April 2014

Improvements to MCTS Simulation Policies in Go

Christopher Paul Nota
Worcester Polytechnic Institute

Dale Joseph LaPlante
Worcester Polytechnic Institute

A Major Qualifying Project Report
ON

Improvements to MCTS Simulation Policies in Go

Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE

In Partial Fulfillment of the Requirement for the
Degree of Bachelor of Science

by

Dale LaPlante
Christopher Nota

UNDER THE GUIDANCE OF
Professor Gábor N. Sárközy

May 1, 2014

Abstract

Since its introduction in 2006, Monte-Carlo Tree Search has been a major breakthrough in computer Go. Performance of an MCTS engine is highly dependent on the quality of its simulations, though despite this, simulations remain one of the most poorly understand aspects of MCTS. In this paper, we explore in-depth the simulations policy of Pachi, an open-source computer Go agent. This research attempts to better understand how simulation policies affect the overall performance of MCTS, building on prior work in the field by doing so. Through this research we develop a deeper understanding of the underlying components in Pachi's simulation policy, which are common to many modern MCTS Go engines, and evaluate the metrics used to measure them.

Acknowledgements

We would like to thank Professor Gábor Sárközy for his counsel, teachings, and constant willingness to help, Levente Kocsis for his guidance and astonishing insight, the institution of SZTAKI and our colleagues there who have made us feel so very welcome, WPI for giving us this opportunity, and all those who have come before us for their profound research into Monte Carlo Tree Search and its application to Go.

Contents

List of Figures

Chapter 1

Introduction

Monte-Carlo Tree Search is a search algorithm that combines deep search with random simulations to determine an optimal action [6]. The algorithm has been hugely successful in complex problems, especially those with large branching factors [7]. The most noteworthy example is its success in the realm of computer Go [6]. Go is a board game that is noteworthy in its difficulty to computer players due to its large branching factor and tree depth. Prior to the development of modern MCTS methods, the best computer Go programs played at the level of a weak amateur. Modern MCTS based programs now play at a master level on small boards (9x9) and at a strong amateur level on standard sized boards (19x19) [4].

One of the critical components of MCTS is the random simulation (or "playout"). Despite the success of MCTS, and the fact that high-quality simulations are critical to the success of the algorithm, simulations remain poorly understood [9]. In this paper we attempt to further the understanding of MCTS simulations in the realm of Go, expanding upon and adding depth to past research in the field. We focus our research on the playout policy used by Pachi, a reasonably strong open-source MCTS Go engine developed primarily by Petr Baudis[4]. In particular we expand upon the work of Müller and Fernando in their 2013 paper analyzing playout subpolicies in Fuego, another open-source Go engine [9], and the work of Silver and Tesuaro in their 2009 paper on simulation balancing [2].

Initially, simulations would select moves completely randomly. It was quickly shown that encoding expert knowledge into the simulation policies in order to produce more realistic playouts produced a substantial improvement in the strength of MCTS [10], and it seemed that further improving

MCTS would be a simple matter of learning to encode more expert knowledge into the simulation policies in a fast, efficient manner. However, this task soon proved to be much more difficult than immediately thought, with many attempts to add expert knowledge actually *worsening* results. Counterintuitively it was shown by Gelly and Silver that increasing the strength of a playout policy as a stand-alone player *does not* necessarily produce and increase in the strength of MCTS[10]. In 2009, Silver and Tesuaro showed that it is more important that a playout policy is *balanced* than strong. Huang, Coulom, and Lin applied simulation balancing in a practical setting in 2010 [15]. Finally, in 2013, Müller and Fernando analyzed in detail the subpolicies in Fuego, and explored new ways to measure simulation policies [9].

Despite the work that has gone into attempted to understand these simulation policies, designing a quality simulation policy remains what some in the field have called a "dark art" [13]. In this paper, we continue the study on simulation polices. We expand Müller and Fernando's work by analyzing Pachi's playout policies in detail, and evaluate various metrics for their ability to measure the quality of playout polices. We then adapt machine learning algorithms presented by Silver and Tesuaro in attempt to improve Pachi's playout policy. In doing so we contribute a deeper understanding of the aspects comprising a strong playout polices, and work towards eliminating their status as a "dark art".

Chapter 2

Background

In this chapter, we give an overview of Monte-Carlo Tree Search, Go, simulations in Go, Pachi, and Pachi's simulation policy, Moggy.

2.1 Monte-Carlo Tree Search (MCTS)

The idea of using Monte-Carlo techniques for Go dates back to 1993[14], but the idea did not find much success until 2006, starting with Coulom's proposal of the Monte-Carlo Tree Search (MCTS) algorithm[12]. Traditional Monte-Carlo techniques involves repeatedly sampling a position using random simulations. MCTS expands on this idea by progressively building a search tree based on the results of these simulations.

2.1.1 Steps in MCTS

A Monte-Carlo Tree Search consists of four steps: selection, expansion, simulation, and backpropagation. The details may vary depending on the specifics of the algorithms being used, but in general each step works as described below.

Selection refers to the initial descent of the tree. Each pass starts at the root of the tree. A child node is then selected based on some criteria. Usually child nodes are selected based on how promising they are (how likely they are to result in a victory), and how few times they have been visited (how unexplored they are). The algorithm then repeats this process for the child node, selecting the next level-deeper child node using the same criteria.

This continues until the algorithm reaches a leaf node. At this point, the expansion step begins.

Expansion is the step taken once a leaf node is reached. It will create zero, one, or more child nodes. If it created child nodes, it will select a child, and then begin a simulation from there. If it did not create a child, it will begin a simulation from the leaf node.

Simulation - At this point, a random playout is performed from the node selected in the expansion phase. It will return either Win, Loss, or in some cases Draw. For example, in a chess match, moves are played randomly until a king is taken. In Go, moves are played until the board is filled and both players pass.

Backpropagation - The result of the simulation is then propagated backwards through each node that was passed during the descent. Each node has a count for number of visits and number of wins. If the simulation resulted in a win, both the visits and wins counts are incremented. If the simulation resulted in a loss, only visits is incremented.

2.1.2 Upper Confidence Bound 1 Applied to Trees (UCT)

Upper Confidence Bound 1 Applied to Trees (UCT) is a node selection algorithm introduced by Levente Kocsis and Csaba Szepesvri [1]. It resolves what is known as the exploration/exploitation tradeoff. The selection policy is affected by two conflicting ideas. On one hand, it should try to allocate more resources towards exploring nodes that are more promising (simulations passing through this node have resulted in victory more often). This is known as exploitation. On the other hand, it also needs to examine unexplored or poorly explored nodes in order to find moves that it may have missed initially, and to avoid search traps (nodes that initially appear good, but after a large number of simulations are realized to be). This is known as exploration. So, there is a tradeoff between exploration and exploitation. UCT resolves this tradeoff by applying an algorithm known as UCB1 to node selection.

2.1.3 UCB1

UCB1 (Upper Confidence Bound 1) is an algorithm designed to solve the multi-armed bandit problem. This problem is usually described by the example of a gambler in front of a series of slot machines (each of which is

referred to as an "arm"). He does not know anything about the machines, and they all may be different. The question is which machines to play, and in which order. The problem is defined in [1]:

> A bandit problem with K arms is defined by the sequence of random payoffs, $X_{i,t}, i = 1, ..., K, t \geq 1$, where each i is the index of a gambling machine (the "arm" of the bandit.). Successive plays of machine i yield the payoffs $X_{i,1}, X_{i,2}, ...$

So the exploration/exploitation tradeoff is as follows: How often should the gambler play the machine that has yielded the best results so far (exploitation), and how often should he try other machines (exploration)? UCB1 seeks to minimize the *regret*.

Definition 1. *Regret is the loss experienced from not always playing the best machine [1]. Exploration will often lead to regret because it is unlikely that any given unexplored machines is the best. It is defined at time n by the expected sum of the rewards from the best machine minus the expected sum of the rewards of the machines chosen by the policy. This is given by the equation below [1]:*

$$R_n = \max{}_i \mathbb{E}[\sum_{t=1}^{n} X_{i,t}] - \mathbb{E}[\sum_{j=1}^{k} \sum_{t=1}^{T_j(n))} X_{j,t}] \tag{2.1}$$

A policy will solve the exploration-exploitation problem if the growth rate of the regret is kept within a constant factor of the best possible growth rate.

UCB1 is applied to MCTS during the node selection phase. Namely, UCB1 treats node selection as a multi-armed bandit problem for each individual internal, explored node. Each arm corresponds to an action that can be taken at the given node, and each payoff corresponds to the cumulated, discounted reward for each of these paths. UCB1 will select the node that maximizes the following equation[1]:

$$Q_t(s, a, d) + c_{N_{s,d}(t), N_{s,a,d}(t)}$$

where $Q_t(s, a, d)$ represents the value estimated for action a in state s at depth d and time t, $N_{s,d}(t)$ represents how many times state s has been visited up to time t and depth d, and $N_{s,a,d}(t)$ is the number of time action a was performed while in state s up to time t and depth d. By treating

every node like this, UCT resolves the exploration-exploitation problem at the time of node selection, while also guaranteeing convergence as the number of simulations goes to infinity[1].

2.2 The Game of Go

The game of Go is relatively simple in terms of rules. It is fully observable by both players, deterministic, static, discrete, and adversarial. It is played by two players in alternating turns on a 19x19 board. Each turn, the turn player places one stone on an intersection of lines (called a "point"). The goal is to surround a greater total area of the board than the opponent. Additionally, when a group of one player's stones has no adjacent empty/friendly points (they are filled by the opponents stones or the edge of the board), these stones are removed from the board and are worth one point each at the end of the game. This is called a "capture". The game ends when both players pass, at which point the scoring phase begins.

Players may play at any point on the board, with two exceptions. The first is that under most rule sets, it is illegal to play a move that results in the immediate capture of the stone being played, without the opponent making move (called a "suicide"). The other exception is called "Ko". The Ko rule states that a player may not play in a way that reverts the game to a previous state (so no loops may occur). For example, if black plays at location **a** in Figure 2.1, he will take white's piece:

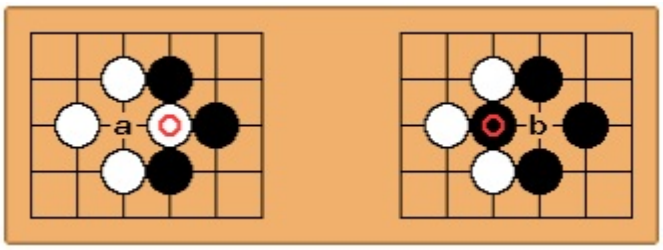

Figure 2.1: An example of a Ko situation [3].

Due to the rule of Ko, white is not allowed to then play at **b** because it would revert the game to a previous state (white would place a stone down directly where their previous one was captured, and in so doing capture blacks

stone that was just put down). White is, however, allowed to play elsewhere and then play at **b**, because at that point the state would have changed.

While these rules seem simple, they lead to very elaborate and complicated game play, and deep strategy. To see just how complicated the game can become, we compare Go against Chess [1]:

Metric	Chess	Go
Average Possible Moves	37	200
Average Length of Game	57 moves	300 moves
Possible Positions	10^{47}	10^{170}
Possible Playouts	10^{123}	10^{360}

Figure 2.2: Complexity of Go compared to Chess

This demonstrates the sheer branching factor of the game, however the intricacy of Go is difficult to see using just these numbers. As stated in the introduction, a Go player must be able to do two things well in order to be effective. They must be able to judge the current state of the game, meaning not only understanding the positions of each stone but also the general status of stones (such as how safe they are, what territory they affect, etc.). They must also have strategy, meaning they can interpret multiple future states and decide which course of action leads to the most desirable outcome. For humans, the concept of judgement and strategy usually takes on multiple meanings in a game of Go due to the size of the board. Human players often interpret different structures on the board both separately and together in order to develop strategies that can keep their stones alive while also obtaining the maximum amount of territory. It would be difficult to judge a single structure without knowing its context in the overall board, and likewise it would be difficult to create an overall strategy without first ensuring those smaller structures are actually alive. Humans are naturally adept at being able to interpret a number of varying parts of the board both separately and together, as well as figuring out what should and shouldn't be considered when thinking about strategy. Computers, on the other hand, have a difficult time doing so because they are limited by their programming. While it may be easy to code 3x3 structures into a Go algorithm, things become exponentially more complicated when looking at larger and larger portions of the board (as we can see from the chart above). Because of

this, it is very difficult for any realistic Go algorithm to judge the state of a game in its entirety, and therefore difficult for it to understand context. Instead, modern Go algorithms judge the board based on specific criteria it can look for (usually pertaining to single or very few groups at a time). It then develops its strategy by predicting (simulating) a future state in which it wins and playing the move that is most likely to get there.

Ranking

Ranking in Go is slightly different than in other games. In Go, the lower level ranks are referred to as "kyu" and the higher level ranks are called "dan". Beginners start somewhere between 30 an 20 kyu and work their way downward towards 1 kyu. When players progress beyond this level, they enter the expert ranking of dan. These ranks begin at 1 and progress up to 9. It is generally accepted that 1 through 7 dan are amateur expert rankings, while 8 and 9 dan are professionals[3].

2.3 Pachi

Pachi is an open source program designed to play Go using a MCTS engine. It currently holds a 7 dan rank on 9x9 boards (as measured by KGS) [4]. KGS is an online Go server that is used by players around the world (both machine and human) to test their skills against one another. Pachi is run on KGS to obtain its rank.

While Pachi can achieve the highest amateur dan ranking on 9x9 boards (7 dan), its capabilities drop when playing on 19x19 (a problem for all MCTS-based engines). For these 19x19 games, Pachi is ranked as 1 kyu when run on reasonable hardware, 2 dan when using higher end machinery, and 4 dan if run on large clusters [4]. Pachi's playout phase during MCTS is determined by its playout policy Moggy, which will be discussed in detail in later sections.

Pachi also provides a "simple, modular framework" for developing Go programs[4], which made it ideal for our purposes.

2.4 Playout Policies in Go

During the simulation phase of MCTS, a Go algorithm needs to be able to depict the playout resulting from the simulated move with some amount of

accuracy. If it cannot then the simulations results will be unreliable and the updated node will be unpredictably biased. In order to produce reliable simulations, a number of different heuristics are used with varying probabilities of being picked. These heuristics attempt to imitate normal behavior in a game of Go, such as capturing pieces, defending, expanding territory, etc.. The more accurate a policy is at predicting the resulting playout from a move, the more valuable each simulation will be.

2.4.1 Speed vs. Quality

The most basic tradeoff in a simulation policy is that of quality vs. speed. In the optimal case, a simulation policy could simply return the minimax value of a position, and the MCTS would converge almost immediately. Of course, if this were possible, there would be no need for MCTS to begin with. Nevertheless, the intuition follows that a policy that is accurate with respect to the minimax score is better than an inaccurate policy [citation needed]. However, in order to make a more accurate prediction, the policy must devote more computational resources to the simulation. Therefore, if a policy attempts to be too accurate, it may be too slow, and less nodes are explored. As a result, it is also very important for a policy to be able to be run quickly.

2.4.2 Balance vs. Strength

Early in the development of modern MCTS engines, it seemed obvious that a simulation could be improved simply by increasing the strength of the simulation policy as a stand-alone player. However, it was quickly discovered that this was not the case, and that in some cases there was even a negative correlation between the strength of the simulation policy and the strength of the MCTS using that policy [10]. It was speculated that policies needed a certain amount of randomness to be effective, and that policies that were too deterministic would perform poorly [9].

In 2009, Silver and others explored the idea of simulation balancing [2]. The goal is to ensure that the errors committed by a simulation policy cancel out. This is in contrast to optimizing the strength of a simulation policy, which attempts to simply minimize errors. They showed that a balanced policy performs better than a strong one [2].

Strength and balance can be defined more formally. $V^*(s_t)$ defines the minimax value of a given state, s_t at time-step t. Any non-optimal move that is made results in an error, δ_t:

$$\delta_t = V^*(s_{t+1}) - V^*(s_t) \tag{2.2}$$

A "strong" policy is one that has a low error, while a "balanced" policy is one that has a low *expected* error[2]. A strong policy will make few mistakes, while a balanced policy can make many mistakes, as long as they balance out in the minimax estimate is minimally affected.

Thus the strength J of policy p over positions ρ is given by[**?**]:

$$J(p) = \mathbb{E}_\rho[\mathbb{E}_p(\delta_t^2 | s_t = s)] \tag{2.3}$$

And the full imbalance $B_\infty(p)$ is given by (where z is the outcome of a simulation run by policy p):

$$B_\infty(p) = \mathbb{E}_\rho[(\mathbb{E}_p(z|s_t = s) - V^*(s_t))^2] \tag{2.4}$$

2.4.3 Accuracy vs. Stochasticity

Finally, it has been observed that there is a tradeoff between the accuracy of a simulation and its stochasticity [2]. Playouts with a high level of accuracy and low stochasticity will be biased towards observing certain moves multiple times, while not exploring other less-common moves. This trade-off can cause the playout policy to be incomplete, as it only observes a static, limited subset of possible playouts which may disallow variance/coverage.

2.5 Moggy

Moggy is the default file for handling the playout policy of Pachi. It has two main policies it uses to simulate playout, namely Seqchoose and Fullchoose (Seqchoose being the default). It also has a number of heuristics (also referred to as sub-policies) which are used by these two playout policies to obtain predictable moves. These two policies, along with their sub-policies/heuristics, are described in detail below. These descriptions are derived from in-depth analysis of the code used to implement them, meaning that the information here may not be applicable to all Go-playing engines.

2.5.1 Seqchoose

Seqchoose tries heuristics (each with a certain probability of being tried) sequentially. The heuristics are tried in this order: ko, Nakade, local Atari capture/defend, ladder, local groups with 2 liberties, local groups with 3 or 4 liberties, patterns, global Atari capture/defend, Joseki, or fill board (each heuristic's execution in Moggy is described in-depth below). If these all fail then Seqchoose defaults to random (returns pass). If a heuristic is tried and then successfully passes, Seqchoose will immediately return the resulting coordinate to be used in the next step of the playout (if multiple coordinates satisfy the given heuristic, one will be chosen randomly). The probability each heuristic has of being chosen is set by default, however they can be reset by passing in certain parameters to Pachi. The default probability each heuristic has of being tried is given in the chart (Figure 2.5.2) below.

2.5.2 Fullchoose

Fullchoose, unlike Seqchoose, attempts each heuristic and adds all successful moves to a list. It then calls Taggedchoose to pick from this list. Taggedchoose condenses the list by merging any entries that suggest the same coordinate (merging their tags as well). Tags are characters that represent where a move came from (from which heuristic(s)). Taggedchoose then constructs a probability distribution for each move based on the gamma values for their tag(s). Gamma values are what determine the probability of a certain move being chosen (the higher a tag's gamma value, the higher probability it lends to a move). The gamma values of each tag is set by default in Moggy, however they can be changed based on input. Tagged choose continues by adding up the total of all the probability distributions (plus Tenuki probability) and picking a random number between zero and the total. It then checks each move's probability distribution against this random number and, if it has a probability higher than the random number, that move will be chosen and returned. As the move queue is being checked, this random number decreases by the probability distribution of each move checked (in an attempt to converge). If Tenuki is enabled (enabled by default) and no move is chosen after the move queue is exhausted, a random move is played (returns pass). The gamma values for each heuristic are as follows:

Heuristic	Rate of Seqchoose	Gamma Val. for Fullchoose
Ko	20%	6
Nakade	20%	5.5
Local Atari	-1U*	5
Ladder	0%	4
2 Lib Check	-1U*	4
N Lib Check	20%	3.5
Pattern	100%	3
Global Atari	0%	2
Joseki	-1U*	1
Fillboard	0%	0

*-1U is defined based on the size of the board. On large boards (19x19) it is given the value of 80%, while on small boards (9x9) it is given a value of 90%.

Figure 2.3: Probabilities and gamma values of Moggy's sub-policies

2.5.3 Ko

Ko is checked for by looking up several values being stored in the board structure. First, the last coordinate on the board where a stone was taken (and therefore the last coordinate where ko was initiated) is looked up (this is called last_ko.coord) as well as the current coordinate ko is active at (called ko.coord). In order to proceed, last_ko.coord must be a valid coordinate and ko.coord must be empty. Moggy then checks how old last_ko.coord is and, if it's greater than the defined koage (four moves by default), returns unsuccessfully. If it is younger than the koage, the coordinate ko was found at will be tested for sensibility (no suicides). If this passes, last_ko.coord will be returned.

2.5.4 Nakade

Nakade checks the last move played to see if it is part of a structure that has a large space in the middle (see Figure 2.4). This is important because if three or more spaces are enclosed, then the defending player can create two "eyes" in the provided space (making the structure invincible). Some

of these structures can be played inside of (filling one or more of the empty spaces) before the defending player creates these eyes, and by doing so the attacker will have prevented the two eyes from being formed (see Figure 2.5). This is called Nakade. To check for Nakade, Moggy looks up the last move played and searches its neighbors to find an empty one. After one is found, the remaining empty neighbors are checked in relation to the first one. If any of these empty spots are not eight-adjacent to the first one, then this will return unsuccessfully. Next the neighbors of the empty spots are checked to ensure the attacker does not have any connected pieces and that there are at most 6 empty spots. After this is done the algorithm checks the surrounding pieces to determine the shape of the group. It then compares this shape and number of empty spots with known Nakade shapes. If any return successfully, this will return with the coordinate of where to play for Nakade.

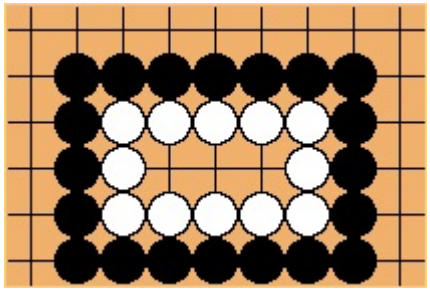

Figure 2.4: An example of a Nakade situation. If white plays in the very center, this structure will be invincible and black cannot capture it [3].

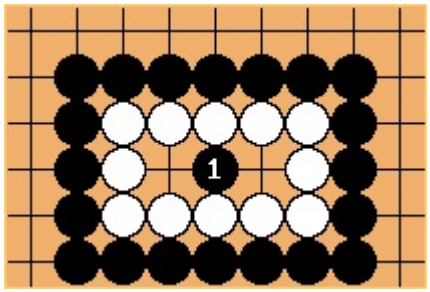

Figure 2.5: An example of a resolved Nakade situation. Because black played in the center (at **1**), there is no way for white to make this structure invincible anymore and it is considered dead [3].

2.5.5 Local Atari

This checks to see if the last play resulted in any neighboring groups being put in Atari. Atari simply means that a group has been reduced to one liberty and is in danger of being captured (see Figure 2.6). To do this, Moggy checks each neighbor of the last play and counts the liberties of any group found. For each group with only one liberty, it runs group_atari_check on it, which decides if the group is savable (savable meaning the defending player can play in such a way that the group lives). If the turn player is the attacker, group_atari_check looks at neighbors of the threatened group to see if there are any mutual Ataris. If there are, the group is captured (to immediately protect the turn player's pieces). If there are no mutual Ataris, the one liberty of the threatened group is checked for quality. If playing the liberty for the defending player does not result in additional liberties, the defending group is considered to be dead and there is no need to attack it. If the defending group can play on the liberty to gain additional liberties, then they will be captured in order to prevent escape and the liberty coordinate is returned by this heuristic. If the turn player is the defender, group_atari_check will return any coordinate that either counter-captures a neighbor or successfully increases the defending group's liberties. It will not play on a coordinate that results in a ladder (unless the ladder forces an Atari in the process), nor will it save a Ko unless it is being used for an eye.

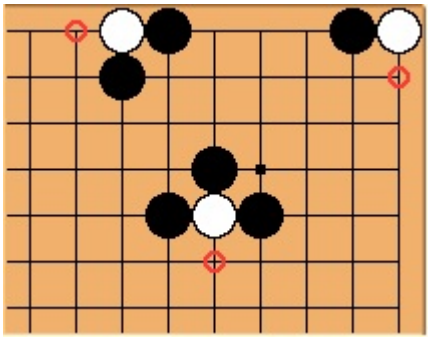

Figure 2.6: An example of Atari. All of white's pieces shown here are considered to be in Atari. Each of their one and only liberties is indicated by the red circles [3].

2.5.6 Ladder

A ladder occurs when a group has one liberty left in certain situations (see Figure 2.7). The situation has to be such that playing the liberty as the defender results in the group having two liberties. The attacker then responds to this by blocking the stone's path, reducing its liberties back down to one. If the structure is a true ladder, this cycle will repeat until a wall or another group is hit (see Figure 2.8). True ladders are considered to be dead shapes, and as such should not played out by the defender (as it only loses points). Ladders can be broken if there is a group owned by the defender in the ladder's path. Another way to break a ladder is to Atari one of the attacking stones through the process of laddering (often known as a false ladder). In Moggy, ladders are checked by first looking at the chase stone. This is the stone (or group) at the head of the group currently threatened, and will inevitably be threatened by the opponent through the process of laddering. If this stone is already threatened (has less than 3 liberties), the ladder is false and will not be played. After this, Moggy goes through a series of moves to play out the ladder. If at any point the attacker becomes Atari'd or the defending group gains more than two liberties, the ladder is breakable and will not be played. If it is played out successfully then this will return the coordinate to play in order to initiate the ladder.

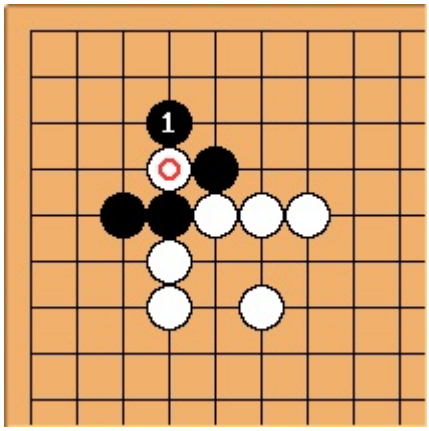

Figure 2.7: An example of a ladder situation. White's stone in Atari (designated with the circle) is considered to be dead [3].

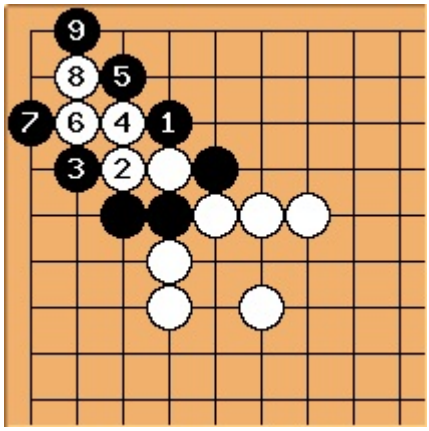

Figure 2.8: An example of a ladder that has been played out. As one can see, no matter how much white tried to gain more liberties, black was able to stop them. The order of the moves is indicated by the stones' numbers [3].

2.5.7 2 Lib Check

When the last play resulted in a neighboring group (or the stone itself) having only two liberties, this heuristic checks what to do about it. The first

step is checking the initial condition, which is done by observing the last play's liberties as well as each of its neighbors' liberties. Any groups with two liberties then have the qualities of those two liberties checked. If the group can use either of those liberties to run (gain 3+ liberties) or connect to another group owned by the same player, the function returns immediately because the group is considered to be safe. If the turn player is the attacker (does not own the defending group), each of the two liberties is checked to see if Atari is feasible. Atari is not feasible if playing it puts the attacker in Atari and there is no nearby group (with two or more liberties) that can be used to protect the Atari position first. If it is feasible then each position is checked to see which causes less clutter for the attacker, and any position not playable by the defender is taken out of consideration. The best resulting Atari position(s) will have its coordinate(s) returned by this function. If the turn player owns the defending group, they will check if a counter-Atari is possible by playing in one of their two liberties and return the coordinate if so.

2.5.8 N Lib Check

It is important to check the safety of local groups, even if they seem to have a good number of liberties. If the last move resulted in a neighboring (eight-adjacent) group having 3 or 4 liberties, the liberties of that group will be checked for counter-Ataris. That is, if there is an enemy group nearby with only two liberties that can be attacked by extending the defending group, that coordinate(s) will be returned by this heuristic. Otherwise the defending group is left alone. Note that the number of liberties the defending group needs in order to be checked is variable. While the default value is 3-4, it can be changed by the passed-in parameters (which is why this is called N Lib Check). The tradeoff between high and low values for this is that high values give a good chance of not missing threats while not taking one's own liberties, however results in spending more time on this heuristic and possibly wasting moves on live groups.

2.5.9 Pattern

The playout policy Moggy has a predefined database of 3x3 patterns that it uses for pattern recognition. When the pattern sub-policy is called, it checks each spot in this 3x3 pattern with the area around where the last

piece was played. If the area is identical to the pattern, the pattern subpolicy returns the coordinate that is generally accepted as the proper move in said pattern. Some positions in the pattern allow for variable pieces to occupy them, meaning that there may be more than one scenario that a certain pattern can be applied to.

2.5.10 Global Atari

The board structure in Pachi naturally holds a list of groups with only one liberty. When this heuristic is called, if that list is empty, it returns immediately. Otherwise global Atari checks all Ataris on the board using group_atari_check (see Local Atari). If the boolean capcheckall is true, group Atari will construct a list of all feasible Atari moves (whether capturing or defending) and return them. If capcheckall is false (the default case), this will only return the first feasible Atari move found. Capcheckall can be toggled with parameters passed into Pachi.

2.5.11 Joseki

Joseki is a common term used for sequences of play that result in a fair outcome for both players. These are generally used in the beginning of the game when the board is relatively empty and there is little influence from outside sources. Moggy has a Joseki library for 19x19 boards that it uses by default, however users can pass in custom Joseki libraries or even choose to use none. Pachi does not have a default Joseki library for 9x9 boards.

2.5.12 Fill Board

This heuristic attempts to fill empty areas on the board for the purpose of gaining territory and creating structures that can be used by other heuristics. It is not used by default due to poor performance. When it is used, the heuristic retrieves the number of free spots on the board and divides by eight. It then compares this number with fillboardtries (default 0), and if it is higher than fillboardtries, fillboardtries is used instead. It then picks a random free coordinate from the board and sees if all of its eight-adjacent spots are free. If they are, the coordinate is returned immediately. If not, it randomly chooses another free coordinate and tries again. The number of

times it attempts to find a free spot is defined by the number determined at the start of this algorithm.

Chapter 3

Analysis of Playout Policies in Go

Quality playout policies play a crucial role in the success of Monte-Carlo Tree Search. While the search tree aspect of MCTS is well understood, the same cannot be said of the playouts [9]. This paper first attempts to analyze these simulations in the game of Go, specifically using the Pachi framework, in order to establish a better understanding of the factors that influence the effectiveness of a playout policy.

3.1 Methodology

All experiments were run using Pachi version 10.00 with modifications as noted.

3.1.1 Notation

For playout policy p, we define MCTS(p) to be a Monte-Carlo Tree Search using p to simulate games at leaf nodes. M refers to Moggy Seqchoose, which is the default policy used in Pachi, with its default settings. R refers to a random simulation policy.

In order to get a good spread of simulation policies, we took an approach similar to Müller and Fernando's treatment of Fuego playout subpolicies[9]. We divided the set of policies into two groups: subtractive policies and additive policies. Subtractive policies use Moggy Seqchoose but disable one

Subtractive Policies		Additive Policies	
Command	Description	Command	Description
fullchoose=0	M	Random	R
fullchoose=1	Use Fullchoose	patternrate=100	R + pattern
atarirate=0	M - Atari	lcapturerate=100	R + lcapture
josekirate=0	M-Joseki	nlibrate=100	R + nlib
korate=0	M-ko	ladderrate=100	R + ladder
ladderrate=100	M + ladder	korate=100	R + ko
lcapturerate=0	M - lcapture	nakaderate=100	R + Nakade
nakaderate=0	M-Nakade	josekirate=100	R + Joseki
nlibrate=0	M - nlib	atarirate=100	R + Atari
patternrate=0	M - pattern	fillboardtries=1000	R + fillboard
fillboardtries=1000	M + Fillboard(1000)	capturerate=100	R + Global Atari
capturerate=100	M + Global Atari		

Figure 3.1: Policies variations used in analysis. Commands for Additive Policies abbreviated.

subpolicy. Additive policies use a random policy, but enable one Moggy subpolicy. We also tested Moggy Fullchoose, and the enabling of Moggy's ladder subpolicy, both of which are included under subtractive policies, as they are more similar to M than to R.

For more information about Moggy and its subpolicies, see Section 2.5.

3.1.2 Strength of MCTS(p)

Ultimately, the test of a playout policy is its actual performance when used in MCTS, so the most basic and accurate test of a playout policy p is to measure the playing strength of MCTS(p). For each policy we did this by continuously playing MCTS(p) against MCTS(M). For each game, we ignored the clock and ran 2000 simulations per turn. The sides alternated colors each game, and a 6.5 Komi was used. Pondering was disabled as it was found to unfairly advantage certain policies.

3.1.3 Playout Subpolicy Time Measurements

As discussed in Section 2.4.1, there is a tradeoff between the "quality" of a simulation policy and the speed at which it runs. The strength of a playout

policy is correlated to the amount of time it takes to run a single playout. This value is inversely proportional to the number of simulations that it is possible to run. Increasing the number of simulations increases the depth of the constructed search tree, enabling more accurate results.

We used Google's gperftools to measure the time that was being spent in each subpolicy. This utility routinely samples the stack in order to determine which subroutines the code is currently executing. Therefore, probabilistically, functions which take more time will be sampled from more frequently.

3.1.4 Position Database

In order to gather a database of realistic positions to run tests on, we turned to game records gathered from the KGS Go server. KGS is one of the most popular online Go servers. We gathered the records from Ulrich Goertz's SGF Game Records [11], which is an archive of all high level games played on KGS. These games all include one player who is at least 6 dan. We then narrowed this down by removing games that had handicaps or were lost on time.

Handicap games were removed because they would give bad information on the strength of certain positions compared to results, because a worse player with a big handicap may be in a strong position early on, but still lose due to their poor skill. Thus, a program might misinterpret the strong position as a losing position. Further, handicap games would also often include weaker players, even in the kyu range, which would poison attempts to build an expert policy based on moves by strong players.

Games lost on time were removed because the winner of these games would be determined by time-out and not by board position. Hence, a player could be far ahead and still lose. This would introduce misleading data.

3.1.5 Moggy as a Move Predictor

To evaluate the strength of playout policy p, we compared the action a selected by policy p to the action selected by expert policy, $\mu(s, a^*)$, which produces action a^* in state s. $\mu(s, a^*)$ was defined by the actions selected by players in our database of games.

Because $\mu(s, a^*)$ represents a strong player, the error δ_t (see Equation 2.2) of $\mu(s, a^*)$ is expected to be low. Moves selected by p that match the move selected by the expert policy have the same low error. Therefore, the

more closely policy p resembles $\mu(s, a^*)$, the lower the average error, and the higher the strength.

While it therefore seems to be desirable to have a playout policy p that closely resembles $\mu(s, a^*)$, there are a number of variables that can make the results misleading. In any given situation there may be a number of different "strong" moves to play (moves with a low error δ_t). Because μ only selects ones move for each state, p may select an equally strong move, but still be "penalized" for it by this metric. Another thing to consider is that in order to maintain a diverse sample, playout policies tend to be inherently stochastic. Therefore, a playout policy may fail to select $a*$ simply due to chance. While this makes most playout policies somewhat poor oracles, stochasticity has been shown to improve the overall effectiveness as a playout policy (see section 2.4.3).

We tested roughly 75 thousand games and about 15 million moves from our database in order to collect a good representation of Moggy's predictive capabilities. The test was performed on 23 different policies, and the predictions of each sub-policy as well as the overall policy were recorded.

3.1.6 Error of Playout vs Game Results

While comparing the moves selected by p is an estimate of the strength of p, Though stochasticity and generating diverse samples is important, the outcome of a good playout policy is expected to be somewhat accurate in evaluating positions (see Section 2.4.3). Thus, a position that wins a majority of playouts should hopefully result in a win for the actual player. While there are no known samples of perfect play (with the possible exception of certain endgames), strong players should present a reasonably strong and unbiased estimate of the outcome of a position (or the minimax score). Therefore, we define the Mean Squared Error of a playout policy vs. the game result as:

$$\frac{1}{P}\sum_{\rho}\frac{1}{N}\sum_{n=0}^{N}(r(\rho)-s_n(\rho))^2, \tag{3.1}$$

where P is the total number of positions, ρ is a given position, N is the total number of simulations per position, $s_n(\rho)$ is the result of running a simulation policy for the nth time on p, and $r(p)$ is the actual game result of position ρ, as determined by the high-level players in the position database.

This is also a measure of the *balance* of a playout policy (see Section 2.4.2). A balanced policy is expected to preserve the minimax winner of a position. Results from actual high level play are believed to be our closest estimates of the minimax score in each position. Therefore, results from high level player are used as our estimate of $V^*(s)$.

3.1.7 Error of Playout vs Pachi Score

Another estimate of the minimax score is given by the Pachi score of a position, $\mathcal{P}(s)$ (Pachi's estimated win percentage for the current player in state s). The Pachi score is simply the UCT score of the root node for the given state. This has a number of benefits over comparison to game results–it allows playout policies to be analyzed in positions that have not been encountered in real play, it will be subject to less noise, and a UCT-based expert will converge to the minimax result given enough simulations [1] (though in practice this number is too large to be achieved). The Mean Squared Error compared to the Pachi score is defined similarly as above:

$$\frac{1}{P}\sum_{\rho}[\mathcal{P}(\rho) - \frac{1}{N}\sum_{n=0}^{N} s_n(\rho)]^2, \tag{3.2}$$

where P is the total number of positions, ρ is a given position, N is the total number of simulations per position, $s_n(\rho)$ is the result of running a simulation policy for the nth time on ρ, and $\mathcal{P}(\rho)$ is the Pachi score of position ρ.

This is essentially the same as the equation for the *full imbalance* (given by Equation 2.4), with $\mathcal{P}(\rho)$ being used as an estimate for $V^*(s)$.

3.2 Results

3.2.1 Strength of MCTS(p)

The results of the strength test are shown in Figure 3.2.1. These results are used in later analysis.

Disabling certain subpolicies, namely Local Capture and Pattern, had a very negative effect on the overall strength of MCTS(p), suggesting that they are very important to the success of M. Other subpolicies seemed to have little to no effect on the outcome. Fullchoose performed surprisingly

poorly. It uses the same subpolicies as Seqchoose, so better results would were expected. We had believed that Fullchoose's primary weakness was its execution time compared to Seqchoose, but these results suggest that there are further problems.

Enabling the Ladder subpolicy did increase the performance of Seqchoose slightly when a set number of simulations was used, which more or less meets expectations. However, as seen below in Figure 3.3, it is considerably time consuming, explaining the fact that it is not enabled by default.

Policy p	Wins	Games	Win Rate
Josekirate=0	114	218	52.29%
ladderrate=100	103	193	52.02%
nlibrate=0	111	219	50.68%
fillboardtries=1000	108	218	49.54%
korate=0	107	220	48.63%
nakaderate=0	90	219	41.09%
atarirate=0	86	220	39.09%
capturerate=100	81	218	37.15%
fullchoose=1	37	197	18.78%
patternrate=0	20	218	9.17%
lcapturerate=0	0	220	0.00%

Figure 3.2: Win Rates of MCTS(p) of modified policies against MCTS(M), with 2000 simulations per turn.

3.2.2 Playout sub-policy Time Measurement

Figure 3.3 shows the time spent in Fullchoose subpolicies. Figure 3.4 shows the time spent in Seqchoose subpolicies. Fullchoose runs each policy once for each move, so it was the best option for comparing the time it takes for each subpolicy to run once. The ladder check took by far the most time, taking around 5 times as long as the next slowest policy. apply_pattern was the second slowest, but was still considerably faster than the ladder check. The other policies took little time by comparison, with local_atari_check taking a mere 1% of the time.

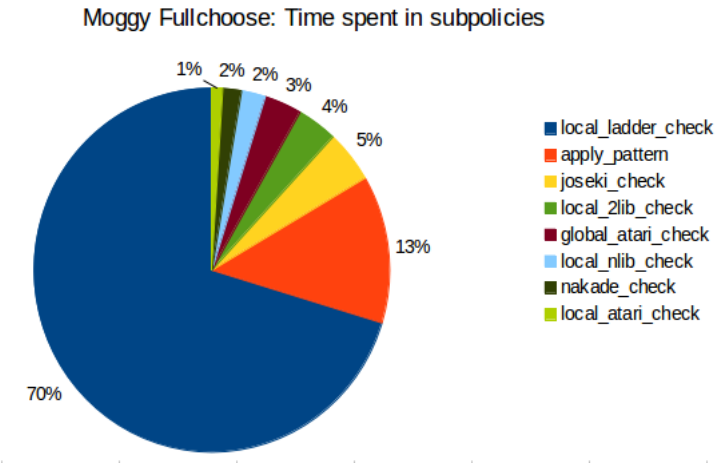

Figure 3.3: Time distribution of subpolicies in Fullchoose

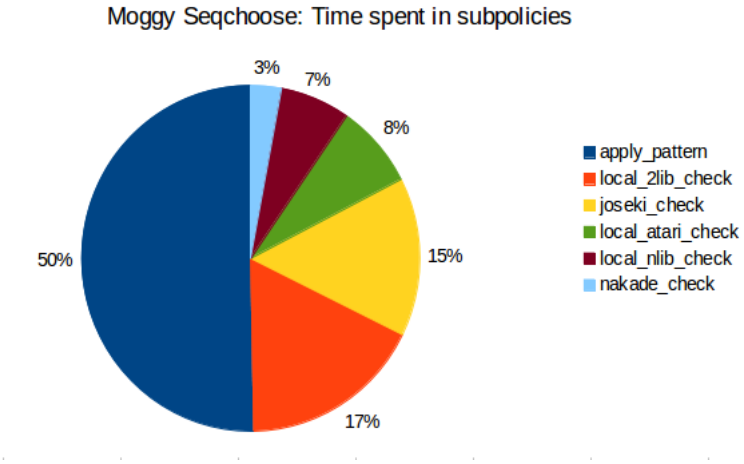

Figure 3.4: Time distribution of subpolicies in Seqchoose

3.2.3 Moggy as a Move Predictor

The results show that Moggy's default settings yield an approximate 20-21% success rate at predicting the move selected by expert policy $\mu(s, a*)$ at any point in a game. When varying the input, it can be seen that the majority of the predictions (and thus most moves selected by the playout policy) come from Moggy's 3x3 pattern recognition (Figure 3.7). Pattern appears to have only about a 19% success rate, however, at predicting moves. While this may seem low considering the subpolicy plays moves that are supposed to represent well-known expert patterns, it is important to understand the context-dependency of the patterns in real play. A 3x3 section only covers about one fortieth of the overall board, meaning most of the board is not being considered. Furthermore, multiple patterns may be present on different parts of the board. While 19% is not very high, given the circumstances that the algorithm is used under, it is also not surprising.

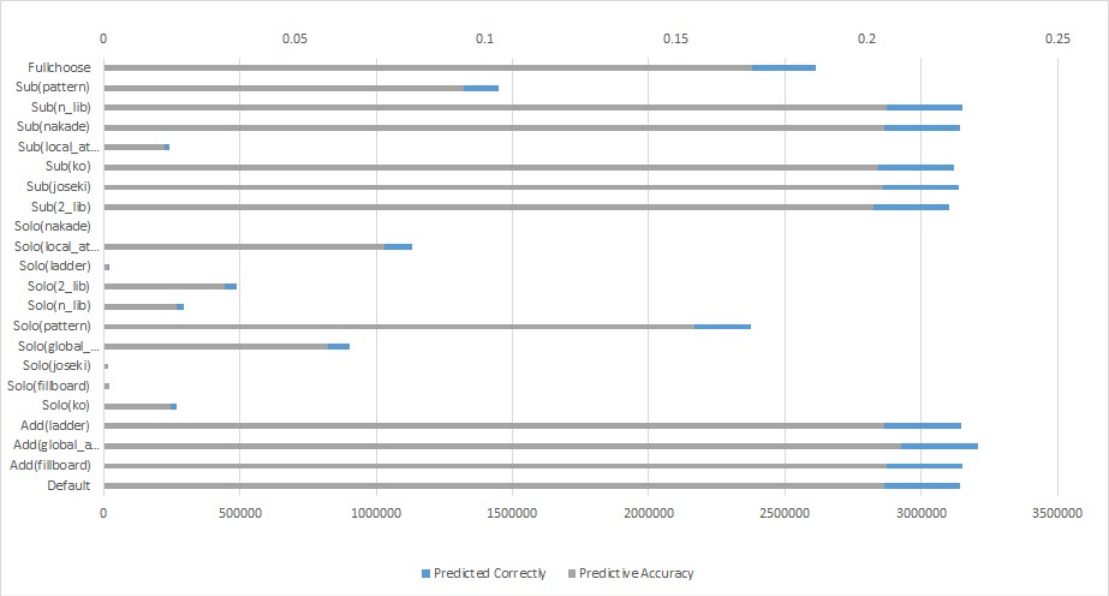

Figure 3.5: Predictive rates per policy. Sub means that Seqchoose was run without the listed subpolicy. Add means that Seqchoose was run with the listed subpolicy included (done for subpolicies not included by default). Solo means that the policy solely consists of the listed subpolicy.

The subpolicy with the second highest influence is local_atari_check (which captures/defends Ataris), which has moderate influence (though small compared to pattern) and a very high success rate (about 60%). This is unsurprising because Ataris are usually obvious plays in real games (so long as capturing or defending the Atari is sensible). Furthermore, they are often *urgent*, meaning that is important to play the move before the opponent can prevent it. Because the subpolicy checks the quality of surrounding liberties and potential liberties before defending a group, it will not try to save a dead group and therefore has highly accurate prediction when defending (though this is not the case in small-point scenarios when the number of points gained by defending is outweighed by larger potential point sources on the board). Similar to defending, the subpolicy will also not consider playing on dead group when attacking (if there is a group on the board that can be captured but has no chance of surviving, Moggy will wait to capture it). This is very good at predicting because it avoids unnecessary moves while still capturing when appropriate. Similar to defending, however, there are some cases where it may be inappropriate to leave a group living, even if they seem to be dead already. It is not uncommon for one group on the board to travel across a space and come within range of another one that's in danger. If this is the case, it is important for Moggy to reassess the group it determined to be dead earlier, because the quality of its potential liberties may have increased dramatically. As it stands, local capture/defending of Ataris does not take this into account (because it only looks at groups around the last stone played), however global Atari does (it will reassess all Atarid groups on the board). This may give reason to include a rate higher than 0 as the default for global Atari capture.

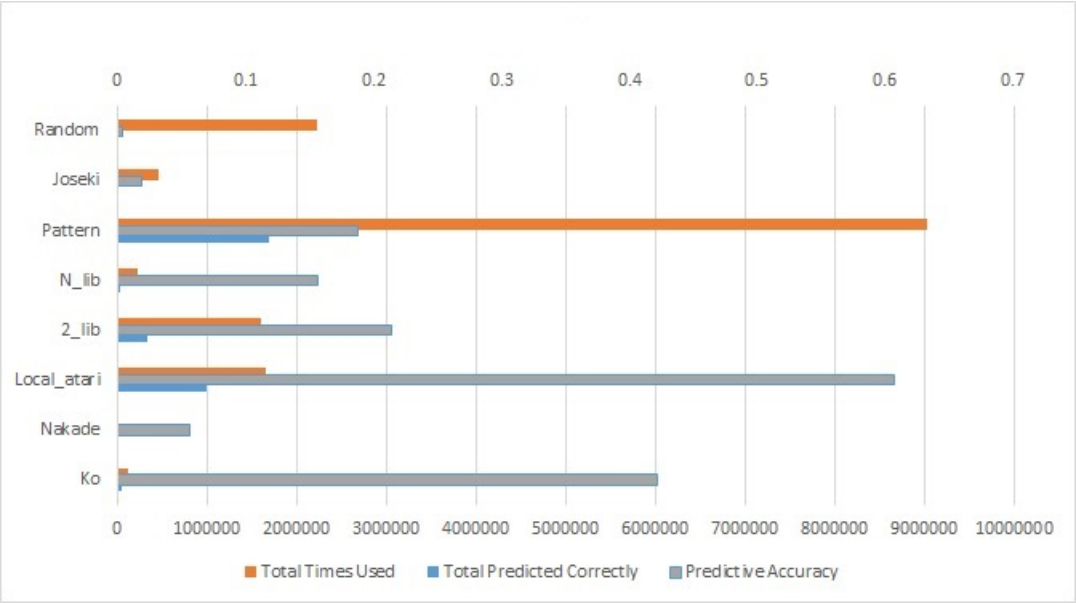

Figure 3.6: Predictive rates per subpolicy. Note these are only the subpolicies used by Seqchoose. These do not include fill board, global Atari, or ladder.

Ko also had a good success rate, around 40% (and 55% in Fullchoose), though it was not used often due to the relative rarity of Kos. Nakade had very poor success rate (near 0%), which is somewhat unclear due to how strict Moggy is at checking Nakade. It is not influential however due to the number of times performed. Making/preventing Ataris (2 Lib) was moderately influential and about 20% successful. This was as expected as making/preventing Atari is common, though situational for successful Atari. Defending a 3 or 4 liberty structure (N Lib) had about 15% success rate and wasn't very influential Joseki proved to be one of the worst predictors, as it had near 0% success rate and minor influence (about as much as Ko or N lib). This seems to be a little strange, due to Joseki relying on pattern recognition similar to the Pattern sub-policy. However, there are also often a large number of patterns that match in a given move, that it does not differentiate between. Ladder, Global Atari, and Fill Board were not used by Moggy's default policy, though we observed their results in our modifications. Ladder plays were successful around 30% of the time, though were rarely used and therefore not influential. Global Atari had about 5-10% success rate

and were moderately influential (though this could be increased by raising its priority in Seqchoose). Finally, the Fill Board sub-policy had moderate influence, though was not successful (success rate near 0%).

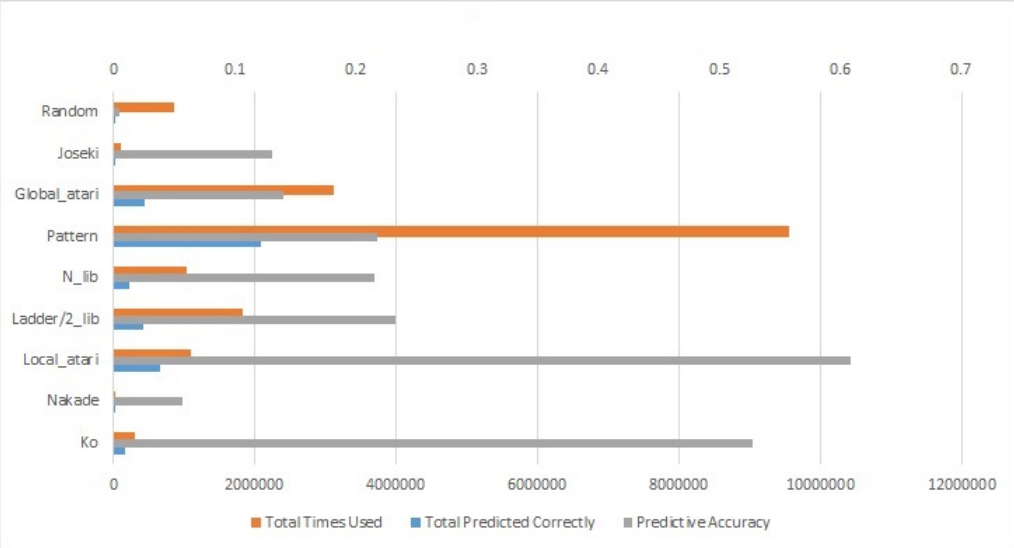

Figure 3.7: Predictive rates per subpolicy. Note these are only the subpolicies used by Fullchoose. These do not include fill board. Additionally local_atari and ladder are counted together due to Fullchoose's method of storing these tags.

3.2.4 Error of Playout vs Game Results

The error rate of the playouts was very high overall. This is somewhat expected, since the database of positions included many games that were close. When the minimax score $V^*(s)$ in terms of point differential is small, the playouts are expected to have varying results.

Figure 3.10 shows the error versus the playing strength of MCTS(p). The correlation is not very strong. Given that balance has been shown to be an effective measure for improving the quality of a playout policy by Silver and Tesuaro [2], a stronger correlation was expected. Increasing the number of data points to encompass more varied error rates may reveal a stronger correlation.

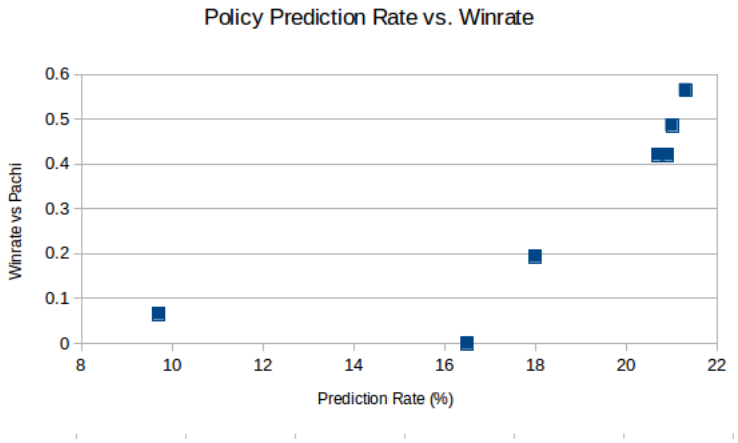

Figure 3.8: Policy Prediction Rate vs. Winrate

Subtractive Policies		Additive Policies	
Command	MSE ($\%^2$)	Command	MSE ($\%^2$)
capturerate=100	43.97	patternrate=100	45.25
fillboardtries=1000	44.10	lcapturerate=100	46.06
ladderrate=100	44.16	capturerate=100	46.08
nakaderate=0	44.17	atarirate=100	46.68
nlibrate=0	44.21	nlibrate=100	47.04
korate=0	44.24	ladderrate=100	47.18
fullchoose=0	44.25	fillboardtries=1000	47.23
fullchoose=1	44.36	josekirate=100	47.24
atarirate=0	44.39	korate=100	47.25
lcapturerate=0	45.50	nakaderate=100	47.31
patternrate=0	45.54		

Figure 3.9: Mean squared error of Playouts vs Actual Results for each policy variation.

Figure ?? shows how each policy variation performed. Interestingly, many of the subtractive policies actually outperformed the default policy. The top three policies were the addition of Global Atari, Fill board, and Ladder to the default Seqchoose policy. However, setting each of nakaderate, nlibrate, and korate to 0 also improved results, which is somewhat surprising.

Similar to the prediction results, local captures and pattern moves proved vital to performance on the metric. In fact, using Seqchoose with *only* patterns outperformed using Seqchoose with everything *except* patterns, and local capture was nearly as important. This is likely related to the fact that they are both used very often, and applicable in a wide variety of situations, while most of the other policies are used much more infrequently. This perhaps shows that it is more important to have expert knowledge that can be applied generally and frequently than detailed but infrequently used expert knowledge.

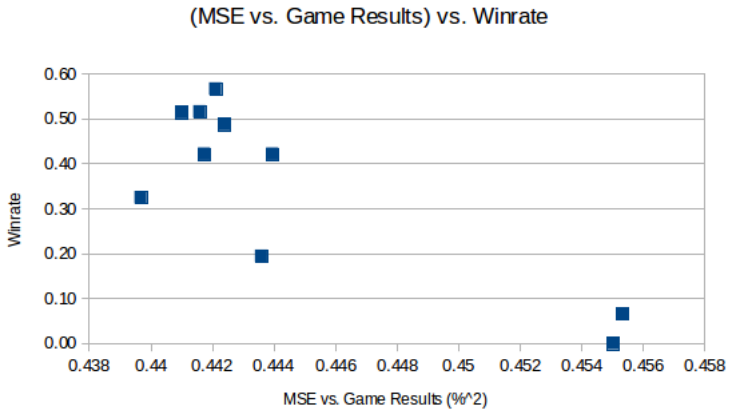

Figure 3.10: (MSE vs Game Result) vs Winrate

3.2.5 Error of Playout vs. Pachi Score

The measured error versus the Pachi score correlated remarkably well with the error vs. actual results (Figure 3.13). This suggests that the Pachi score is a sufficient estimation of minimax value $V^*(s)$. This is a very useful result, as it verifies the methodology used by Silver and Tesuaro[2] and others in simulation balancing. It is impressive given the low number of simulations used in this test (5000 simulations).

Subtractive Policies		Additive Policies	
Command	MSE ($10^{-4}\%^2$)	Command	MSE ($10^{-4}\%^2$)
nakaderate=0	31.08	patternrate=100	56.25
korate=0	31.72	lcapturerate=100	80.01
nlibrate=0	31.93	atarirate=100	97.10
ladderrate=100	32.23	capturerate=100	117.12
atarirate=0	33.41	nlibrate=100	121.47
fillboardtries=1000	34.65	ladderrate=100	126.67
capturerate=100	35.39	fillboardtries=1000	128.12
fullchoose=1	42.32	josekirate=100	128.19
lcapturerate=0	59.30	korate=100	129.41
patternrate=0	60.28	nakaderate=100	131.12

Figure 3.11: Mean squared of Playout vs Pachi Score for each policy variation.

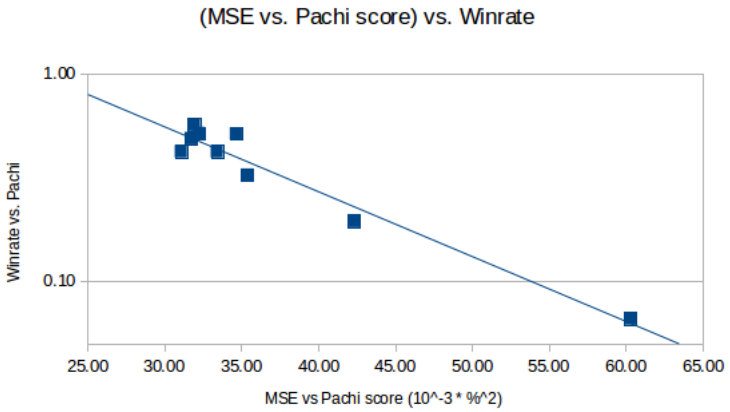

Figure 3.12: MSE (Policy vs Pachi score) vs Winrate. Winrate shown on logarithmic scale.

Interestingly, the correlation with the win rate for this value seems stronger. This may be due to a slightly better spread of data points. More data would still be useful for establishing a tighter correlation. However, the correlation seems strong enough to be a candidate for machine learning.

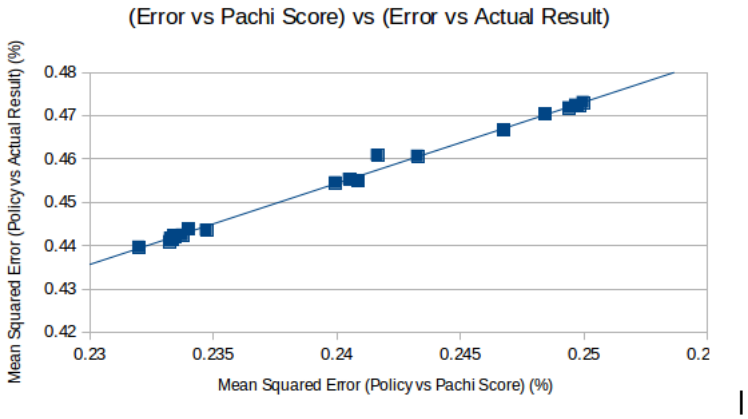

Figure 3.13: MSE of Playout Results vs. the Pachi score and vs. actual game results

3.3 Conclusions

After testing various characteristics of multiples policies within Pachi, we had gathered a large amount of information regarding Pachi's playouts. It is clear that the "pattern" and "local capture" subpolicies are the most valuable Moggy subpolicies. Removing them hurt the win rate, expert prediction, and the error vs. both actual game results and the Pachi score. Interestingly, the strength of the local capture module contrasts strongly with Fuego, where it was shown that disabling its corresponding capture module actually *increases* the strength of Fuego slightly[9]. However, in Fuego, the capture atari and defend atari modules are separate, whereas in Pachi they are combined in local_capture. Fuego was shown to be significantly weakened by removing "defend atari," which suggests that the strength of local_capture is more in defending than capturing. The pattern module was also shown to be very important in Fuego.

In contrast to the pattern and local capture modules, some of the subpolicies had surprisingly little effect. For example, disabling the nlib (low liberties) module resulted in a roughly 50 percent win rate against the default. Similarly, disabling the Ko and Joseki modules also had little effect.

As was expected, the ladder subpolicy took an exceptional amount of time (70% of all playout policy time), proving its infeasibility. Upon further inspection of ladder's implementation in Pachi, this time consumption appears to be unavoidable without losing a significant portion of the subpolicy's safety measures. As stated in 2.5.6, ladders can be very risky shapes if handled incorrectly. In order to ensure there is no chance for escape, the subpolicy must run down every possible path the ladder can take, resulting in high time cost, while only improving the policy slightly. The other subpolicy with significant time consumption was the pattern module. Pachi uses a somewhat brute force approach to matching each pattern. Despite this, it still took considerably less time than the ladder subpolicy while contributing much more to the success of MCTS in all metrics considered.

Both the prediction of expert moves and the measured error vs. Pachi score showed reasonable correlation with the playing strength of MCTS(p), while also being relatively inexpensive to compute. For this reason, they both make good candidates for machine learning. Furthermore, the error vs. the Pachi score showed to correlated very well with the error vs. actual results. If the results of games played by high level players are assumed to be reasonable estimates of the minimax score (0 or 1), then this suggests that the Pachi score is an acceptable estimate of $V^*(s)$ for the purpose of simulation balancing (see equation 2.4). The application of these metrics to machine learning is explored and tested in Chapter 4.

Chapter 4

Improvements to Moggy

In this chapter we use the metrics used in the previous chapter to perform machine learning on the weights in Moggy Fullchoose. Silver and Tesuaro [2] applied machine learning techniques Go in an artificial setting, using these techniques to learn weights for 2x2 patterns on 5x5 and 6x6 boards. We apply these techniques to learn weights for Moggy Fullchoose in a 19x19 setting.

4.1 Methodology

4.1.1 Policy Gradient

The Moggy Fullchoose playout policy is described by

$$\pi_\theta(s,a) = \frac{\prod_i w_i^{\phi(s,a)}}{\sum_b \prod_i w_i^{\phi(s,b)}} \tag{4.1}$$

where $\pi_\theta(s,a)$ is the probability of selecting move a at state s. θ is a vector containing weights w_i corresponding to each Moggy subpolicy, and $\psi(s,a)$ is vector describing which subpolicies suggest move a in state s. We will later need the gradient of the policy with respect to the subpolicy weights:

$$\nabla_\theta(\pi_\theta(s,a)) = \frac{\nabla_\theta(\prod_i w_i^{\phi(s,a)})}{\sum_b \prod_i w_i^{\phi(s,b)}} - \frac{\prod_i w_i^{\phi(s,a)} \nabla_\theta(\sum_b \prod_i w_i^{\phi(s,b)})}{[\sum_b \prod_i w_i^{\phi(s,b)}]^2}$$

$$= \psi(s,a) = \frac{1}{w_\theta}\pi_\theta(s,a)[\phi(s,a) - \sum_b \pi_\theta(s,b)\phi(s,b)] \tag{4.2}$$

4.1.2 Apprenticeship Learning

Apprenticeship Learning seeks to maximize the playing strength of a policy by learning strong moves from expert examples. In other words, the goal is to create a policy that behaves as closely as possible to a given expert policy $\mu(s, a^*)$[2].

We ran Fullchoose on half of our database of 80,000 games, constructing a move queue for each move (about 8,500,000). For each move, we then extracted each of the following pieces of information: each move present in Fullchoose's move queue, all of the policies used to suggest any given move in the queue, and the actual move chosen by the player in the given situation.

A simple algorithm for gradient ascent is provided by Silver and Tesuaro[2] for maximizing the likelihood $\mathcal{L}(\theta)$ of choosing action a^* given by the expert policy at state s. For each action/state pair, we updated the weights according to:

$$\Delta\theta = \alpha\psi(s, a^*) \tag{4.3}$$

The derivation is as follows[2]:

$$\mathcal{L}(\theta) = \prod_{t=1}^{L} \pi(s, a^*)$$

$$\log \mathcal{L}(\theta) = \sum_{t=1}^{L} \log \pi(s, a^*)$$

$$\nabla_\theta \log \mathcal{L}(\theta) = \sum_{t=1}^{L} \nabla_\theta \log \pi(s, a^*)$$

$$= \sum_{t=1}^{L} \psi(s, a^*)$$

4.1.3 Simulation Balancing

To minimize the full imbalance B_∞ of Moggy Fullchoose, we use the gradient descent algorithm presented by Silver and Tesuaro [2]. This algorithm learns from $\hat{v}^*(s)$, which is an estimation of the minimax value function generated by a deep Monte-Carlo Tree Search. In this case, we use Pachi with 5000 simulations to construct $\hat{v}^*(s)$, as the error vs. this value was shown to correlate very well with the error vs. real game outcomes (see 3.2.5).

Silver and Tesuaro's Policy Gradient Simulation Balancing [2]

For a given state s, the gradient used for gradient descent is the product of two terms: the *bias*, $b(s)$, and the *policy gradient*, $g(s)$. The bias given by $b(s)$ is the direction that the mean outcome needs to be adjusted in order to match the minimax estimate $\hat{v}^*(s)$. For example, is black winning too often, or not often enough? The policy gradient $g(s)$ tells us which policy parameters to modify in order to adjust the mean outcome. Formally, where z is the outcome of the simulation policy π_θ:

$$b(s) = \hat{v}^*(s) - \mathbb{E}_{\pi_\theta}[z|s] \tag{4.4}$$

$$g(s) = \nabla_\theta \mathbb{E}_{\pi_\theta}[z|s] \tag{4.5}$$

Therefore it follows that the full imbalance B_∞ and the gradient of the full imbalance are as follows:

$$B_\infty(\theta) = \mathbb{E}_\rho[b(s)^2] \tag{4.6}$$

$$\nabla_\theta B_\infty(\theta) = \mathbb{E}_\rho[b^2] = -2\mathbb{E}[b(s)g(s)] \tag{4.7}$$

In practice, it is easy to get a good estimate $\hat{b}(s)$ for the bias as well as a good estimate $\hat{g}(s)$ for the policy gradient by sampling simulations. Because $\hat{b}(s)$ and $\hat{g}(s)$ are usually correlated, it is necessary to sample them separately.

To get $\hat{b}(s)$, we sample M simulations from state s:

$$\hat{b}(s) = \hat{v}^*(s) - 1/M \sum_{\xi \in \chi_M(s)} z(\xi) \tag{4.8}$$

To get $\hat{g}(s)$, we sample an additional N simulations (where T is the length of simulation ξ, and $\psi(s,a)$ is defined in Equation 4.2):

$$\hat{g}(s) = \sum_{\xi \in \chi_N(s)} \frac{z(\xi)}{NT} \sum_{t=1}^{T} \psi(s_t, a_t) \tag{4.9}$$

Finally, for each state s in the training set, we update our policy weights according to:

$$\Delta\theta = \alpha \hat{b}(s) \hat{g}(s) \tag{4.10}$$

Where α is the learning rate.

4.2 Results

Subpolicy	Hand-Tuned	Apprenticeship	Simulation Balancing
Ko	6.0	10.15	1.99
Nakade	5.5	1.34	0.04
Local Atari	5.0	23.96	5.45
l2lib	4.0	2.34	2.59
lnlib	3.5	1.94	1.23
pattern	3.0	3.21	6.22
Global Atari	2.0	0.81	5.36
Joseki	1.0	0.10	0.33

Figure 4.1: Learned Moggy Fullchoose weights from hand-tuning, apprenticeship learning, and simulation balancing.

4.2.1 Apprenticeship Learning

After training Fullchoose on 40,000 games to attain the proper weights, we passed these values back into Pachi and tested it on the remaining 40,000 games. The predictive capabilities of Fullchoose increased by about 2.5% (as shown in 4.2.1, which is significant relative to the previous predictive value of Fullchoose. This brings it on-par with Seqchoose for predictability.

Predictive Accuracy of Fullchoose	
Hand-tuned Weights	17.00%
Learned Weights	19.62%

Figure 4.2: Predictive capability of Fullchoose hand-tuned default weights and weights learned by apprenticeship learning.

4.2.2 Simulation Balancing

The subpolicy weights learned through simulations balancing (Figure 4.1) seem to agree with the results of our earlier experiments (see section 3.2.5). The most weight was given to the Pattern, Local Atari and Global Atari subpolicies. Pattern and Local Atari both showed to be very important to balance (error vs. Pachi score) in Section 3.2.5, and that is reflected here. In the same section, setting the Nakade rate to zero had shown the highest balance, which is reflected here by a very low Nakade rate.

Though the learned weights differ significantly from the original weights, the overall improvement at the task was convincing but limited (Figure 4.4). This may be due to a limited amount of "expressiveness" by the weights in Fullchoose. The learned weights may not be near the global optimum because the balance is non-convex [2], however it is possible that the best achievable results by modifying the weights are not significantly better regardless.

There was an increase in balance, but this improvement did not translate to an improved win rate. While significant, the balance of the simulation policies is not the only factor in the success of MCTS. The simulation balancing algorithm used attempts to minimize the expected error overall all positions, giving them even weight, but it may be that balance is very important in some situations, and less important in others. Thus, the balance should be maximized for those situations more than the position. It was also raised as a possibility by Müller and Fernando that different phases of the game might require different policies in order to achieve balance[9]. In either case, maximizing the balance in the manner used may not produce the optimal results.

While the error vs the Pachi score was shown to correlate very well with the error vs. expert game results (Figure 3.13), learning based on the Pachi score remains prone to certain biases. For example, MCTS engines are known to frequently handle Nakade situations incorrectly [13]. The low weight given

	Mean Squared Error vs. Pachi Score ($\%^2$)
Hand-Tuned Weights	$3.530 \times 10^{-}3$
Learned Weights	$3.116 \times 10^{-}3$

Figure 4.3: Mean Squared Error vs Pachi Score of learned simulation balancing weights and default weights

	Games Won	Games Played	Win rate
Apprenticeship Learning	79	133	59.40%
Simulation Balancing	103	197	52.28%

Figure 4.4: Win rate of learned policies against hand-tuned default policy

to Nakade may say more about the Pachi score than the simulation itself. It is not clear otherwise biases in Pachi may have affected these weights.

4.2.3 Playing Strength of Learned Policies

The policy learned through apprenticeship learning showed a strong improvement in strength over the hand-tuned weights, winning nearly 60% of the games. Simulation balancing won a more modest 52% percent in strength, still showing a slight improvement. This suggests that in practice both techniques are viable and perhaps superior method of tuning the weights to hand-tuning.

4.3 Conclusions

Overall the results suggest that using machine learning to learn weights simulation policies is a viable and perhaps superior alternative to setting weights by hand, even in cases where the policy is not particularly expressive with respect to the weights. The machine learned weights are competitive with hand-tunes weights, even when they are very different. Surprisingly, and in contrast to Silver and Tesuaro's results[**?**], apprenticeship learning actually outperformed simulation balancing. Both learning techniques produced policies that were more effective at their respective tasks than the default hand-tuned weights, but the improvement was relatively small. Despite this, there was still a measurable increase in playing strength.

One issue preventing the improvement from being larger was the lack of "expressiveness" achievable through modifying the Fullchoose weights. The Pachi source code claims that the average move queue length in Fullchoose is just 1.4 moves [16]. This means that for many moves in the simulation, the relative weights of the subpolicies are not important. Therefore, modifying the weights fails to have a strong impact on the total playing strength. Silver and Tesuaro [2], and later Huang, Coulom and Lin [15] in their works on simulation balancing used policies considerably more expressive with respect to the weights, and therefore found much better more noteworthy results.

Chapter 5

Summary

In this project, we contributed to a deeper understanding of the individual subpolicies that comprise Pachi's simulation policy, examining why they were effective and in which ways they were ineffective, building on Müller and Fernando's work in Fuego[9]. We analyzed metrics used to measure these simulation policies, using them as tools for improvement, while also developing an understanding of their weaknesses and limitations. We demonstrated machine learning techniques in MCTS and showed that they were competitive with hand-tuning, and that they increased the accuracy of the simulation policy. This research will be valuable in future work on improving simulation policies in both MCTS Go engines and MCTS in general.

Chapter 6

Future Work

Our analysis of Pachi's playout policy and its many components have led us to find several areas for improvement. While we attempted to enhance Moggy ourselves with this data, there is still much more work to be done. Some of the possible areas to follow up on this research are:

Understanding Local Errors: In this paper we measured the global error of a playout policy, and balanced Moggy Fullchoose based upon it. However little attempt has been made to systematically measure local errors. It is difficult to understand why a policy misevaluates the board as a whole, and thus hard to make improvements or add subpolicies to correct for this misevaluation. However, local errors are much easier to understand and thus correct for. Furthermore, there has been little study into how improving a playout policies performance in certain local situations can hurt the simulation as a whole. If global error really is best understood as the sum of local errors, it seems logical that improving performance in a local situations that hurts the simulation as a whole must also hurt performance in a number of other local situations.

Information Sharing Between Playouts and Search Tree: Some minor attempts have been made to share additional information from playouts with the outer search by adding additional terms to RAVE. For example, Pachi examines the probability of each point being controlled by each player at the end of the game, and gives additionally bias towards points that are less well decided[4]. However, we have not discovered any attempts to share information in the opposite direction. For example, the deep search may discover the answer to certain Semeai ("capturing race") positions that are too complicated for a simulation policy. Simulations in other parts of the

tree may have the position being decided randomly, thus causing moves to be misevaluated. It would be useful to have a way to pass such information into the playout policies somehow.

More Expressive Weights for Fullchoose: Can Fullchoose be modified to allow for more expressive subpolicy weights? For example, could the pattern module be expanded and parameterized with a Softmax policy or something similar? Would this allow for the possibility of a more significant increase in playing strength through machine learning?

Pachi as a move tutor for Moggy: Using an expert policy constructed from game records meant that the strength of Moggy was being measured on a small number of moves, and in a binary fashion. A more accurate estimate of the error may be to use Pachi to determine how worse the move selected by Moggy is from the optimal move (according to itself). Müller and Fernando created a data set containing positions annotated by Fuego[9], that perhaps would be a strong candidate for performing machine learning on.

Improvements to Fullchoose: While Seqchoose has been the clear favorite of Pachi's policies so far, it is still reasonable to expect Fullchoose to be able to outperform it. Fullchoose is capable of considering multiple purposes of a single move (due to a single move being able to have multiple tags), whereas Seqchoose only chooses a move based on a single criteria at a time. The game of Go often requires the player to consider these multi-purpose aspects of a move so that they may be as efficient as possible. Our research has shown that it is in possible to improve upon Fullchoose with its current framework (we did so through changing the weights of each tag). We would suggest future researchers look at undesirable characteristics for moves (ie heuristics that would assign tags with negative values) as a place to start.

Subpolicy optimization: The subpolicies *ladder* and *pattern* currently take the most time to perform. Ladder takes so much time that it isn't even included in the default policy, due to ineffectiveness. Future work could look at optimizing these subpolicies (prioritizing the optimization of ladder).

Additional domain knowledge: Currently Moggy has no understanding of a Ko fight. It uses a very basic test when deciding whether to perform Ko, however does not consider any currently active (ie unplayable) Ko spots. The concept behind *2 lib check* could be utilized to implement Ko fights. If the board object kept track of which groups had only two liberties (similar to how it keeps track of all atari'd groups), it could compare the territory of

each of these groups against the one currently in Ko. At that point, deciding which group to attack would be trivial (the one with the highest territory would be chosen). This could also be expanded to looking at non-two liberty groups as well, so long as they initiate a Ko fight (demand a response).

There is a lack of understanding in Pachi when it comes to complicated fights. This can result in it failing to find a move necessary for capturing a group if it is at the beginning of a specific chain of events. This is due to the semi-randomness of Moggy. Even after a large number of simulations, it is still unlikely that Moggy could foresee a chain of events unless each of them corresponds to a high-ranking heuristic. One way this could be implemented is through thermography. We conducted some preliminary research on implementing thermography in Pachi based on how contested a given spot is as well as how valuable it is). The thermograph was successful in finding these hotspots with a some amount of accuracy, while also producing false positives. By using a thermograph, it may be possible to isolate an area of the board where an order-dependent fight is about to take place. With the size of the problem drastically reduced, it may then be feasible to integrate Tsumego solvers into MCTS.

Pachi naturally keeps dead shapes alive for the purpose of efficiency (capturing a dead group is usually a wasted move). While this is desirable in most cases, it should not be overlooked that many dead groups can in fact be saved. This often happens when a live group is close enough to connect to the dead group, thus giving it life. If Pachi doesn't recognize that there is a dead group in the live group's path, it may allow the opponent to connect. Likewise Pachi doesn't consider helping a dead group if it has a live group nearby, which could be helpful for either giving the group life or for attaining the initiative. The heuristic *global atari* may be used for re-assessing dead groups on the board in case one of their liberties become life-saving.

Bibliography

[1] Kocsis, L., & Szepesvari, C. (2006). Bandit based Monte-Carlo planning. 15th European Conference on Machine Learning.

[2] D. Silver and G. Tesauro. Monte-Carlo simulation balancing. In A. Danyluk, L. Bottou, and M. Littman, editors, ICML, volume 382, pages 945952. ACM, 2009.

[3] n.pag. Sensei's Library. Web. 11 Apr 2014. http://senseis.xmp.net.

[4] Baudis, Petr, and Jean-loup Gailly. "Pachi: Software for the Board Game of Go / Weiqi / Baduk." Pachi. Pachi, 12 Aug 2012. Web. 11 Apr 2014.

[5] Good , Irving John. "The Mystery of Go." New Scientist. 21 Jan 1965: n. page. Web. 11 Apr. 2014.

[6] Browne, C. (n.d.). About. Retrieved from http://mcts.ai/about/index.html

[7] Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., et al. (2012). A Survey of Monte Carlo Tree Search Methods. : IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES.

[8] Müller, M., Berlekamp, E., and Spight, B.. Generalized Thermography: Algorithms, Implementation, and Application to Go Endgames. Diss. Berkeley University, privately published, 1996. Print.

[9] Müller, Martin, and Sumudu Fernando. Analyzing Simulations in Monte Carlo Tree Search for the Game of Go. Edmonton, Canada: University of Alberta, 2013. Print.

[10] Gelly, S., & Silver, D. (2007). Combining online and offine learning in UCT. 17th International Confer-ence on Machine Learning(pp. 273-280).

[11] Grtz, U. (n.d.). Game records in SGF format. . Retrieved April 1, 2014, from http://www.u-go.net/gamerecords/

[12] Rmi Coulom (2007). "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 2931, 2006. Revised Papers. H. Jaap van den Herik, Paolo Ciancarini, H. H. L. M. Donkers (eds.). Springer. pp. 7283. doi:10.1.1.81.6817 Check —doi= value (help). ISBN 978-3-540-75537-1.

[13] Chaslot, G., Chatriot, L., Fiter, C., Gelly, S., Hoock, J., Perez, J., et al. (). Combining expert, oine, transient and online knowledge in Monte-Carlo exploration. : TAO (Inria), LRI, UMR 8623 (CNRS - Univ. Paris-Sud), bat 490 Univ. Paris-Sud 91405 Orsay, France, teytaud@lri.fr.

[14] Brgmann, Bernd (1993). Monte Carlo Go. Technical report, Department of Physics, Syracuse University.

[15] Huang, Shih-Chieh, Remi Coulom, and Shun-Shii Lin. Monte-Carlo Simulation Balancing in Practice. Taiwan: National Taiwan Normal University, Dept. of CSIE, . Print.

[16] Baudis, Petr . "Pachi Simple Go/Baduk/Weiqi Bot." . repo.or.cz, 1 Feb. 2014. Web. http://repo.or.cz/w/pachi.git.