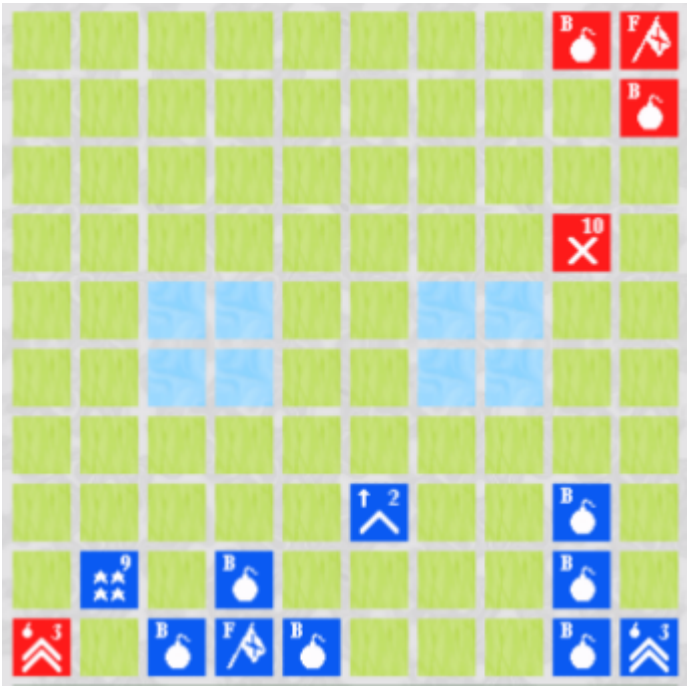

FIGURE 2.1: An example provided by de Boer, showing a situation that is easily solved by humans but is very difficult for computer players

the general by the miner. These pieces will alternate between moving up and down, until the blue general reaches the limit set by the two-squares rule (see section 1.2.2). Once the limit is reached, red will not have to use their turn to have the miner evade, and can do just one move to further the other two plans (the capture of the miner and the chasing of the general).

A human player, particularly one with some experience, will quickly see that the miner in the bottom left is safe from harm, due to the two-squares rule. They can then focus on executing the other two plans. A computer player however, particularly one that uses a search tree, will still have to consider these moves. In this tree, the optimal path is one that is 200 moves long, and roughly four out of five moves involve the miner and the general repeatedly chasing and evading. This greatly increases the size of the tree.

To work around this problem, de Boer created an AI agent that also thinks in plans rather than moves. He used his expert domain knowledge to define a large set of plans, e.g. 'Have the marshal evade the spy', or 'Trap an enemy piece'. Every turn, the AI agent determines how much each move would contribute to each plan. The move that ends up contributing most to all plans collectively is the move that is chosen by the bot to play.

To create a setup for the bot to use, De Boer uses a method that utilises statistical data from a database of Stratego games played from the Gravon database. He uses this data to calculate the chance that a certain piece is placed in a certain square, and how likely it is that other pieces are next to it. He then generates a large number of semi-random setups, and uses specific heuristics to select the best setup to use.

De Boer also uses heuristics to determine the odds that a certain enemy piece on the board has a certain rank. Every move made by the opponent is evaluated by these heuristics and the odds are updated accordingly. He uses normalisation to make sure the sum of all percentages per piece and per rank remains close to 100%.

Evaluation

De Boer effectively decides against any type of dynamic lookahead analysis, due to its inefficiency, and focuses strictly on crafting a very strong static lookahead analysis. De Boer may be uniquely positioned to do this, as likely few people exist with a similar grasp of both Stratego as well as programming. In his evaluation, he states that Invincible is able to defeat some human players, but not others.

It seems that in particular, Invincible struggles playing against players who use an unexpected setup, or who play unconventionally. This makes sense; de Boer has used his expert knowledge to teach Invincible how to play against typical opponents, e.g. the ones he himself would likely encounter most, but not against a player who might be far less skilled but a lot more unpredictable. He mentions that a player who blocked off two lanes using bombs cost Invincible some of its high-ranked units, ultimately leading to a defeat. A human however would be able to recognise this situation and adapt accordingly, as in this case it is very easy to control the remaining opened lane while slowly threatening the blocked off lanes. De Boer had simply not taught Invincible how to do this yet.

Ultimately, Invincible's potential seems limited. De Boer says he can improve Invincible by adding more plans and refining the existing ones, but it seems inevitable that at some point the amount of plans will begin conflicting with one another. It is also clearly limited by de Boer's knowledge of the game (which may be great, but is ultimately still limiting).

2.2.4 Stratego Senior Design Report

Jeff Petkun and Ju Tan produced a senior design report on a Stratego AI agent in 2009 [PT09]. Their agent is a Minimax agent with α-β-pruning. A novelty in their agent is that they have developed three different static lookahead analysis methods.

As de Boer mentions in his master thesis, the value of certain pieces and information changes over the duration of the game [Boe07]. To account for this, Petkun and Tan have developed a single evaluation function that can be tuned using various weights. These weights are then adjusted based on the progression in the game. It starts by prioritising discovery, material advantages and flag safety. In the midgame, it adds the distance between the miners and opponent pieces that have not yet moved. Once the game reaches the final phases, it prioritises this distance more, as well as the distance between owned pieces and enemy unknown pieces.

The report does not mention how exactly the Minimax tree search deals with the uncertainties of the ranks of the enemy pieces. It makes mention of a basic probability assignment to each piece, simply based on the number of remaining unknown pieces of each remaining rank that could be assigned to the unknown piece. These probabilities can only change through a capture (revealing the rank), movement of two squares or more (revealing a scout) and moving (revealing it is not a bomb or a flag).

The evaluation shows their tests against a random agent and against an agent that uses their own evaluation function but only searches one move deep instead of doing a tree search. Their results show that their agent is capable of beating both, though they mention an interesting caveat; their agent ties the random agent in 28% of games instead of winning. Usually, ties in AI Stratego games happen in the game runs for longer than X moves. It is assumed that that is also the case in this paper, however the exact number of moves after which a tie occurs is not mentioned. It is

therefore difficult to estimate exactly how significant this 28% of games ending in a tie is.

2.2.5 Using Domain-Dependent Knowledge in Stratego

In a paper from 2009 Mohnen attempts to use domain-dependent knowledge to improve the evaluation function for Stratego and attempts to use move-specific heuristics to apply for example forward-pruning to optimise an AI agent based on the Minimax algorithm [Moh09].

The Minimax algorithm is a tree-search algorithm, and as a result it largely suffers from the problems described by de Boer in his thesis [Boe07] (see also section 1.4.2). Mohnen attempts to use domain knowledge to at least partially mitigate some of these issues.

The Minimax algorithm is a depth-first search algorithm. This has some notable downsides, for example that if a bad move is made near the root of the tree, the algorithm will first need to finish searching this branch before moving on to the next, despite the bad move at the root probably negatively impacting the final evaluation score of the leaf nodes. If that impact is large enough, then it may be more efficient to do forward-pruning; reducing the search depth for this branch so that less resources are wasted on searching it completely.

Mohnen applied a number of heuristics to see in which cases forward-pruning has a positive effect on the AI agent. Unforunately there was little success; while the heuristics as 'Losing the last miner', 'Losing the spy' and 'Losing the marshal or general' seemed to provide a marginal improvement (winrate of 50-52% compared to an agent without these pruning heuristics), the other attempted heuristics significantly worsened the performance of the AI agent.

Mohnen also attempts to use domain knowledge to improve the evaluation function. To do this, a basic evaluation function is defined based on simply counting the number of friendly and enemy pieces on the board. Mohnen then defines a number of additional heuristics that could improve the evaluation function, e.g. capturing the enemy spy, having pieces near the enemy's flag, etc...

Interestingly, Mohnen finds that almost all of these heuristics individually seem to improve the evaluation function when compared to the basic evaluation function. However, when all heuristics are added simultaneously performance degrades significantly instead. This may stem from a problem where the heuristics are not weighted correctly compared to the basic evaluation function.

Mohnen concludes stating that domain-dependent knowledge can improve the AI agents performance, provided there is enough tuning done. Unfortunately, the exact way in which the agent and the heuristics have been implemented and how the tests are executed remains vague, with few details given. It therefore remains difficult to estimate how much these heuristics actually contribute, or what the performance of the AI agent is compared to a human.

2.2.6 Opponent Modelling in Stratego

Stankiewicz wrote a paper in 2009 on opponent modelling in Stratego [Sta09]. In this paper Stankiewicz attempts to more correctly predict the ranks of the opponent's unknown pieces, based on several statistically determined metrics. It is effectively an extension to the efforts of de Boer [Boe07] to find a good way to predict enemy piece ranks.

Stankiewicz used the Gravon database of 70000 games and tracked how often a piece of a certain rank would move near a piece of another rank, and how often they would move away instead. During the game, the AI agent also keeps track of how often a revealed piece was determined to be a bluff. It uses this to determine a bluffing probability for the opponent.

On every move made by the opponent, the AI agent reads the probability that the opponent would move a piece of each rank closer to the agent's pieces from the precalculated conditional probability table. It takes the chance that the opponent might be bluffing into account. It uses these odds to update the probabilities for each piece that they may be of a certain rank. These odds are then used in an AI agent that uses Expectimax to determine the next move.

To evaluate the accuracy of the method, Stankiewicz tracked how often the agent guessed the opponent's piece rank correctly. A correct guess in this case is having the actual rank as the highest probability for that piece when it is involved in a capture (either offensive or defensive) as this reveals the rank. The agent ultimately manages to guess 40% of piece ranks correctly.

While this may seem impressive, some crucial details are omitted. For example, it is not known if these 40% correct guesses are evenly distributed over the full game, or if the correct guesses all happen towards the endgame, when guessing correctly is much easier. There is also no measure of exactly how wrong the wrong guesses are. To a marshal, wrongly guessing an enemy piece has a rank of a sergeant when it is in reality a lieutenant will not matter much. It is also unfortunate that the correctness is only checked on an attempted capture, when the rank is revealed regardless. At this point, guessing correctly is not too important anymore, it is mostly important in the few moves leading up to the attempted capture.

These caveats may have a considerable impact, as the winrate of the agent with opponent modelling against an agent without it is not considerably greater. One would assume that knowing 40% of piece ranks leads to a significant advantage, but the winrate never exceeds 56%. The effect of the measures intended to detect bluffing is also unknown here, as the Expectimax agent may not bluff (this depends on the evaluation function used, which is unfortunately not given).

2.2.7 Quiescence Search for Stratego

In 2009 Maarten Schadd and Mark Winands propose a method for quiescence search [SW09]. They propose that tree-based search methods in Stratego suffer from the horizon effect, which can be solved by briefly extending the search depth but limiting the search to 'volatile' moves. This method is known as a quiescence search. In their paper the authors also suggest a more specific method called an 'Evaluation-Based Quiescence Search' (EBQS).

The horizon effect is an effect where a tree search will not accurately be able to assign the correct value of a move, because a large change to the evaluation value would only happen just beyond the search depth of the tree. As an example for Stratego, imagine a tree search with a depth of five (three moves for the player, two for the opponent). Between a friendly piece and the enemy flag there are three empty squares, but there is also an enemy piece of a known lower rank just two squares away. The tree search will be able to explore the move capturing the enemy piece of a lower rank, which improves the evaluation score for the move moving the friendly piece in that direction. It will not however quite reach the move capturing the enemy

flag, which would end the game. That means the best move in this case is undervalued. Because this game-ending move is just beyond the 'horizon' of the search tree, the tree suffers from the horizon effect.

To solve this problem, a quiescence search can be done. This is a brief extra search at the search depth, only searching moves that would alter the evaluation score by a significant amount. Any heuristics could be applied, e.g. only search capture moves, or only search capture moves that are known to result in a successful capture. This method effectively extends the search depth just a little bit beyond the horizon, and thus reduces the horizon effect.

Unfortunately, directly applying quiescence search to Stratego did not improve the agent's performance much. The first problem is the large overhead of doing quiescence search. Despite only searching specific moves, the branching factor remained fairly large due to the large amount of chance nodes in the Expectimax search algorithm. Even with low limits on the quiescence search depth, the algorithm ended up spending almost as much or even more time on the quiescence search than on the regular search. The second problem is the limited improvement to the search results. In only 44% of cases was an improvement seen, and the average improvement was approximately merely the value of a single scout (in their evaluation function).

Quiescence search ended up not improving the winrate of the agent. Only when looking at the very next move (and none after) was a slight improvement seen. This inspired the authors to do create a new method they called 'Evaluation-Based Quiescence Search'. This method involves looking at the results of a single move in specific situations. The first situation is when two known pieces are next to one another. In that case, the capture is searched, and the results are the QS-value. The second situation is where there is one known piece and one unknown piece. Again the capture is searched, and the results are the QS-value.

This method provided a very small improvement over the normal agent without quiescence search (52%) and the agent with a small quiescence search (56%). The authors note that large gains are not expected in the Expectimax framework, so the result is still good. Unfortunately, it seems that the method amounts to little more than a very brief extension of the search depth, of which we would already expect it to be performance improving. It seems the authors have actually proven that it is better to search through more likely taken moves, rather than unlikely taken moves. This unfortunately has not yet lead to a change in direction away from the Expectimax algorithm.

2.2.8 Competitive Play in Stratego

In 2010 Sander Arts wrote his master thesis on using multiple heuristics such as the history heuristic, transposition tables, α-β-pruning and more notably STAR1, STAR2 and STARETC to improve the performance of the Expectimax algorithm [Art10]. He achieved a reduction of 75% of evaluated nodes. He also did the first complexity analysis of Stratego.

Arts calculated a number of exact figures and upper bounds for Stratego in his complexity analysis. These are as follows:

- *No. of setups*: $\sim 10^{23}$ different possible starting setups.
- *Upper bound on state-space complexity*: $\sim 10^{115}$.
- *Average game length*: ~ 381 plies (analysed from the Gravon database).

- *Average branching factor*: Starts at ~ 23, quickly rises to peak at ~ 25 and then linearly goes down to ~ 15. Total average is ~ 21.7.
- *Game tree upper bound*: $\sim 10^{535}$ including chance nodes, $\sim 10^{509}$ excluding chance nodes.

Arts notes that this complexity is considerably greater than a game like Chess, which only has a state-space complexity of $\sim 10^{46}$ and a game tree complexity of $\sim 10^{123}$. This seems to support the findings of de Boer that show that tree search algorithms struggle with the complexity of the game [Boe07].

To overcome this complexity hurdle, Arts applied specific heuristics to the Expectimax search algorithm, which will allow it to prune non-promising branches earlier from the tree. The most important of these are STAR1, STAR2 and STARETC. These are all ways to prune in the chance nodes of the Expectimax search tree. As STARETC showed the greatest increase, it will receive a brief explanation of how it works.

STARETC is a version of Enhanced Transposition Cutoff modified so that it can deal with probabilities. It uses a transposition table to retrieve lower and upper bounds, as well as the exact value. This information is retained between moves, so that once the agent has a new turn it can look up a considerable amount of data from the transposition table, instead of fully recalculating it. The lower and upper bounds are used in a similar fashion as α-β-pruning. From the probabilities in a given chance node, we can calculate the upper and lower values a certain result should at least provide in order for the node not to be pruned by regular α-β-pruning. Suppose for a chance node with two possibilities, both equally probable, we have an upper bound of 9 and a lower bound of 8, with possible values between 0 and 10. If the first possibility returns with a value of 5.5, then there is no value that the other possibility could return that brings the average value for the chance node between 8 and 9. It can therefore be safely pruned.

Using these heuristics Arts manages to prune up to 75% of nodes from the explored search tree. While this is an impressive result, the exponential nature of the algorithm used means that still an incredible number of nodes has to be searched, even with fairly small search depths. For a search depth of five total moves, the STARETC method still searches an average of 1.17 million nodes, with a standard deviation of 3.76 million nodes and a maximum of 33.6 million nodes. At higher search depths, this will keep rising exponentially.

While Arts' contributions are certainly great improvements, it seems unlikely that considerable further improvements will be made to the method. The best improvements can probably be obtained by finding a better order in which to explore moves in the tree, so that the upper and lower bounds can be found sooner and thus more nodes can be pruned. Nonetheless, a considerable breakthrough is required to make this method suitable for deeper searches.

2.2.9 Designing Agents for the Stratego Game

In 2018 Sergiu Redeca and Adrian Groza wrote a paper named 'Designing Agents for the Stratego Game' [RG18]. In this paper they describe the design for two agents; the probabilistic agent and the multi-ply agent. These agents are fairly simplistic and do not seemingly improve on any previous results. They will be briefly explained here, for completeness of the literature study.

The probabilistic bot evaluates all possible moves the agent can take in its current turn. If these moves involve attacking a piece with an unknown rank, the agent

chooses the rank with the highest probability. This probability is very basic; the probability that a certain piece is a certain rank is proportional to the number of pieces of that rank that are still unknown and left. Note that if a piece has moved, it can no longer be the bomb or the flag. Since at the start of the game the pieces with the rank of scout are the majority of the unknown pieces, the agent assumes that any piece that it attacks will be of the scout rank. As the game progresses, the proportions of remaining pieces shift and will indicate different ranks.

The multi-ply bot is very similar. It is effectively identical in its approach to the probabilistic bot, except it searches through multiple plies. It simulates which move the opponent would take by assigning possible ranks to the opponent's pieces. It is thus very similar to a simple Minimax tree search, where the probabilities are handled by simply picking one option and using that. This avoids having to explore many chance nodes that vastly increase the branching factor.

The effectiveness of both of these approaches is low. Ultimately, the agent configuration best identified ended up being a 2-ply bot that only won in 91.3% of cases against a bot that only selects random moves every turn. For an agent to be considered at least somewhat capable it would have to win more than 99% of games against a random agent. Losing approximately one in ten games suggests a serious deficiency in the applied strategy.

2.3 AI agents designed without using expert domain knowledge

The amount of literature that uses techniques that do not require expert domain knowledge is considerably smaller than the amount of literature that does use it. There are notable advantages to methods such as these, specifically that no domain knowledge is required and that the agents have the potential to exceed the ability of the creator. The limits of agents relying on expert domain knowledge become apparent in de Boer's work on Invincible [Boe07] and for example in the papers by Mohnen [Moh09] and Petkun and Tan [PT09].

It should be noted that few attempts have been made so far to design an AI agent without expert domain knowledge, and the attempts so far are somewhat basic; for most of these, it is the first time a certain technique has been applied to Stratego, so there was little to no previous research they could base their research off of. Most of the research is therefore not very refined or shows a proof of concept.

2.3.1 Optimizing Stratego Heuristics With Genetic Algorithms

In 2003 Ryan Albarelli wrote a paper on his attempts to use genetic algorithms to optimise a set of weights that define how much certain heuristics contribute to the evaluation function [Alb03]. This evaluation function is then used in a Minimax agent. It should be noted that Albarelli does not appear to mention how the agent deals with unknown piece ranks.

Albarelli suggests a basic evaluation function, which consists of a number of heuristics and assorted weights. The heuristics include simple things like 'number of enemy pieces left' and 'number of friendly generals left', etc... To introduce some extra play variation, he introduces a random 'tip' value which modifies the heuristic value by at most 3%. Each weight is defined as a signed 8-bit value.

The evolutionary process uses N-point crossover to generate child candidates from two parents. The fitness function is composed of whether or not there was a

victory, how many remaining pieces there are and how long the game took. Unfortunately, the random tip factor, the skill of the opponent and the starting setup impact the evolutionary process in a negative way, as a good solution may simply have bad luck and therefore be assigned a lower fitness than it should have, or vice versa.

The result from the experiments are somewhat unclear. The first problem here is that Albarelli seems to have mislabelled the axes on his graphs (they appear to be swapped). Albarelli observes that there is a large amount of fluctuation in the maximum and average fitness of the population, though there is an upward trend in the maximum fitness.

He also notes that with a low mutation rate, the population quickly becomes homogeneous. This may indicate a local optimum that is easy to end up in, or that the heuristics as defined by Albarelli simply do not allow for a more refined evaluation function. Note here that while Albarelli uses genetic algorithms to ultimately find and tune the evaluation function, he is still defining the heuristics himself. This will impose a limit on the performance of the agent. The best weight configuration and the exact heuristics are not mentioned, so no conclusions can be drawn from it.

2.3.2 Feasibility of Applying a Genetic Algorithm to Playing Stratego & Reachable Level of Stratego Using Genetic Algorithms

In 2012, Vincent Tunru and Roseline de Boer worked together on their bachelor theses on applying genetic algorithms to Stratego [Tun12; Boe12]. They each wrote their own bachelor thesis, but the work they discuss is the same. For that reason, the two theses are discussed as one.

De Boer and Tunru take a similar approach to Albarelli [Alb03]. The notable differences include the lack of a Minimax tree search in de Boer and Tunru's work, as well as different heuristics that are applied to moves instead of the overall board state. De Boer designed heuristics based on her experience as a Stratego world champion (note: both Roseline and Vincent de Boer have been Stratego world champions). The weights that should be assigned to the heuristics also differ per rank; a scout may weigh a heuristic based on exploration higher than a spy might. Heuristics include but are not limited to whether or not a move would be a piece's first, whether or not it would successfully capture an enemy piece and being in a strategic position relative to the lakes.

De Boer and Tunru apply universal crossover and use a parent selection method based on an exponential relation between the fitness values (e.g. a solution with a fitness twice as large is four times as likely to be chosen). The fitness function is based on whether or not victory was achieved, the number of remaining pieces and the total amount of moves required, similar to Albarelli.

Contrary to Albarelli, de Boer and Tunru find that the average fitness seems to increase fairly steadily. This may in part be due to a larger amount of fine-tuning of the parameters of the evolutionary process by de Boer and Tunru.

In their evaluation, one of the very few comparisons to a different agent is done, specifically against STARBOT, the best agent produced by Arts [Art10]. They find that for a search depth of four STARBOT is able to defeat the agent by de Boer and Tunru, which they named VICKI. This is a fairly impressive result, as VICKI does not use a search tree and thus effectively has a search depth of one. It is also much faster than STARBOT, which they note struggles to complete a game at search depth five within two days (which is in part due to it running on poor hardware), whereas VICKI can find its move in less than a second.

The resulting agent appears to be fairly aggressive, preferring to explore using lower ranked pieces and capturing any known pieces with its higher-ranked pieces. This is fairly atypical when compared to human play; exposing the higher ranked pieces so early tends to be disadvantageous. It may be possible that VICKI has evolved to a local optimum of very aggressive play, which may be exceedingly effective against weaker players, such as current AI agents that do not have the capacity to think far ahead.

2.3.3 Learning to Play Stratego with Convolutional Neural Networks

Schuyler Smith details in his paper from 2015 how he trained a convolutional neural network for use in an AI agent for Stratego [Smi15]. He developed a simple baseline AI and two neural networks, each employing a different strategy. It appears to be the first attempt at an AI agent for Stratego that does not employ any expert domain knowledge at all (previous attempts using genetic algorithms still used a small amount of expert domain knowledge to define heuristics [Tun12; Boe12; Alb03], though no knowledge was required to tune them).

Smith uses a baseline AI for comparisons and to generate some of the training data. This baseline AI uses a Monte Carlo strategy, where every move is assigned a random board state, where the ranks of the pieces are consistent with the current board. It then simulates a number of games for 10-50 moves and evaluates the results based on if victory was achieved and the relative number of pieces left. This agent is very weak, but does defeat a random agent in 94% of games.

For both neural network strategies Smith uses the same input representation, which consists of six layers:

1. The player's immovable pieces (bombs and flags)
2. The player's movable pieces
3. The opponent's movable pieces of a known rank
4. The opponent's known bombs
5. The opponent's unknown movable pieces (not bombs or the flag)
6. The opponent's unknown, unmoved pieces

The second and third layer are encoded using the piece rank, e.g. a scout is encoded as 2, a miner as 3, etc... The other layers are encoded in binary. Smith notes that this representation is only able to capture first-order inferences, e.g. if a piece has moved, it is not a bomb or a flag.

The first strategy was to have the neural network predict which move it should make. Smith recorded the moves made by the baseline AI for a thousand games and used this as training data. This training process was very unforgiving, as the neural network would only be credited if it predicted the exact move the baseline AI made. This is particularly difficult, as the baseline AI uses some level of randomness to determine its move. To make the move, a single forward pass through the network was done, and the move with the highest probability that is also valid was chosen as the move to make.

The second strategy involves using a neural network to replace the evaluation function in a traditional Minimax search algorithm. Here, Smith selected 15000 board states and ran a number of random games on them to estimate the probability

of victory in each state. The neural network would then train on this information. Compared to the first strategy, this method made for a much easier training process.

In his evaluation, Smith notes that strategy 2 appeared to be the strongest agent, followed by the baseline AI. Strategy 1 did not work very well, though it could still beat a random agent in 87% of games. It should be noted that this is still a fairly good result, as strategy 1 takes 1̃ms to select its move, whereas the baseline AI and strategy 2 take somewhere between 0.5-1 seconds.

There are still considerable improvements to attempt here. For example, Smith's input encoding does not take into account what information the opponent knows about the player, as well as any deeper inferences. Moreover, encoding information about known pieces as a rank may not be optimal; it may be better to split this into multiple binary channels. It may also be possible to combine strategies 1 and 2 into one agent, which does some kind of tree search but prioritises exploring moves that strategy 1 thinks are good moves and evaluates boards using strategy 2.

Chapter 3

Research Question

As mentioned previously, the goal of this thesis is to create an AI agent that is capable of playing Stratego at a decent level. Ideally it would surpass previous attempts at computers playing Stratego, but as previous attempts have had mixed results this may not be quite attainable within the scope of a Master thesis. Nonetheless, I intend to create an AI agent that can at least pose some challenge to a beginning human Stratego player.

The research question is therefore the following:

> *What methods can be used to let an AI agent play Stratego at higher levels than previously achieved?*

Stratego quite naturally lets itself subdivide it into smaller subproblems. While the quality of the solutions to each of these subproblems will of course impact one another, they can mostly be treated separately for development purposes. These subproblems are the following:

- *The Setup Problem*: The first thing any Stratego player needs to do is come up with a strong starting setup. There are a number of things that make any setup strong, but some unpredictability is also very helpful.
- *The Information Problem*: Stratego is a game with hidden information, e.g. the opponent's starting setup. Being able to accurately guess the ranks of the opponent's pieces will help any Stratego AI.
- *The Static Lookahead Analysis Problem*: Ultimately, an AI player has to decide which move to take. Finding out which move that should be appears to be no trivial task, as there are considerable complications to consider when evaluating the board. A necessary step is to develop a method that can do a good static lookahead analysis.
- *The Dynamic Lookahead Analysis Problem*: If a static analysis method is perfect, a dynamic one is unnecessary. However, given that it is extremely unlikely that a perfect method can be developed, a dynamic lookahead analysis is very important to improve the performance of the AI agent. Here, complications arise that make tree-search methods less effective. Some kind of mitigation for the issues that Stratego poses needs to be found.

It is fairly easy to see why these subproblems are mostly disconnected and can be solved independently. An AI agent would create a setup and during play would guess what the enemy piece ranks are. It would then perform a lookahead analysis in every turn. The performance of the agent will depend on how well each of these problems are solved. Because of this, we can define four smaller research questions:

1. *How can a setup for Stratego be created that is of a high quality?*
2. *What methods can be used to guess the enemy piece ranks as accurately as possible?*
3. *In what way can we best perform a static lookahead analysis?*
4. *How can we overcome the complications posed by Stratego in a dynamic lookahead analysis?*

3.1 Relevance of research question

The simplest and most direct reason for why this research question is relevant is that the current best Stratego AI agent still poses little challenge to an average or slightly-above average player. For more advanced players, playing against it is simply not as entertaining. For these people, it would be nice if they had an AI opponent that could actually give them a hard time during play.

There are also more general reasons why developing an AI agent for Stratego is relevant. Stratego is quite distinct when compared to Chess or Go due to the large amount of information that is hidden from the player. This also adds a bluffing aspect to the game. These are elements that so far have not been researched to a great extent. A comparable game that comes to mind is Poker, which has only recently seen an AI agent beat expert players in a tournament setting. But even here, the hidden information can be probabilistically estimated; players don't select the cards they are dealt. With Stratego this is different, as players do choose their own starting setup.

This may also have practical implications. A good Stratego AI agent is one that understands risks and can balance multiple different complicated factors, such as the importance of hiding and revealing information. This may be useful when dealing with situations where certain actions taken by an agent are visible, but not the information that they are based on. Being able to act accordingly may be very valuable. Perhaps in those cases, techniques that may be applied in this thesis may prove to be useful.

Chapter 4

Methodology

As outlined in chapter 3, Stratego can be divided into multiple subproblems. In this thesis I will attempt to solve these problems separately as well. For each of these problems I will outline a number of ways in which these subproblems could be potentially solved. I will most likely rely on methods and techniques that do not require large amounts of expert domain knowledge, as I simply do not possess such knowledge. I am a decent Stratego player, but I would rate myself as a beginning or average player. I will therefore focus on methods that do not utilise expert knowledge, either by self-improvement or information extracted from public databases.

It should be mentioned that solving each subproblem separately in complete isolation of the others is not really feasible. Depending on the quality and characteristics of a solution for a subproblem, it may become easier or harder to estimate the quality of the solutions for the other subproblems. It may also create new ideas for solutions for other subproblems. As an example, work on the *Setup Problem* led to new insights that helped create solutions for the *Information Problem*.

As a result, during development I often switched between the different subproblems. Presenting the work in a chronological order however would be confusing, so this thesis presents the work categorised per subproblem. This means that occasionally there may be a reference to a section further ahead, but I have tried to keep these to a minimum.

4.1 StrAItego

The AI agent will be developed within the StrAItego framework. This is a framework that I have written in C#.NET, the programming language I am personally most comfortable with. I have used the Windows Forms library for building the UI and will use the SciSharp package which offers TensorFlow.NET, a .NET binding for Google's TensorFlow, which will allow me to train neural networks. To use the networks, I will use the more lightweight TF.Lite format which is more suited to individual network calls instead of batch processing like regular TensorFlow is.

The framework is a fully functional Stratego game. It also offers an interface for AI agents to play games. The framework does a couple things for every AI agent, including:

- *Board inversions*: The board is always inverted so that the AI agent is always playing as the red player. Every AI agent can therefore assume that a blue piece is an enemy piece, and the top side of the board is the opponent's side of the board.
- *Move validity*: When being requested to make a move, the framework provides the AI agents with methods that will create a list of all valid moves that it can

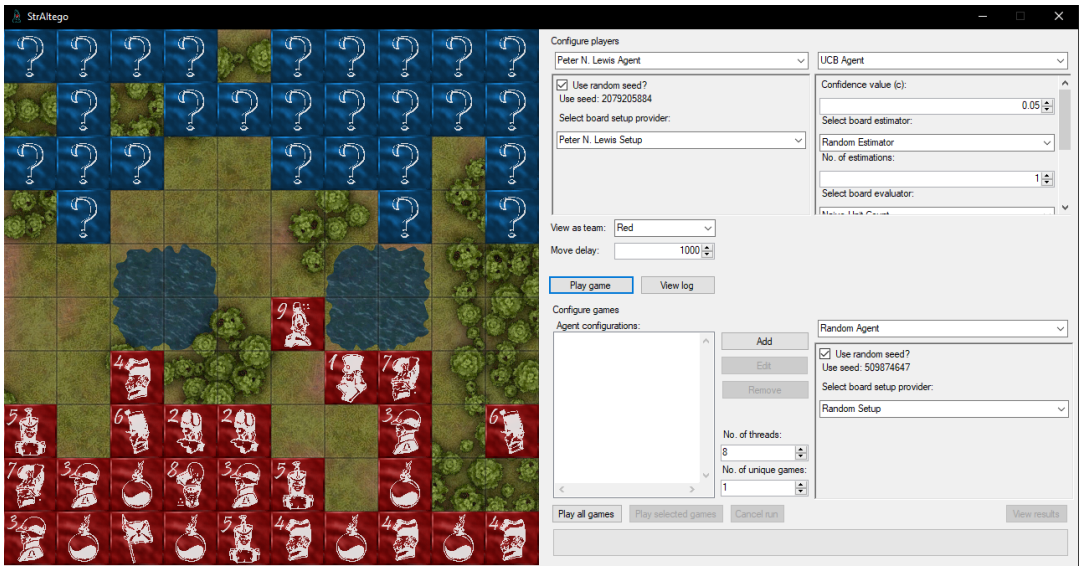

FIGURE 4.1: A screenshot of the StrAItego interface.

make. AI agents therefore will not have to for example check if they are compliant with the Two-Squares rule, as any move violating that rule will simply not be in the list.

- *Full information and inferences*: The framework tracks for every piece which rank it could possibly have. This includes the fact that moved pieces can not be a bomb or the flag, but also inferences such as eliminating the possibility that a piece is of a certain rank, if all pieces of that rank have already been revealed elsewhere.
- *Execute all game logic*: AI agents only have to select which move they wish to play. The execution of the move and all logical inferences resulting from it (e.g. revealing information about certain pieces) are all handled by the framework.
- *Logging capabilities*: To aid the development process, AI agents can log messages that can be reviewed during or after the game.

Additionally, the framework has options for running many games with different AI agents pitted against one another for testing purposes. These 'tournaments' are run in parallel using multi-threading and are heavily optimised to make as much usage of the processing capabilities of the computer it is running on as possible. For example, running 10000 games between random agents takes approximately 4 seconds on an Intel i7-4790k. Results are also automatically collected and displayed in tables.

4.2 The Gravon database

The Gravon database [Jun15] is a database of 51338 games of classic Stratego played online via the Gravon website, played between 2003 and 2015. Players on this website can opt to have their game be recorded in the publicly accessible database for review with the StraDoS2 application, which allows you to replay games from the database. Each game is recorded in its own file, with either a .xml or .gsn file extension, using a human-readable notation. This makes parsing the files in a different project fairly straightforward.

The purpose of this database within this thesis is to provide some basis from which to build an AI agent upon. By analysing the games, it is for example possible to see what humans would do in certain situations or what starting setups they like to use. There are some caveats to this; not all games are of a high quality and the games have been fully anonymised, which means the comparative strength of the players is unknown. This means that it is hard to estimate if either player is actually playing well. It is possible that a good move by a good player is countered by a brilliant move from a brilliant player, or that an absolute blunder from a good player is followed by a move from a bad player that does not capitalise on the mistake.

However, one thing that can be assumed is that whatever starting setup is used or whatever moves are played during the game, will on average be much stronger than a random starting setup or a randomly chosen move. This means that even if an AI agent based on data from the Gravon database may be flawed in certain ways, it will still be considerably better than a random agent and offer a good base to develop further.

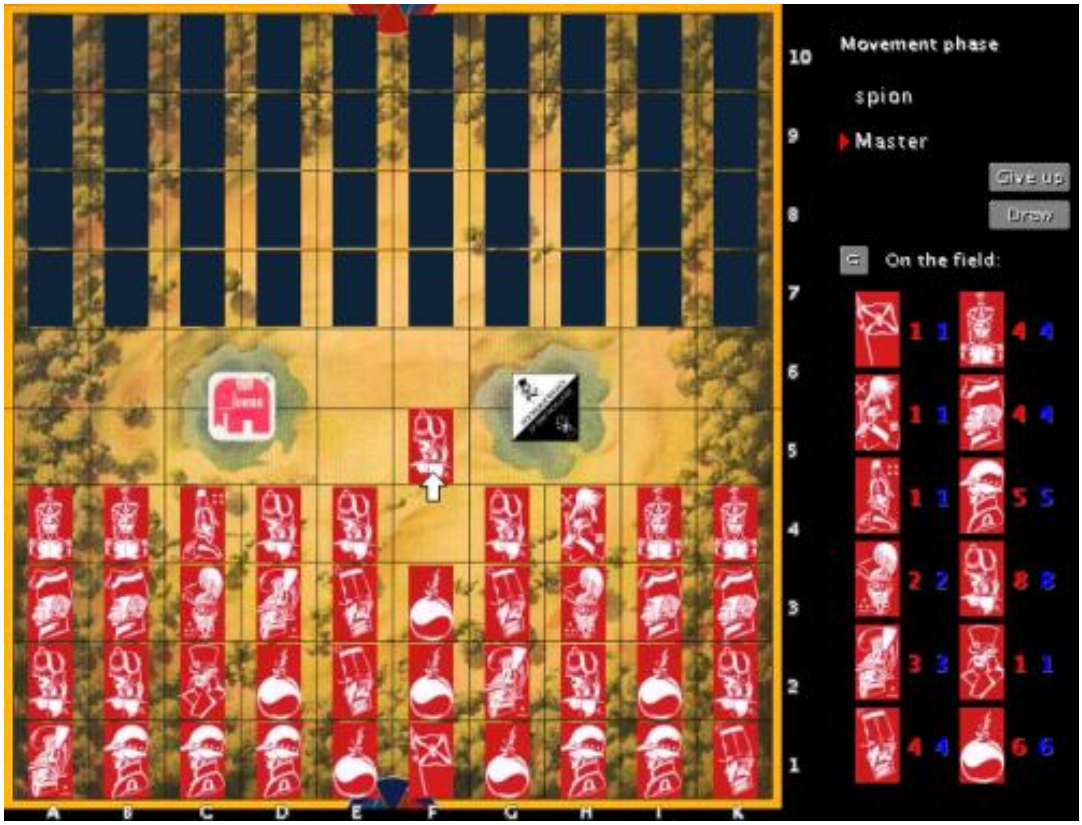

FIGURE 4.2: A screenshot of the StraDoS2 viewer.

4.3 Determining agent quality

In order to determine if an agent is actually playing the game well, we need some kind of quality evaluation method. As it is not really possible to directly assign some number to the performance of an AI agent, this will have to be some comparative method. This involves pitting the test subject against some benchmark AI agent and recording the results of a large number of games (typically at least 50 or 100). In this thesis I typically use one or multiple benchmark AI agents to evaluate if the current AI agent has strengths or weaknesses against specific AI agents. As the gameplay

from each AI agent can drastically differ (e.g. one agent may be overly aggressive or perhaps rather defensive), testing against multiple agents seems necessary to prevent a type of rock-paper-scissors scenario, where agent A may be effective against agent B, which in turn is strong against agent C, which then is very good against agent A. Only by testing them all against each other can a proper comparison be done.

The AI agents that are typically tested against are the following:

- *Random Agent*: A very simple agent that plays a random valid move each turn. It needs little explanation that this is a very weak agent, and really only serves as a simple test to see if something is very wrong.
- *Random Agent (Avoids Piece Loss)*: A very simple agent that plays a random valid move each turn that will not result in the guaranteed loss of that piece (e.g. it will not attack a known enemy Marshal with a Scout). If no such move exists, it plays a random valid move instead. Once again a very weak agent, but one that is a bit more cautious than the Random Agent.
- *Peter N. Lewis Agent:* A C#reïmplementation of the agent written by Peter N. Lewis in 1997 that went on to convincingly win the MacTech Programming Challenge (see section 4.3.1). A fairly simple agent that uses 12 heuristics to determine which move to make. This agent poses a much greater challenge than the random agents, but has some obvious flaws that can be exploited.
- *"Best-yet" Agent*: The best agent developed during the thesis so far. Tested against to see if the new agent is better than what was developed before.
- *Variation of the test subject*: A slight modification of the test subject (e.g. with different hyperparameters) to tune and optimise the test subject.

The results of the games are expressed as a *winrate*, which is the percentage of games that ended in a victory out of the total number of games. Ties are not included in this total as they are quite rare and only really happen if the turn limit for a game is exceeded (e.g. a game takes more than 2000 turns to finish). Although the StrAItego program considers this a tie, it can probably be more accurately marked as "indeterminate". These incredibly long games with no clear outcome do not really contribute to the comparison in a meaningful way and are thus excluded.

4.3.1 Peter N. Lewis Agent

In 1997 MacTech journal held a programming contest to see who could develop the best Stratego AI [Boo97]. The winning solution was created by Peter N. Lewis from Perth, Australia and was written in C++. Out of the 32 games his agent played, it managed to win 30 games, far outscoring his competition (the second-best agent only managed to win 13 games). Peter's agent uses 12 heuristics that are traversed in order to determine which move it should make. This results in a very aggressive AI agent that usually wins by eliminating all enemy pieces instead of capturing the flag (19 games were won this way, compared to only 10 flag capture victories).

This agent has been reïmplemented in StrAItego in C#to use as a benchmark agent, as only testing against random agents does not give a great indication of how strong an agent is exactly.

The agent has 12 heuristics that it tries out in order. The move that matches the highest heuristic in the list is selected. For example, if no moves match heuristics 1

and 2 but one does match 3, that move is executed and no further evaluations are made. The heuristics are as follows:

1. *Use Spy*: If the spy is next to the enemy marshal, capture it.
2. *Defend against Spy*: If the marshal is next to the enemy spy, capture it. If it is next to unknown pieces, evaluate if it should be captured or if the marshal should move away.
3. *Attack weaker*: If a known piece is next to a weaker known piece, capture it unless the new position is dangerous.
4. *Explore attack*: If a scout, sergeant, lieutenant or captain is next to an unknown piece, capture it.
5. *Retreat*: If a known piece is next to a stronger enemy piece, run away.
6. *Scout*: Try advancing scouts forward rapidly.
7. *Attack distant*: If a known piece is distant but a path exists that moves a stronger piece towards it, advance the stronger piece over the path.
8. *Explore distant*: Advance lower-ranked pieces towards unknown pieces.
9. *Attack known with same distant*: If a known piece can be attacked by a known identical piece, advance towards it and capture it.
10. *Move forward*: Move any piece we can forward.
11. *Move*: Move any piece we can.
12. *Resign*: No moves can be made at all, so the agent should resign.

In the source files, Lewis mentions a thirteenth heuristic called *Find Flag*, but due to time constraints it was not implemented. This may also be why the Peter N. Lewis agent tends to win by capturing all movable pieces of the opponent, rather than capturing the flag.

The Peter N. Lewis agent behaves very aggressively, trying to capture as many enemy pieces it can without losing too many of its own. Because the *Attack distant* and *Explore distant* heuristics use an actual pathfinding algorithm, it is capable of moving pieces from one place on the board towards a faraway position. This is a notable advantage compared to typical tree-search-based agents, as exploring such a long path in a tree-search with a high branching factor is largely infeasible.

The aggression and determination of the Peter N. Lewis agent is a considerable strength against AI agents. However, this method is fairly weak against human players. A player who realises how this agent works will quickly see that a piece that is moving towards it is likely of a higher rank. This predictability makes the Peter N. Lewis agent fairly easy to trick. Simply revealing the general will see the agent immediately trying to move its marshal towards it, as it is the only piece that can capture it. Because *Defend against Spy* only moves the marshal out of a potentially dangerous situation, but does not prevent it from moving into such a situation, the marshal can be eliminated fairly easily.

Regardless of its flaws, the Peter N. Lewis agent makes for a good benchmark to test how well an AI agent deals with an extremely aggressive opponent.

Chapter 5

Setup Problem

The very first thing an AI agent playing Stratego has to do is come up with a starting setup. Even though at this point the game has barely started, what may be the most important single decision that determines the outcome of the game is made at this point. A good starting setup allows a player to play the right balance of offence and defence, allows a player to move important pieces around the board more easily and protects their flag. It is important to note that these qualities can never be condensed into a single perfect starting setup, as perhaps a quality that is even more important to take into consideration is the unexpectedness of the setup. One of the primary factors that determines who wins in a game of Stratego is how well a player can guess the enemy's piece ranks. If there were a single perfect starting setup, or even a small number of them, it would be easy to deduce which setup is being used by discovering just a few pieces.

A good example of a way to subvert he enemy's expectations is the so-called "Shoreline bluff" or "Lakeside bluff". The idea is that to protect the flag, players would typically place their flag in the back row. This makes sense, as it is furthest removed from the opponent's pieces and provides ample opportunities to stifle their attacks with other defensively used pieces. The trick with the Shoreline bluff is to place the flag on the front row next to one of the middle lakes instead. This is a dangerous location and would be fairly easy to reach, but is also a lot less expected. A player who is not expecting this could break through the initial defences, spend their time to reach the back row, maybe even defuse a bomb structure that looks like it is protecting the flag only to find the flag was never there. An example of such a setup can be found in figure 5.1.

The process of creating a good setup is largely independent of the rest of the agent. As the setup creation only happens once at the start of the game, before any information is revealed at all, we can mostly treat the *Setup Problem* as separate from the other problems. There is no setup that we can begin countering yet, nor is there any opponent behaviour available that suggests we should either take a more offensive or a defensive approach.

The question of "what exactly makes a good setup?" is a very difficult one. It is possible to quantify how often a setup wins or loses, but that does not necessarily indicate quality; it may also be possible that stronger players prefer using a specific setup against weaker players, but would not use the same setup against players of a greater strength. It is possible to simply check which setups are being used, for example using the Gravon database [Jun15], as a setup that is being used will probably be at least better than some randomly generated setup.

This is not to say that randomly generated setups are necessarily bad, or can be improved using some expert knowledge. As de Boer has shown [Boe07], it is possible to generate random setups and evaluate them based on certain hand-crafted features. After finding a good potential candidate, some small adjustments can be

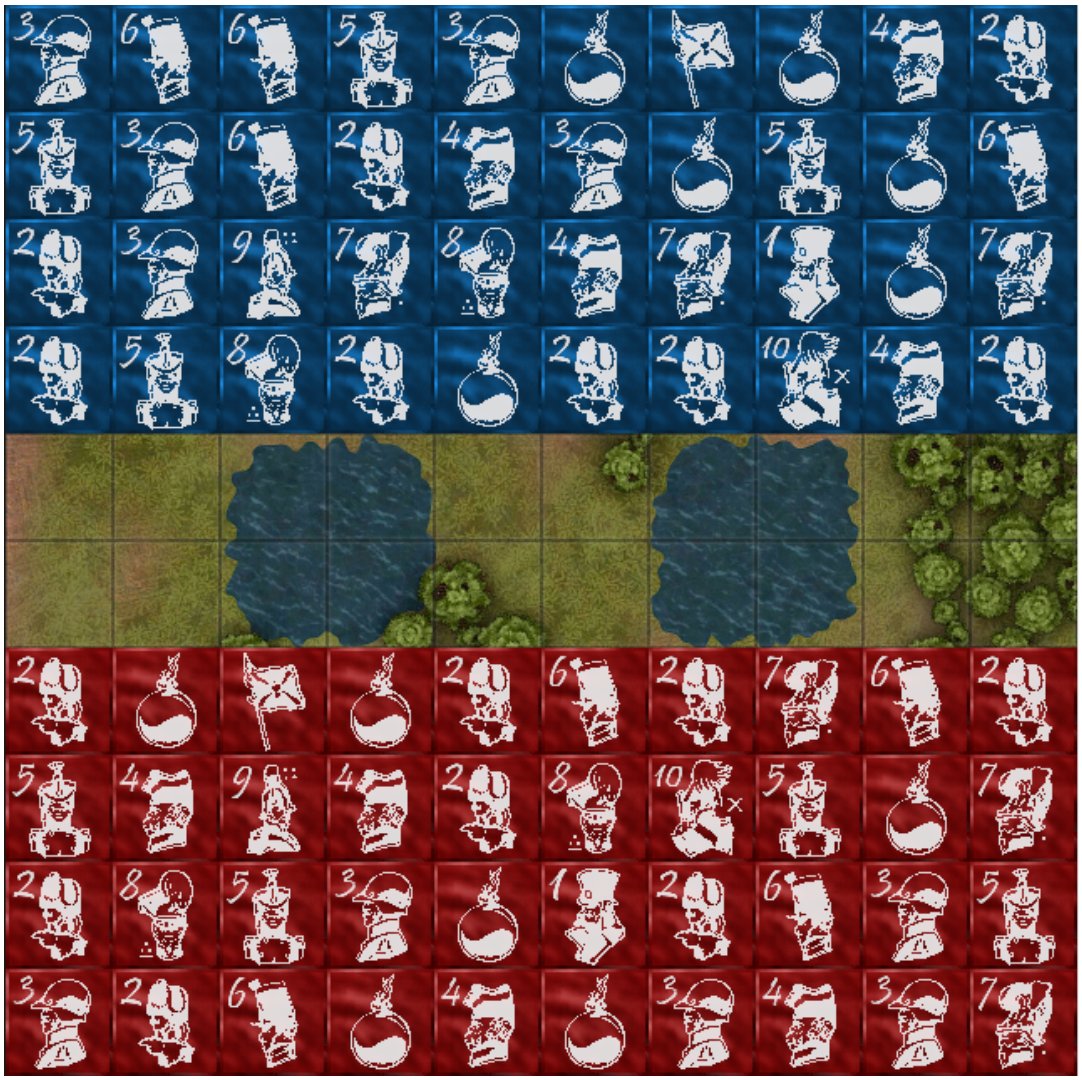

FIGURE 5.1: A game of Stratego where the player in Red is using a version of the Shoreline bluff, whereas Blue is playing with a more standard setup.

made to create a setup that someone like de Boer could feasibly use, or at least would not immediately discard as a terrible setup.

As I personally lack the expert knowledge to hand-craft strong Stratego setups or evaluate them in a meaningful way, other than filtering out *really* bad setups, I will have to rely on using setups from specific sources or developing an evaluation method that can evaluate setups based on machine learning. I have created 6 different *setup providers*, e.g. a method that (given a random seed) provides a starting setup to an AI agent. Out of these 4 pull a setup from a provided dataset, the remaining 2 create a completely new setup.

5.1 Setup providers using a dataset

This section contains 4 setup providers that pull a setup directly from a dataset. These setups are usually of a fairly high quality, but they may be considered too predictable if overused. Moreover, it should be considered that an AI agent using any

of these setup providers is not really creating its own setup; it is incapable of producing something original or something better than what a human would typically create.

Most of these setup providers contain very few setups (except for the Gravon Setup Provider). Because of this I would consider them to be too predictable for use against human players, particularly if they have played a few games against an agent that is strictly using those setups. I will detail them here but not typically use them for further comparisons or experiments.

5.1.1 Peter N. Lewis Setup Provider

This setup provider is the simplest; it provides a single setup used by the Peter N. Lewis-agent when it was originally developed. Lewis in his source code defines three possible starting setups that were intended to be used by his agent, but only one is actually used. This setup is fairly offensively oriented, which matches the kind of "hunter-killer"-playstyle employed by the Peter N. Lewis-agent.

FIGURE 5.2: The setup provided by the *Peter N. Lewis Setup Provider*, intended to be used by the agent of the same name.

5.1.2 Accolade Setup Provider

In the 1990s a company called Accolade published a video game version of Stratego for the Commodore 64. The game featured a very basic, not very good AI agent that always used one of 13 different starting setups. These have been collected in an online strategy guide for Stratego that details the setups and their strengths and weaknesses [Ult].

According to this strategy guide, various players have pointed out that the setups provided by Accolade are not very good setups. Nonetheless, I have included them in the *Accolade Setup Provider*.

5.1.3 Vincent de Boer Setup Provider

In his master thesis de Boer details 6 setups that he has either used in the past or still uses [Boe07]. He uses these setups as a type of test for his system that tries to

score setups based on certain features he believes are beneficial. The setups work as a kind confirmation; his system should confirm de Boer's belief that these setups are good. If the system gives the setups a high score as he expected, the system works.

The highest-scoring setup is shown in figure 5.3. It achieved a score of 22 under de Boer's scoring system.

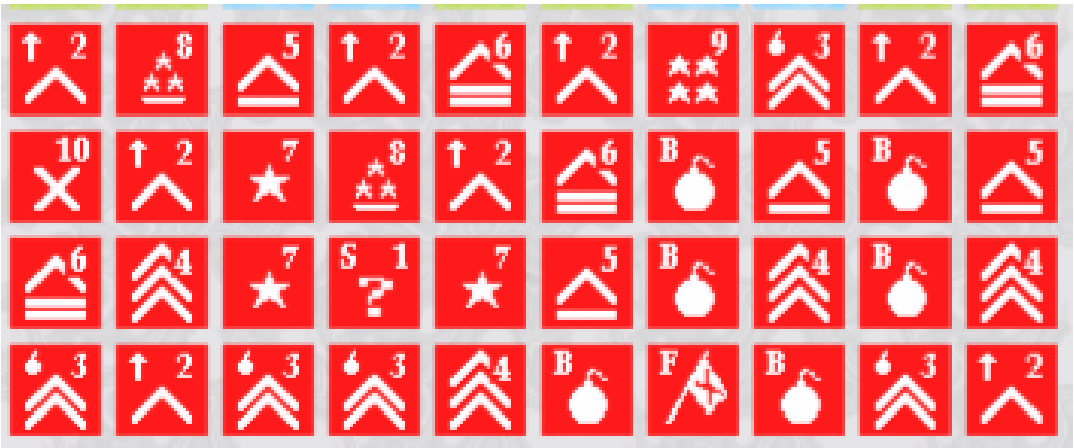

FIGURE 5.3: The setup in the Vincent de Boer Setup Provider that scores highest according to de Boer's scoring system.

5.1.4 Gravon Setup Provider

The Gravon database contains full records of over 50000 games [Jun15]. As every game necessarily has a setup for either player, this means the database has more than 100000 setups that were at some point utilised by human players. These have been compiled into a single *Gravon Setup Provider*, which provides a random setup from this set. This makes it possible to run experiments with setups that AI agents can realistically encounter when playing against human players.

Working with the assumption that humans are pretty good at playing the game, we can assume that on average the setups in the Gravon database are at the very least 'not terrible', certainly better than a fully randomly generated setup. There are some notable exceptions of course, from players who were maybe not playing too seriously or who were experimenting with something completely new that may not have worked out too well.

This setup provider is particularly useful when testing e.g. the effectiveness of piece rank estimators, as the variety in setups is large and diverse. Additionally, because these are human-made setups we avoid any bias that could be introduced by creating original setups in a specific way.

5.2 Setup providers that create original setups

As calculated by Arts [Art10], Stratego has up to $\sim 10^{23}$ different possible starting setups. But a large database such as the Gravon database only has just over 100000 setups, a very small fraction of this incredibly large number. This raises an immediate question: does the possible reuse of setups pose a risk in terms of predictability? It may be that only a few pieces have to be revealed before an opponent could deduce what the rest of the setup might look like.

According to de Boer, this is not unrealistic [Boe07]. He mentions that he reused a certain setup a couple times in a tournament, until he played a game against an opponent who was unusually immune to bluffing tactics and always seemed able to make the right captures. When this opponent attempted to capture a piece that de Boer had switched with a bomb in the setup, losing a high-ranking piece in the process, he revealed that he had written down and memorised the exact setup and could, to de Boer's amazement, perfectly reveal the exact ranks of de Boer's remaining 25 pieces.

Although it would likely require far too much memorisation from a human than can be reasonably expected to do something similar for a more significant amount of setups, computers of course have no such limitations. It is thus not infeasible that a computer could quickly deduce which setup is being used if the opponent is using one from the Gravon database, for example. Such a method to determine enemy piece ranks is put into practice with the *Database Estimator* (see 7.2.4).

Because such vulnerabilities would pose a large risk to an agent using setups directly from a database, it is a good idea to also be able to create original setups that are not vulnerable to these techniques. We will consider two methods for creating new, original setups. A third variation was originally developed, but it did not provide meaningfully different setups in both diversity and quality and thus was dropped earlier in development in favour of a focus on the other variation.

5.2.1 Random Setup Provider

The easiest way to create a totally original setup is to simply fully randomise the position of each of the pieces. This creates a unique setup that is completely unpredictable to the opponent. While the predictability factor may be incredibly large, the strategic qualities of the setups produced this way are often very lacking. For example, there is a 15% chance that the *Random Setup Provider* creates a setup where the flag is positioned directly in front of one of the three open lanes, a position which the opponent could theoretically capture with a Scout on turn 1.

Even if the flag is placed more favourably, the other pieces likely are not in good positions. This makes using these setups not very advisable. Because it is possible that a setup is extraordinarily weak, this setup provider is not really recommended for actually playing games. It can however serve as a baseline, to indicate how much of an advantage is gained by using better setups.

To quickly illustrate this, an experiment was done where two Random-agents which select completely random moves play against each other for 20000 games. The only difference is that one agent uses the *Gravon Setup Provider* (see section 5.1.4) and the other is using the *Random Setup Provider*. The agent using the Gravon setups achieves a winrate of 64.9%, which shows that with high significance we can conclude that the quality of random setups is not very good.

5.2.2 Naive RvH Setup Provider

Creating setups that are better than random setups may seem like a simple task for a human. However, figuring out exactly which features actually make a setup better or stronger poses a significant challenge. Doing this in a truly accurate manner would require playing thousands of games with very strong AI agents, to see what setups do well and which do not. Because a very strong AI agent has not really been developed yet, we have to stick to the best Stratego-playing agents that we do have some access to: humans.

We can safely assume that human players can reliably produce setups that are stronger than random setups. We could thus change our goal slightly; instead of making setups that are strong, we could try making setups that are *human*. If a setup looks human enough, it is probably also better than a fully random setup.

To estimate how human a setup is, a neural network has been trained to try and make the distinction between a *random* versus a *human* setup. We do this by creating a dataset which features all the setups from the Gravon database[Jun15] and an equal number of setups that have been randomly generated.

Training a neural network

The network has an input vector of 480 nodes. The setups are converted to a one-hot encoding. Each of the 40 positions in the setup can hold a piece which has one out of 12 ranks, which means we need 480 inputs to encode each possible pair of position and rank. This input layer is followed by five fully-connected hidden layers of 150 nodes each, using a ReLu activation function. The softmax output layer has two nodes; one for indicating how human a setup is and the other for how random a setup is. Because it is a softmax-layer, this is translated to a percentage indicating the likelihood of a setup belonging to a certain class.

Cross-entropy loss initially starts out at $\sim$ 0.7, and after training went down to 0.21. At this point, an accuracy had been achieved of $\sim$ 96%. Testing shows that the network at this point was very capable at discerning between human and random setups, although it was mostly accurate at finding human setups, whereas occasionally it would still misidentify a random setup as a human setup. It would be more preferable if the network was a bit more strict in what it classifies as human.

To make the network more strict, a number of adversary setups were generated. This is done by generating random setups and evaluating them with the network. If the network is more than 90% confident that the random setup is human, it is considered an adversary setup. These adversary setups are then mixed with the Gravon setups and random setups, such that the number of adversary and random setups is equal and that the sum of the adversary and random setups is equal to the number of Gravon setups.

After continuing the training on the new dataset, the network now only reaches an accuracy of 93%, which is still fairly high. A notable difference however is that it now is 99% accurate in detecting random setups, whereas it might occasionally mark a human setup as random. This is not really an issue; after all we are more interested in recognising setups that are typical for humans, rather than a setup that was man-made but looks mostly random.

Generating setups

A neural network that can discern between a human and a random setup is not immediately capable of actually creating a human-like setup on its own. The most simple and perhaps naive way of doing it would be to create a large amount of random setups, evaluate them with the neural network and select either the most human setup or the first sufficiently human setup. This is the idea behind the *Naive RvH Setup Provider*.

The provider generates at most 10000 setups, evaluating them in order. Once it finds a setup that is considered human with a confidence of at least 99%, it immediately returns that setup. If it does not find such a setup, it selects the most human one out of the 10000 setups generated.

Because the setups are still fully randomly generated, there is a high degree of unpredictability in these setups. From visual inspection it is clear that these setups are probably not *perfect*, but they clearly show certain features. As an example, the flag typically has a few bombs near or around it. Bombs also tend to cluster near each other and high-ranking pieces seem to be spread out a bit.

Unfortunately, there is an apparent downside to a lot of the setups that are generated. The network makes a distinction between human and random, but it does not make a distinction between setups that are frequently used by humans and infrequently used by humans. For example, the network has learned that setups employing the *Shoreline bluff* tactic of placing a flag next to the lakes are very human. This means that the setup provider produces a disproportionate amount of setups that also employ this tactic.

To counteract this issue, the generation of random setups was adjusted. The random generation works by assigning each piece a random number between 0 and 1, and then simply sorting them. The lowest sorted piece is placed on the bottom-left corner and the rest of the pieces are placed row by row until the highest sorted piece is placed in the upper-right corner. To push the flag back away from the shores, the randomly generated number for the flag is divided by two, meaning it is sorted far lower in the list and thus placed further towards the back. This means that the flag is usually near the back row but it is still possible for it to be closer to the front.

A simple experiment was done to evaluate if these setups are indeed better than just a random setup. Two Random agents played against each other in 20000 games, one agent playing with the *Random Setup Provider* and the other with the *Naive RvH Setup Provider*. The agent using the Naive RvH Setup Provider managed to achieve a winrate of 66.1%, which means we can conclude with certainty that these setups are better than random setups.

5.3 Conclusion

The goal of this chapter was to look at multiple ways of supplying good setups to an AI agent. Using a good setup is paramount to success. It is very important to find the right balance between setup quality and setup predictability. It is fairly easy to create a setup provider that produces one of the two, but it is hard to create one that produces both at the same time. The *Gravon Setup Provider* is good at providing quality setups, yet they are potentially very predictable. The *Naive RvH Setup Provider* in contrast is much better at creating something unpredictable, yet in terms of quality it may be lacking compared to the Gravon Setup Provider.

The selection of setup provider therefore depends on a number of external factors. For example, when playing against an agent which is using the *Database Estimator* (see 7.2.4), it may be unwise to use the *Gravon Setup Provider*. Yet when playing against a human player, those same setups may be more than sufficient, as humans are not capable of memorising over 100000 setups and quickly deducing which one is being used.

Therefore, it is not really possible to designate a single setup provider as the 'best' or 'most optimal' setup provider. Even comparing them in terms of setup quality is already difficult, as much depends on the agent using them. For this thesis, we typically use the *Gravon Setup Provider* or where necessary the *Naive RvH Setup Provider*, if the results of an experiment may be disproportionally affected by an agent figuring out exactly which setup is being used very quickly.

Chapter 6

Dynamic Evaluation Problem

Stratego is a highly complex game for a computer to solve effectively. Not only does the lack of information pose a significant issue, the large branching factor each turn and relatively small game impact of individual moves is not an insignificant problem either. In Chess for example, almost every move is important as it immediately changes the strategic value of the position by a considerable amount. Pieces can move across the entire board in one move, and project large amounts of influence over the rest of the board (e.g. by covering or threatening other pieces). In Stratego, a piece can only move one square at a time, and the only piece that can move further is the low-ranked scout, whose only tactical purpose is to reveal enemy pieces, not capture them. This also makes games last much longer on average. As Arts calculated, the upper bound on the game tree complexity is $\sim 10^{535}$ including chance nodes and $\sim 10^{509}$ without chance nodes [Art10], which is much higher than Chess at 'only' $\sim 10^{123}$.

This complexity puts any tree-search based algorithm at a significant disadvantage. The depth of the tree that needs to be searched is large, as moves have little individual impact, yet the tree itself is also very wide. The chance nodes in this tree further complicate this issue. For this reason de Boer opted to not use a tree-search algorithm at all and instead opted for his solution based on plans [Boe07]. This approach relies on a lot of expert domain knowledge, which de Boer as a Stratego world champion certainly has. Unfortunately, it also required de Boer to carefully balance and fine-tune the weights for each of the plans he developed. This means introducing new knowledge to his AI agent does not scale well and thus becomes increasingly difficult.

To develop an AI agent without a significant amount of expert domain knowledge therefore still requires some kind of tree-search algorithm. The effectiveness of the tree-search will determine how much it can contribute to the final solution quality. Making the tree-search as effective as possible is at the heart of the *Dynamic Evaluation Problem*.

6.1 Minimax

The Minimax algorithm is likely the most well-known tree-search algorithm. It is also the most well-studied algorithm when it comes to its application in Stratego. Considerable efforts have been made to optimise it as much as possible, dealing with chance nodes and aggressively pruning as many branches as possible without affecting the final result, or to use it in conjunction with other new techniques [Bal82; PT09; Moh09; Sta09; SW09; Art10; RG18; Alb03; Smi15].

In the Minimax algorithm, the game-tree is searched fairly exhaustively up to a certain depth, only pruning branches when the outcome can no longer be changed

by them. The optimal move is then found by alternating picking the best and the worst move for the playing agent. It simulates what would happen if both players consistently made the most optimal move possible, and tries to find the best possible outcome. It can be made more efficient by various techniques such as α-β-pruning, heuristics-based pruning, quiescence search, etc..., which seek to limit the amount of the game tree that needs to be explored.

Despite these efforts, agents based on the Minimax algorithm still lack in quality. It seems that Arts' STARETC agent has the best implementation [Art10], but it can still only search up to a depth of at most 5 moves before the search costs become incredibly high. Even at that search depth, Arts found that a maximum of 33.6 million nodes were searched in a particularly bad case. Searching this many nodes every turn costs far too much processing power for a relatively low search depth.

Although there are potential optimisations to be made with the Minimax algorithm, in particular in connection to the chance nodes, we will instead opt to explore different tree-search methods that have not been explored as fully as Minimax, but still hold considerable potential.

6.2 Monte-Carlo Tree Search

More recently, the Monte-Carlo Tree Search (MCTS) method has been successfully applied to several games, most notably the AI developed by Deepmind to master Go [Sil+16]. It has also previously been applied to Stratego by Mets [Met08], which was a fairly limited application on the algorithm but showed good potential nonetheless. Compared to other tree-search algorithms, MCTS often manages to make high quality decisions while exploring far less of the tree. This can partially be explained by how MCTS evaluates nodes at every level in the tree instead of only at the leaf nodes, allowing it to better direct the search and exploration of the tree.

The MCTS algorithm follows 4 steps a number of times, which can be a predetermined number or a dynamically selected number. The steps are as follows:

1. *Selection*: Starting from the root node, select successive child nodes (moves in the game) until a leaf node is reached that has some child node that has not yet been simulated, according to some *selection method*.

2. *Expansion*: From the leaf node, attach a new leaf node that represents a specific move (unless no more moves can be made) and move to the newly created child node.

3. *Simulation*: The current state is simulated according to some kind of *simulation method* and receives a value representing the desirability of that state.

4. *Backpropagation*: The resulting value is backpropagated to the root node, updating the values of these nodes along the way.

The *simulation method* can be any method that converts the node state into a numerical value that represents the desirability of the state. A commonly used simulation method is the random rollout, where random decisions are made until some kind of end state is reached, the type of which determines the value assigned to the node. This is the method used by Mets [Met08]. Other methods exist that do not necessarily require a full playout of the game but instead perform an evaluation of the current state, such as neural networks. I will refer to these methods as *Evaluation methods* and discuss them further in chapter 8.

The *selection method* determines which nodes are explored. A naive option would be to randomly select a node, but better methods typically use the value of the child nodes as well as how often they have been visited. The aim of the selection method is to select the child nodes that expand the search tree towards the most useful nodes. To determine what moves to explore, we can use domain-agnostic methods such as ε-greedy and Upper-Confidence Bound. These are fairly easy to implement and should provide a selection method of acceptable quality already. However, it is also possible to add some knowledge about Stratego to the selection method so that it is able to make better selection.

This chapter will consider a few selection methods. The primary concern is AI agent quality, although it is not possible to fully ignore performance. Whichever selection method is used, it may be called $O(dn)$ times per move, with d the maximum *depth* of the tree and n the maximum number of *simulations*. That means that whichever method is used should not be too computationally intensive, or the AI agent will take too long to make a move.

6.2.1 Hidden information

It is important to mention the *Information Problem* here briefly. When exploring a tree with hidden information, issues quickly arise. What happens for example when a piece attempts to capture a previously unseen enemy piece? The lack of information means we are unable to decide how the tree should continue.

There are two ways of dealing with this issue. The first option is to use *chance nodes*: every time we are unable to decide how the tree should continue, we instead add several chance nodes that represent every possible outcome. For example, if we attempt to capture a piece that has moved but not yet revealed itself with a sergeant, we add ten chance nodes, each revealing a different piece and thus having a different outcome. This immediately presents a problem however, as this widens the already very wide search tree considerably. This problem worsens when the opponent needs to make a move in the search tree, as it is not clear which moves could even be made.

To counteract the issue of chance nodes widening the tree, we can use the general approach of Mets [Met08]. This involves making an educated guess on all piece ranks of the opponent simultaneously at the start of the search, that is consistent with the known information on the opponent. The agent can then use this information to decide how the events with unknown information should play out. This has some obvious drawbacks, in that it makes the agent implicitly 100% sure that a certain event will go a certain way. For example, suppose we estimate that an enemy piece has the highest chance of being a lowly Sergeant. We may not be certain of this at all, perhaps we are only 30% certain that it is a Sergeant, and the other 70% is split over the other ranks. Now, despite the fact that we are unsure about the exact rank of the piece, the agent will decide that if we attack it with a Marshal we will surely capture it. The agent will likely see attacking this piece as very beneficial, disregarding the risks. In reality, that enemy piece may be a Bomb which would make the agent lose its Marshal, but that possibility is not considered at all. Particularly in cases like this where there is no outcome that is much more probable than other outcomes this can lead to inaccuracies in the full search, which impacts the final found values of the moves. If these errors are large and frequent enough, this will cause the AI agent to make bad decisions.

Despite these issues, the MCTS agents in this thesis will always use the second option, as the use of chance nodes would impact performance too heavily. Details on this problem and the exact methods to solve them can be found in chapter 7.

6.2.2 ε-greedy

One of the simplest selection methods is known as ε-greedy. In this method, we simply select a random child node with ε probability, and we select the child node with the highest value in $1 - \varepsilon$ cases. The value of ε thus determines the ratio of *exploitation*, e.g. how often we simply go down the most optimal move, versus *exploration*, e.g. how often we look at some random move to determine its potential.

There are numerous variations of the ε-greedy method, that typically change how ε behaves. In the most simple variant ε is simply constant throughout the full search, whereas certain variations try to modify ε depending on the number of simulations so far or the found values for each move. Typically those variations try to lower the value of ε in some way, so that early on the search favours exploration and later on the search favours exploitation instead.

For many applications, ε-greedy is not the most optimal selection method, even when all kinds of variations are applied. This commonly found lack of potential means I have only implemented the constant-ε version of the ε-greedy method. In this version it is important to select the right value for ε, so that there is a good balance between exploration and exploitation.

Experiment to determine ε

To find the right value, an experiment is done for all values of ε in the $[0.05, 0.95]$ range, with 0.05 intervals. In this experiment an ε-greedy-agent using the Database Estimator (see 7.2.4), NUC Evaluator with 10000 evaluations (see 8.1.1) and a Gravon setup (see 5.1.4) is pitched against a Peter N. Lewis-agent using the Peter N. Lewis setup in 100 games. All ε-greedy agents use the same starting seed, so they are playing identical matches against the Peter N. Lewis-agent. The results of this test can be found in figure 6.1.

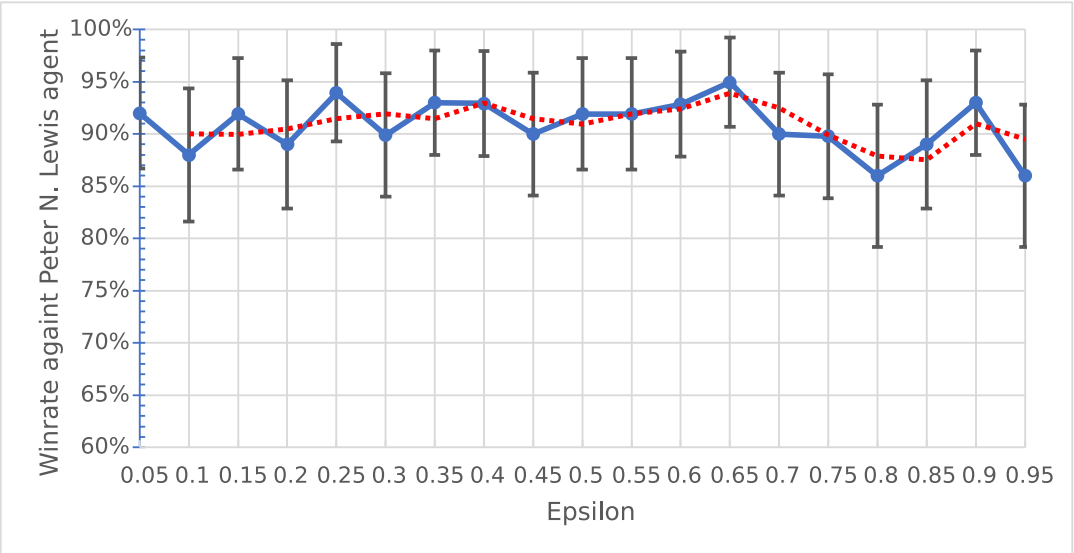

FIGURE 6.1: Graph depicting the experimental results of the ε-greedy-agent pitched against the Peter N. Lewis agent, with various values for ε. Error bars depict $p \leq 0.05$.

The blue line in the graph depicts the actual results, the red dotted line shows a moving average of the results. The moving average helps to filter some noise from the results, as the results are all roughly in the same range. From this graph we

can see that the performance of the ε-greedy-agent appears to improve until $\varepsilon = 0.65$ after which the agent performance slightly seems to drop, although the error margins are quite high. It thus seems that more exploration is beneficial, favouring a wider search of the tree, up to a certain point, though it is hard to conclude this for certain.

Exploration being beneficial could be in part explained by the nature of Stratego, where individual moves tend to have little impact, and the true impact of a series of moves only becomes apparent after some time. This means that the values assigned to the states for the different moves will not differ too much from each other, and thus the 'best' move so far according to the ε-greedyagent may not actually be much better than the other moves. In fact, if the agent explores this 'best' move too often, it is more likely to reach a state that is considerably different from the starting state. Because this information propagates backwards, it will become difficult for the other moves to essentially 'catch up' to it. By using a fairly large value for ε, this effect is somewhat mitigated. Based on this experiment, we select $\varepsilon = 0.65$ as the optimal choice for this agent.

When using the agent with this ε, we find that it can search to a depth of 10 moves when doing 10000 simulations. Increasing this to 100000 means it can achieve a search depth of 13. This is surprisingly deep, as various Minimax approaches did not manage to search this deep even with millions of nodes explored [Art10].

6.2.3 Upper-Confidence Bound

The major problem with the ε-greedy-agent is that it is fairly inflexible with how it decides to explore or exploit. The ε-value is always fixed and even in the variations where it does change it often does not do so in an optimal way. A generally better way to balance exploration and exploitation is known as the Upper-Confidence Bound method (UCB). The UCB method not only evaluates the value of the child nodes, but also the *confidence* it has in those values. The UCB method then tries to explore the child nodes of which the value has a low confidence and exploit the child nodes with a high value and confidence.

To select the next child move to explore, the UCB method uses the following formula: $f(t) = \arg\max_t \left[Q(t) + c \cdot \sqrt{\frac{log(n)}{n_t}} \right]$, where t denotes the move, $Q(t)$ the average return value, n the total amount of simulations ran, n_t the amount of simulations ran for move t and c a constant. This constant c determines the balance between the average return value and the confidence the agent has in that value. As c becomes larger, the confidence value becomes more pronounced, making the agent take exploration moves more often. If c decreases, the agent will choose exploitation moves more often.

Experiment to determine c

To determine the optimal value for c, the UCB-agent is pitched against the best agent thus far, the ε-greedy-agent Specifically, an experiment is done for all values of c in the $[0.05, 0.95]$ range, with 0.10 intervals. In this experiment a UCB-agent using the Naive RvH Estimator (see 7.2.5), NUC Evaluator with 10000 evaluations (see 8.1.1) and a Gravon setup (see 5.1.4) is pitched against a ε-greedy-agent ($\varepsilon = 0.65$) using the same estimator, evaluator and setup provider for 100 games. The agents are thus virtually identical, the only difference is the selection method. The results of this test can be found in figure 6.2.

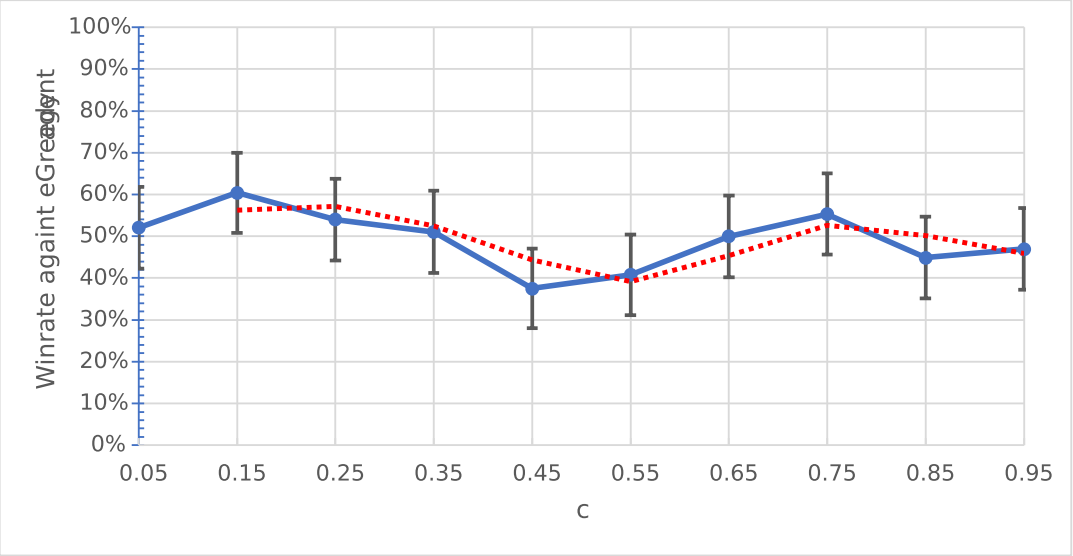

FIGURE 6.2: Graph depicting the experimental results of the UCB-agent pitched against the ε-greedy-agent, with various values for c. Error bars depict $p \leq 0.05$.

The blue line in the graph depicts the actual results, the red dotted line shows a moving average of the results. The moving average helps to filter some noise from the results. From this graph we can see that the performance of the UCB-agent appears to start strong with low values of c, after which performance appears to decline. Based on the general trend, it appears the values for $c = 0.45$ and $c = 0.75$ in particular may be outliers (unusually low and unusually high, respectively).

It is noteworthy to see that the winrate is far less 'convincing' than the winrate for the ε-greedy-agent against the Peter N. Lewis-agent. This is because both agents at their core use the same algorithm: Monte-Carlo Tree Search. Because the agents are mostly identical, large advantages are not expected if only one part of the agent is changed. It is possible that this effect is similar to the effect Schadd and Winands observed in their attempts to optimise an Expectimax-based agent [SW09].

From the graph it appears the performance peaks at roughly $c = 0.15$, and then declines somewhat. This is expected: because the simulation method used typically only provides a small range of different values, using values for c that are larger makes the confidence factor absolutely dominate the value factor, which means the agent explores all moves equally often. This would entirely nullify the advantage that UCB provides in how it balances exploration and exploitation.

With $c = 0.15$ and 10000 simulations, we find that the search manages to get a search depth of 5, considerably less deep than the ε-greedy-agent. Nonetheless, it does manage to get an advantage over ε-greedy, if only slight. It thus appears that the UCB-agent is capable of searching the lower depths more efficiently, whereas ε-greedy searches considerably deeper but may be missing other branches that should have been explored at lower depths. When we increase the number of simulations to 100000, it occasionally manages to search up to a depth of 8.

6.2.4 pUCT

Both ε-greedy and Upper-Confidence Bound attempt to steer the tree search in the right direction using data acquired during the search. This works fairly well, particularly if the search has been going on for a while and a lot of simulations have been

done. Nonetheless, these methods are not perfectly optimal. These two methods essentially slowly build up a sort of 'idea' or 'hunch' about which moves it should select. However, it can only do so by acquiring data from the search. It would perhaps be more efficient to start out with this 'hunch' instead.

Being able to generally steer the search before the search has even started requires some outside knowledge. In this case, we require some prediction function P that takes a state s and for all N moves that can be made from S outputs a probability p_{move}, for which $\sum_{i=0}^{i<N} p_i = 1$. We can then use such a function to help steer the search, by selecting moves with a higher probability more often.

This concept was put into practice by David Silver et al. in their efforts to master the game of Go [Sil+16]. They approached Go using a novel approach based fully on neural networks and Monte-Carlo Tree Search. The selection method they employed is called *pUCT*, or "Predictor + Upper-Confidence bound applied to Trees". The exact formula used is the following:

$$a^k = \arg\max_a \left[Q(s,a) + P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \left(c_1 + \log\left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right) \right] \tag{6.1}$$

In this formula a^k represents an action a taken from state s at timestep k, $P(s,a)$ a prediction function that outputs the probability of a being selected from s, $Q(s,a)$ the value for action a taken in state s found during the search, $N(s,a)$ the number of visits to a taken at state s. c_1 and c_2 are constants used to tune the formula so that a good balance is found between the found value, the predicted value and the confidence value.

Prediction function

The prediction function $P(s,a)$ has been created using publicly accessible data from the Gravon database [Jun15]. This database contains more than 50.000 games played online by humans. We can use these games to train a neural network that outputs a softmax probability that predicts which move is most likely to be made. It is important to note that ideally we want a function that predicts which move *should* be made next, e.g. the best possible move, rather than the move that humans are most likely to make next. Such a prediction function is possible, but would require extensive self-training and a very strong agent to use for data generation. Considering such an agent is not available to me for this purpose, we assume that whatever a human does is considered 'good enough' for now.

From the Gravon database I have extracted over 1.7 million states and associated moves. These have been converted to a one-hot encoded input and output representation. For the input, we use 3312 input nodes, which can be divided into three sets of 1104 nodes.

- The *first set* contains the true owned piece ranks on every square. For each of the 92 squares we have 12 possible ranks that can be placed there. As this is a one-hot encoding, each input corresponds to a single (rank, square) tuple that is set to 1 if that rank is actually on the square or 0 otherwise.
- The *second set* contains the owned potential piece ranks on every square. Similar to the first set, we not only mark the true ranks but also all ranks that that piece could potentially have according to the information revealed to the opponent. In case something has moved, the bits for the bomb and flag are set to

0. If we have revealed all our scouts, then none of the other pieces will have the *scout*-bit enabled, etc...

- The *third set* contains the enemy potential piece ranks on every square. It is thus identical to the second set, but for the opponent's pieces instead of the owned pieces.

This representation ensures that all information regarding piece positions and ranks is passed to the neural network. Note that this is not *all* information that is potentially available; moves made thus far and the current state of the Two-Squares rule are excluded. The first was excluded due to this not easily fitting in a standard input representation, and the second was excluded because it rarely has a large enough impact (as it only affects a single move, which the tree-search will detect anyway).

The output representation is again a one-hot encoded output vector with 1368 nodes, one for each possible tuple of two squares that could serve as an origin and destination for a move. Note that this isn't every pair of two squares; it is not possible to move across a lake or over diagonals for example. Only the pairs between which a piece can move are included (specifically a scout, as they can move considerably further than other pieces). Note that this output vector uses softmax, e.g. the output vector predicts the likelihood that a specific move is picked next.

The network has five hidden layers of 300 nodes each. Each of these layers is fully connected. This was largely done for simplicity and as a proof of concept, and also to compare a little bit with Smith [Smi15], who opted to use convolutional neural networks instead.

For training, a α of 0.001 was used. After approximately 11 hours of training, cross-entropy loss had decreased from the starting value of 5.56 to 2.91, and the accuracy had increased from a mere 0.022 to 0.246, or roughly $\sim 24\%$. This accuracy is surprisingly high, meaning that given a random state from the game the network can accurately predict the next move in more than one in five cases. Note that this not merely means that the network has figured out which moves are *valid* moves given a state, as on average a player can make just over 20 different moves per turn. The network has in fact learned to estimate which move will be made next.

There are some limitations however. Most notably there is some considerable data imbalance. In historical Stratego games, it appears that moves made in the opposing corners appear much less frequently than moves made in the center. This means that corner moves are considered "less likely" than other moves simply based on the location. Because the neural network will not recommend these moves very often, the agent may play considerably worse when the game has important pieces in corners.

Another important limitation is *performance*. The selection methods discussed so far have been relatively cheap to compute. However, using a neural network as a prediction function requires that once we try to look up the predicted value for a child node, we need to convert the parent node to the input representation, run it through the network, retrieve the output and apply softmax to it. This process would take far too long when done through regular Tensorflow. This is because Tensorflow is not optimised for single network calls, but instead prefers batched calls, e.g. applying a vector of thousands of inputs to the network simultaneously. To optimise for single calls, the network has been converted to the TF.Lite format, which is more optimised for this scenario. Nonetheless, compared to UCB the pUCT agent takes roughly 10 times longer to compute a move.

Tuning c_1 and c_2

There are two constants that are used to tune how important the confidence and predicted values are compared to the discovered value, c_1 and c_2. In the original paper, the authors find that values of 1.25 and 19652 respectively work well and produce good results. From initial testing, I found very similar results. Because test results from this kind of tuning can be quite noisy and radical changes in values for these two constants produced worse results, I opted to select $c_1 = 1.25$ and $c_2 = 19000$ as default values, as this seemed to produce good results from observational testing. A full test has not been performed, as the pUCT agent is a fair bit slower than the other agents tested so far. This means that a test which would usually take a couple hours would now take a full day or more to process, which is too long for me to execute.

Comparison to UCB

The pUCT agent behaves noticeably different from the UCB agent. Compared to the UCB agent, this agent appears to behave more aggressively. It is more willing to move multiple pieces forward at the same time, and move them deeper into enemy territory. The UCB agent in contrast acts a bit more defensively, preferring to intercept enemy pieces rather than go on the offensive. This is likely because of the added direction pUCT brings to the search. UCB initially has to explore all moves equally, both offensive and defensive ones, as well as 'nonsensical' moves (e.g. moves that make little sense to make regardless of the strategy employed). pUCT provides more direction towards the offensive moves and a bit towards the defensive moves, but is much better at hinting that the 'nonsensical' moves should not be searched too much.

The predicted problem with the corner moves is also visible. It appears that the prediction function is in general less likely to recommend moves that move a piece into the corners. It also appears that moves that move a piece onto the back row are a bit more rare. This may initially seem odd, as flags are usually placed in the back row. But typically they aren't captured from there; it is more likely that a bomb in front of it is discovered, hinting at the location of the flag, rather than discovering it from the side.

To determine its exact effectiveness, the pUCT agent was pitched directly against the UCB-agent ($c = 0.15$) in 100 games. Both used the Naive RvH estimator (see 7.2.5), the NUC evaluator (see 8.1.1) with 10000 evaluations and a Gravon setup (see 5.1.4). pUCT managed to achieve a winrate of 41.4%, which is not enough to be superior to UCB, though it is not too far off either.

In theory, pUCT should be stronger than UCB. The prediction function should, assuming it is correct enough, guide the UCB search so that it can more effectively search the game tree. As the test results show, pUCT has not beaten UCB yet, which means that the prediction function is not good enough.

There are multiple potential reasons for this.

- *Lack in data quality*: The data fed to the move prediction function is based entirely on human data. This involves players from all skill levels, including beginning players who may not be playing optimally. There is also the issue that nearly half of the players who played of course lost the game, meaning their moves made may not have been the best.
- *Lack in data spread*: The data is compiled from a lot of games played, but not all scenario's that could happen appear often enough in those games. This

is exemplified by the move prediction function rarely predicting moves that move a piece into the opposing player's corner or back row. There are only the situations that are apparent when observing games, though there may be other deficiencies in this regard.

- *Overfitting on parts of the data*: The lack in data spread makes it so that certain parts of the board are much more rarely visited. There are however also places on the board, particularly in the center, that may be overrepresented instead. This makes the move prediction function naturally biased towards these moves, meaning it overfits on these positions and moves and recommends them far too often.

It should be noted that pUCT also manages to do a little bit better than UCB (and ε-greedy) in certain aspects. Most notable is the search depth; with only 10000 evaluations it can reach a depth of 11, and with 100000 simulations it reaches a depth of 14. In some rarer occasions it can go even deeper, down to 16. This is deeper than both UCB and even ε-greedy, making it more likely that if the prediction function were to improve, pUCT could beat both UCB as well as ε-greedy.

In addition it appears that pUCT does aggression, particularly in the center of the board, particularly well. This makes sense in that this scenario is likely overrepresented in the Gravon database. It only starts failing when it reaches more unknown territory, at which point it misses some more obvious plays. This could in theory also be compensated a little bit by giving each move a base chance in the prediction function, so that no move is entirely 'filtered out' when it should not be.

6.3 The effect of information

From testing it appears that UCB is better able to search through a slightly wider scope of moves, through setting the c-constant at the right level. UCB can also be made more specific by setting c even lower, but from the testing that does not appear beneficial. Searching in more specific directions seems to be more suited to ε-greedy and pUCT. This suggests that *if* more information is known about the opponent's pieces, ε-greedy and pUCT could be more effective.

To test this hypothesis, we run a new experiment where we let the ε-greedy-agent and the pUCT-agent play against the UCB-agent, all with the optimal settings found before (e.g. $c = 0.15$ and $\varepsilon = 0.65$). However, this time we use the Database estimator (see 7.2.4) and a Gravon setup (see 5.1.4), which means the agents are much more capable at discovering the opponent's piece ranks. ε-greedy now achieves a winrate of 55.7% against UCB, and pUCT achieves a winrate of 46.4% against UCB. pUCT may only see a slight improvement, but the advantage that ε-greedy gets from having more information appears to be considerable compared to the performance when there is less accurate information available. However, that advantage does require the ε-greedy-agent to actually be able to obtain this information, which in this specific setting it may be able to but might remain out of reach for more realistic settings.

6.4 Discussion

It appears that Monte-Carlo Tree Search, regardless of selection strategy, poses a strong and viable alternative to Minimax. It manages to fairly efficiently search the game tree. Most mistakes that can be observed are not due to MCTS or the specific

selection method, but rather mistakes in the used simulation method or piece rank estimator.

The strongest selection method appears to be UCB. This is especially notable, as it is the method that searches least deep of the three. It is very well possible that the lack of depth works out as a strength due to the lack of information in the game. Piece ranks have to be guessed, and it makes sense that pieces that are further behind enemy lines have less information revealed about them, whereas the frontline pieces are likely already known or have some information known about them. This makes the outcome of any encounters that appear at lower depths in the tree search more likely to be accurate, whereas outcomes of encounters at greater depths are less certain. Any large changes in the simulated value may thus be misleading if it happens after many moves in the tree search.

ε-greedy also seems to be a surprisingly good method, despite its simplicity. Specific to this case, the implemented version is also the least complex one, with no tricks to reduce the value for ε if necessary or other ideas to make ε-greedy more effective. There may still be more to gain here, which could push it beyond what UCB achieves now.

The pUCT method still needs considerable work. It definitely shows good potential, but it is hampered by imbalanced training data. If data were to be generated using a strong agent in more kinds of scenarios, the move prediction function may improve enough to begin beating UCB consistently. The neural network for pUCT already seems to be fairly good at predicting human moves in many common scenarios, but there are still plenty more less common scenario's that it has not trained on sufficiently yet. Additionally, it would be good if a better measure of a 'good move' than 'human move' could be used.

6.5 Conclusion

The goal of this chapter is to explore different ways to solve the Dynamic Evaluation Problem. In particular, we have looked at three different selection methods for Monte-Carlo Tree Search.

From the experiments, it appears UCB is the best selection method overall. It is stronger in the early- and midgame, but it can lack the necessary depth for the later stages of the game. It can handle lack of information or poor piece rank estimations best out of the three methods tested. If more information is available or can be accurately guessed, then it appears UCB is less optimal and ε-greedy appears a bit stronger. This is likely due to its ability to search deeper, making it more viable for the later stages of the game than UCB.

The potential for a hybrid method definitely exists. In such a hybrid UCB would handle the earlier stages of the game and once the board is sparse enough, information is plenty enough or the confidence in the piece rank estimations is high enough it may be beneficial to have ε-greedy take over for the final stages of the game.

The third method, pUCT, also exhibits great potential. It can search to even greater depths than ε-greedy can, though the fact that the move prediction function is not good enough severely hampers its ability to direct its search more accurately. To make pUCT more viable, more work is definitely needed.

Chapter 7

Information Problem

The main property of Stratego that sets it apart from other board games is the fact that the ranks of the enemy's pieces are hidden from view. The only way to find out what rank a piece has is to either try to capture it or have it attempt to capture one of your pieces. Of course this has drawbacks; you may have to reveal one of your own unrevealed pieces and more importantly you may lose a piece if the enemy piece has a higher rank than yours.

Because of these drawbacks it would be much more preferable if it was somehow possible to guess the rank of an enemy piece *before* interacting with it. Unfortunately this is very difficult to do. The reality is that for many pieces there is very little information, and the information that is revealed can be intentionally deceiving. For example, an enemy piece that is rapidly advancing towards your own pieces *could* be a high ranking piece that you need to be careful of, requiring the attention of your own high ranking pieces. Alternatively, your opponent could be trying to deceive you and it could instead be a low ranking piece, with the goal of revealing your high ranking pieces.

Having correct information, particularly in a tree search-based AI agent is critical to the quality of the agent. The effect of making an incorrect guess of an enemy piece's rank has an obvious immediate effect. Making the wrong guess could easily see you lose a piece because you incorrectly believe an attempted capture would go your way. But for a tree search-based AI agent, there are additional negative effects. The first 'layer' of the search tree only shows the moves an agent can make at that point. But the next layer contains all the moves that the opponent might make. Depending on what the agent believes the opponent's pieces to be, the moves that the agent will explore on the opponent's behalf also change. This can have a big impact on the rest of the search tree and what values are ultimately assigned to each move, as we may be incorrectly guessing what the opponent would do.

7.1 Specifying the problem

There are a number of ways that an AI agent could potentially use to work with the hidden information in Stratego. It is therefore important that we first specify exactly *how* we intend to work with it in this thesis, as this has an impact on what methods we can use. We can do this by exploring when exactly an AI agent would encounter hidden information, and how we at those points would like to deal with it. In this thesis we focus primarily on Monte-Carlo Tree Search (see 6.2), so we will look at where an MCTS-agent encounters hidden information and how dealing with it in specific ways would impact the tree search.

Recall that the MCTS-agent repeats four steps an X number of times: *selection*, *expansion*, *simulation* and *backpropagation*. For convenience, we will assume that the

MCTS agent has already explored a little bit so that a part of the tree structure already exists.

1. *Selection - Start node*: The MCTS-agent starts its *selection* step in the starting node. At this point the agent only has to deal with the moves it can make by itself. The agent knows which of its pieces are where, so it knows what moves we can make. So far there is no problem, and the agent can select some move m to go down.

2. *Selection - Leaf node (opponent)*: Suppose m takes the agent to a node that it has not previously explored, e.g. a *leaf node*. Here the agent is already potentially faced with hidden information. If the selected move m moved a piece into an empty square or attempts to capture a previously seen enemy piece, we know exactly what the new state looks like and there is no issue. However, if m were to be an attempted capture of an unseen enemy piece, the agent faces hidden information. This is the first problem: **How do we resolve an attempted capture of an unknown enemy piece?**

3. *Expansion - Opponent's turn*: If the agent runs into a leaf node, it needs to be able to expand the node. This means the agent needs to know what state transitions can happen from the current state, e.g. what moves the opponent can make. Unfortunately this presents the agent with a mountain of hidden information, as it needs to know the ranks of all enemy pieces that could feasibly move. This results in the following problem: **How can we figure out what moves an opponent can make?**

4. *Selection - Opponent's turn*: Suppose m takes the agent to a node that it has previously explored. That means that we have already done an expansion here, and know what moves the opponent could make. This means that at this point, we do not face a new problem concerning hidden information. We select an opponent's move o to explore.

5. *Selection - Leaf node (agent)*: Similarly to what is described in 2, the agent needs to decide how the opponent would decide what the current state looks like, as the agent is potentially faced with information hidden not from itself but from the opponent. This gives us a new problem: **How do we resolve an attempted capture of a piece unknown to the opponent?** This may seem like it has an obvious answer as the agent knows what the truth is, but the opponent does not and this can alter its decision making (e.g. it may *believe* that its Spy is attacking a Marshal, which would be very beneficial, but it may in reality be attacking something else which would be detrimental instead).

6. *Expansion - Agent's turn*: At this point the agent needs to decide what the opponent believes to be its potential moves. However, this introduces a divergence: the opponent may believe it is capturing a piece in move o that leads to this state, but we may know in reality that this is not true and it loses its piece instead. This means that the agent not only has to find out what the opponent believes the agent can do, but also consider what moves it can actually make. This causes a split: there is a branch that the opponent considers correct or probable, and there is a branch that the agent considers correct. **How do we deal with the search tree splitting based on what either side believes to be correct?**

7. *Selection - Agent's turn*: The agent has to select which moves it wants to explore. It knows what moves it can make as this has been decided in the expansion step, so there is no issue here.

8. *Simulation*: Suppose we have found a state that we wish to evaluate. The outcome of the simulation step heavily depends on what we have decided that the current state looks like. Although a lot of hidden information may be considered here, what we believe the state looks like has already been partially decided in the preceding steps where a number of decisions have already been made. Nonetheless, not all hidden information may have necessarily been dealt with. **How do we deal with undecided hidden information in the simulation step?**

9. *Backpropagation*: Once a state has been simulated and been assigned a value, that value must be backpropagated back to the root or starting node. The agent does not encounter any new hidden information here, so this step goes without issues.

We are thus left with five important questions that need an answer to define what the MCTS tree looks like and how it is explored. They are:

1. How do we resolve an attempted capture of an unknown enemy piece?

2. How can we figure out what moves an opponent can make?

3. How do we resolve an attempted capture of a piece unknown to the opponent?

4. How do we deal with the search tree splitting based on what either side believes to be correct?

5. How do we deal with undecided hidden information in the simulation step?

Unfortunately there are few, if any options that could answer all these questions in a way that is truly accurate without also increasing the branching factor beyond a level that makes the tree so large that it becomes effectively unsearchable. We thus have to look at ways that we can still be *somewhat* accurate, but without also increasing the branching factor too much (which in Stratego is unfortunately already quite large). This is the primary goal that we try to answer in this chapter.

We leave question 5 to be answered by the *simulation method*, as it does not impact our ability to search the tree. It only impacts the eventual value assigned to states and it may alter how we direct our search, but the structure of the search tree is the same regardless of our answer to this question. The agent would provide the simulation method with the information that it knows and what hidden information it has already decided on, and the simulation method has to decide what to do with that information. It is thus out of scope for this chapter.

There are two potential methods of dealing with with the remaining four questions: introducing chance nodes to the tree, or making a single estimate of the truth.

7.1.1 Chance nodes

Chance nodes are the traditional answer to chance-based events in tree search problems. They effectively act as a new intermediate layer between two states that are separated by some stochastic event. Each chance node in this layer represents an outcome of the stochastic event and has the probability of that outcome associated

with it. In the selection and backpropagation steps, this probability would be used to determine which branch to select and how to weigh the values backpropagated from the branches, providing little weight to very rare outcomes and much weight to very probable outcomes.

We will look at how chance nodes could be used to answer each of the four questions.

1. How do we resolve an attempted capture of an unknown enemy piece?

When attempting to capture an enemy piece, we need to decide which outcomes are possible. One might initially guess that there are three different outcomes: a successful capture, a draw or a failed capture. Unfortunately that is not enough; depending on *which* piece is captured or successfully evades capture, probabilities and hidden information for the rest of the board also changes.

Suppose a Marshal attempts to capture an enemy piece. The draw and failed capture outcomes are very well-defined here: a draw can only happen if it encounters another Marshal, and a failed capture can only happen if it encounters a Bomb. A successful capture however is not that simple. If the Marshal captured a General, of which the opponent has only one, then the hidden information changes for all other pieces as well, as they can no longer potentially be a General. Even if the Marshal were to capture a different piece of which the opponent still has multiple, the hidden information is affected by altering the probabilities that each piece has for each rank.

This means that a capture of an unknown piece necessarily introduces at most 12 chance nodes, one for each rank that is potentially captured. This creates a new layer with a branching factor of 12 every time there is a capture, which makes the search tree considerably larger.

2. How can we figure out what moves an opponent can make?

When working with chance nodes, dealing with figuring out what moves the opponent can make is difficult, though not impossible. The biggest issue is that it is not really possible to determine the *list* of moves the opponent could make, e.g. deciding on all moves that are possible at once. To do so, for each piece we would have to determine if it could be a Scout, a regular piece or an unmovable piece (Bomb and Flag). To find out what list of moves are possible, we would need to know which pieces have been assigned what kind of rank, as that determines their possible moves. Unfortunately, this would create far too many chance nodes, as for each combination of pieces we would have to create a new chance node and assign a probability. With hundreds of possible combinations, this would create far too many chance nodes to effectively deal with.

A better alternative would be to only create chance nodes based on the specific move that is selected instead of the entire list. A chance node would be created for every possible move, no matter what rank a piece might have. This creates a more manageable number of chance nodes, even though it would still be large due to the large number of potential Scout moves. The downside here is that the chance nodes need a probability assigned to it, which may be a bit difficult to calculate as it would have to take the likelihood of the move as well as the likelihood of the rank into account.

It is important to note that this extra layer of chance nodes would be larger when the tree is not very deep yet. In greater depths, more hidden information would have

already been decided on, which means that there are far less possible combinations to consider at this step.

3. How do we resolve an attempted capture of a piece unknown to the opponent?

When an opponent in the search tree attempts to capture a piece, the outcome is unknown to them. To be completely accurate, they would have to do the same thing that we do; estimate the probability that the captured piece has a specific rank and change the state accordingly. They would use the predicted outcome to decide what move they would make themselves, which is exactly what we are trying to find out. However, we want to decide what we would do based on what would actually happen, e.g. resolving the capture by simply using the rank that we know our piece has.

This would create a divergence, e.g. we would both have to explore a tree where the opponent uses chance nodes to determine their move, and a tree where we use the true rank to determine our next move. The alternative option is leaving out the tree that the opponent uses to determine their next move and only use the true rank to determine the outcome of enemy attempted captures. Of course, this impacts accuracy to some degree; we are suddenly far more pessimistic about the strength of our opponent. By assuming the opponent can accurately simulate the captures they try with absolute certainty, we likely vastly overestimate their capabilities.

This impacts our playstyle in a fairly large way. Most importantly, it is not really possible to bluff anymore, as we assume the opponent can see through any deception thrown at them. This seems like a large price to pay, but it comes with the very notable advantage that this approach would not require any additional chance nodes or layers, which simplifies the tree structure considerably.

4. How do we deal with the search tree splitting based on what either side believes to be correct?

As mentioned before, to be completely accurate we would need to occasionally split the tree based on what we know to be true and what we believe the opponent believes to be true. This would only need to happen when the opponent has just made a move, as for our own moves we simply do not have the absolute truth available and have to work with what we do know.

How these alternate branches are best dealt with is hard to determine. Assuming that we are most interested in what moves we are going to make and only want to know a rough estimate of the value the opponent would predict for its own moves, it may be possible to do a shallow tree search on the tree that the opponent uses for its own move selection, up to a limited depth. This approximation could then be enough to give a rough estimate of which moves the opponent would consider, and afterwards we could focus exclusively on the part of the tree that uses the true rank.

Conclusion

The use of chance nodes in the search tree for Stratego adds a great deal of complexity to the structure of the tree, making it require much more effort to traverse sufficiently. Additionally, there are important problems to consider, for example how to exactly deal with problem 2. This makes the use of chance nodes largely impractical, and it would probably be better to search for alternatives.

7.1.2 Making a single estimation

An alternative to using chance nodes was proposed by Mets [Met08]. He suggested a method he called "strooien" or "scattering" in English. This means that at various points all pieces that did not have their rank known would get a randomly assigned rank ('scattering' the possible ranks on the unknown pieces), after which every piece would now be 'known' to have a rank. This in effect turns Stratego in a perfect information game. All piece ranks are known to either player, even if for one player those ranks are not entirely correct. In this thesis we call the assignment of ranks to all unknown pieces an *estimation*.

If we pretend Stratego is a perfect information game, the problems with hidden information effectively disappear. The MCTS can simply be done without any issues or additional layers or nodes. This is a huge improvement to how efficiently the tree can be searched, although it comes at a significant cost to accuracy.

The cost to accuracy

Making an estimation of what the board looks like and subsequently treating Stratego as a perfect information game means that there is an implicit 'truth' in the search tree. For example, attempting to capture an unknown piece suddenly becomes capturing a known piece, which has a set outcome. Because of this, certain moves that may be very uncertain and have risky consequences are treated as completely certain, risk-free moves.

This can not only trick the AI agent into making very risky moves, which could cause significant blunders. It also has consequences for what we believe the opponent is likely to do. Because we assume that Stratego is suddenly a perfect information game, we also assume that our opponent can and will play optimally. This makes the AI agent a bit more defensive, as it assumes that the opponent is likely to make any capturing moves that will go their way.

Making a single estimation and strictly using it as the truth also completely eliminates the aspect of bluffing from the game, at least as far as the search tree is concerned. Because we no longer consider the possibility of being wrong, we also do not have to concern ourselves with attempts to deceive us. There is no probability of pieces being certain ranks to minimise or maximise, as we do not need probabilities in the tree anymore. The only place where they could be necessary is the simulation method or when we actually create the single estimation used in the search tree.

The impact of all these issues appears to hinge on a single factor; how good are the estimations that are made? If we can completely accurately guess what the enemy piece ranks are, or even just the ones that are on or close to the front line, then most capture moves will play out accurately in the tree search. At that point it may no longer matter much if there are a few mistakes in the estimation, or if we overestimate what the opponent knows. The tree search would be able to find a course of action that is perhaps a bit conservative but also effective.

Conclusion

Making a single estimation of enemy piece ranks heavily simplifies the tree structure, compared to the alternative of using chance nodes. This comes at a cost to accuracy, although if the estimation is good enough this may not be too impactful on the quality of the agent. Although Mets has used a similar method with some success [Met08], it seems there is a lot of unexplored potential to this method, which Mets also acknowledges.

In this thesis we will focus on this estimation strategy rather than chance nodes for dealing with hidden information, as it appears that there is enough to improve here. We will look at a few *estimation methods*, e.g. methods that provide an assignment of ranks to all enemy pieces, to see which method works well and if the estimation strategy has merit.

7.2 Estimation methods

When using the single estimation method to solve the problem of having hidden information complicating the structure of the search tree, the accuracy of the estimation becomes critically important to the quality of the AI agent. After all, if the estimation is good and pieces are assigned the ranks that they actually have (or perhaps a rank close to it), the tree search will become more accurate and the resulting move will work out better in the actual game.

Though various rank-guessing methods have been proposed in literature, it appears most of these are focused on predicting single pieces instead of predicting all pieces at once. Moreover, most methods are not very well described and lack a lot of implementation details. Additionally, it is very hard to determine the effectiveness of the proposed methods. There is no real standard, and if any figures are cited it is often unclear if those figures are consistent throughout the entire game or only applicable to certain phases of the game.

7.2.1 Setup Reconstruction

The general approach employed for all estimation methods (where applicable) is called *setup reconstruction*. This is a novel way of making piece rank estimations. Whereas previous attempts have largely focused on predicting single pieces at a time, we will focus on methods that attempt to reconstruct what the original setup looked like, and extrapolate the piece ranks from there. This is done by only taking into account what information has been revealed about each of the pieces so far. It does not use the current position of the pieces on the board but rather the position where the pieces originally started.

By focusing purely on reconstructing the original setup, this method is far less prone to fall to bluffing attempts by the opponent, e.g. by aggressively moving low-ranked pieces towards a high-ranked piece to 'threaten' it. This can be very advantageous to the AI agent, as the tree search itself is unable to deal with these kinds of bluffs.

Another advantage is that when the setup was created, no information about how the game would progress is available yet. The creation of the setup was thus entirely isolated from the rest of the game. This means that information from during the game, such as the position of pieces or moves made may be deceiving rather than helpful. Of course information about the piece ranks is useful, as that helps filling in the blanks in the setup. This also includes whether or not pieces have moved at all or if they have moved multiple squares in one move, as this can reveal whether or not they are movable (e.g. not a Bomb or Flag) or a Scout.

In this chapter we will focus on five *estimation methods* or *estimators*. Three of these explicitly use the setup reconstruction strategy, whereas the other two are mostly used as a comparison method.

7.2.2 Omniscient Estimator

The first estimation method is not meant for a serious agent. The Omniscient Estimator simply cheats and provides a 100% accurate estimation, where all piece ranks are correctly 'guessed'. It is however possible to use this estimator as a kind of benchmark, or to see how well an agent would do given perfect information. Typically, we expect an agent that does well with perfect information to also do well with a reasonably accurate estimator.

7.2.3 Random Estimator

The simplest estimation method would be one that simply makes all decisions at random. However, in Stratego this is not entirely trivial. Suppose there are four pieces, of which two have moved. We know these to be a Marshal, a General, a Bomb and the Flag, but we do not know exactly which is which. If we were to fully randomly assign the ranks, we could assign the General or Marshal to one of the unmoved pieces, which would be a correct placement for that single piece, but is not valid if we then try to assign the Bomb and the Flag somewhere.

To avoid this, we use an algorithm that randomly assigns the ranks in a way that always produces an estimation that is valid. Realise that there are only three types of pieces on the board; unmoved pieces, moved pieces and known pieces. A piece can only move from the unmoved group to the moved or known group, but never in the other direction. For example, a Scout starts as an unmoved piece, can become a moved piece by moving once or can become a known piece by moving more than one square in a single turn. Because this can only go in one direction, we can order the assignments such that the end result is always valid.

1. Place all the pieces in a list and randomly order the list.
2. Loop over the list once to assign ranks to all *known* pieces keeping track of which ranks still need to be assigned somewhere.
3. Loop over the list a second time, now assigning the *Flag* and the *Bombs* to the first valid piece found in the list.
4. Loop over the list a third and final time, assigning all other ranks to the remaining pieces.

After this you end up with a random assignment of the piece ranks that must be valid given the current information. However, a random assignment is not a very good assignment. To illustrate this, an experiment is done where two ε-greedy-agents ($\varepsilon = 0.65$), a Gravon setup (see 5.1.4) and a NUC evaluator (see 8.1.1) with 10000 evaluations play against each other, one using the *Random Estimator* and the other using the *Omniscient Estimator* (see 7.2.2). After 100 games, the agent using the Random Estimator only manages a measly winrate of 5%, which is incredibly low. It would thus be better to use an estimator that is more accurate.

7.2.4 Database Estimator

Making better estimations requires some knowledge about what typical setups look like. The simplest way to use that knowledge is by putting it in a database and searching it. This is what the *Database Estimator* is at its core. It has all 100000+ setups from the Gravon database [Jun15] sorted by frequency (highest frequency

first) in a long array of setups. It finds a setup by simply starting at the beginning of the array and iterating over it, taking the first setup in the array that matches the given information (which is by extension also the most frequently used setup that matches the given information).

If an opponent uses a setup from the Gravon database, this method works flawlessly and will eventually find the exact setup that the opponent is using. It finds the setup incredibly quickly as well; tests show that typically it only requires 5-9 pieces to be fully revealed for the estimator to pinpoint the exact setup, which in several test games happens after a mere 15-30 moves (depending on the aggression of the players). As Stratego takes on average 381 moves to finish, this means that this estimator can reveal the entire enemy board after less than 10% of the game has passed. Even before this point is reached the estimator can be fairly accurate sometimes, as plenty of setups share similarities with each other.

Fallback method

If an opponent does *not* use a setup from the Gravon database, this method would eventually run out of setups to try and thus fail to produce an estimation. There needs to be some fallback method that the estimator can use so that it can provide a valid estimation even if the used setup is not exactly in its database. That way the method can still provide an estimation that is hopefully somewhat close.

The fallback method works as follows: during the iteration over the database, we keep track of the setup that matches *most* so far. Specifically, the setup that has most individual pieces in common, consistent with the currently known information. Every 'violation' of the given information, e.g. a piece that in a database setup can not move but on the board *has* moved, or a piece that in a database setup is a Scout which on the board is a Miner all count as a piece *not* in common or consistent with the currently known information.

After iterating over the database where no perfect match was found, we instead find a match that is not perfect but as close as there is in the database. The next step is to change that match into a valid estimation. This is not entirely trivial, as we need to assign the ranks such that it is completely consistent with the known information while also being as close as possible to the closest match.

We make the assignment by stating that the estimation that we need is effectively a *minimum-cost perfect bipartite matching*, with the 40 ranks on one side and the 40 pieces on the other. The sides are fully connected, e.g. every rank has an edge to every piece. The costs for the edges between a rank r and a piece p are as follows:

- *Cost 0*: Edges between r and p get cost 0 if p has r assigned in the closest match and is consistent with the known information.
- *Cost 100*: Edges between r and p get cost 100 if p does not have r assigned in the closest match, but is consistent with the known information.
- *Cost 10000000*: Edges between r and p get cost 10000000 if such an assignment is inconsistent with the known information (e.g. assignment would produce an invalid estimation).

By setting the cost for an invalid assignment extremely high, we prevent the minimum-cost matching from ever being inconsistent with the known information. Additionally, by setting the cost for an assignment consistent with a match at 0, we maximise the number of assignments that are also found in the closest match. To find the actual matching, we use the Hungarian Algorithm [RVD20].

The final estimation after using the fallback method is not always a good estimation, as the fallback method does not really care which pieces have their ranks swapped from the closest found setup. An initial attempt was made to favour keeping certain ranks in the same places, but these did not seem to improve the estimations. Because the estimations are not great if the opponent is using a setup not seen before, the quality of an agent using is expected to drop.

To verify this, an experiment was done where two UCB-agents ($c = 0.15$) (see 6.2.3) using the Database Estimator and the NUC Evaluator (see 8.1.1) with 10000 evaluations played 100 games against each other. One agent uses the Gravon Setup Provider (see 5.1.4, the other uses the Naive RvH Setup Provider (see 5.2.2). Typically, we would expect the agent using the Gravon Setup to do slightly better as the setups tend to be of a higher quality, but in this case the agent using the Naive RvH Setup Provider wins instead, with a winrate of 61.4%. This is fairly high considering the only difference between the agents is the setup.

To see how well this method does in general, an experiment was done to compare it to the *Random Estimator* and the *Omniscient Estimator*. Once again each agent is a UCB-agent with $c = 0.15$ using the NUC Evaluator with 10000 evaluations and the Gravon Setup Provider. The agent using the Database Estimator plays against an agent using the Random Estimator and an agent using the Omniscient estimator for 100 games each. Against the Random Estimator, a winrate of 86.5% was reached, whereas against the Omniscient Estimator a winrate of 38.6% was reached. These are fairly good results, as this method seems to convincingly beat the Random Estimator and has fairly decent results against the agent that is cheating to know all ranks right from the start.

7.2.5 Naive RvH Neural Network Estimator

Recall that with the setup reconstruction strategy, we try to estimate the setup originally used by the opponent as accurately as possible. Unfortunately, it is often the case that little information is available, thus it becomes difficult to determine the setup accurately, particularly if we assume that a database approach will not work (for example because the opponent likes to create new setups). In theory, we could assume that the opponent knows what they are doing and has made a good setup, and instead try to make a good setup that matches the known information.

Unfortunately, what determines if a setup is 'good' is not entirely clear. We can assume however that human players tend to make better setups, and that random setups in general are worse. If we can then somehow make a setup that matches the information given *and* looks human enough, we could use that setup as our estimation.

Thankfully, a method to distinguish a human setup from a random one was already developed as part of the *Naive RvH Setup Provider* (see 5.2.2). This neural network is capable of estimating how human a certain setup looks. We can also use this network to help make a good estimation.

The neural network is not capable of directly producing estimations, but it can rate a setup. To produce an estimation using the neural network, we first need some setups to be rated by it, and then we can pick the highest-rated one. We thus use a different estimator, the *Random Estimator* (see 7.2.3), to generate estimations that are consistent with the currently known information. This is a somewhat naive approach, but it should allow us to use the neural network to make estimations.

The estimation thus works as follows: the Random Estimator generates 1000 estimations, all valid considering the currently known information. The neural network

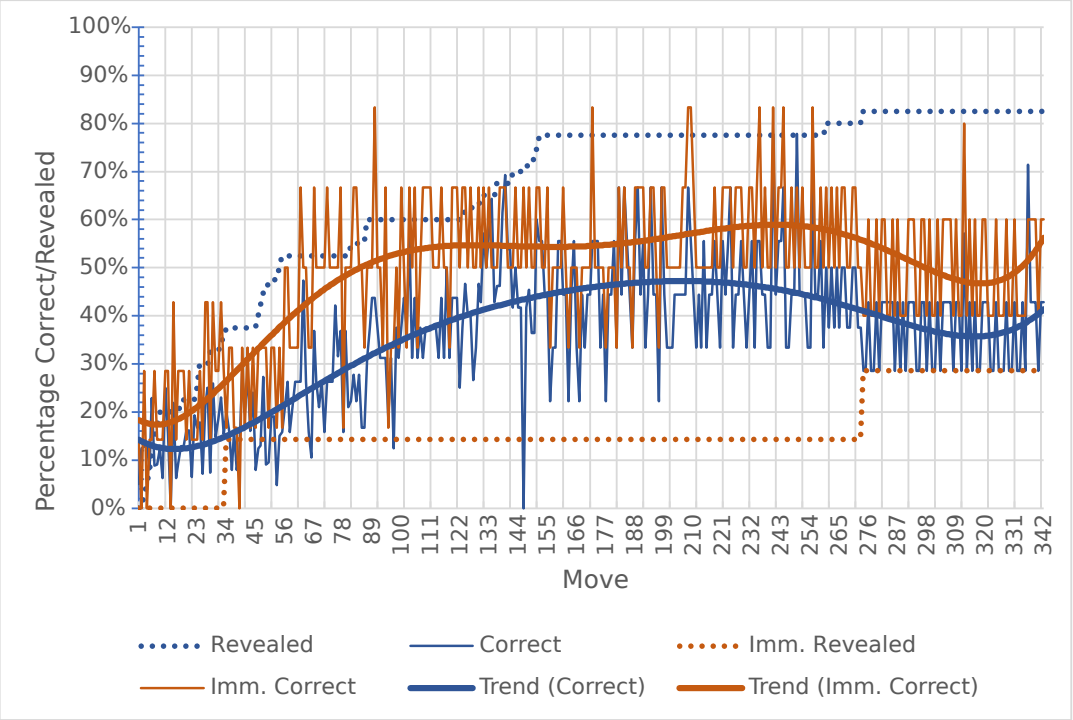

FIGURE 7.1: An example game measuring the performance of the Naive RvH Neural Network Estimator. This graph shows the perspective of the Red player. The trendlines are 6th order polynomials.

then evaluates them one by one, estimating how 'human' they look (a confidence value between 0 and 1), remembering the highest-rated setup. After all 1000 have been evaluated, *or* if a setup is rated as 0.99 human, the highest-rated setup is returned.

To illustrate how effective this method is, an experiment was done whereby two identical agents, both an ε-greedy-agent ($\varepsilon = 0.65$) with the Naive RvH Neural Network Estimator, NUC Evaluator (see 8.1.1) with 10000 evaluations and a Gravon Setup Provider (see 5.1.4), play a game against one another. During this game we measure the following:

- *Revealed*: The percentage of enemy pieces that had their rank fully revealed.
- *Correct*: The percentage of enemy pieces that had not yet had their rank revealed, but which the estimator guessed correctly.
- *Immobile Revealed*: Same as Revealed, but only for the immobile pieces (e.g. Flag and Bomb).
- *Immobile Correct*: Same as Correct, but only for the immobile pieces (e.g. Flag and Bomb).

The game was ultimately won by Blue and lasted 684 turns. By plotting the measurements we get a rough idea of the effectiveness of the method. These results can be found in figures 7.1 and 7.2.

The graphs both show that the performance of the estimator can be very noisy. There are estimations where it does not get a single rank correct, yet also estimations where it suddenly gets over half of the pieces correct. This is because the estimator generates completely new setups every turn, rather than keep using its prediction

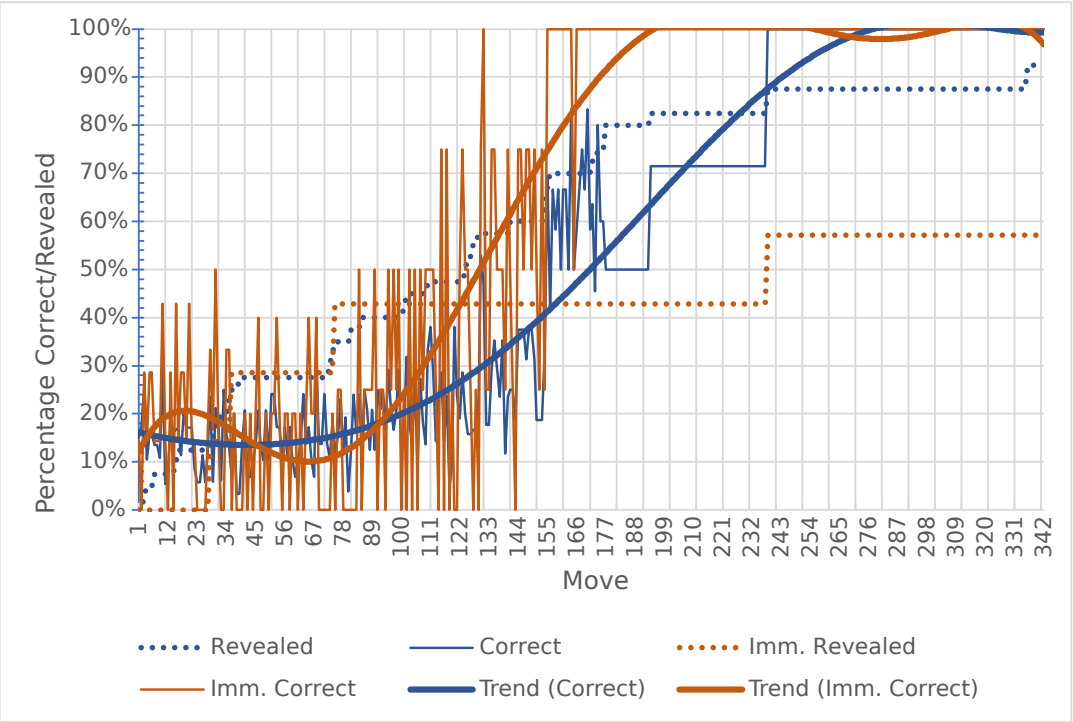

FIGURE 7.2: An example game measuring the performance of the Naive RvH Neural Network Estimator. This graph shows the perspective of the Blue player. The trendlines are 6th order polynomials.

until it is invalidated (or potentially improved upon). It may be interesting to see if keeping the estimations more consistent is beneficial because the agent tends to be able to stick to its plan better, or if this causes the agent to stick to a bad guess and lose important pieces because of it.

It seems that Blue had an easier time predicting the setup of Red, particularly the positions of the immobile pieces which were consistently guessed correctly in the second half of the game. Red has a harder time, getting stuck at guessing roughly half the remaining pieces correctly once more than 75% of the pieces have been revealed. This may be because Blue discovered the opponents immobile pieces much sooner, whereas Red for most of the game only saw 1 immobile Blue piece, only discovering a second after making its 274th move. It is possible that the positions of the immobile pieces is important for estimating where the other pieces could be located.

To see how well the method does overall, an experiment was done to compare it to the *Random Estimator* and the *Omniscient Estimator*. Once again each agent is a UCB-agent ($c = 0.15$) using the NUC Evaluator with 10000 evaluations and the Gravon Setup Provider. The agent using the Naive RvH Neural Network Estimator played 100 games against each opponent. It managed to obtain a winrate of 68.5% against the agent using the Random Estimator and only got a winrate of 6.4% against the agent using the Omniscient Estimator. The winrate against the Random Estimator is fairly good, at least convincingly enough to conclude this method is much better than random. However, the winrate against the Omniscient Estimator leaves much to be desired. From observing several games, the reason appears to be that the agent using the Omniscient Estimator plays much more aggressively, whereas the other is much more defensive or even hesitant at times. This is likely because the Omniscient agent can see all the risks and use some high-ranking pieces to capture large amounts of pieces. This puts the agent using the Naive RvH Neural Network

Estimator at a considerable disadvantage, particularly because the opponent reveals very few pieces by using this tactic. It may therefore not have enough clues to go on to accurately estimate the rest of the pieces on the board, leading to the disappointing winrate.

7.2.6 Direct Rank Estimator

One downside of the *Naive RvH Neural Network Estimator* (see 7.2.5) is that the neural network used is not very well suited to making estimations. It essentially requires us to generate a lot of random estimations and then helps pick the best one. However, these estimations are still randomly generated; there is no real thought that goes into generating them and as such certain structures appearing happens entirely by coincidence.

A different approach would be to have the neural network actually try to help generate estimations, rather than merely evaluate them. The *Direct Rank Estimator* uses a neural network that outputs probabilities for each tuple of piece and rank, and then converts those probabilities to an estimation that should be the 'most probable' estimation.

The neural network used is slightly special, in that it has multiple output layers. There are 40 output layers, one for each piece, with 12 different neurons each, one for every rank. These output layers are activated with Softmax, e.g. each neuron represents a probability, and all probabilities together sum to 1. The network has 480 input nodes for a one-hot encoding of the known information; each piece has 12 input nodes that are active if that piece could potentially have one of the 12 different associated ranks.

After the input layer, the network immediately splits into 40 different branches. Each branch contains 4 hidden layers containing 240 neurons each using a ReLu activation function, after which an output layer of 12 nodes follows. Each piece thus has its own branch in the network.

The dataset for this network is extracted from the Gravon database [Jun15]. From each game, we extract every board state. From these board states, we then extract the set of known information about the enemy pieces to function as the input data, along with the correct ranks to function as labels. These sets of information are then aggregated into a single dataset of more than 1.7 million information sets and associated correct ranks.

The training results for this network were not great, despite having tried many variations of this network, either with differently sized layers or a shared core between the branches and the input layer. The final combined cross-entropy loss appears to hover between 32 and 40, or 0.8-1 per branch. This is not amazingly low, but not horribly high either. The achieved accuracy however leaves something to be desired; while the accuracy is initially low, it quickly rises to about 25%. After this the accuracy starts dropping again as the loss decreases further, eventually settling on about 18%.

Although the training results are not very promising, the development on this estimator was continued. The accuracy in particular was worrying, but the loss could be considered somewhat low which held some promise.

To convert the produced probabilities by the neural network into an actual estimation, we use the Hungarian Algorithm [RVD20]. This algorithm finds a *minimum-cost perfect bipartite matching*. We make the assignment by stating that the estimation that we need is effectively such a matching, with the 40 ranks on one side and the 40 pieces on the other. The sides are fully connected, e.g. every rank has an edge to

every piece. The costs for the edges are the probabilities as given by the neural network, but inverted so that a high probability of a piece being a certain rank leads to an edge between that piece and a rank with a low cost. After running the Hungarian Algorithm we obtain a matching, which is then returned as the estimation.

To see how well the method does, despite its poor training results, an experiment was done to compare it to the *Random Estimator* and the *Omniscient Estimator*. Each agent is a UCB-agent ($c = 0.15$) (see 6.2.3) using a NUC Evaluator (see 8.1.1) with 10000 evaluations and the Gravon Setup Provider (see 5.1.4). The agent with the Direct Rank Estimator played 100 games against each opponent. The results are fairly disappointing; it achieved a winrate of only 34.2% against the agent using random estimations, and a winrate of 9.6% against the omniscient agent.

These results are, to be blunt, quite poor. Investigating the debug logs appears to reveal why: the estimations are typically of a low quality, only getting between 2 and 5 pieces correct, less than half of what the Naive RvH Neural Network Estimator typically does. Because this estimator always produces the same estimation given the same information, these poor estimations are kept over multiple turns. However, there is the occasional 'stroke of genius', where the estimator suddenly makes an estimation that gets up to 60% of the remaining pieces correct, even when there are over 20 pieces still completely unrevealed. As long as that estimation is kept, agent quality noticeably improves. Unfortunately, it seems that these sudden high-quality estimations are too rare for it to impact the overall agent quality much, causing the agent to lose frequently due to making too many mistakes.

It seems that this approach has some merit, but needs considerable refinement in order to truly pose a threat to a human player (or even other AI agents). There is an advantage to be gained from providing consistent estimations given the same information, but this can only be utilised effectively if the estimations are also consistently good. That is not yet the case for this estimator unfortunately, so estimation quality is lacking.

7.3 Estimators' "Home advantage"

It is fairly obvious why the Database Estimator has an advantage when the opponent is using the *Gravon Setup Provider*. However, the *Naive RvH Neural Network Estimator* is effectively based on the same neural network that is used for the *Naive RvH Setup Provider*. An interesting question here is if the estimator then also has an advantage when playing against an opponent that uses this setup provider. To test this, an experiment was done where two UCB-agents using a NUC Evaluator with 10000 evaluations play against each other. One agent uses the Database Estimator, the other uses the Naive RvH Neural Network Estimator. They played 300 games against each other, 100 where both agents use the Gravon Setup Provider, 100 where both agents use the Naive RvH Setup Provider and 100 where both agents use the setup provider that the other agent has an advantage against, e.g. the Database Estimator is combined with the Naive RvH Setup Provider and the Naive RvH Neural Network Estimator is paired with the Gravon Setup Provider.

In the games where both agents use a Gravon Setup Provider, we find that the agent using the Database Estimator has a winrate of 91.8%, meaning it can consistently and convincingly win against its opponent. However, in the games where the Naive RvH Setup Provider is used the agent using the Naive RvH Neural Network Estimator has an edge, getting a winrate of 64.8%. This is not as convincing, but still a strong result. In the games where both agents use a setup that is weak against

the opponents estimator, the agent with the Database Estimator has a slight edge, getting a winrate of 58.9%.

This experiment shows that there is a home advantage for certain estimators. However, it is hard to exactly quantify this advantage, as we only have a single setup provider that creates completely original setups available.

7.4 Discussion

Solving the *Information Problem* efficiently and accurately is a major challenge and is key to creating an AI agent that can perform well, both against other agents but also against human players. Even a poorer agent sees remarkable improvements in gameplay when given better estimations, and a strong agent does very poorly if it only has bad information to go on.

The strategy of *Setup Reconstruction* appears to work quite well, especially given the limited amount of information that is required for it to work. Considering the fact that the positions of the pieces on the board are completely disregarded and only their positions in the original setup are used, estimation methods using this strategy can get surprisingly good results.

The *Database Estimator* in particular is extremely strong, *provided* that the opponent is using a setup it has seen before. When this is not the case performance does decrease, but not to the point where the entire method becomes unusable. It is by far the best option to use against a player who reuses their setups, as this method can achieve effective omniscience in the early game already. It would be a particularly interesting method to use in an online setting that would collect the setups that players try to use against it, so that it 'learns' from the opponents. The estimator could however do with a better fallback method, so that if a setup is not in its database it has a better way of dealing with that issue.

The experiment in section 7.3 shows that from the estimation methods described, there is no single 'best one'. The effectiveness of any estimation method is highly dependent on the setup provider that the opponent uses. If this is known beforehand, we could make an educated guess as to which estimation method should be used, though this is not always clear.

7.5 Conclusion

In this chapter we have considered multiple ways to solve the Information Problem. In particular, we have looked at the setup reconstruction method, which tries to rebuild the original setup that the opponent uses and mostly focuses on that in favour of considering the current positions of the pieces. By doing so we have developed three new estimations methods.

However, there is still a long way to go when it comes to estimation methods. The best methods described in this thesis are the *Database Estimator* and the *Naive RvH Neural Network Estimator*. Both of these have different use cases, as well as agents that they do particularly well against. There are still improvements to be made to these estimation methods, for example a better fallback method for the Database Estimator and a better setup generator for the Naive RvH Neural Network Estimator.

The *Direct Rank Estimator* still needs a fair amount of work before it becomes truly viable. It seems the general approach has some merit, but either the implementation

or the training data is flawed in some way. In general it is probably better to use one of the other two methods.

It is however clearly evident that the Information Problem is still mostly an open problem, with a lot of potential improvements to be made that can significantly improve the performance of AI agents playing Stratego.

Chapter 8

Static Evaluation Problem

Any AI agent playing any game needs to have a general sense of what to do, otherwise playing against it would not be much of a challenge. That 'general sense' typically comes in the form of some *evaluation function*, that takes the current state of the game and assigns it some value. That value represents the desirability of the state, and gives the agent an idea of what kind of states it should try to reach (or avoid, for that matter).

An AI agent could simply look at the potential moves it has and evaluate the states that follow them, and pick the move that leads to the highest-valued state. Provided that the evaluation function is *perfect*, e.g. a function that can perfectly order all possible states based on likelihood of victory, such an agent would be completely unbeatable. After all, the agent would always take the move that improves its situation most, until it reaches a state of victory.

For small and simple games, such as Tic-Tac-Toe, these functions may exist and be feasible to use. However, in larger and more complex games these functions are often far from perfect. For games like Chess, Go and Stratego, if such a perfect function existed it would be far too complex to design or learn via machine learning algorithms. What we do instead for these games is create evaluation functions that *approximate* how desirable a state is. These approximations may give a general idea on the desirability of a state, but can often be quite imprecise. They are better suited for comparing states that are further away from each other, rather than states that only have a single state transition between them.

To make using these approximate evaluation functions practical, we employ *Dynamic Evaluation*, e.g. we use some kind of algorithm to search a large number of states, and rely on the evaluation function being broadly accurate. Examples include Minimax and Monte-Carlo Tree Search, although it should be noted that in the context of MCTS we often speak of a *simulation method*. Because these algorithms can search deeper than just a single state transition, we can reach states that are more distanced from the current state. While the evaluation function may not perfect, it can approximate the truth closely enough to see if these farther states are desirable to move towards. The algorithm then works out which state transitions need to be followed to reach it.

Clearly, the choice of search algorithm has a large impact on agent quality. However, the evaluation function itself is also incredibly important; a good function can guide the AI agent towards victory, but a bad function could deceive it into moving towards less than desirable states and thus a defeat. For Chess and Go, a lot of research has been put into finding good evaluation functions and using these as optimally as possible. For Stratego, several attempts have been made but so far it seems that none have truly found a strong evaluation function.

Having a good evaluation function is vital to the AI agent quality. However, developing such a function is not trivial. For Stratego, many elements factor into

how desirable a certain state is, such as the pieces on the board, their specific ranks, their exact locations, what pieces have been revealed, etc... The importance of these also changes over time, e.g. the Spy is important as long as the opponent still has their Marshal. If they lose their Marshal, the Spy becomes almost useless.

The evaluation function does not have to be perfect, but it does need to be a good enough approximation so that the agent can play the game properly. The evaluation function is one of the factors that also affects the playstyle of the agent most. Creating a good evaluation function is the core of the *Static Evaluation Problem*.

8.1 Evaluation Functions

Although several people in literature have attempted to make evaluation functions, results so far have been limited, and some methods have not been described in detail. We thus do not focus on these methods much and focus more on newly developed methods. Nonetheless, we include two evaluation functions that also feature in the work by Mets [Met08]. These are the *Random Rollouts* method and what I have named the *Jeroen Mets Evaluator*, which features points values for discovering and capturing certain pieces. The other evaluation functions are new.

To standardise the usage of these evaluation functions, they must all abide by some rules. Most notably, the values output by the *Evaluator* must be a floating point value between 0 and 1. A state that is a defeat must return a 0, and a victorious state must return a 1.

The Evaluator receives a board state, from which it can read any and all information, including hidden information. The reason for this is that the given board state is not an *actual* board state, but a state where the hidden information has been *estimated*. Because the hidden information in the given state is fabricated and not based on the actual hidden information, using it is not considered cheating.

8.1.1 Naive Unit Count (NUC) Evaluator

As de Boer stated [Boe07], there are a number of factors important to the desirability of a certain board state. One of the most important factors is the pieces that are still on the board. It is hard to quantify exactly how much each individual piece contributes, but perhaps it is possible to make a naive approximation by simply stating that every piece is equally important. This is the core idea behind the first new evaluation function introduced in this thesis.

The *Naive Unit Count* or NUC Evaluator is very simple. It counts the number of friendly pieces, or *friendlies*, and the number of enemy pieces, or *enemies*. Every piece is counted equally, so a Marshal is counted equal to a Sergeant. The number of friendly and enemy pieces is then converted to a score value by using the following formula:

$$score = 1 - \left(1 - \frac{friendlies/enemies}{40}\right)^8 \tag{8.1}$$

The division by 40 is to make sure the value of the formula is always between 0 and 1. Raising the value to a power of 8 is done to make the values where the agent is ahead of its opponent more spaced out, meaning it should prefer capturing enemy pieces over protecting friendly pieces. This makes an agent using this evaluator a bit more aggressive.

Despite the apparent simplicity of this method, the effectiveness is surprisingly high. Even though the consideration of many factors are left entirely to the tree search, such as the exact positions of the pieces, agents using the NUC Evaluator achieve remarkably good results. For example, when a UCB-agent (see 6.2.3) with the Database Estimator (see 7.2.4) and the NUC Evaluator with 10000 evaluations played 100 games against a Random Agent, both using the Gravon Setup Provider (see 5.1.4), the achieved winrate was 100%. When playing against a Peter N. Lewis-agent using a Gravon Setup for 100 games, the winrate was 87.8%, which confirms the strength of the evaluator.

The NUC Evaluator was designed at the start of the thesis as a function that is 'good enough' to use while testing solutions to the Dynamic Evaluation Problem and the Information Problem. However, each time the subject of evaluation functions was revisited to try out new things, the new ideas were surprisingly never quite able to consistently beat the NUC Evaluator. It has therefore stayed as the benchmark to test other evaluation functions against. The evaluator is also pleasantly computationally inexpensive, meaning it is possible to for example scale up to 500000 evaluations per turn without the agent becoming so slow that it takes too long before a move is made. An ε-greedy-agent with that many evaluations takes between 3.5 and 4 seconds to make a move on an i7-4790k, which is not bad especially considering that the agent is single-threaded.

8.1.2 Random Rollouts Evaluator

A common way of simulating how good a certain state is in any game, is by randomly executing state transitions until a terminal state is reached. The type of terminal state then determines the value for the initial state. The process of executing random state transitions is called a *Random Rollout*. The idea is that if we do many random rollouts, the average value will converge to the true value. If the random rollout ends in a victory, a value of 1 is returned, and if a rollout ends in defeat a value of 0 is returned. By averaging a lot of these values over the course of the entire tree search, we can roughly estimate the desirability of a state.

For Stratego however, this method is incredibly inefficient. The first issue is that Stratego games are quite long; games can take hundreds or even more than a thousand moves before reaching a conclusion, not to mention that with randomly executed moves games may take even longer. Doing a random rollout thus takes a lot of time.

The second issue is that in general there are a lot of moves possible each turn, more than 21 on average [Art10]. Yet the moves considered by Stratego players is usually much lower, as many moves are simply not promising enough to even consider. A random rollout will still execute those moves equally often. As an example, take a situation where a player has a piece right next to the enemy flag. In a typical random rollout, the winning move is only selected immediately less than 5% of the time. It is of course possible that the move is selected after other moves, but then there is the risk that the piece has already moved away or been captured by the opponent. The random rollouts are thus not incredibly indicative of the actual desirability of a state.

The weakness of random rollouts can be experimentally verified. We pitched an ε-greedy-agent ($\varepsilon = 0.65$) (see 6.2.2) with the Omniscient Estimator (see 7.2.2) and the Random Rollouts Evaluator with 100 rollouts against a Random Agent, both using a Gravon Setup (see 5.1.4) for 100 games. The achieved winrate was 83.8%. This does

Rank	Moved	Discovered	Captured
Bomb	-	100	750
Marshal	100	100	500
General	100	50	250
Colonel	100	25	100
Major	100	20	50
Captain	100	15	20
Lieutenant	100	10	10
Sergeant	100	5	5
Miner	100	20	50
Scout	100	-	2
Spy	100	-	100
Flag	-	-	1000

TABLE 8.1: The points assigned to each piece of a certain rank, based on having moved, being discovered and being captured.

not seem to bad, but if we swap out the Random Rollouts Evaluator for a NUC Evaluator (see 8.1.1), the winrate jumps to 97.9%. Not only is this winrate considerably better, but the test was also completed nearly one hundred times faster, displaying the incredible inefficiency of using random rollouts in the context of Stratego.

8.1.3 Jeroen Mets Evaluator

Mets describes an alternative method to Random Rollouts in his work [Met08]. As the method does not have a specific name, it will be referred to as the *Jeroen Mets Evaluator*. The method is fairly simple: every piece rank has a specific value based on it having moved, being discovered and it being captured. These values can be found in table 8.1.

If we sum all the points together, we find that each side could have 11087 points at most. The points are thus summed for each side to find the values for *redPoints* and *bluePoints*, and then we use the following formula to determine a score value:

$$score = \frac{redPoints/BluePoints}{11087} \tag{8.2}$$

The found score value is then returned as the value to be assigned to the given board state.

A surprising value is the high value for capturing a Bomb, which is even higher than the value for capturing a Marshal. Mets argues that as Bombs are typically close to the Flag, losing one is indicative of an impending defeat. Other interesting values include the low value for the Scout and the lack of a value for discovering one. This is done so that Scouts can effectively discover enemy pieces without harming the agent's score too much, thus seeing a net positive which should promote scouting. Miners receive a slightly higher capture value, due to their utility in capturing Bombs.

To see the effectiveness of this method, an experiment was done to compare it to the NUC Evaluator (see 8.1.1). Two ε-greedy-agents ($\varepsilon = 0.65$) (see 6.2.2) using a Database Estimator (see 7.2.4) and a Gravon Setup (see 5.1.4) have played 100 games against each other, each using either the Jeroen Mets Evaluator or the NUC Evaluator (see 8.1.1), both with 10000 evaluations. The Jeroen Mets Evaluator only achieved a

winrate of 14.3%, showing with high confidence that it is considerably worse than the NUC Evaluator.

8.1.4 Flat NUC Evaluator

The Naive Unit Count Evaluator (see 8.1.1) has seen considerable success in the experiments it has been used in. However, it was originally intended just to be a placeholder. It is possible that variations of this method could feasibly be a bit better than the NUC Evaluator.

Recall that the NUC Evaluator uses a formula to determine the score value based on the total number of pieces for each side. In said formula, the count for Red is divided by the count for Blue. This has an interesting property, where the situations where Red is ahead spans a larger part of the range of possible values than the situation where Blue is ahead. In the case where Red is most favoured, they would have 40 pieces to 1, which after division becomes 40. If Blue is most favoured, it is 1 to 40, which after division is 0.025. In the case where they are equal, the division becomes 1. This means that the range 0.025-1 is used for when Blue is ahead, and $1 - 40$ is used for when Red is ahead, which is a much greater range.

To see if this has a large impact, we will look at a version that 'flattens' this difference, hence the name *Flat NUC Evaluator*. In this version, the formula to calculate the score value based on the number of *friendlies* (friendly pieces) and *enemies* (enemy pieces) becomes

$$score = \frac{(friendlies - enemies)}{80} + 0.5 \tag{8.3}$$

In this formula the situation where both players are tied in the number of pieces is exactly in the middle of the 0-1 range, at 0.5. The rest of the situations are equally divided over the range.

To check if this version is more or less effective than the NUC Evaluator, an experiment is done with two ε-greedy-agents ($\varepsilon = 0.65$) (see 6.2.2), both using the Database Estimator (see 7.2.4) and the Gravon Setup Provider (see 5.1.4). One agent uses the regular NUC Evaluator, the other the Flat NUC Evaluator, with 10000 evaluations per turn each. After playing 200 games, the Flat NUC Evaluator achieved a winrate of 42.3%. This is certainly not a poor result, but it does show that the Flat NUC Evaluator is not better than the original NUC Evaluator.

8.1.5 NUC Without Flag Evaluator

At the start of this chapter, we stated that any evaluator should return 1 in a winning state, and 0 in a losing state. After all, if you capture the Flag the game immediately ends in victory, so there is no real reason to consider the nuances of what the rest of the board might look like. There is however a slight issue with this assumption, namely that we are not working with perfect information but with hidden information. The location of the Flag also happens to be the *most* hidden information, as the location can only be revealed by discovering the locations of any remaining Bombs, as well as discovering that any of the other pieces are movable. In other words, we are very rarely certain of the location of the Flag before actually attempting to capture it.

This has consequences for an AI agent that does not realise these nuances. After all, it considers whatever the estimator that it is using provides as the absolute truth in its tree search. And because in most cases the value assigned to the other states

is not close to 1, accidentally guessing the position of the Flag wrong can severely skew the search in a single direction. If an agent is convinced the Flag is right next to it, it will always try to capture it regardless of how certain the estimator actually is. This can lead to the agent making a blunder and losing an important piece.

The *NUC Without Flag Evaluator* loses this assumption, and does not consider capturing the Flag a victory. It does however consider losing its Flag a defeat, as it is of course certain of the position of its own Flag. The rest of the evaluator is completely identical to the NUC Evaluator.

To see if losing the assumption regarding capturing the opponents Flag improves the agent, we run an experiment where two ε-greedy-agents ($\varepsilon = 0.65$) (see 6.2.2) play 100 games against each other, both using the Gravon Setup Provider (see 5.1.4). They both use the Naive RvH Neural Network Estimator (see 7.2.5), as this estimator is not capable of correctly estimating the entire board and will thus frequently misplace the Flag in various positions. One agent uses the NUC Without Flag Evaluator and the other uses the regular NUC Evaluator, with 10000 evaluations each. The agent using the NUC Without Flag Evaluator achieved a winrate of 61%, in fact defeating the regular NUC Evaluator.

While this is a good result, it is only valid if we are using an estimator that is not always capable of correctly finding the Flag. If we replace the Naive RvH Neural Network Estimator with the Database Estimator (see 7.2.4), we find that the agent with the NUC Without Flag Evaluator only manages a winrate of 48.9%, slightly worse than the NUC Evaluator. Because this value is so close to the 50% mark, the experiment was repeated with 200 games instead. This found a new winrate of 47.2%, slightly lower. The lower winrate makes sense; although the agents mostly tend to win by eliminating all enemy pieces, the occasional win by Flag capture may not be as attractive to the agent using this version than the agent with the regular NUC Evaluator.

We thus conclude that the NUC Without Flag Evaluator may be a bit better than the regular NUC Evaluator when used in situations where finding the Flag may be harder than usual, e.g. when playing against an unpredictable opponent. However, in cases where the estimation of the flag location is more reliable, the regular NUC Evaluator seems like the *slightly* better option.

8.1.6 Naive Unit Value Count (NUVC) Evaluator

It may seem obvious that not every piece is equal; some pieces simply have much more utility than others. Take the Marshal for example: capable of capturing any enemy piece that is not a Bomb or another Marshal, it is an extremely useful and dangerous piece. It would not require much thinking to conclude that it is more valuable than a simple Sergeant. However, as de Boer demonstrates [Boe07], the value of these pieces can change over time. The Spy is an excellent example of this; if the opponent loses their Marshal, the Spy loses most of their value as it can now only capture the Flag and draw against another Spy, which almost every other piece on the board can do as well.

Nonetheless, the idea of assigning a value to specific pieces is not new. Plenty of attempts have been made to assign ranks to pieces to find a good evaluation function. The *Naive Unit Value Count Evaluator* is another one of those attempts. Each rank has an assigned value, and the total value for each side is calculated by multiplying the value for each rank by the number of respective pieces of that rank are still on the board. The values assigned are as follows:

- *Flag*: 0.
- *Spy*: If the enemy has a Marshal, 10. Otherwise, 1.
- *Scout*: 1.
- *Miner*: If the enemy has 2 or more Bombs, 6. Otherwise, 3.
- *Sergeant*: 4.
- *Lieutenant*: 5.
- *Captain*: 12.
- *Major*: 14.
- *Colonel*: 16.
- *General*: 18.
- *Marshal*: If the enemy has a Spy, 20. Otherwise, 40.
- *Bomb*: 10.

The evaluator then calculates the value for both sides, and calculates the score using the following simple formula:

$$score = \frac{redValue/blueValue}{324} \tag{8.4}$$

The division by 324 ensures that the output values are always between 0 and 1.

To find out if the NUVC Evaluator works better than the regular NUC Evaluator, we run an experiment where two ε-greedy-agents ($\varepsilon = 0.65$) (see 6.2.2) play against one another. Both are using the Database Estimator (see 7.2.4) and the Gravon Setup Provider (see 5.1.4). One agent is using the new NUVC Evaluator and the other is using the regular NUC Evaluator, both with 10000 evaluations. After 100 games, the agent using the NUVC Evaluator achieved a winrate of 57.2%, suggesting that the NUVC Evaluator is slightly better than the NUC Evaluator. However, as we have seen with the NUC Without Flag Evaluator (see 8.1.5), there is a chance that in a situation where the estimator can not provide information that is as accurate, this evaluator will perform more poorly.

To verify this, we take the same setup but swap out the Database Estimators for the Naive RvH Neural Network Estimator (see 7.2.5). After 200 games, the NUVC Evaluator now reaches a winrate of 52.8%. Once again this is slightly better than the results for the NUC Evaluator, but unfortunately they are not significant. If we check for $P \leq 0.05$, we find that the actual winrate is likely between 45.9% and 59.7%. So while there is a good chance that the NUVC Evaluator is slightly better, it is not entirely certain in this case.

8.1.7 Neural Network (NN) Evaluator

In existing literature, the use of neural networks to solve (parts of) Stratego has not been explored much. The only case appears to be Smith [Smi15], who used convolutional neural networks to predict moves and evaluate states. Smith however seems to have used a poor input representation, as well as flawed training data that hampered the resulting agents. The results were still not bad, but they did not seem to be as good as they could potentially be.

One of the goals of this thesis is to try to apply neural networks to the problem of playing Stratego. Here, we will consider the application of a neural network to the *Static Evaluation Problem*, by using one as an evaluation function.

Ideally, a neural network learning to evaluate states is training using a self-training loop, where a good agent using the neural network uses the MCTS algorithm to find the correct labels for each state while playing games against an identical agent, as is for example done by Silver et al. [Sil+16]. However, due to time constraints we will have to limit ourselves and instead use data extracted from the Gravon Database [Jun15]. From this database we extract all board states that appear and attach a label that states whether or not the state eventually led to a victory. States from games that led to a draw are discarded.

Naturally, this is not a perfect labelling. Preferably, we would attach labels with a bit more granularity; if a game happens to swing from being in favour of Red to being in favour of Blue, all states where Red was ahead would still be marked as 'leads to defeat'. Additionally, states very early in the game where the outcome is by no means clear are also labelled as if the outcome *is* already clear. Nonetheless, the hope is that because there are over 1.7 million states, these 'mistakes' cancel out and the network is still able to learn a general sense of what a desirable state looks like.

Network architecture

The neural network itself has an input layer of 3312 neurons. This corresponds to three channels of 92×12, e.g. for every square on the board we can one-hot encode each of the 12 ranks. The three channels are used for the following:

1. A channel marking the ranks and locations of all friendly pieces.
2. A channel marking the information that the opponent knows of all friendly pieces. If a piece on square B4 could only be a Bomb or a Flag for example, two neurons would be activated for that location.
3. A channel marking the information that the agent knows about all enemy pieces, marked similarly as in the second channel.

Following the input layer are five hidden layers of 300 neurons each, using the ReLu activation function. The network ends in a Softmax output layer of two neurons, one to mark winning states and the other to mark losing states.

The training process went surprisingly well, with the cross-entropy loss decreasing from ~ 0.7 to ~ 0.18, and accuracy increasing to $\sim 93\%$. This means that the network is fairly capable at learning the given training data, and should be able to give a rough estimate of the desirability of a certain state.

The *Neural Network (NN) Evaluator* takes the resulting neural network and uses it as an evaluation function. It converts the given board state to the correct input representation and feeds it to the neural network. It then takes the resulting confidence in the state being a winning state as the score value, and returns it.

In order to find the effectiveness of this agent, we run an experiment where two ε-greedy-agents ($\varepsilon = 0.65$), both using a Database Estimator (see 7.2.4) and a Gravon Setup Provider (see 5.1.4). One agent uses the NN Evaluator, the other uses the NUC Evaluator (see 8.1.1). They both only get 1000 evaluations, as the NN Evaluator takes much longer to execute due to the neural network calls and the experiment would otherwise take too long to execute. After 100 games, we find that the agent using the NN Evaluator achieves a winrate of only 8.2%, which is very poor.

When observing the agent using the NN Evaluator play and viewing debugging logs, it becomes clear why the agent is not doing well. In general, the agent appears to be very defensive, rarely initiating a capture attempt. In addition, it seems that the data is not good enough and has some issues with regards to balance, e.g. certain positions are overrepresented in the data, which causes the neural network to learn the wrong thing. This manifests itself in two main ways: First, there are certain positions on the board that the agent really likes to place certain pieces on. Getting a piece there seems to increase the perceived chance of winning to nigh-certainty. Second, it seems that positions where pieces are in the opponent's corners are fairly rare, which means the agent generally tends to avoid them. The same goes for the back row in general, as human players typically tend to not touch the back row much until they are certain that the Flag is there. This means that positions where there are friendly pieces on the enemy back row are also very rare.

Nonetheless, there are some upsides. The agent *does* seem to have a general sense of how good its position is, e.g. when it appears to be ahead it does generally also perceive itself to be ahead, and when it begins losing it does actually perceive itself as such. This suggests that with better training data, this approach could work much better.

8.1.8 Double NN Evaluator

Unfortunately, the Neural Network Evaluator (see 8.1.7) does not seem to work very well. However, there are some things that can potentially be done to improve the evaluator without changing the neural network.

One of the major ways in which the NN Evaluator blunders is by considering certain positions to be much stronger than others, for no apparent reason. This is likely due to seeing certain positions too often compared to others, an imbalance in the training data. One way to potentially mitigate this is by using the neural network twice instead of only once per evaluation.

Of course, using it twice with the same input would lead to the same output. But what can be done is inverting the board state and feeding that to the neural network. We are thus not only interesting in the chance of winning for Red, but also the chance of losing from Blue's perspective. The neural network is not symmetrical in that regard, meaning the confidence that Red will win is not necessarily equal to the chance that Blue thinks it has of losing.

The *Double NN Evaluator* uses the neural network twice, once to determine Red's chance of winning $rWin$ and then again to determine Blue's chance of losing $bLose$. We then put these values in the following formula to take the average:

$$score = \frac{(rWin + bLose)}{2} \tag{8.5}$$

The idea is that this should make the situations where the neural network is overconfident in certain positions less impactful, as it can be averaged out. Of course, this comes at a significant cost to performance, as doubling the amount of neural network calls also doubles the execution time required to evaluate a state. For reference, making 10000 evaluations per turn using the regular NN Evaluator takes approximately 3-5 seconds on an i7-4790k, whereas the Double NN Evaluator takes 6-10 seconds. Taking an average game length of 381 moves [Art10], this means the agent takes between 30 and 45 minutes to play the average game. This is not too bad if the agent were to play against a human, but it is not great either.

The most important part however is how well this method performs. We run an experiment with two ε-greedy-agents ($\varepsilon = 0.65$) (see 6.2.2), both using a Database Estimator (see 7.2.4) and the Gravon Setup Provider (see 5.1.4). The first agent uses the Double NN Evaluator, the second uses the NUC Evaluator (see 8.1.1). Again, they both only get 1000 evaluations, as the experiment would otherwise take too long to do. After 100 games, the agent using the Double NN Evaluator has achieved a winrate of 12%, which is an improvement over the regular NN Evaluator, but still rather bad.

Despite seeing a slight improvement, the evaluator is still pretty bad. Doubling the number of calls to the neural network seems to have helped with the problem where the network was overconfident in certain positions. The largest issue now appears to be the defensive attitude that the agent seems to get from using it. Normally a defensive attitude would not be necessarily a poor idea, but it is typically bad against AI agents, where more offensive agents seem to be much stronger.

8.1.9 Double NN NUC Evaluator

The issue with both the Neural Network (NN) Evaluator (see 8.1.7) and the Double NN Evaluator (see 8.1.8) appears to be its extremely defensive strategy. In general, defensive strategies seem to work somewhat poorly against other AI agents. One reason for this seems to be the fact that they usually rely heavily on the given starting setup, which may be completely unsuitable for a strong defense. Another problem is that tree search algorithms have a limited range beyond which they have a hard time seeing advantageous moves. Defensive agents typically lose a considerable portion of their front before an enemy attack loses steam, at which point they may no longer be able to see far enough to realise that certain potential avenues of attack exist.

The NUC Evaluator (see 8.1.1) makes an agent much more aggressive. It sees states where the opponent has less pieces as more desirable, which makes it aggressively go after the opponents pieces, even if it in the long term could make it lose pieces of its own. A downside of the NUC Evaluator is that it does not have much positional awareness, e.g. the positions of the pieces on the board do not matter to it, it only cares about the number of pieces. This contrasts with the NN Evaluator, which seems to do a bit better on positional awareness, but does not like capturing enemy pieces all that much and prefers to play defensively.

Because of these contrasting qualities, it may be worth it to try combining both evaluators into one. The NN Evaluator would provide it with positional awareness, whereas the NUC Evaluator would give it aggressive impulses to move its pieces forward and begin capturing enemy pieces. This is the idea behind the *Double NN NUC Evaluator*; two evaluators working as complements to one another.

In practice it works as follows: both evaluators calculate the score values for the state they are given. Then, the average of the two is taken and returned as the score value for the state. The formula thus looks like this:

$$score = \frac{1 - \left(1 - \frac{friendlies/enemies}{40}\right)^8 + \frac{(rWin + bLose)}{2}}{2} \tag{8.6}$$

In this formula *friendlies* is the number of friendly pieces on the board, *enemies* is the number of enemy pieces on the board, *rWin* is the perceived chance of victory that Red has and *rBlue* is the perceived chance of defeat that Blue has.

To see if this combination of evaluators actually works better, we run an experiment with two ε-greedy-agents ($\varepsilon = 0.65$) (see 6.2.2) using the Database Estimator

(see 7.2.4) and the Gravon Setup Provider (see 5.1.4). One of the agents uses the Double NN NUC Evaluator, the other uses the regular NUC Evaluator. Each agent only gets 1000 evaluations, as the experiment would otherwise take too much time to run. After 100 games, the agent using the Double NN NUC Evaluator has achieved a winrate of 16%. We see an improvement over both the NN Evaluator as well as the Double NN Evaluator, though the agent still performs fairly poorly.

It seems that combining the evaluators in this way is not performing very well. Judging by the debug logs, it seems that the Double NN Evaluator tends to return values within a greater range than the NUC Evaluator generally does. That means that the NUC Evaluator is often 'overshadowed' by the Double NN Evaluator. The idea for this evaluator might work better if the NUC Evaluator was given a larger weight than the Double NN Evaluator. The risk to doing that is of course that the Double NN Evaluator no longer really matters and could just be left out for a minimal loss in evaluation quality. At the same time, an agent could then simply use far more evaluations to make up for this issue.

It seems that the neural network developed so far is not good enough for it to be worth the hit to performance. It seems more efficient to use a simpler evaluator, like the NUC Evaluator, more often instead. This leads to a higher agent quality compared to using the neural network. With improvements to the neural network, perhaps via self-play in some kind of training loop, the difference in evaluation quality can be overcome as the neural network certainly shows some potential.

8.2 Discussion

From the experiments done we find that the Naive Unit Count (NUC) Evaluator (see 8.1.1) and the Naive Unit Value Count (NUVC) Evaluator (see 8.1.6) do best out of the tested evaluators. An agent using them typically uses a more aggressive playstyle, typically preferring to capture enemy pieces rather than protecting its own. There is some merit to the NUC Without Flag Evaluator (see 8.1.5) as well, which seems to perform better than NUC in situations where it is harder to make accurate estimations.

From this it seems that we can conclude that there is an important connection between estimation quality and evaluation quality. It appears that in situations where the estimation quality is high, evaluators with a high quality also do better. However, in situations where estimation quality is lower, these same evaluators may not perform as well as before. In particular, evaluators that make large changes to the estimated value of a board state based on specific features of that state are vulnerable to this, such as setting the value of a state to 1 if the Flag is captured. Such values make sense when viewing the evaluation of a state in a vacuum, but when we consider that many things about the state can be very uncertain this may not be such a good idea. After all, if the real board state does not share the same features, then the evaluations will be considerably off, whereas if an evaluator ignores such specific features an AI agent may improve in quality.

Assigning specific values to specific ranks is a difficult thing to do, as can be seen when looking at the poor results that the Jeroen Mets Evaluator (see 8.1.3) obtained. Many things impact what value a rank should have, so keeping the values the exact same over the course of an entire game may not be the best way to do it. The values from the NUVC Evaluator seem to do fairly well, although it may be that there is more potential to be realised here with a better tuning of the values. However, there is of course the risk that an evaluator could run into the same pitfall described before,

where a disproportionate assignment of values to specific ranks could mislead an agent working with poor estimations.

The neural networks developed so far are not quite ready to perform. The largest issue here seems to be a lack of well-balanced training data, as well as poor labelling. The largest constraint was time, as with a proper self-training loop this could theoretically be improved much. In their current state however, they cost far too much processing time for too little reward. There is a sense of positional awareness in there, but it is too flawed to be of any practical use at this point.

Whatever the best approach exactly is, it appears that random rollouts is not it. Stratego games simply take too long for a random rollout to be performant enough, and because there are so many possible moves the results are often not very indicative of the desirability of a state. Considering that simple methods like NUC perform so much better for far less processing power and that there are little to no avenues to significantly improve the performance and quality of the Random Rollouts Evaluator (see 8.1.2), it seems that the method of random rollouts applied to Stratego is little more than a dead end.

8.3 Conclusion

We have considered multiple ways of solving the Static Evaluation Problem. In this chapter a number of evaluation methods have been described that have different approaches to assigning a specific value to a state and which have their own strengths and weaknesses based on how it does that.

The best evaluators to use are the Naive Unit Count (NUC) Evaluator (see 8.1.1) and the Naive Unit Value Count NUVC Evaluator (see 8.1.6). Both of these evaluators perform well, whether estimation quality is high or low. These evaluators are also highly performant and can thus be used at scale, doing hundreds of thousands of evaluations per move without taking too much time. This makes these evaluators excellent for use against humans, who would prefer not to have the opponent take minutes per move.

There is still significant potential in the neural network approach. With a better training process, much better results should be achievable. This will however take considerable time and processing power to do. A limiting factor of using neural networks is the relatively poor performance, however with a good multithreading approach this should be more manageable.

Chapter 9

Human Testing

The primary purpose of any AI Stratego agent is of course playing the game against a human. A good Stratego agent should be able to occasionally win against a human of course, as otherwise the human player may not feel challenged enough. Few AI agents for Stratego have ever cleared this bar, and so far no agent can actually consistently beat humans. The best agent described in literature appears to be de Boer's Invincible [Boe07], which in his tests of the AI against a few people managed to win a couple games.

In this chapter we will be looking at four games played against two people by slightly different AI agents each time. Because the experiments so far have not indicated that a single agent is the best in all situations, a few different AI agents have played against the humans in different scenarios. Specifically, both players played two games each; one game in which they had to pick a setup from the Gravon Setup Provider (see 5.1.4) and one in which they were free to create their own setup. When picking from the Gravon setups, they were shown a random setup and were allowed change to another if it was not to their liking. They were allowed to keep changing until they found a setup that they sufficiently liked.

The point of playing against an agent with a Gravon Setup is to test the effectiveness of the Database Estimator (see 7.2.4) specifically, as that estimator can often pinpoint the exact setup used in only a few moves. From that point on, the AI agent would know the ranks of every piece, and it would be interesting to see if that is enough to beat a human player.

9.1 The players

I convinced two close friends to play two games against the AI agents: Tom and Gijs. Tom is a friend I have known since elementary school and went to high school with. We often played strategy games together and have played Stratego together when we were younger. While he definitely has experience in playing the game, he did mention that 'it had been a while' but he would give it his best.

Gijs is a friend I met in high school. He is very proficient in many strategy games such as Chess, Diplomacy, Catan and also Stratego, which he until fairly recently often played online against another mutual friend. Although Tom is definitely a good player, Gijs is almost certainly a much stronger player when it comes to Stratego.

Their AI opponents are the following agents:

- *Tom w. Gravon setup*: ε-greedy-agent ($\varepsilon = 0.65$) with a Database Estimator, NUC Evaluator with 100000 evaluations and a Gravon Setup
- *Tom w. Custom setup*: UCB-agent ($c = 0.15$) with a Database Estimator, NUC Evaluator with 100000 evaluations and a Gravon Setup

- *Gijs w. Gravon Setup*: ε-greedy-agent ($\varepsilon = 0.65$) with a Database Estimator, NUC Evaluator with 500000 evaluations and a Gravon Setup
- *Gijs w. Custom Setup*: UCB-agent ($c = 0.15$) with a Direct Rank Estimator, NUC Evaluator with 150000 evaluations and a Naive RvH Setup

Because Gijs is the expected stronger player, I gave his AI opponents more evaluations (e.g. 'time to think') than the AI opponents for Tom. In the games where the players are allowed to create their own setup, I changed the ε-greedy method (see 6.2.2) for the UCB method (see 6.2.3), as that seems to perform better when the estimation quality is likely to be a bit lower.

For the second game of Gijs, he asked if the AI could also create unique setups, which prompted me to have him play against an agent using a Naive RvH Setup Provider (see 5.2.2). I also let him play against an agent using the Direct Rank Estimator (see 7.2.6), which in hindsight was not a good idea. The hope was that the fact that this estimator produces more consistent estimations would lead to it acting more decisively against Gijs, even if the moves chosen were not always ideal. This turned out to be a bad idea, and I should have used the Naive RvH Neural Network Estimator (see 7.2.5) instead.

All games were played using the Naive Unit Count (NUC) Evaluator (see 8.1.1), as that appears to be the evaluator that is able to well most consistently.

9.2 The games

The four games were played in the StrAItego program. As the program tracks all information for every AI agent, it also displays this information to the human players. Both players had a *graveyard*, which shows them which pieces are still on the board and which have been captured. Additionally, they can see which ranks a piece on the board could still have. Naturally, this includes whether or not a piece has moved. They can also see the same from their opponent's perspective, e.g. if the opponent knows if a piece has moved or seen its rank. In a normal real-life game players are expected to remember this information themselves, meaning they have an advantage that they would normally not have. Nonetheless, this 'advantage' may be considered fair as it can be more difficult to remember what exactly happened on the board if you do not have it physically in front of you.

9.2.1 Tom with Gravon Setup vs. AI agent - AI Victory

For the first game, Tom played against an ε-greedy-agent ($\varepsilon = 0.65$) with a Database Estimator, NUC Evaluator with 100000 evaluations and a Gravon Setup. He selected the second setup from the Gravon database that we was presented, which can be seen in figure 9.1.

In the opening phase of the game, both Tom and the AI agent aggressively moved their Majors forward into their opponents center. However, upon realising that the Blue Major would be able to take a considerable amount of pieces in his center, Tom pulled his Major back a bit. This did not help much, as the Major was already too deep in his center and his own Major would not be able to catch up. After taking six pieces Tom decided to capture the Major with the only piece nearby that could capture it, revealing his General. In figure 9.2 is the situation before Tom captured the Major with his General.

Losing a Major early can be a considerable blow, however in this case the AI traded it for the reveal of Tom's General. The location of the General was enough for

FIGURE 9.1: The Gravon setup Tom used in his first game.

the Database Estimator to do what it does best: pinpoint the exact setup Tom was using. After only 19 moves in total, the AI agent is working with effectively perfect information.

The AI uses this information to move its Marshal forward through the center, where it decides to capture Tom's Major. However, Tom had moved his Spy forward, and manages to capture the Blue Marshal, a major blow to the AI. The NUC Evaluator thus shows a weakness, in that it considers trading a Marshal for a Major an advantageous trade, whereas a human player would likely not consider that a good trade at all.

From this point on the AI becomes noticeably less aggressive, occasionally moving a piece forward through the center but preferring to shift pieces around on its own side of the board. This is likely because Tom has both his Marshal and his General positioned behind the lakes, making it easy to catch most incoming pieces. This means attacking looks unattractive to the agent. Tom takes initiate and moves a few pieces, including his General, up through the left lane, where he encounters a number of Bombs blocking the left side of the board. This also explains why the AI was not moving down the left lane; the Bombs were blocking the entry there. Tom brings up a Miner to clear one of the Bombs (see figure 9.3), but after clearing a single mine it gets captured by a Sergeant, preventing him from clearing any other Bombs.

After losing his Miner, Tom pulls his General back. This appears to be just in time as the AI agent moves a Colonel down the center, capturing a second Major before losing it to the Red General. Because a large number of pieces have moved in the center, Tom decides to take his General up through the center and begins capturing as many pieces as possible, frequently checking if pieces have moved yet.

What is perhaps the most interesting moment of the game happens during this offensive. Tom's General on D8 has Blue pieces above it and to its right. On D10 is the Blue Flag, which Tom did not know for certain yet but was considering as a potential location for the Flag. At this point, the AI agent attempted to capture the Red General with a Blue Lieutenant on D9. Because the General had already revealed its rank, this may seem like a complete misplay, but from the AIs perspective it made sense.

The reason for this is that the AI agent considers Toms reasoning to be the same as its own. In other words, Tom should like capturing pieces. If the AI does not attempt to do D9 to D8, then Tom could consider doing D8 to D9, which would place him next to the flag at D10, an incredibly dangerous position. So it decided to sacrifice a piece to Tom in order to coerce him to move right instead, which Tom did,

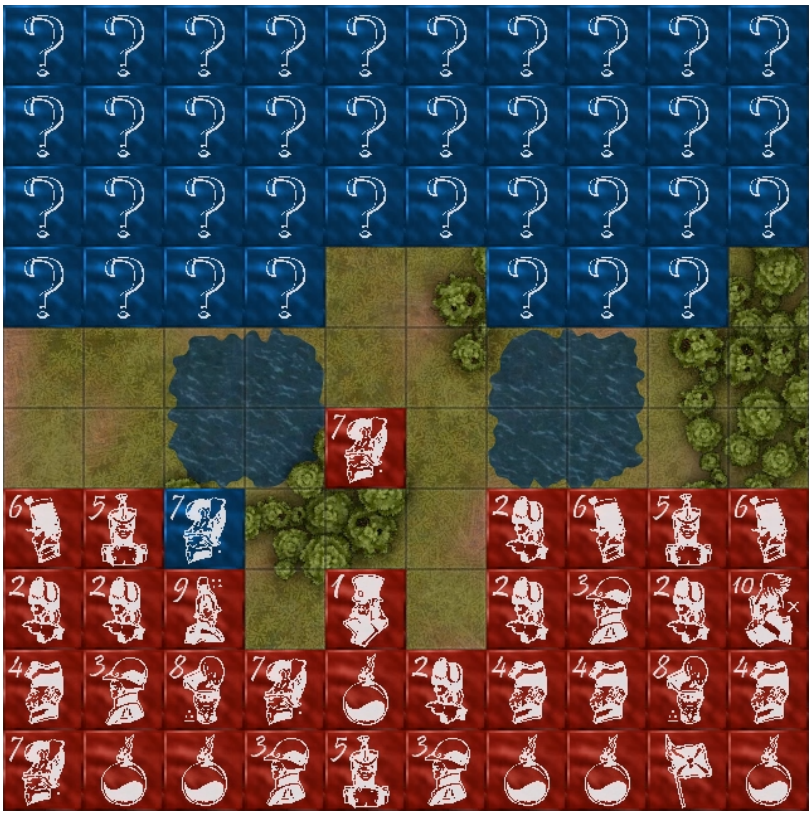

FIGURE 9.2: The Blue Major takes six pieces, prompting Tom to reveal his General

stating that he did not think an AI would sacrifice a piece like that if the Flag was behind it. From the logs we can also conclude that this was not a fluke; the move E8 to D8, sacrificing the Scout, was the second-lowest valued move, meaning the AI considered it detrimental to its position.

Tom using his General to clear out pieces works well until he attempts to capture the Blue General, stopping his offensive (see figure 9.4).

After the loss of his General, Tom decided to take the Colonel that was in front of the Flag up the right flank, to begin capturing more moved pieces. However, by capturing a piece above the right lake he exposes his Flag to an unseen Blue Scout at the very top of the board, allowing the AI to capture his Flag and win the game, as seen in figure 9.5. In total the game took 190 moves, which is well under average for a Stratego game.

9.2.2 Tom with Custom Setup vs. AI agent - AI Defeat

In Tom's second game he played against a UCB-agent ($c = 0.15$) with a Database Estimator, NUC Evaluator with 100000 evaluations and a Gravon Setup. He was allowed to create his own setup this time, which can be found in figure 9.6.

In the initial opening phase, a few pieces are traded back and forth. A notable exchange happens when the AI once again gets its General deep into enemy territory, even capturing the Red Spy before being traded against the Red General. This can be seen in figure 9.7. So far, the AI is playing pretty similarly to the first game. A notable difference however is that the Database Estimator is having a much harder time coming up with quality estimations. It seems that the revealed pieces are slightly

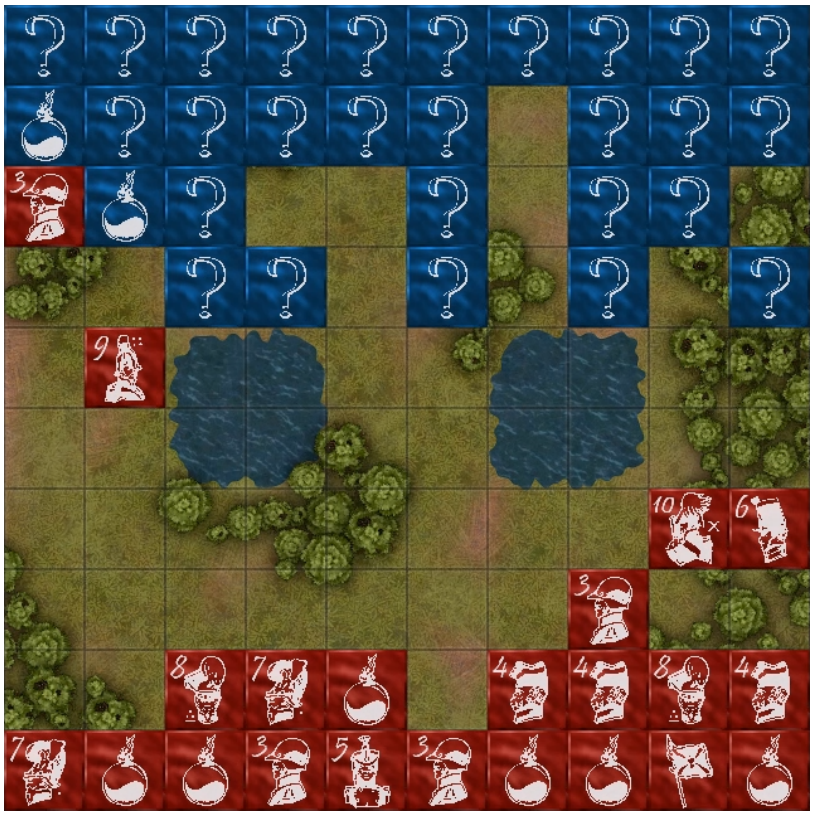

FIGURE 9.3: Tom moves up with a Miner and his General in the left lane, encountering a number of Bombs.

misleading, making it believe that Tom's setup is based on a setup that is otherwise very different from what Tom is actually using.

Tom decides to go on the offensive, bringing his Colonel further up the center, but loses it to a Blue Colonel before he can do too much damage. He then decides to do a slightly more coordinated assault, using a Major and the remaining Colonel to move up in the center. The AI tries to meet them in the center with a Captain, not knowing that the Colonel is there, which gets promptly captured. The Red Major moves further forward, but gets captured by a Blue Colonel. The resulting situation can be seen in figure 9.8.

After the loss of his Major, Tom initially advances with his Marshal all the way to the back row, before retreating back to defend his right flank where the AI has been posturing with one of its pieces. This gives the AI an opening to use its Marshal to come down the left flank and begin capturing some pieces. The Marshal even gets to C2, merely two moves away from the Flag with no pieces that could oppose him, but the AI decides to move right instead to capture a Captain (see figure 9.9), and afterwards blunders its Marshal away to the Bomb on D1.

Tom now realises that only one piece can still realistically threaten his Flag, the Blue Colonel. He moves his Marshal to the left flank to defend and moves his own Colonel up through the center. Tom trades Majors on the right flank. After clearing out some moving pieces, Tom realises that the Blue Flag is in the top-left corner surrounded by Bombs. He brings up a Miner on the right flank to move via the back to the Flag, but he fails to defend against a Blue Lieutenant and after a long chase the Miner gets trapped between two Red Bombs on J4, where it gets captured.

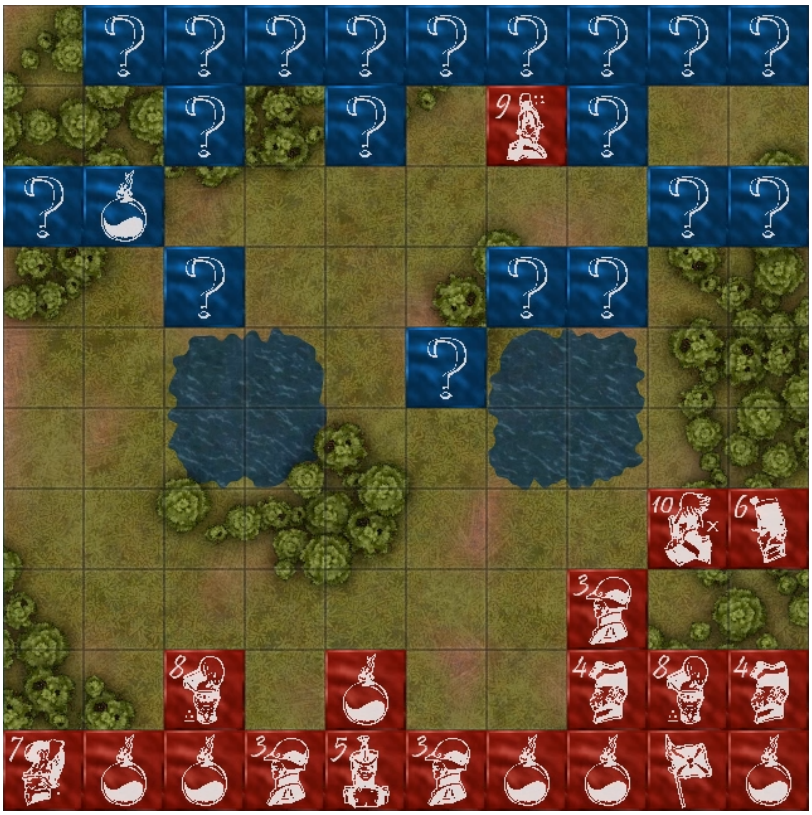

FIGURE 9.4: Tom starts clearing out pieces on the opponent's side, right before losing his General to the opponent's General to his right.

Once more Tom returns to the strategy of clearing pieces that have moved. However, the AI agent moves a Captain down through the center, to which Tom responds too late with his Marshal. The Blue Captain takes a Scout and a Sergeant before it is right next to the last Red Miner that Tom has left (see figure 9.10). At this point the AI could force a draw by capturing the Miner, but it instead opts to move right and loses its Captain to a Bomb. Looking at the logs reveals why: the estimator believed that the Bomb was the Flag instead, which means the agent thought it could win the game.

After this the game is virtually decided. Tom uses his Marshal to move up the center and scare the movable pieces away, which are two Miners and a Colonel. After having done this he moves his last Miner up the left flank, which is now defenceless, and effortlessly captures the Flag. The game lasted much longer than the first, taking 449 moves, which is slightly over the average game length for Stratego.

This game was an excellent example of why having correct information is vital to the AI agent's decision making. It was incredibly close to victory, before losing its Marshal to a Bomb, and again incredibly close to forcing a draw, before losing the Captain that could take the last Miner to another Bomb.

9.2.3 Gijs with Gravon Setup vs. AI agent - AI Defeat

Gijs' first game was against an ε-greedy-agent ($\varepsilon = 0.65$) with a Database Estimator, NUC Evaluator with 500000 evaluations and a Gravon Setup. This is the same agent that Tom initially faced, but with five times as many evaluations in an attempt to scale up the difficulty as Gijs is a stronger player. The setup Gijs decided to use can be seen in figure 9.11.

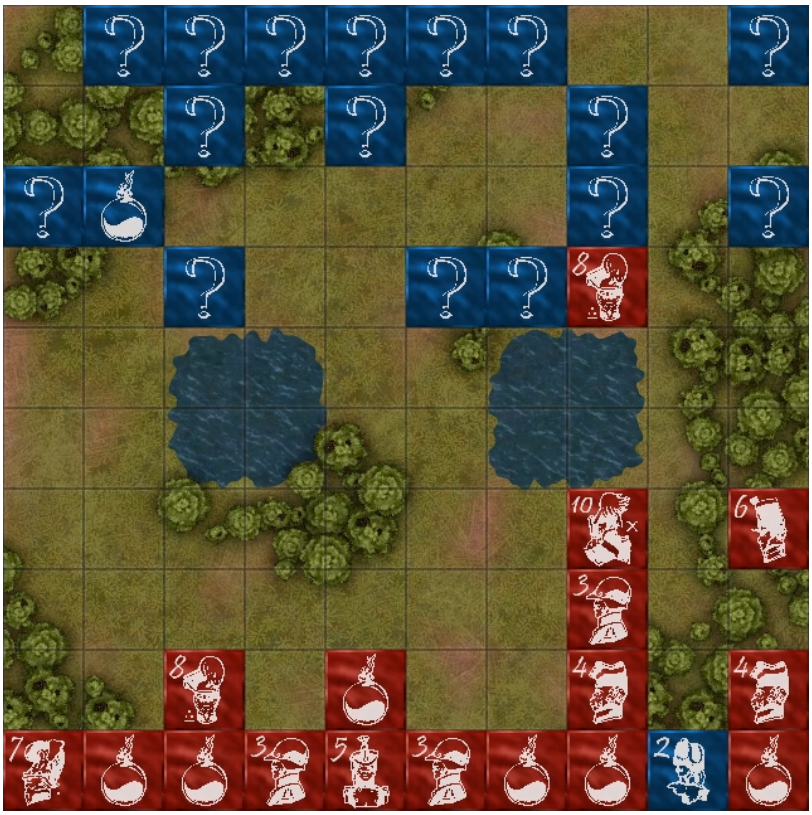

FIGURE 9.5: Tom starts clearing out pieces on the opponent's side, right before losing his General to the opponent's General to his right.

The AI agent immediately used the same strategy that it had used against Tom as well: use a General to move forward and start taking as many pieces as possible. Gijs initially did not anticipate this and thought the AI would move back, which caused him to lose a few lower-ranked pieces, his Spy and his General which the AI agent traded his General for. At this point Gijs remarked that a human player would probably not be this aggressive.

After discovering a Blue Bomb in the center lane, the AI began moving a piece down through the right flank. Because Gijs used a Scout to find the Blue Bomb, it gave the Database Estimator enough information to once again determine the exact setup that Gijs was using, meaning that after move 15 the AI was playing with perfect information.

Gijs did not trust the piece moving down on the right flank after the very aggressive opening from the Blue General, and began moving his Marshal over to the right. This ended up being a good idea, as shortly after it was revealed that the piece moving down was indeed the Blue Marshal. The Marshal cleared out most of the right flank, evading the Bombs that are there, forcing Gijs to trade his Marshal against the AI's to prevent further captures (see figure 9.11).

At this point, Gijs was severely behind in pieces. Both players had lost their General and Marshal at this point, but in addition to those Gijs had also lost his Spy, 6 Scouts, 2 Miners, 2 Sergeants and 2 Lieutenants. In contrast, the AI had not lost any pieces besides the General and the Marshal, and had only revealed a single Bomb and a Miner. Gijs was extremely lucky that the AI had most of its high-ranking pieces on the right flank, whereas Gijs had most of his in the middle and the left flank. Had either of these positions been swapped, Gijs could have lost a lot of high-ranking

FIGURE 9.6: The Custom setup Tom used in his second game.

pieces very early in the game.

At this point Gijs decided to be more aggressive. He moved his Colonel up the center and began carving a path through the AI's pieces, claiming two Captains. However, on the left flank the AI once again pushed forward, this time with a Colonel. Opting not to trade and lose his own Colonel, Gijs moved his piece out of harms way, leaving the Blue Colonel to claim his last three Miners (see figure 9.13).

Because Gijs knew that he now had to either find a Flag not protected by Bombs or capture all movable Blue pieces, he moved his Colonel up to join his other Colonel on the AI's side of the board. This left the Blue Colonel to capture even more pieces, including another Captain and the second Major.

Up until this point, the AI had been playing very well and had mostly dominated its opposition, capturing all but 6 movable Red pieces. Unfortunately, the last piece it captured was a Sergeant on the back row that on two sides is flanked by Bombs. This means that the Blue Colonel is now in a little 'nook' and it appears that due to the Red Colonels on its own half of the board, it was unable to actually see far enough ahead to begin moving its Colonel to other pieces. This suddenly made the agent play much more defensively as it could no longer see a good offensive option (see figure 9.14).

This gave Gijs the opportunity to start clearing out any pieces that had moved on the AI's side of the board. He eventually traded one of the Colonels to the other Blue Colonel. The agent now realised that it could potentially capture a Red Captain with one of its Majors, promptly chasing it around for a while on Gijs' side of the board. Eventually, Gijs moved the Captain back up through the center and placed it next to the remaining Colonel, letting it get captured in order to capture the Major in return.

A human at this point would have been able to win as Blue. Gijs did not really have enough pieces to cover all avenues of attack, and a human would have easily realised where his Flag was located and capture it with a Miner. However, for the AI agent this was a difficult task, as it would take far too many moves for a Miner to reach the Flag. Gijs continued clearing out any pieces that moved with his Colonel.

After clearing out enough pieces, the AI could once again reconsider moving its own Colonel out of the little 'nook' it was stuck in. This prompted Gijs to chase it around with his Colonel. He had also captured enough enemy pieces to realise that the Flag was unprotected by Bombs, which prompted him to move his remaining Sergeant and Captain up to the enemy side of the board (see figure 9.15).

After chasing the Colonel around for a bit, Gijs sacrificed his Captain so that

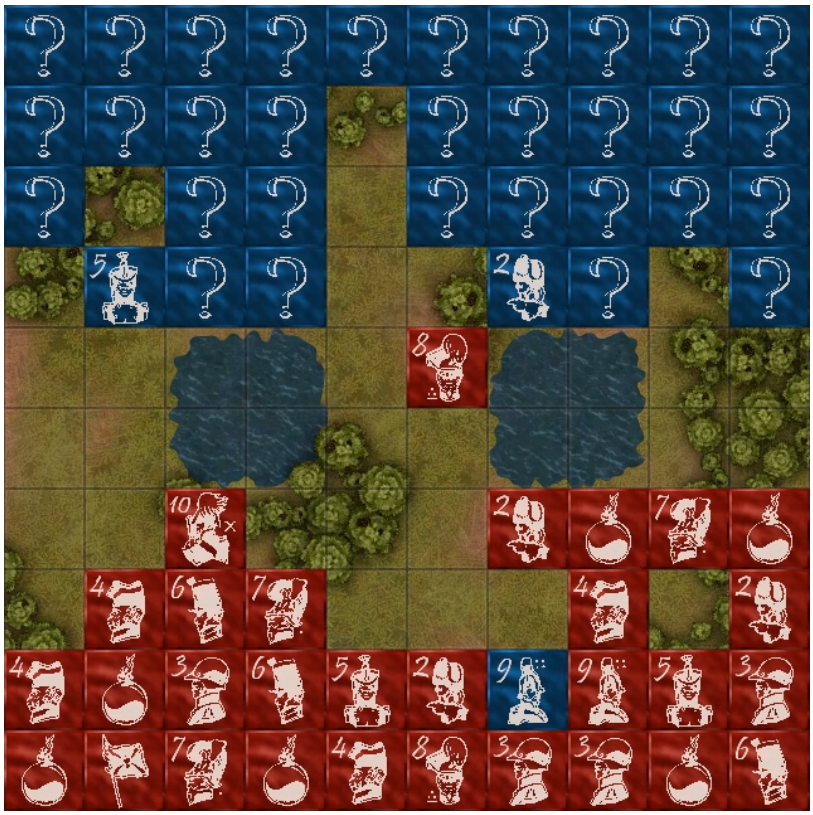

FIGURE 9.7: After the initial opening the Blue General captures the Red Spy, before being traded against the Red General.

he could get the Colonel in a position where he could trade it for his own Colonel, winning the game. Gijs' only remaining movable piece was a Sergeant.

Gijs could have attempted to take any of the unmoved pieces in the hope that it was a Flag, and was at times close to doing so. However, there were five unmoved pieces that were also undiscovered, so he would have to get a bit lucky. As Gijs is a somewhat conservative player who prefers to be certain of his victory, he opted instead to try and take the last Colonel.

The AI agent played very well and quickly had Gijs in a bad position. However, it failed to actually capitalise on its very advantageous position, and ultimately lost due to it being unable to see enough moves ahead. The game ultimately lasted 335 moves, resulting in a very narrow victory for Gijs.

9.2.4 Gijs with Custom Setup vs. AI agent - AI Defeat

For his second game, Gijs was pitched against a UCB-agent ($c = 0.15$) with a Direct Rank Estimator, NUC Evaluator with 150000 evaluations and a Naive RvH Setup. In hindsight, this was a very weak agent due to the poor estimator, but I wanted to observe its performance against a human player. It led to a fairly one-sided game with few interesting things happening, so it will be described more briefly than the other games.

The custom setup Gijs used can be seen in figure 9.16. He blocked off the left flank with Bombs, and focused most of his high-ranking pieces around the center lane, creating a very offensively focused setup. This may in part be a reaction to the AI agent being very offensive early on in his first game, which this setup would counteract fairly well.

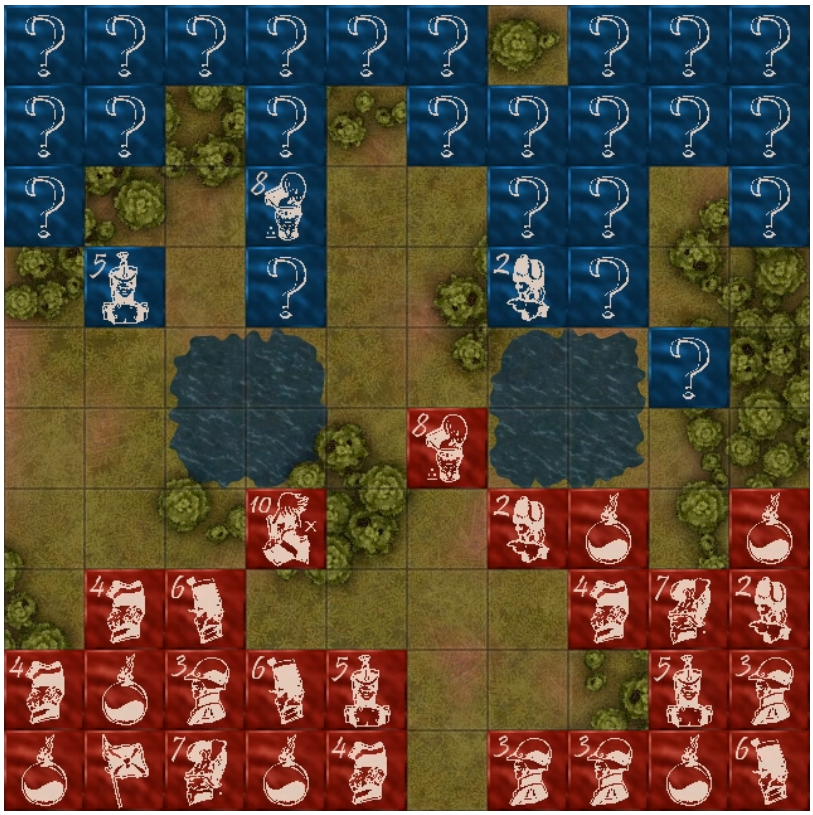

FIGURE 9.8: After a brief assault the Red Major gets captured by the Blue Colonel on D8.

Interestingly, Gijs found that the Naive RvH Setup Provider that the agent used had blocked of the right lane of the board. Gijs moved a few pieces including a Miner up to the Bombs, so that he could begin an offensive there at any time. Similarly, the AI agent moved a few pieces down the left lane, including a Miner, which captured both Bombs on the left side of the board. After capturing the Blue Spy early, Gijs moved his Marshal up through the center and his General up through the left lane, backed up by his own Spy. He cleared out a large number of moving pieces, including a Colonel, a Major, most Lieutenants and all but one Miner, before losing his Marshal by attempting to capture the enemy Marshal (see figure 9.17).

At this point, the AI agent sees no possible avenues for offense, and only tries to evade the pieces that have been discovered. It manages to trade its General for Gijs' General, but this leaves Gijs with both his Colonels and all three of his Majors, whereas the highest-ranked pieces that the AI agent has left are its three Captains.

Gijs thus decides to open up the right flank and captures one of the Bombs. This gives him opportunity to march a Major directly into the right flank of the AI agent's side of the board. Gijs decides to play it safe and moves his Colonels up through the left flank and the center, effectively pinning all Blue pieces in the top-right side of the board (see figure 9.18).

At this point Gijs can easily capture the remaining Blue pieces. Eventually he realises that he only has to capture a Scout, but he decides that he would rather win by capturing the Flag. He uses a Scout to test the first undiscovered piece, which ends up being a Bomb. He then captures an undiscovered piece on G8 with his Colonel, which would be risky if it were not for the fact that no Blue pieces can effectively threaten him anymore. This piece turns out to be the Flag, and Gijs wins

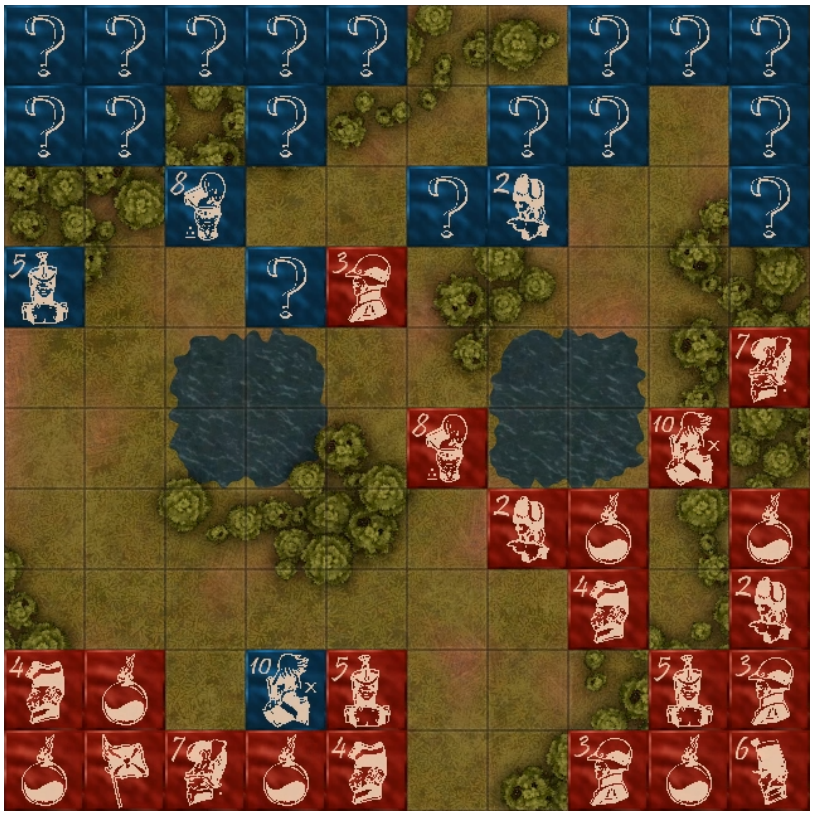

FIGURE 9.9: The Blue Marshal gets incredibly close to the Red Flag, but the AI loses the piece to the Bomb on D1.

the game after 289 moves, which is roughly 100 moves faster than the average game takes.

Gijs clearly was the dominant player this game, leaving nothing to chance. The AI agent was also working with a bad setup for its typically aggressive playstyle, having both the Marshal and the General on the back row and blocking one of the lanes with Bombs, leaving virtually no good avenues of attack for it to exploit. This led to a very one-sided game, where Gijs had ample opportunity to give himself an advantage and take as many pieces as possible.

9.3 Discussion

The AI agents managed to win only one of the four games, but it was only truly outclassed in one game, where it was using a poor estimator. In the other three games, it repeatedly managed to get very close to victory or enforcing a draw, but a single mistake often kept it from realising these potential outcomes.

In the first game, Tom was quickly behind on pieces, but he kept his high-ranking pieces around. This allowed Tom to reestablish himself and go on the offensive. He eventually only lost because he moved a piece out of the Scout's way which could then capture the Flag. Had this not happened, the AI could probably not have prevented a defeat.

The AI's major limiting factor in this game appeared to be the limited search depth, which caused it to lose sight of potential offensives in the second half of the game. Its main strength was the quick pinpointing of the exact setup Tom was using, which provided it with a lot of useful information for the rest of the game. This

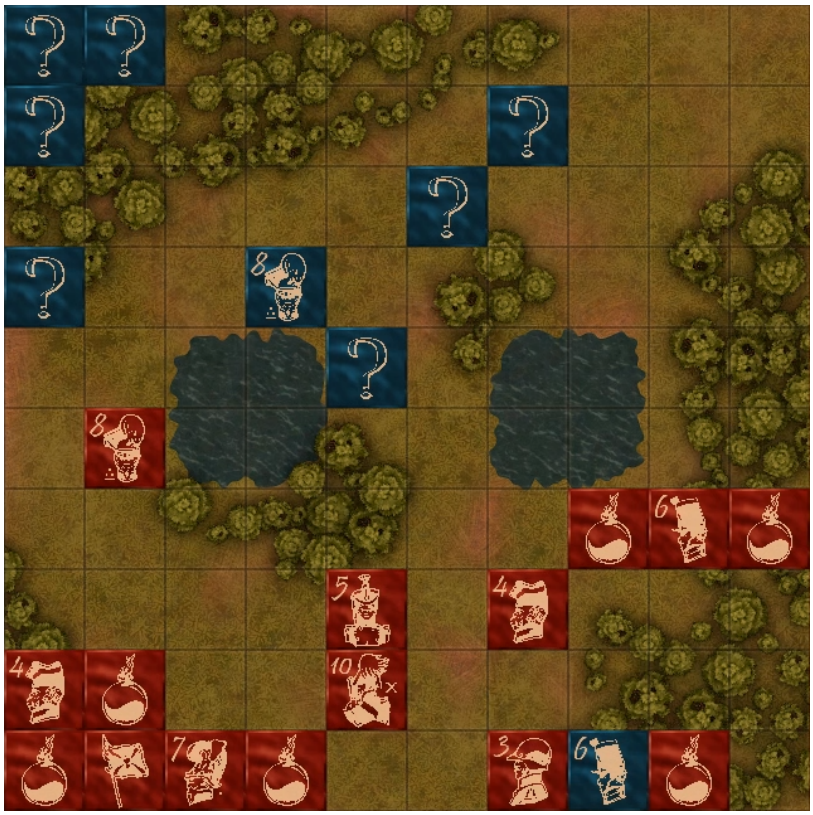

FIGURE 9.10: A chance to force a draw: the Captain must choose to take the Miner or the Bomb, but unfortunately not knowing which is which decides to take the Bomb.

also highlights the incredible importance of using a somewhat unexpected setup, particularly against AI agents.

In Tom's second game, the AI got very close to victory with its Marshal, but made a mistake by attempting to capture a Bomb. Here, the fact that the AI could not exactly pinpoint the used setup was its main limiting factor, causing it to lose a number of important pieces. Otherwise the AI played fairly well, countering Tom's assaults pretty effectively until it ran out of pieces to defend with, allowing Tom to capture the Flag.

In Gijs' first game, the AI initially played very well and established a very good position for itself, capturing a large amount of Gijs' lower ranked pieces, which would otherwise be used to scout with. Had either of the players used the same setup but mirrored, the AI would have likely won this game decisively. However, a bit of luck allowed Gijs to become more aggressive. Initially the AI could handle this, but then essentially 'lost' a Colonel because it could not see a way to use it within the search depth that it could explore. Had it been able to do so, it could have countered Gijs' advance and potentially brought a Miner down to capture his Flag. Once again, the limited search depth proves to be the main reason the AI could not bring this game to a good end.

Gijs' second game was very easy for him. The AI was not using a setup that it was suited for, and was also using a poor estimator. When it comes to gameplay there are not many things that we can conclude, other than the fact that selecting the right estimator and setup can be vitally important to the quality of the AI agent.

FIGURE 9.11: The Gravon Setup Gijs used for his first game.

9.4 Conclusion

We reviewed a number of games where various AI agents were tested against a human player. These games were played to test the effectiveness of the AI agents against human players, as well as to try to find their major strengths and weaknesses.

The major strength for the agents was definitely the use of the Database Estimator (see 7.2.4) in conjunction with the opponent using a Gravon Setup (see 5.1.4), although even when Tom used a custom setup the AI agent could have won the game. Even when playing against a stronger player like Gijs, the AI agent could, given the correct information, fairly easily win the early game.

In the mid- and endgame, the AI agent begins to falter. This is mostly due to the fact that the distance between friendly and enemy pieces begins to increase on average, and the human players typically begin to bring high-ranking pieces closer, which the AI agents have a harder time dealing with. Due to the increased distance, the AI agent finds it difficult to begin new offensives and keep its momentum going. It loses initiative, and relies on a mistake by the opponent to regain it. If that mistake does not come, it simply loses the game.

In conclusion, we find that the tested AI agents are very capable of playing the opening phase against human players, especially if it can determine the exact setup that the opponent is using. However, the endgame is still very weak. This is consistent with what others have found in existing literature. It would require either a larger search depth or a better evaluator to improve.

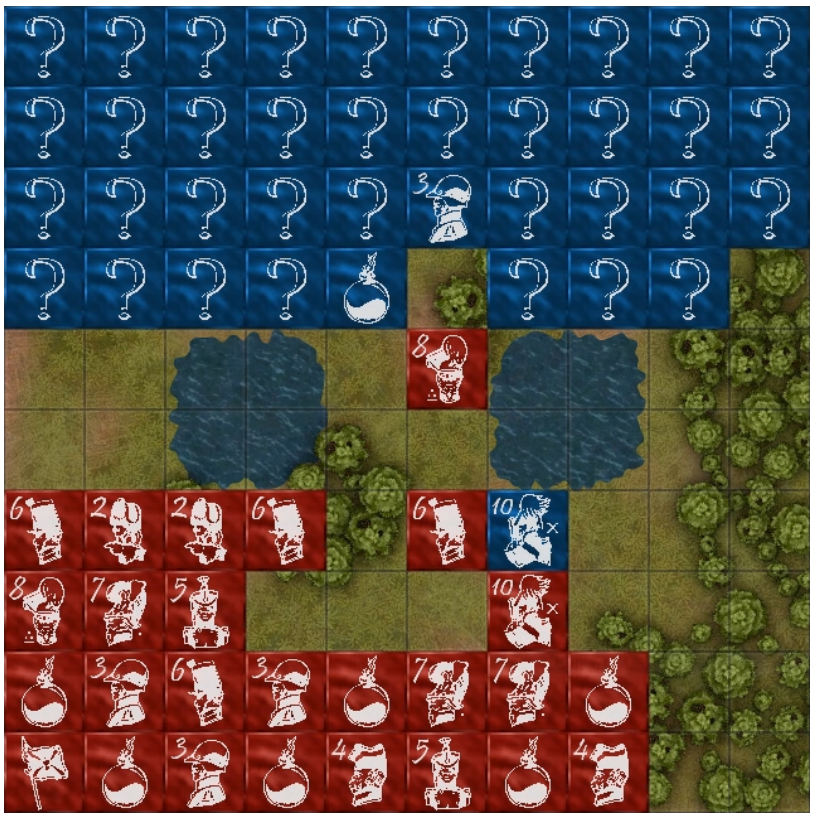

FIGURE 9.12: Gijs is forced to trade his Marshal against the AI's Marshal to prevent further captures.

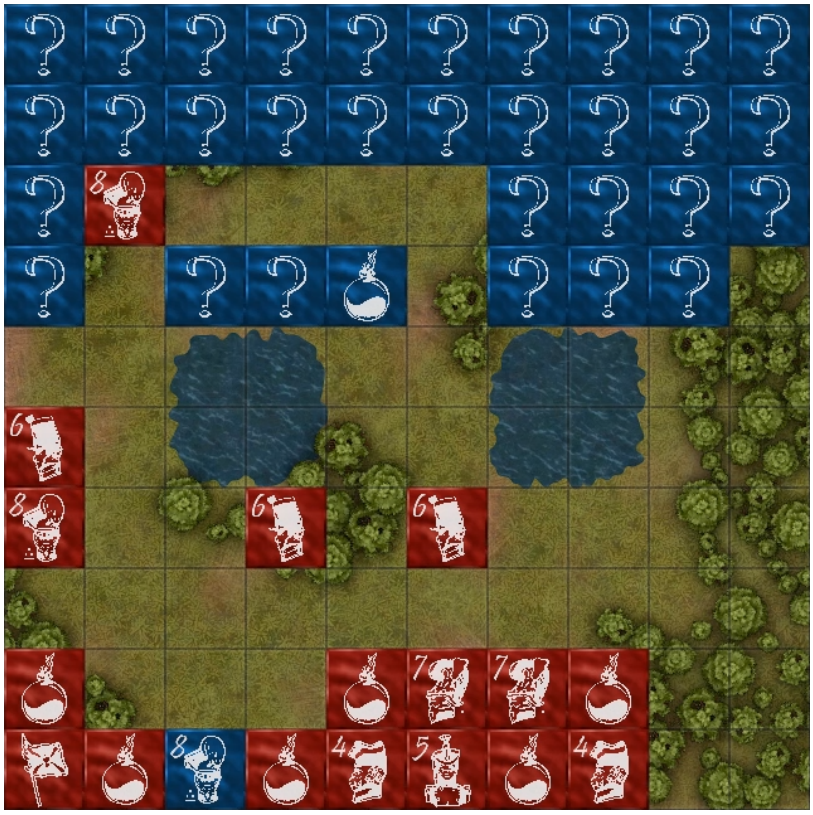

FIGURE 9.13: After letting the Blue Colonel pass, the AI captures Gijs' last three Miners.

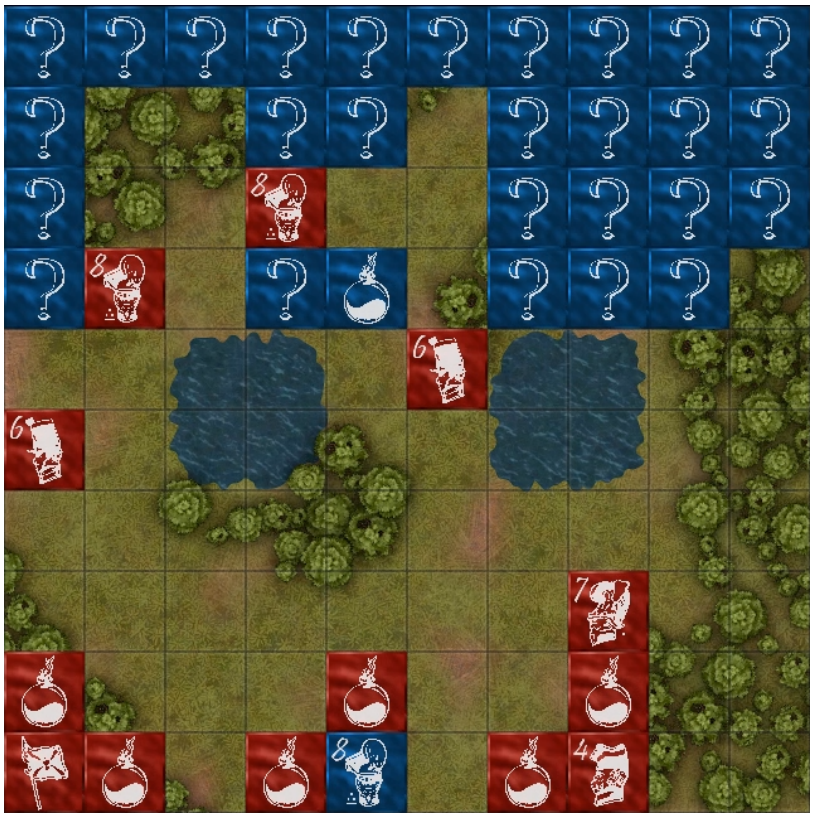

FIGURE 9.14: The AI's offensive loses steam, as its Colonel gets 'stuck' with no clear path to further offensive action.

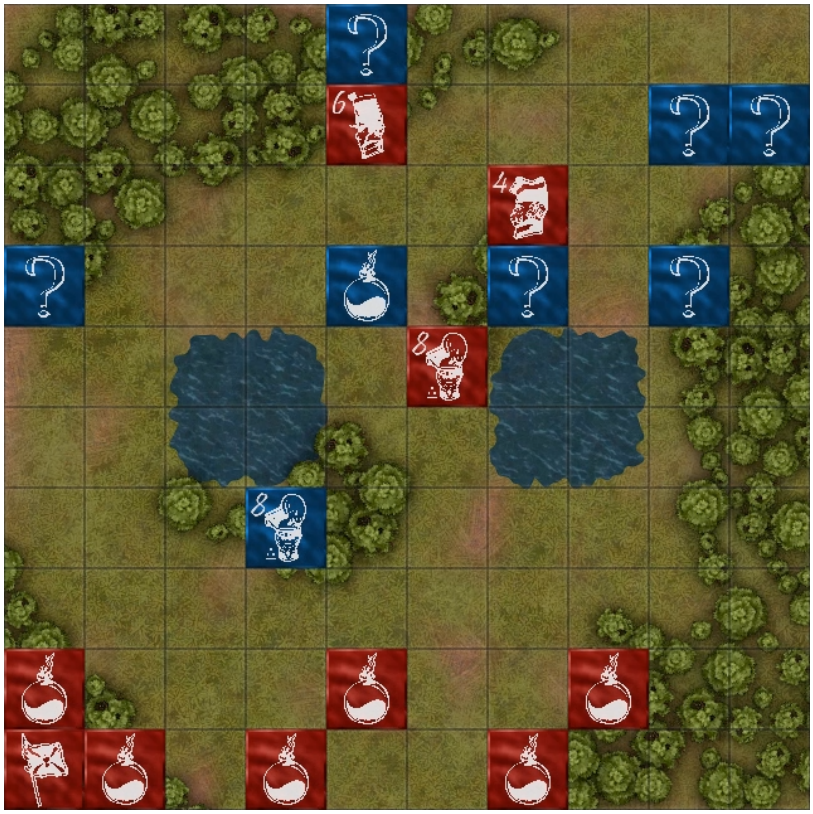

FIGURE 9.15: Gijs begins chasing the last Blue Colonel around.

FIGURE 9.16: The Custom Setup Gijs used for his first game.

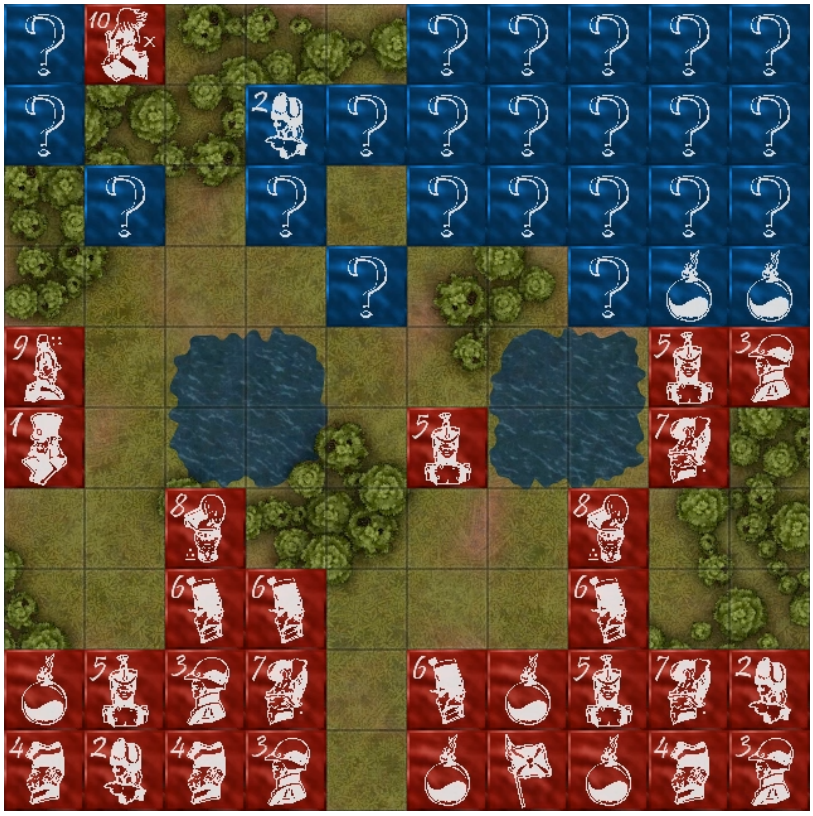

FIGURE 9.17: Gijs' offensive goes well and he manages to capture a large number of pieces, before losing his Marshal to the Blue Marshal on A10.

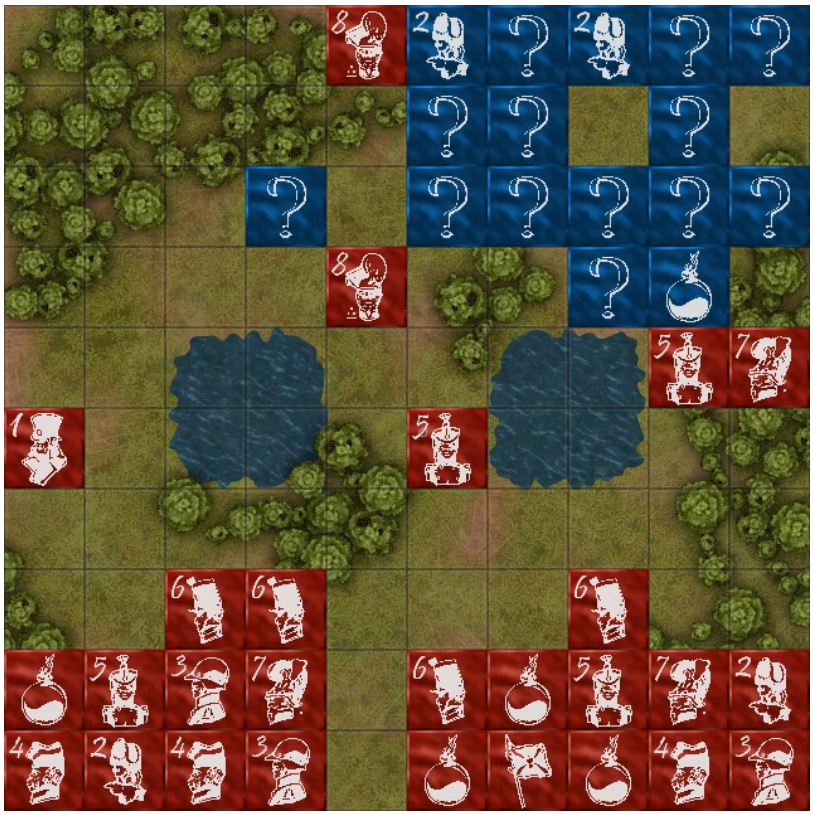

FIGURE 9.18: Gijs plays it safe and traps all Blue pieces in the top-right corner of the board, pinning them with his Colonels and a Major.

Chapter 10

Discussion

During the course of this thesis we have looked at creating an AI agent for Stratego by dividing Stratego up into four different subproblems, the *Setup Problem*, the *Information Problem*, the *Dynamic Evaluation Problem* and the *Static Evaluation Problem*. Each of these problems are difficult to solve and we discovered that the exact solutions found for each can impact the effectiveness of solutions to other problems as well.

For the *Setup Problem*, we found that in general picking a setup from the Gravon database [Jun15] using the Gravon Setup Provider (see 5.1.4) works fairly well. It seems clear that human players do have a decent sense of what makes a good setup. The neural network approach to creating new setups using the Naive RvH Setup Provider (see 5.2.2) also worked fairly well, and there are still potential improvements to be made there. This method is largely limited by the fact it selects the 'best' setup out of a number of random setups, meaning it still relies on a little luck that there is a good setup between them.

Therefore, the use of a Gravon Setup seems to work well, unless the opponent has decided to solve the *Information Problem* using the Database Estimator (see 7.2.4), which can often pinpoint the exact used setup within the first 20 moves as it did in the games played against humans. Having perfect information is incredibly powerful, as the agent quality for most agent types gets a very large boost, regardless of the exact details of the agent. Of course, this relies on the opponent using a Gravon setup, which is definitely not always a given. When this is not the case, it seems to be more efficient to use the Naive RvH Neural Network Estimator (see 7.2.5). This estimator does not assume that the opponent is using a setup that it has seen before. One limitation here however is that this was tested against setups that were generated using the same neural network that the estimator uses to determine what setup was used, which could have boosted the estimation quality for this estimator.

When it comes to the *Dynamic Evaluation Problem*, much seems to depend on the quality of the available estimations. The pUCT method (see 6.2.4) does not seem quite ready for proper usage yet, even though it shows great potential. If the agent can be expected to quickly obtain perfect information, the ε-greedy method (see 6.2.2) seems to do slightly better than the Upper-Confidence Bound method (see 6.2.3). However, if it may be difficult to make high-quality estimations, using the UCB method seems to be slightly more advantageous. This may simply be because the UCB method searches less deep than the ε-greedy method, which means that it searches more in the areas close to its own pieces, which are more likely to have some information revealed, whereas the ε-greedy method can search much deeper towards entirely undiscovered pieces. If these were estimated accurately, then ε-greedy can use this property to its advantage, but otherwise it can become a limiting factor.

Poor estimation quality can also be a limiting factor when it comes to solving the *Static Evaluation Problem*. Here we found that if the estimation quality is low, the NUC Without Flag Evaluator (see 8.1.5) does well whereas if estimation quality is high, Naive Unit Value Count Evaluator (see 8.1.6) can do slightly better than the regular Naive Unit Count Evaluator (see 8.1.1). Although the difference is not very large, it does show that very complex evaluation functions can become a risk factor if they rely on specific features of the board state. If those features are misidentified by the estimator, the evaluator can overestimate the desirability of a state, causing the agent to make mistakes. In situations where estimation quality was worse, we found that the NUC Evaluator is the most consistent performer.

The human testing has gone well, and an AI agent even managed to win one of the games and got close in two others. A weak endgame is its largest weakness, where the limits of the Monte-Carlo Tree Search become apparent as the agent is not able to search deep enough to find good moves. These limits could be overcome with a more directed search or a better evaluator.

In hindsight it was unfortunate that the Naive RvH Neural Network Estimator was not tested against a human player. It would be interesting to see if an agent using that method could do well. Instead, we found that the Direct Rank Estimator (see 7.2.6) performed poorly against a human player, although the poor setup also played a part.

Chapter 11

Conclusion

During this thesis we set out to find a good AI agent for Stratego. We considered various solutions to a number of different problems, either through the usage of neural networks or via more traditional means. This has paid off; we have seen that certain AI agents are capable of playing against human players and giving them a hard time beating them, although there is still much to improve.

The best setup providers appear to be the Gravon Setup Provider (see 5.1.4) and the Naive RvH Setup Provider (see 5.2.2), the latter best used if a new original setup is needed.

When it comes to making good estimations, the Database Estimator (see 7.2.4) is unrivalled, provided that the opponent is using a Gravon setup. If this is not the case, then the Naive RvH Neural Network Estimator (see 7.2.5) is a viable replacement that can provide fairly decent estimations.

As far as selection methods go, what is best depends on the quality of the estimations. If the estimation quality is high, then the ε-greedy method (see 6.2.2) is the superior option. However, if the estimation quality is expected to be lower, it is wise to select the Upper-Confidence Bound method (see 6.2.3) instead. The pUCT method is a promising candidate, but requires more work before it can really compete with the other methods.

For evaluating states, the best overall method is the Naive Unit Count Evaluator (see 8.1.1). If estimation quality is low, it may also be a good idea to try the NUC Without Flag Evaluator (see 8.1.5), as this is less prone to making mistakes when estimating that the Flag is in an incorrect position. If estimation quality is high however, the Naive Unit Value Count Evaluator (see 8.1.6) becomes a good alternative to the regular NUC Evaluator.

When playing against humans, AI agents still have much to improve upon before they can become truly competitive. Their weak endgame hampers their gameplay a lot and can often cause them to lose, despite doing very well in the early game. Nonetheless, the agents do pose at least some challenge to a human player, and can on occasion even win against one.

Chapter 12

Future Work

There is still plenty to improve when it comes to creating AI agents for Stratego. Although we have AI agents that can occasionally beat a human player, we would of course much prefer an agent that can consistently beat a human player. Already we have seen a number of shortcomings to the solutions presented to the four problems, all of which provide possible avenues of new research to improve these solutions and thus improve the AI agents using them. We will review the most important ones here.

When it comes to generating setups, we have introduced a way to generate completely original setups using a neural network. However, this works by generating random setups and rating them using the network, which means that the final setup may have some obvious flaws due to the random generation. This could be improved by using certain heuristics to make changes to randomly generated setups, or a new method that creates new setups that is perhaps based on known frequent structures of pieces, e.g. a placement of Bombs surrounding the Flag or the placement of the Marshal on one side of the board and the General on the other. Additionally, it could be interesting to see if a neural network can be made to distinguish between good and bad setups, instead of relying only on the assumption that what a human would do is necessarily good.

There are plenty of options to look at when improving the solutions to the Information Problem. Although our efforts have mostly been focused on setup reconstruction, it could be interesting to incorporate positional information into the method as well. The given methods for setup reconstruction could be easily improved too, for example by providing a larger database for the Database Estimator or creating a better fallback function for when a setup is not in the database. Additionally, the Naive RvH Neural Network Estimator could be improved by either using a better neural network or by providing a better method to generate setups that fit the given information.

In dealing with the Information Problem, we have decided to focus on the strategy of making single estimations in favour of chance nodes. That approach could potentially be improved, for example by no longer assuming that the opponent has perfect information. This assumption essentially eliminates the concept of bluffing, which is otherwise frequently used in Stratego. Chance nodes themselves may also be something to look into again to use in perhaps a more limited way, to prevent them from overcomplicating the structure of the search tree.

The selection methods for the MCTS algorithm work pretty well, and it might be difficult to improve them. However, there are variations of ε-greedy that could easily be tried out. Additionally, the pUCT method could still do with a lot of work, and probably holds the most potential. With better training data and potentially a better network architecture, this method could be good enough to overtake both Upper-Confidence Bound and ε-greedy in terms of agent quality, as it would enable

a far more directed search that focuses on the correct moves. It may also be possible to use more traditional methods to prune certain moves out of the tree, for example by focusing on pieces that are close to the last-moved piece, or on capture moves as suggested by Schadd and Winands [SW09].

Improving the evaluation functions could be very difficult, if the solutions for the other problems are not also improved. Nonetheless, it could be interesting to see what evaluation functions are optimal if perfect information is guaranteed. Perhaps it is possible to find out how pieces of different ranks should be valued compared to one another. The evaluators using neural networks are also potential points of improvement, as they should also be able to improve given better balanced training data and better defined labels. The input representation could also potentially be changed; instead of only one-hot encoding all the different ranks, it might be an idea to also encode if a square is empty with a dedicated neuron, instead of just keeping all other neurons deactivated. The input representation also does not take the two-squares rule into account, which in high-level Stratego is often used to trap pieces and then capture them.

Appendix A

Source Code

The source code for all agents, evaluators, estimators and setup providers are published online on Github: https://github.com/Moranic/StrAItego.

The source code is published under the GNU GPLv3 license. Permissions of this strong copyleft license are conditioned on making available complete source code of licensed works and modifications, which include larger works using a licensed work, under the same license. Copyright and license notices must be preserved. Contributors provide an express grant of patent rights.

Bibliography

[Alb03] R. Albarelli. *Optimizing Stratego Heuristics with Genetic Algorithms*. Dec. 2003.

[Art10] A.F.C. Arts. "Competitive Play in Stratego". Master's Thesis. Maastricht University, Mar. 2010.

[Bal82] B.W. Ballard. *The *-Minimax Search Procedure for Trees Containing Chance Nodes*. 1982.

[Boe07] V. de Boer. "Invincible, A Stratego Bot". Master's Thesis. Nov. 2007.

[Boe12] R.M. de Boer. "Reachable Level of Stratego Using Genetic Algorithms". Bachelor's Thesis. Utrecht University, Feb. 2012.

[Boo97] B. Boonstra. "Column: Programmer's Challenge". In: *MacTech* 13.11 (Nov. 1997). Peter N. Lewis winning solution to the Stratego Challenge. URL: http://preserve.mactech.com/articles/mactech/Vol.13/13.11/Nov97Challenge/index.html.

[Fed10] International Stratego Federation. *ISF Game Rules*. 2010.

[Ism04] M. Ismail. "Multi-agent Stratego". Bachelor's Thesis. University of Rotterdam, Aug. 2004.

[Jun15] T. Jungblut. *Gravon Database*. [Online; accessed 08-February-2021]. 2015. URL: https://www.gravon.de/english/stratego/strados2.php.

[Met08] J. Mets. *Monte Carlo Stratego*. June 2008.

[Moh09] J. Mohnen. *Using Domain-Dependent Knowledge in Stratego*. June 2009.

[PT09] J. Petkun and J. Tan. *Senior Design Project: Stratego*. 2009.

[RG18] S. Redeca and A. Groza. *Designing agents for the Stratego Game*. 2018.

[RVD20] Alex Regueiro, Michael Vivet, and Pavlo Datsiuk. *Hungarian Algorithm*. Nov. 2020. URL: https://github.com/vivet/HungarianAlgorithm.

[Sil+16] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 0028-0836. DOI: 10.1038/nature16961.

[Smi15] S. Smith. *Learning to Play Stratego with Convolutional Neural Networks*. 2015.

[Sta09] J.A. Stankiewicz. *Opponent Modeling in Stratego*. June 2009.

[SW09] M.P.D. Schadd and M.H.M. Winands. *Quiescence Search for Stratego*. 2009.

[Tre00] C. Treijtel. "Multi-agent Stratego". Master's Thesis. Delft University of Technology, Oct. 2000.

[Tun12] V. Tunru. "Feasibility of Applying a Genetic Algorithm to Playing Stratego". Bachelor's Thesis. Utrecht University, Feb. 2012.

[Ult] *Classic Piece Setups from Accolade*. Part of a strategy guide for Stratego by an unknown author. URL: https://www.ultraboardgames.com/stratego/accolade_setups.php.

[Wol18] H. Wolf. *Starting points for improvement in Stratego programming*. Sept. 2018.