INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

ÁREA DEPARTAMENTAL DE ENGENHARIA DE ELECTRÓNICA
E TELECOMUNICAÇÕES E DE COMPUTADORES

Guiding Monte Carlo tree searches with neural networks in the game of Go

Gonçalo Antunes Mendes Ferreira
(BA)

A dissertation submitted in fulfillment of the requirements for the degree of Master in Computer Science and Computer Engineering

Adviser

Prof. Hélder Jorge Pinheiro Pita (PhD)

Jury

Prof. Manuel Martins Barata (PhD, President)

Prof. Gonçalo Caetano Marques (PhD)

Prof. Hélder Jorge Pinheiro Pita (PhD)

MAY, 2016

Abstract

The game of Go remains one of the few deterministic perfect information games where computer players still struggle against professional human players. In this work two methods of derivation of artificial neural networks – by genetic evolution of symbiotic populations, and by training of multilayer perceptron networks with backpropagation – are analyzed for the production of a neural network suitable for guiding a Monte Carlo tree search algorithm. This last family of algorithms has been the most successful in computer Go software in the last decade. Using a neural network to reduce the branching complexity of the search is an approach to the problem that is currently being revitalized, with the advent of the application of deep convolution neural networks. DCNN however require computational facilities that many computers still don't have. This work explores the impact of simpler neural networks for the purpose of guiding Monte Carlo tree searches, and the production of a state-of-the-art computer Go program. For this several improvements to Monte Carlo tree searches are also explored. The work is further built upon with considerations related to the parallelization of the search, and the addition of other components necessary for competitive programs such as time control mechanisms and opening books. Time considerations for playing against humans are also proposed for an extra psychological advantage. The final software – named *Matilda*[1] – is not only the sum of a series of experimental parts surrounding Monte Carlo Tree Search applied to Go, but also an attempt at the strongest possible solution for shared memory systems.

Keywords: Go, Monte Carlo Tree Search, Transpositions, Game Tree, Machine Learning, Neural Networks

[1]After the nickname of the authors favorite World War II tank: the British Infantry Tank Mark II.

Resumo

O jogo de tabuleiro Go é um dos poucos jogos determinísticos em que os computadores ainda não conseguem vencer jogadores humanos profissionais consistentemente. Neste trabalho dois métodos de aprendizagem – por um algoritmo genético e por treino por propagação do erro – são utilizados para criar redes neuronais capazes de assistir um algoritmo de pesquisa em árvore de Monte Carlo. Este último algoritmo tem sido o mais bem sucedido na última década de investigação sobre Go. A utilização de uma rede neuronal é uma abordagem que está a sofrer uma revitalização, com os recentes sucessos de redes neuronais profundas de convolução. Estas necessitam, contudo, de recursos que ainda são muitas vezes proibitivos. Este trabalho explora o impacto de redes neuronais mais simples e a produção de um software representativo do estado da arte. Para isto é complementado com técnicas para pesquisas em árvore de Monte Carlo, aquisição automática de conhecimento, paralelismo em árvore, otimização e outros problemas presentes na computação aplicada ao jogo de Go. O software produzido – Matilda – é por isso o culminar de um conjunto de experiências nesta área.

(Este texto foi escrito ao abrigo do novo acordo ortográfico.)

Palavras-chave: Go, Pesquisa em árvore de Monte Carlo, Transposições, Árvore de Jogo, Aprendizagem Automática, Redes Neuronais

”The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.”

Edsger W. Dijkstra

Contents

List of Figures

Acronyms

AMAF All-Moves-As-First. 16, 28–35, 63, 68, 70

ANN Artificial Neural Network. 2, 16, 40–43, 45, 46, 49, 56, 58, 79, 81–86, 91, 93

CFG Common Fate Graph. 71, 73, 75, 92

CNN Convolution Neural Networks. 1, 2, 17, 41, 42, 47, 49, 79, 91, 93, 94

GTP Go Text Protocol. 16, 19, 21, 44, 45, 50, 51, 55, 56, 58, 77, 87, 88

LGRF Last-Good-Reply with Forgetting. 34, 35, 61, 70, 83

MC Monte Carlo. 16, 23, 29, 31, 32, 34, 35, 37, 63, 64, 87

MCTS Monte Carlo tree search. 1, 2, 15–17, 21, 23, 25, 27–29, 31, 34–37, 41, 42, 49, 51, 53, 55, 60, 63–65, 68, 70, 72, 73, 75–77, 79, 81, 85, 91, 92

MDP Markov Decision Problem. 22, 23

MLP Multilayer Perceptrons. 40, 41, 47–49, 63, 77, 79, 81, 82, 93

MM Minorization-Maximization. 39, 75, 93

RAVE Rapid Action Value Estimation. 16, 28, 29, 31–34, 61, 64, 68, 70, 87

SANE Symbiotic Adaptive Neuro-Evolution. 16, 41–44, 46, 47, 49, 56, 81, 85, 86, 91

SGF Smart Go Format. 15, 16, 21, 59, 77, 87

UCB Upper Confidence Bounds. 23, 25, 26, 29, 31

UCT Upper Confidence Bounds for trees. 16, 25, 27, 29, 31, 33, 36, 37, 39, 49, 63, 64, 70

UCT+T Upper Confidence Bounds for trees with transpositions table. 27, 51, 58, 69, 85

Chapter 1

Introduction

1.1 Motivation

After the defeat of western chess champion Garry Kasparov in 1997 and the availability of affordable personal computers, a number of abstract games that were previously relegated to the backstage experienced a surge of interest. Go was an obvious next objective, given its popularity – the International Go Federation reports on over 40 million Go players worldwide. It has, however, eluded computer scientists to this day, with current top of the line Go programs still having little chance against Japanese professionals. Some upsets gradually happen, with computers needing less and less handicaps to defeat professional players, but the reign of, affordable, master computer play is yet to be seen.

Apart from the challenge or mystery in the future, Go is a beautiful game. While chess has different kinds of pieces, special rules (en-passant, repetition, promotions, castling), Go features simple rules, easy to explain to a child and represent in computer software; with matches usually ending with the agreement of both players. Nevertheless it remains a difficult game to master, with much disagreement on many parts of the game even given its long history and study. It is not uncommon to hear of players dedicating their whole lives to the game and popularizing a particular style of play, opening or continuation.

The world of computer Go was also rocked in the last decade with the development of Monte Carlo tree search (MCTS) algorithms. Before the excitement has dwindled down, the recent surge of interest in Convolution Neural Networks (CNN) has brought new hope for overcoming the last hurdle – defeating a human champion in an even match.

The motivation behind this work is more modest: to explore this area and build a base from which to build a competitive program. This kind of work takes many

years, and may come too late, but working on this area of research and on Go is reward enough for a lifetime.

On a personal note, the interest grew first in the form of a computer chess program, that used a minimax algorithm with $\alpha - \beta$ pruning. Upon discovering the game of Go, the program was rewritten and a Monte Carlo planner was added. This early phase of development, influenced the architecture and availability of multiple playing strategies in Matilda, although since then the minimax algorithm has been removed.

Development continued in parallel, in one approach through the improvement of the MCTS algorithm; and in another through the training and evolution of neural networks. Eventually the efforts would merge, and the work focus on improving the strength of the program as a whole. What is shown here is both what was considered most relevant and what is now present in the resulting software.

1.2 Goals and expectations

This work consists in the exploration of a number of topics for the production of a computer Go program. It attempts to apply research in the field of machine learning with Artificial Neural Network (ANN) to the more recent work done in MCTS for deterministic perfect information games, in order to make a stronger computer Go player. It is also conveniently inserted between last decades MCTS research and the emerging research on the application of CNN.

The first and main goal is to produce a computer Go software that can play Go at reasonable strength with both human and computer players, in consumer grade hardware. This software will use a number of algorithms popularized since 2007 to form a program that is both strong and representative of the current state of the art in computer Go.

The second goal consists in the exploration of techniques for producing Go playing ANN, or helper networks: i.e. a system that is aimed at improving a specific part of the algorithm (MCTS) without being overwhelmed by the entire complexity of the game; and how best to integrate the generated ANN in MCTS.

It is expected that the program at the time of delivery will be significantly weaker that the best programs right now, since a great number of things will invariably have bugs, be incorrect, misparametrized or missing. It is also expected that the use of a simple ANN would weaken the strength of the program in small boards and situations with local fights, but hopefully provide a modest increase in strength in the early game for larger boards.

1.3 Structure of this document

Starting in Chapter 2 this document introduces the rules of the game of Go together with some problem specific terminology that will later on be useful throughout the text. Then, a number of techniques used in computer Go are syntactically presented to the reader, so to bring everyone together – reader and writer – up to date on the work on computer Go of the last decades.

The solution of Matilda is then presented in Chapter 3 in a conceptual overview. There are many parts other than deciding what to play in a Go playing program and while this discussion might not be the most interesting, it is crucial for navigating the source code with ease. All the techniques presented in this chapter were tested in Matilda. In Chapter 4 – Implementation, a number of corrections and performance adaptations to the previous techniques are presented. Problems that are implementation specific are also explored. It is in this chapter that the program strength at playing Go and performance come into play, and determine how and what previous techniques are used. While some implementation decisions can be justified – many are the pragmatic result of long periods of testing.

The document concludes with Chapter 5 with some commentary of the direction of computer Go, the difficulties, development and results obtained in this work, and suggestions for future work; some directed at no one in particular, others at Matilda itself.

The text in this document is enriched with digital textual links. When these links target an outside document, the corresponding address is also printed so the reader does not require a digital copy of the document. It is however suggested the use of a PDF reader. This document also does not feature a glossary, preferring to introduce the many context specific terms throughout the text. As a replacement you can find an index of these terms at the very end of the document, and a list of acronyms in the beginning.

This document is written in American English.

Accompanying software

This work is accompanied by the source code for a series of computer programs, under the umbrella designation of Matilda.

You can find a copy of the source code at https://bitbucket.com/gonmf/matilda. Git is used as the version control system and issue tracking is found from the projects web page.

Chapter 2

Problem

2.1 The game of Go

Go is a board game for two players invented in China before 4th century BC. It was highly regarded as a national art and would eventually spread to Korea, and Japan. The game became the most popular board game in Japan, with schools and ministries formed. Because of this, much Go terminology is borrowed from the Japanese language, like the use of kyu/dan rankings. After the Japanese Meiji revolution the game popularity would receive a serious hit, and again with the advent of World War II. In Korea and China however, it was rekindled and their players would come to match their Japanese counterparts. Ever so slowly Go is also capturing the minds of the rest of the world.

The game itself has suffered little changes since ancient times. The earlier forms of Go appear to have used 17x17 boards, shifting to 19x19 by the 5th to 7th centuries. The *komidashi* was also introduced only in the 1930s. This relative consistency has facilitated the appreciation of older games.

This document will introduce Chinese rules because they are the most commonly used for computer play, but the most popular among humans are Japanese rules, which are significantly different.

2.2 Chinese rules

The game of Go is an abstract, deterministic, perfect information, adversarial, two-player, turn-based game. The first player places a black stone on one of the 19x19 intersections of the board. The second player plays with white stones. The game progresses with the players alternating in either placing a stone in an empty intersection, or passing their turn.

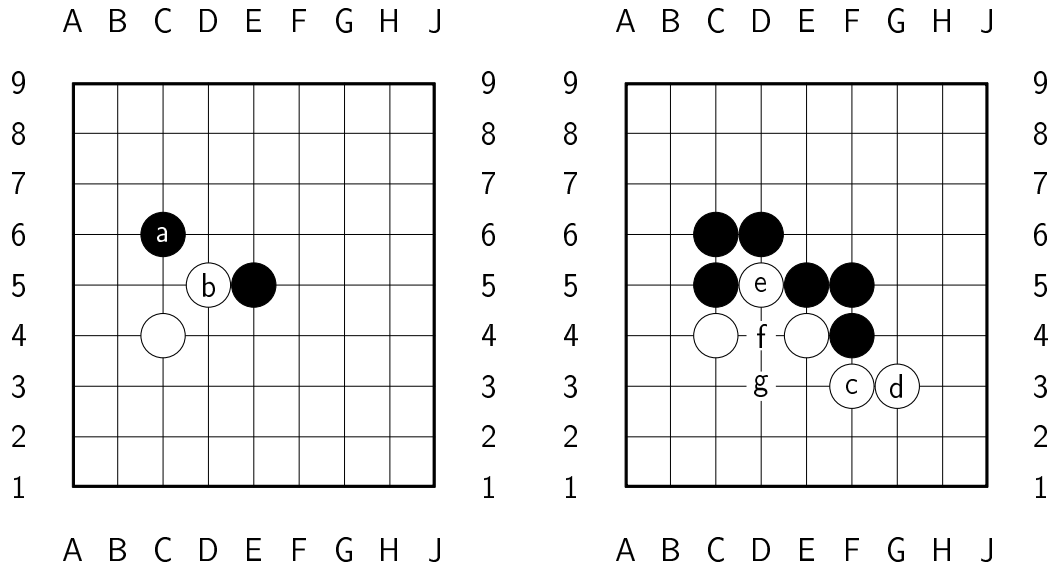


Figure 2.1: Stones and liberties

A possible start of the game in 9x9 boards can be seen in Figure 2.1. To win the game it is necessary to have the most territory, and to have the most territory it is necessary for the players stones to remain on the board. The stones are removed when they are *captured* by the opponent, and they are captured when their number of liberties reaches zero. The liberties are the unique adjacent empty intersections of a group of stones. Stone **a** pictured in Figure 2.1 has four liberties while stone **b** only has three.

The right board shows a group[1] composed of stones **c** and **d**, which has five liberties, and a group with the single stone **e**, which only has one left. When a group of stones only has one liberty it is said to be in *atari* – in risk of capture by the opponent. Stone **e** doesn't share liberties with other stones because it is not connected by adjacency, only diagonally.

From this position we can say that the white player has a larger influence on the bottom of the board, and this intuitive notion can be understood as it being more likely for that area to belong to white, in the end of the game. Since groups of adjacent stones of the same color also share the same fate, this document will use,

[1] In Go literature the term *group* is often used to refer to loosely connected stones that are likely to share the same fate; but are not necessarily connected by adjacency. When connected by adjacency they are often called a *string* or *chain*. In Matilda we found the terms *loose group* and just *group* more descriptive of these concepts.

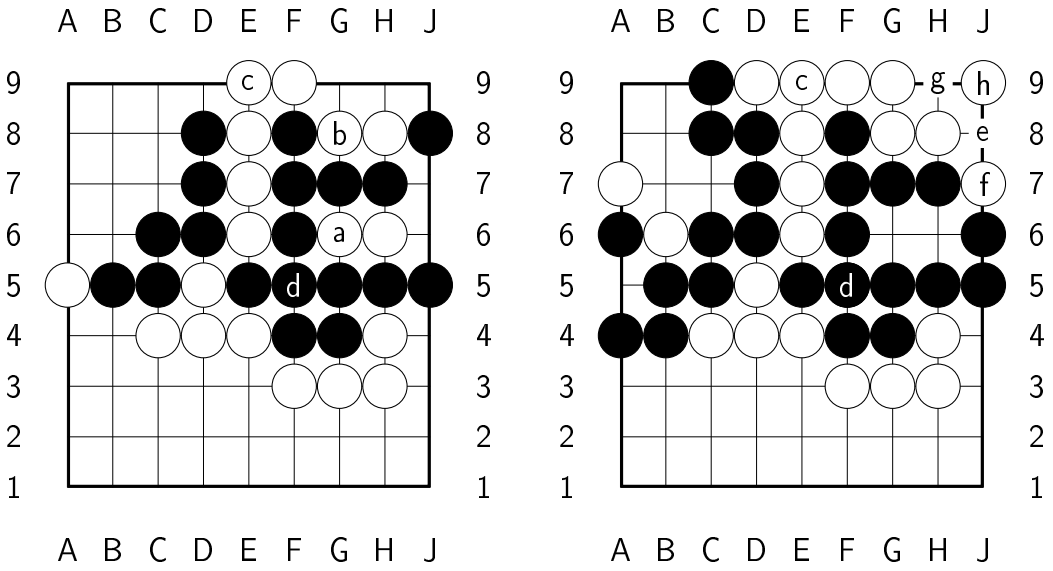


Figure 2.2: Life and death

for brevity, a single letter over a stone to signify the entire group.

If black was to capture by playing at **f**, and white had a stone already in **g**, then white could retake at **e** and capture **f** back. The game would have returned to an immediately prior state. This is prohibited by a rule called *ko*. The *ko* rule is specific to repetitions to the immediately previous state, but repetitions can happen to states further past, when more stones are captured. The rule that prohibits these repetitions is called *superko*.

In Figure 2.2 the group of stones **a** is in *atari*, but groups **b** and **c** can also be said to be *dead*. The terms dead or alive are used to refer to the possibility of stones groups remaining on the board without a mistake from the opponent. Groups **a**, **b** and **c** cannot ensure at least two liberties with which to survive. Group **d** may appear to be dead, since it is almost surrounded, but by playing out the game it can be seen black can capture group **c** first, ensuring it doesn't have a shortage of liberties. Notice that groups on the border of the board have naturally less liberties. This can be used both for offense and defense.

In the right board a possible followup is shown. If black group **d** is threatened, it can be made safe by playing **e** (making one more liberty) – white cannot retake at **f** because of the *ko* rule – and then by playing **g**, capturing both groups **c** and **h**.

When the players are satisfied they can choose to pass, and agree to end the game. At this point the scoring phase begins, which starts by identifying groups of

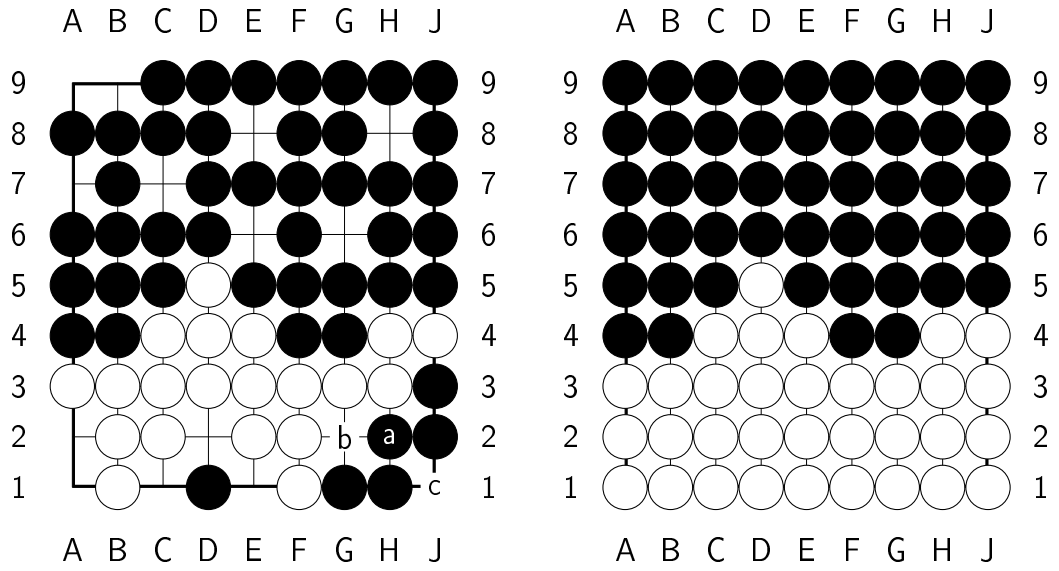


Figure 2.3: Scoring

stones that are dead, and removing them. Figure 2.3 shows group **a**, which is dead because white can play first **b** and then capture with **c**. White playing first at **c** is illegal because it would be a suicide. A play can't be a capture and a suicide at the same time, since a capture frees at least one liberty. Captures have priority over testing for suicide.

After removing the dead groups we count the number of stones in the board plus the intersections that they encircle. This is represented in the right board. Black has 48 intersections and white has 33, black wins by 15 points.

To compensate the white player, who started second, it is customary to award her extra points. This is called the *komidashi – komi* for short – and is usually 5.5 or 7.5 points. The fractional part, when present, is used to ensure the game does not end in a draw.

Having understood how to establish territory and capture stones, let's move on to a 19x19 board. Figure 2.4 will be used to explain eye shape. Play in 19x19 usually starts in the corners, then moving towards the sides and finally to the center of the board. Being close to the sides is advantageous because it is easier to apply influence over that territory, since the border also acts as stones for either color. The areas marked **a**, **b** and **c** all contain four intersections, yet it took very different amounts of stones by black to surround them.

These areas, in the context of providing liberties to a group, are called its eye

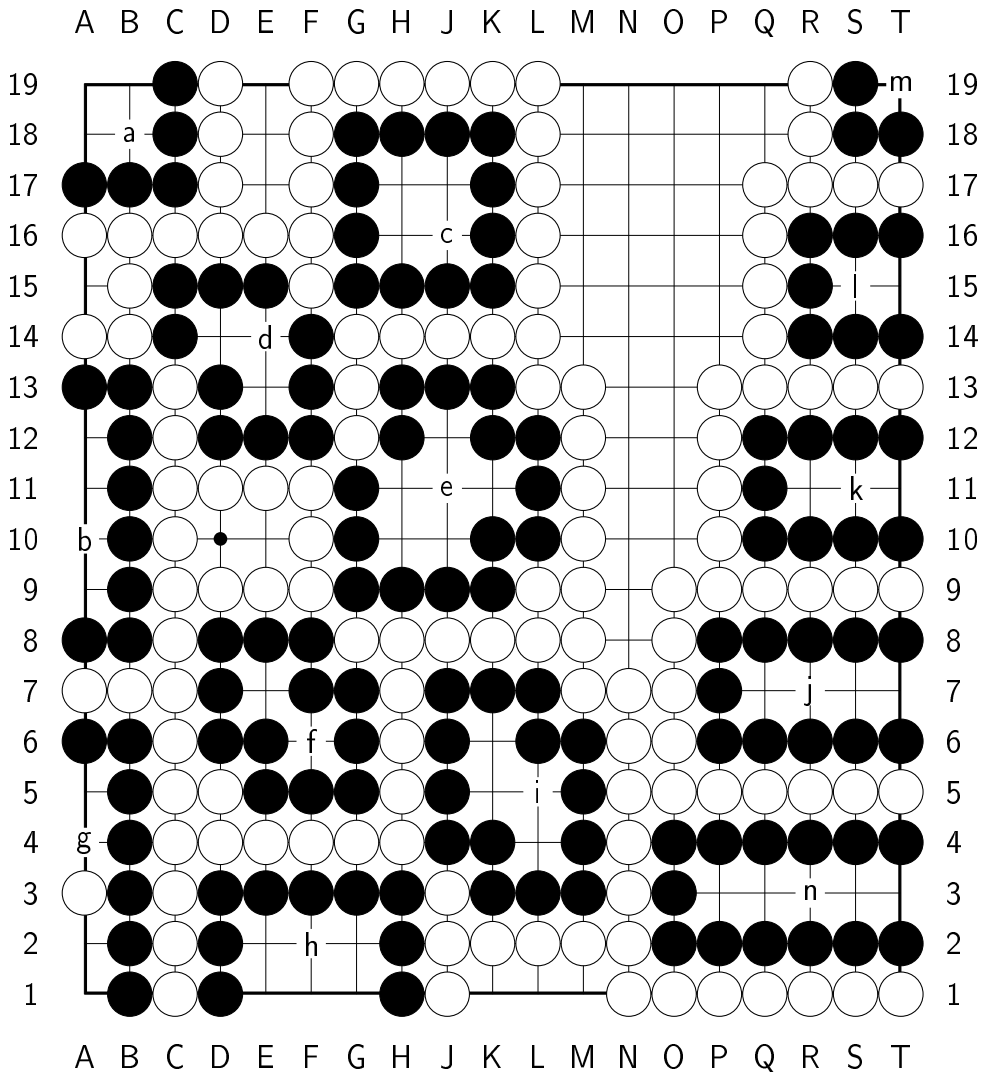


Figure 2.4: Eye shapes

shapes. Some eye shapes are able to guarantee life by providing untakeable liberties, others are guaranteed to be dead, and finally others are disputed – they depend on *nakade* to form what is called *eyes*. Eyes are single empty intersections surrounded by stones of the same player. If a group can secure two or more eyes, it is unconditionally alive, because the opponent needs two plays to capture the group and the first play would be suicide. Figure 2.4 shows examples of eye shapes that are dead: **a**, **c**, **l** and **m** – they cannot make eyes without a mistake by the white player. **b**, **g**, **i**, **j**, **n** and **h** are alive even if white plays first; black can always make at least two eyes. **d**, **e** and **k** depend on good play from black and being her turn to play. There are critical points which if missed by one player, can be played at by the opponent to kill the group. For instance a play in intersection **d** either makes two eyes (played by black) or makes it impossible to make eyes (played by white). These vital points are *nakade*.

Plays aimed at increasing the number of liberties of a group, but which can be put into *atari* again after; and whose group cannot escape being captured, are called *ladders*. Figure 2.5 shows a few examples of ladders, on the left of the board. Notice how white can just pursue the white stones, step by step, until reaching the border of the board. A group on a correctly read ladder can never live. Groups **a**, **b**, **c** and **d** are all ladders and can be killed by white when they reach the border. It is imperative for computers to be able to read ladders; following a ladder needlessly can go on for many turns and decide the outcome of the match.

To the right of the board in the same figure is an example of *seki* – mutual life. *Seki* is a situation where two opposing groups are alive because to move first would mean the death of that players group. A play at either **e** or **f** would be followed by an opponent play at the other intersection, capturing the group. In general two plays where the first one forces the other are also said to be *miai*. When it isn't advantageous for either player to play, both players pass and the game may end with empty intersections that do not benefit either player, if they are liberties of groups in *seki*. Passing is always legal, there is no zugzwang in Go.

The above examples are naturally not representative of actual gameplay. In real play the very first plays could be called the opening stage of the game and are often at the corners. When players start clashing and threatening each others stones there are usually already well studied, solid sequences of plays that can be followed called *joseki*. A standard opening followed by a well known *joseki* is similar to a chess opening, only longer.

When both players seem content with their positions in the corners and sides they may attempt to secure the center of the board, and we are usually in the middle game. At this point areas of influence begin to form and the players begin saving *ko* threats. A *ko* threat is a forcing move – diverting a player from securing a position in *ko*, at a turn where the player can't take the *ko* back herself (because of the *ko* rule).

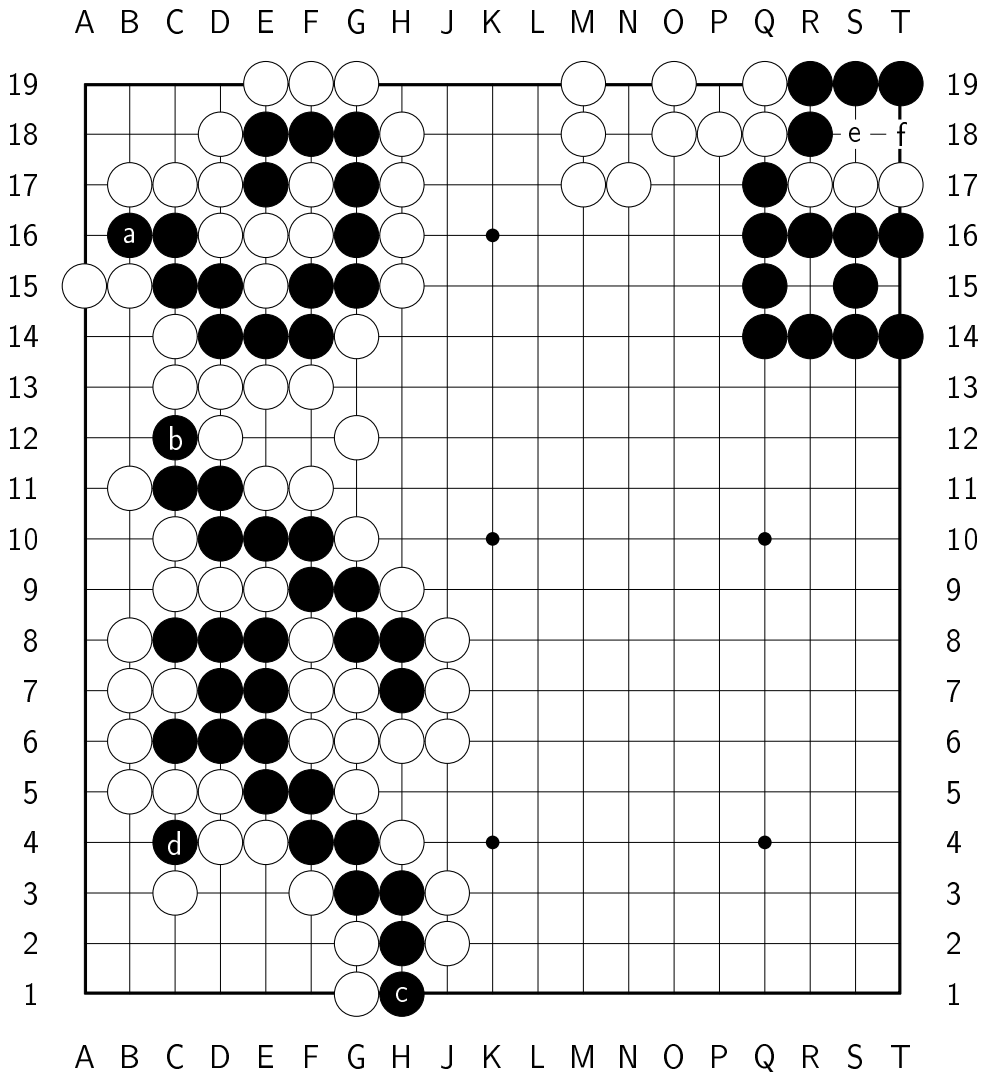


Figure 2.5: Ladders and *seki*

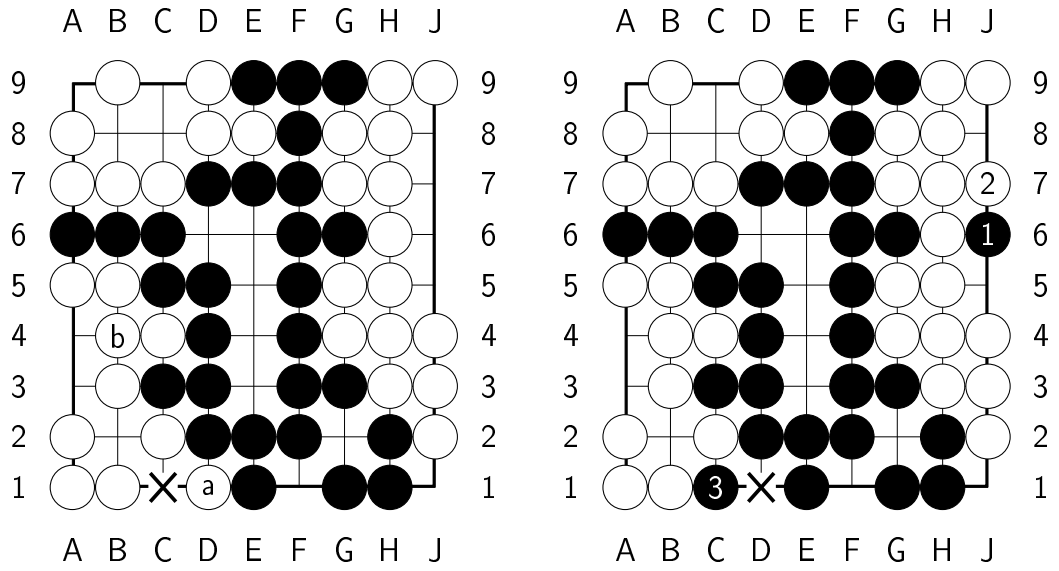


Figure 2.6: *Ko* fighting

The player that has more *ko* threats (or more valuable *ko* threats) is better equipped to settle *ko* fights in her favor; and *ko* fights often decide the life and death of whole groups. Figure 2.6 shows an example of a *ko* fight (left board). White just played **a** and black cannot retake at the marked intersection yet. If she has a good enough forcing move she could, in her next turn, recapture **a**. A possible continuation is shown on the right board of Figure 2.6: **1** is a simple high importance forcing move; white defends and this allows black to later capture group **b**. Figure 2.4 also showed some forcing moves: **b**, **g**, **h**, **i** and **j** are eye shapes whose alive status demands a response from black if attacked.

The ability to lead the game, instead of answering, can be critical to securing points in the middle game, and *ko* threats or simply ignoring an opponents play – *tenuki* – can be used for this.

When reaching the end game the board influences are almost settled, and the players must secure their points, with caution to avoid making mistakes that cost the life of entire groups of stones. Figure 2.7 shows a professional game in the end game, turn 182. Professional play may look overwhelming. Notice how the groups, both the alive and dead, are not exaggeratedly defended or attacked.

When the game is played by players of too big a difference in skill it is customary for the stronger player to have a handicap. This handicap is usually either starting as white without *komi* or starting as white but with the weaker player having already

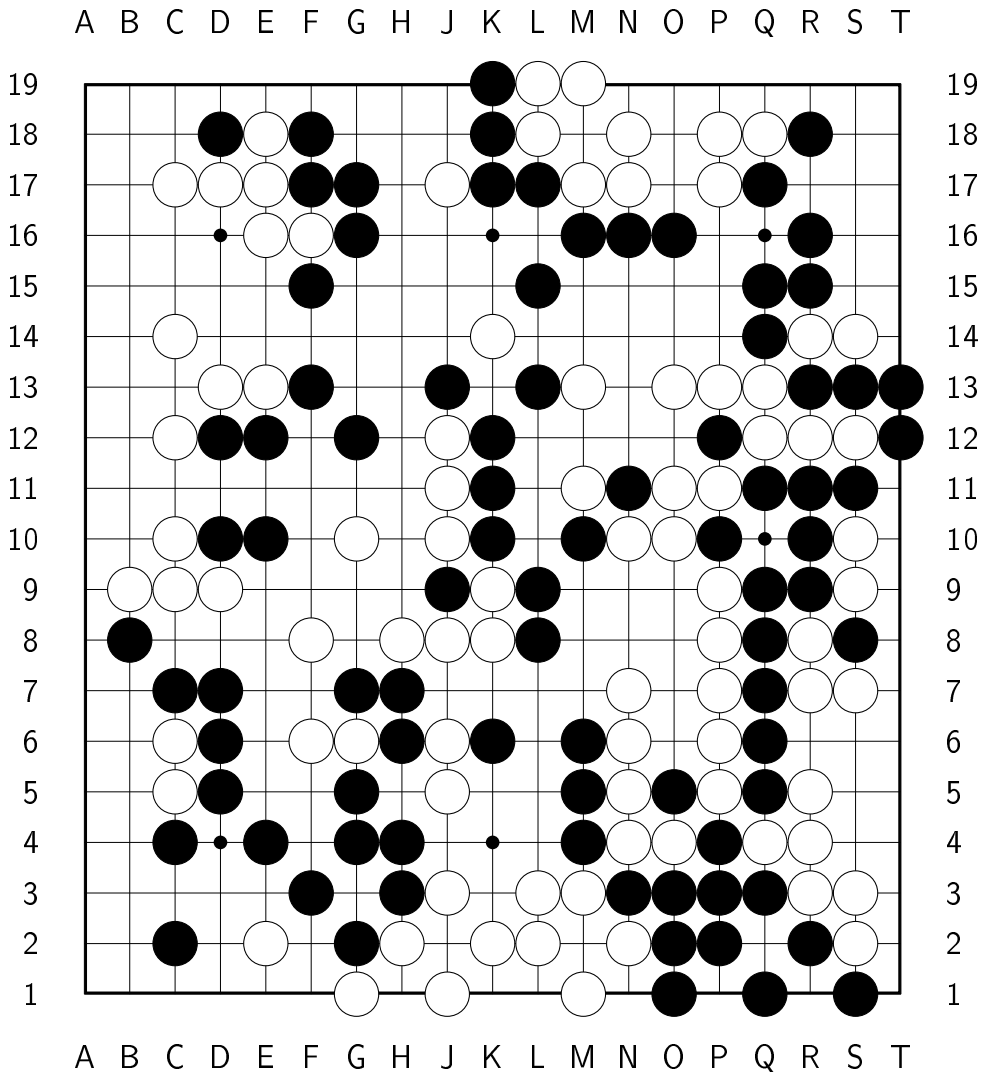


Figure 2.7: Shusaku (B) x Genan, Japan 1846

placed a number of handicap stones in the board. This number usually does not exceed nine stones. As a handicap, the stronger player may also be given less time to think if the match is timed.

Technically, what was introduced before was Tromp-Taylor's concise rules [TT95] without suicides; or Chinese rules with simple positional *superko* and without liberties of groups in *seki* counting for the score. Different rule sets exist today with the Japanese being the most popular. Chinese rules however are more practical for computer players, and will be used exclusively in this work.

2.3 Computer Go

Computer Go is the general term used for the work around computer programs attempting to play or understand Go. It is a field of artificial intelligence that borrows much from attempts to tackle other perfect information games. The greatest difficulty in Go arises from the exponential complexity of the possible game states, result of the size of the board (19x19 or even larger).

Table 2.1 makes evident the complexity growth with board size [TF07], for the most commonly used board sizes. Professional players seldom play boards other than 19x19 though, and computer programs usually play more turns to resolve scoring disputes that would be apparent to humans.

As of 2016 the number of legal positions has been solved for all boards 19x19 and smaller [Tro16].

Traditional search methods that attempt to search all possible states have proven ineffective at playing high quality Go, as have pure pattern based approaches. This due to its complexity, that translates into a prohibitively large branching factor over a game that can be several hundreds of plays long – requiring both short and long term planning.

Modern computer Go software usually features a set of elements:

1. Compilations of openings and joseki are tested to avoid having to exhaust the time limit on the beginning of the match.

Board size	Intersections	Legal positions	Game length (avg.)
9x9	81	10^{39}	45
13x13	169	3.7×10^{80}	90
19x19	361	2×10^{172}	200

Table 2.1: Game variables by board size

2. Some form of game state search coupled with tactical analysis. The tactical analysis may be through the use of automated learning systems, previously compiled knowledge (like handcrafted patterns or rules), programmatic checks or through the simulation of gameplay with branching limitations. The game state search attempts to overcome the limitations and biases of the other methods by providing a more broad strategical feeling for the game, whereas tactical analysis are local and reward driven, which is not useful by itself for problems such as Go.

In the past more attention was given to the tactical analysis, with few programs performing full whole board searches. Different levels of decision systems would try to reduce the actual exhaustive search to a minimum. This method of computer Go programming would prove very difficult and required a Go expert developer. It was also not uncommon for humans to discover and exploit specific difficulties the programs had.

Today a computer Go program is often smaller thanks to MCTS. It uses as much previously generated knowledge as possible to reduce the complexity of the game tree or better simulate the game, invariably introducing a local optima (which would always be present unless the search is exhaustive) and then tries to refine it's chosen play given the time available. The solution presented in this document will not deviate from this general formula.

2.4 Techniques for computer Go

This section attempts the herculean task of identifying the major technical contributions to computer Go in the last decades. While this pertains specifically to Go, or was demonstrated in Go, research on other areas of course has also played a major role. This section is also restricted to techniques that benefit the playing strength of a computer Go program – computer Go research is not restricted to this competitive approach.

The first program to tackle the whole game of Go was written in the late 1960s [Zob70]. It introduced ideas that would later be developed in the programs for several decades, like codifying Go proverbs as criteria for tactical analysis, feature extraction for pattern matching and the efficient hashing of game positions.

In 1987 Anders Kierulf formalizes a file format for exchanging Go records (and compatible games) – Smart Go Format (SGF).

A program first competed in a human Go tournament in the 1980s. In the late 1980s GNU Go is started and in 1990 it would introduce finite automata for pattern matching. In 1993 the work on program Explorer produces research around position

evaluation and solving difficult sub-problems. Its author would also go on to update and extending SGF, renaming it Smart Game Format.

In 1993, the first ideas of using Monte Carlo (MC) methods for Go were proposed. Bruegman goes on to experiment with non-random simulations [Bru93].

NeuroGo [Enz03] in 1996 and Honte [Dah99] in 1999 would integrate trained ANN in a more classical solution for the time. The networks were trained through temporal difference learning (TD), to identify the influence of the players over the board (the likelihood of ownership of each intersection). This was applied to 9x9. Around the same time the method Symbiotic Adaptive Neuro-Evolution (SANE) was invented for evolution of networks for tasks not suited for reinforcement learning methods. SANE and evolutions of it would be successfully applied to Go in 5x5, 7x7 and 9x9 boards in the next years.

Around the turn of the century the first protocol for computers playing Go – Go Modem Protocol – was created. It was a binary protocol and would be mostly replaced by the Go Text Protocol (GTP) when it was released by the team behind GNU Go, almost a decade later.

In 2002 however, a policy for the problem of multi-armed problems that reduces its expected regret was published. This would lead to the beginning of the use of Monte Carlo simulations for Go. In 2003 the program Indigo would first experiment with Monte Carlo searches, but still in the context of a more classical architecture at the time. In 2005 it would apply research on Bayesian generation of 3x3 patterns to guide the simulations and would win its first tournament. Also in 2005 Crazy Stone started using a Monte Carlo tree search, followed by MoGo.

In 2006 MoGo would demonstrate the strength of simple handcrafted patterns in the playout phase of MCTS. Upper Confidence Bounds for trees (UCT) was also introduced in MoGo and was a vast improvement over plain MCTS. Since then work on computer Go has shifted heavily towards MCTS programs, with a focus on improving pattern matching, usually with automated learning; improving information sharing in the UCT game tree; and improving MCTS itself with a focus on high parallelization. MoGo would also benefit from research in TD made for RLGO [SM07].

In 2003 the All-Moves-As-First (AMAF) heuristic is introduced [BH03], and in 2009 it is improved by Rapid Action Value Estimation (RAVE), reducing the expected error from the use of AMAF sampling. The notion of criticality would also be born as a biasing method towards the exploration of intersections with a high covariance with winning the game.

In 2004 Eric van der Werf would explore machine learning techniques for tasks other than just move prediction [vdW04]. He would also *solve* the game for 5x5 boards.

In 2010 contextual MC and nested MC are published, as well as Last-Good-Reply with Forgetting [BD10], which is an advancement of Last Good Reply from 2009.

Rémi Coulom would later successfully explore new methods for parameter tuning of playout policies for MCTS [Cou07, HCL10]. Such techniques are used today in the strongest programs, like Zen and Crazy Stone. Both these games would go on to defeat 9p (professional dan) players with four stones handicap in 2012 and 2013 respectively.

In 2011 Petr Baudiš would explore forms of situational compensation in Pachi, and introduce liberty maps and a definition for the horizon effect [Bau11].

In 2014, two teams [CS14, MHSS14] independently applied CNN to Go, originally at predicting professional play in game records. Maddison et al. also perform the first experiments on integrating CNN with MCTS [MHSS14], with promising results. These findings catapulted CNN to the front of computer Go efforts.

Working for Facebook AI Research Yuandong et al. announced their own research into the application of CNN with MCTS for Go [TZ15], with their first publication exploring the prediction of more than just the next move. Their program written from scratch would start competing in online tournaments in 2016.

In early 2016 it also became known that a team from Google had produced a program – AlphaGo – that for the first time defeated a professional player without handicap (Fan Hui 2p in October 2015). It used different CNN for reducing the candidate plays and then suggesting the next play, in the context of MCTS [Sea16]. In March the program went on to convincingly defeat Lee Sedol 9p – arguably one of the strongest living players.

Research into computer Go enjoys an honest, open discussion online, but has a great majority of computer programs being closed source. Although the general algorithms might be known, what makes the very best programs the best remains in many cases a trade secret. The best free software programs are significantly weaker than their proprietary counterparts as of 2016.

Chapter 3

Solution

3.1 Model

When speaking of computers playing Go it is intuitive to place greater importance in the part that selects the next move, in the context of a game. This section instead starts with a top-down approach.

From the perspective of Matilda it functions as a server in a client-server architecture. The client issues commands that the server fulfills, mostly without the possibility for negotiation. The server keeps a context that includes a Go game representation, time control settings and other minor state. Matilda has a single context and the same instance cannot be used to play in multiple games at once. This architecture moves responsibility from the server agent to the client. This relationship is also asymmetrical; two computer programs of this model cannot communicate directly.

The client in this relationship is a game controller program. In the simplest form, Matilda can play with humans and be its own controller program. This is not very user friendly. A common alternative is to use a separate program for user input and display. This program usually relays the user input to advance the game, controls the time and ensures both players are satisfying the rules.

For computer versus computer play the client takes a form of mediator between the programs, often even geographically distant.

Because of this shift in responsibility a computer Go program can be mainly concerned with requests to play – where given its internal representation of the game, that it keeps up to date, it applies a series of techniques to decide its next move. In Figure 3.1 the communication details of how the commands are exchanged, and therefore if the actor is a program or human being, are omitted.

Most competitive programs, and GTP itself – the protocol most commonly used

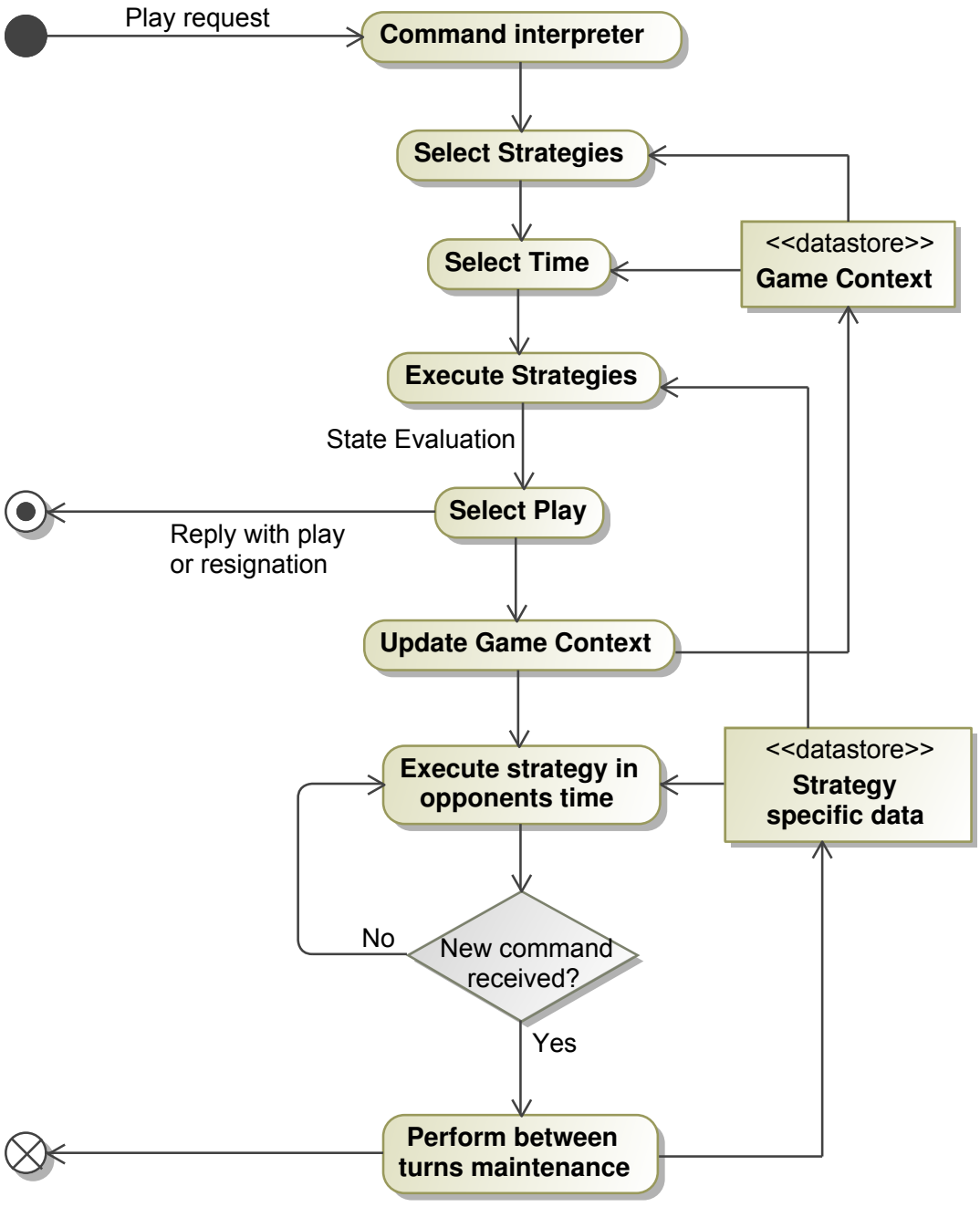


Figure 3.1: Activity diagram for play requests

for Go – are tailored for this use. A request to play in Matilda invokes a short pipeline of instructions. First, transformations may be operated on the state, such as color changes and symmetry operations. Then an opening book and other techniques are invoked. If they satisfy the request, i.e. can immediately provide an answer of satisfying quality, then the request is complete. If not the process continues to the default play selection strategy, which in Matilda is MCTS.

GTP as a protocol has its quirks. It is not solely meant for competitive play. It supports asking the computer to repeatedly play in a row, analyzing positions without actually playing, having the same program play as both players, etc; so it may become confusing for a program to manage its internal state. To make it easier for Matilda to know when it can safely free resources an optional flag is used. It reduces the scope of the protocol for competitive play by informing it will play in a contest between two distinct players, alternating colors.

Before a request to play starts a MCTS, the context of the game is used to decide the amount of time it should run, taking into consideration the time system used and network latency. MCTS will perform as many simulations as possible in that amount of time. After the execution of MCTS or other strategies, a full board outlook is sent back.

It is then used to select the actual play selected, because the best rated play may be illegal: the state evaluation strategies don't have a full match context, and it is necessary to detect *superkos*. Methods of dealing with *superkos* vary from program to program, but in general their detection is usually lax in MCTS.

After replying with the selected play, in a competitive setting, it is the opponents turn to *think*. This gives the program plenty of time to perform maintenance or polish its previous decisions, which it does. The program keeps polling for new commands while extending the MCTS (if it was the strategy previously used), thus preparing itself already for the next turn. This preparation relies on the fact that MCTS builds a large amount of knowledge about part of the game tree, and the more time it has the more accurate its information.

Other program functionality, like manipulating SGF game records, calculating the score and analyzing the current game position is also started from the GTP commands that invoke specialized routines. Because there is little decision making from Matilda the underlying source code is organized in a very flat way, with the exception of MCTS related code.

3.2 State space search

The terms state and state space have already been used in this document. These are used to abstract the game of Go to a problem pieced together with states, which constitute what the system can be at a given time, and transitions, that constitute how

the states can evolve from one to another. A state for the purpose of representing a match of Go must therefore contain all game information necessary for intelligent play – like where the stones are and who is playing. Since this is an abstract representation we can reduce it if we ignore or abstract parts of the problem. For instance if a problem is made of smaller, independent problems and we want to only solve a particular subset of them at a time.

The transition between states is usually done based on rules, in this case the rules of legal play in the game of Go. Transitions are directed which means a directed graph can be constructed of the states and their transitions. This further means that from the perspective of a specific, initial state, we can construct a tree structure from the transitions possible at each state.

We make the distinction between game tree – the abstract directed graph of possible game states – and search tree – the tree like structure that may be physically built (at least on the stack) during an algorithmic search. This tree is acyclic if it detects and prevents state repetitions.

A state space search algorithm in general attempts to navigate the state space in an efficient way. In this we can have many objectives: if for the purpose of aiding a Global Positioning System for traveling, for instance, we would want to find the path from the top of the tree to a specific leaf that is expected to best fit some criteria, like distance or time traveled.

In Go we do not have a particular target leaf to obtain, we instead look to win the match or maximize the score in our favor. Since Go is a zero-sum game the opponent is expected to do the same and one of the players advantage means the disaster of her opponent.

A state space search unfortunately grows exponentially larger with the number of possible transitions at each leaf state. An average match of Go lasts about 200 turns (more for computers playing with Chinese rules) and uses a board of 361 intersections, realistically with 2×10^{172} valid Go states [All94] and a branching factor of about 250 [TF07]. In essence, while a number of algorithms have been invented to search space states efficiently, not one is applicable to a space as vast as this. State space search algorithms however can still be used for local problems, with drastically reduced branching options.

3.3 Monte Carlo searches

A Monte Carlo planning algorithm considers a Markov Decision Problem (MDP) state space, where we learn the reward distribution – and therefore best transition for the player, by random sampling. Each sample consists in navigating the space until a termination criteria, and registering the outcomes of each first transition of the sample. After a satisfying amount of samples, we select the transition with higher average

outcome.

This simple algorithm features very low spatial requirements, is highly parallelizable and works well in simple MDP. It is too simple for Go, because the reward distributions are different at each state and we are solving an adversarial problem. With reinforcement learning however, we can have a MC algorithm that can learn the best solutions (sequences of transitions) by storing the past sample outcomes at each state.

This is called Monte Carlo tree search and has been a focal point of competitive computer Go in the last decade. In MCTS we are learning the approximation of a function – in this case the function of match outcome as the product of playing – for more problem states than the current one. We are not searching for the exact best sequence of plays to victory, which is unfeasible in problems such as Go, but are aiming at the play that will minimize our loss.

The algorithm is usually divided in four steps, performed repeatedly:

1. Selection – From the initial state select and traverse valid transitions until a new state is found.

2. Expansion – Generate the new state.

3. Playout – Randomly play the game until the end from the current state, yielding a playout outcome (win or loss).

4. Propagation – Propagate the playout outcome across the sequence of states that led to the expanded new state, updating their estimated state quality.

These four steps are illustrated in Figure 3.2. So far this simple algorithm is purely observational and random, but it can be improved by directing parts of the search for faster convergence with the real reward distributions.

3.3.1 Guiding Monte Carlo tree searches

In attempting to reduce the regret of our Monte Carlo sampling we have a problem of *exploration versus exploitation*. The Upper Confidence Bounds (UCB) policy formula was initially introduced for the well studied problem of the multi-armed bandit MDP. A multi-armed bandit is some hypothetical gambling machine with a number of buttons, or levers. The gambler will try to produce information about the best levers to pull and will be faced, after an initial more random (exploratory) phase of play, with an exploration versus exploitation dilemma.

The UCB formula itself has a few versions and modifications by different authors, but all variations include selecting the transition that maximizes a combination of the times that transition has been tested, the number of victories (hence the expected

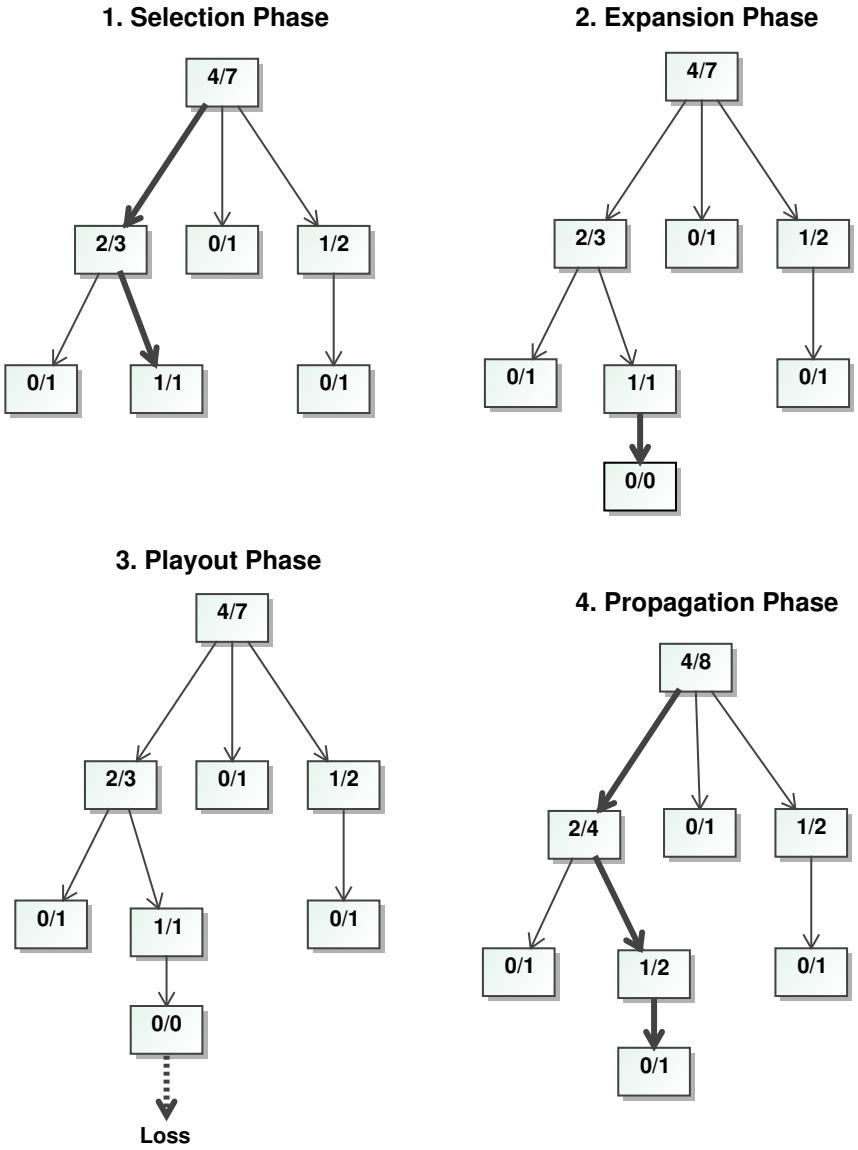


Figure 3.2: One simulation in MCTS with wins/visits

reward) and the total number of plays. Auer et al. [AFK02] introduce UCB1 and variants, and compare them with ε-greedy policies.

Under UCB1, at each selection phase we choose the transition that maximizes $x_i + \sqrt{\frac{2 \times \ln n}{n_i}}$ where x is the current machine we are considering, x_i the average reward for playing at i, n_i the number of plays made at i and n the total number of plays done overall.

The best policy empirically found was an extension of UCB1, called UCB1-TUNED. Both UCB1 and UCB1-TUNED were implemented in this work.

The adversarial multi-armed bandit problem is a variant of the problem where the reward distribution varies for the same state depending on the observer (own player or adversary, in two-player games).

In this new formulation of the problem we need our guiding policy to either minimize or maximize the expected reward, depending on the playing player, for each turn, in a manner similar to the minimax algorithm. The minimax family of state space search algorithms attempts to efficiently prune parts of the search tree based on the previously best observed paths (highest minimum reward). It is pessimistic for assuming good play from the opponent. Since Go is a zero-sum game we need to programmatically invert the reward when at the opponents turn. An exception to this is in situations where a play is only valid for one of the players, which is possible in Go.

3.3.2 Upper confidence bounds for trees

The requirement for the application of any guiding policy for trees is storing statistics about the playout outcomes in every node of the game tree, instead of only the first one. In comparison to the multi-armed bandit, this is exemplified as rows after rows of machines, with distinct reward distributions, with rules for the transition between them.

UCT (also known as UCT0 or UCT-T) is thus a policy for multi-level multi armed bandit problems that unites the use of MCTS with the application of UCB at each transition selection of the selection phase. UCT1 includes the use of transposition tables (also known as UCT+T). UCT2 takes into consideration that, although the game of Go is a perfect information problem, the states that are used in-game may not be perfect – for performance reasons they may not include the information necessary to identify *kos* and *superkos*. Because of this, in UCT2, an estimate of the average value is used instead of the actual value in the saved state.

Bellow is shown a summary of the UCT policies described above. Using the same terminology as [CBK08], $I_s^{\pi}(t)$ denotes the choice of transition in the tth simulation, to be applied at state s using policy name π. $A(s)$ denotes the set of valid transitions from state s, $Q_s(t)$ the average outcome of playouts from state s, and $Q_{s,a}(t)$ the

c	Win rate	Games
0.7	52%	490
1.1	58.1%	903
1.15	59.4%	1086
1.2	58.4%	476
1.3	55.1%	748

Table 3.1: Win rate by UCB c constant

average outcome of the application of transition a at state s. Conversely $N_s(t)$ and $N_{s,a}(t)$ are the number of times state s has been visited, and the number of times transition a has been selected at s, respectively.

$$I_s^{UCT0}(t+1) = I_s^{UCT1}(t+1) = \underset{a \in A(s)}{\operatorname{argmax}}(Q_{s,a}(t) + c_{N_s(t),N_{s,a}(t)}) \tag{3.1}$$

$$I_s^{UCT2}(t+1) = \underset{a \in A(s)}{\operatorname{argmax}}(Q_{g(s,a)}(t) + c_{N_s(t),N_{s,a}(t)}) \tag{3.2}$$

Where $c_{n,m}$ is the bias part of the UCB formula, which is shown again next for the UCB1 and UCB1-TUNED variants. $Q_{g(s,a)}$ is an estimate based on at least $Q_{s,a}$.

$$c_{n,m}^{UCB1} = \sqrt{\frac{2 \times \ln n}{m}} \tag{3.3}$$

$$c_{n,m}^{UCB1-TUNED} = \sqrt{\frac{\ln n}{m} \times \min\{\frac{1}{4}, V_{s,a}(n,m)\}} \tag{3.4}$$

with $V_{s,a}(n,m)$ the estimated variance of the UCB1 upper bound:

$$V_{s,a}(n,m) = (\frac{1}{m} \times \sum_{\tau=1}^{m} X_{s,a,\tau}^2) - Q_{s,a}^2 + c_{n,m}^{UCB1} \tag{3.5}$$

Where $X_{s,a,\tau}$ is the τth reward obtained from transition a in state s, and $Q_{s,a} = \bar{X}_{s,a}$ as stated before. This is, in essence, very similar to the variance formula for random distributions of finite populations. If we are considering playout outcome values as 1 (win) or 0 (loss) then $\frac{1}{m} \times \sum_{\tau=1}^{m} X_{s,a,\tau}^2$ is also equivalent to $Q_{s,a}$.

The impact of the UCB bias contribution (as product of the c constant) is shown in Table 3.1 for different values of c. These results are from an older version of Matilda, playing against GNU Go in 9x9 with 10000 simulations per move.

Transition selection using UCB1 can also be improved by simplifying the square root and calculating the numerator only once for every selection phase:

$$c^{UCB1} = \sqrt{\frac{2 \times \ln N_s}{N_{s,a}}} = \frac{\sqrt{2 \times \ln N_s}}{\sqrt{N_{s,a}}} \tag{3.6}$$

With $\sqrt{2 \times \ln N_s}$ precalculated for low values of N_s.

As already hinted, we will need to store state statistics in some kind of structure, preferably that allows the detection of transpositions. A transposition is merely a state that can be arrived at via different paths in the state space, and because the Go board has natural symmetries, we can treat symmetric states as transpositions as well.

When a state has not been visited yet it is given infinite priority in UCT. This is naturally not desirable since it produces a kind of breadth first search. A few ways of dealing with this problem exist:

1. First play urgency – perhaps the simplest of the methods. We apply a fixed value threshold for preferring a selected state instead of a yet to visit state. The value may be context based.

2. Progressive widening – where the number of possible states to consider widens with the number of visits to the parent state.

3. Progressive bias – an external algorithm is used to initiate the statistics of not yet visited states. It is called progressive because it's impact in the early simulations is lessened over time.

Most MCTS based Go programs use a progressive bias selection with states initialized via domain-specific heuristics.

Regardless of the conceptual formulation of UCT, the underlying structure need not be an actual tree, nor its nodes merely hold the quality of a game state.

3.3.3 UCT with transposition tables

Upper Confidence Bounds for trees with transpositions table (UCT+T) algorithms try to minimize the space required for the search, as well as have it converge faster to the best solution, through the detection of transpositions. Some programs also treat different but similar states as transpositions under the assumption that very similar states will have somewhat similar outcomes.

A game state with no context of the whole game (and thus unable to be used to detect *superkos*) can be represented in vector or exhaustive form. Vector form is

usually preferred for humans when also representing the flow of a game. When concerned only with the current state of the match an exhaustive form is usually easier to recognize. For a computer it is also easier to manipulate, which makes exhaustive representations dominant in the most performance critical parts of computer Go programs.

Vector based representations however have interesting properties that are worth investigating. It can be noticed that all plays between the start of the game, a capture or a pass can be reordered without loss of generality, which can be used for detection of transpositions in a tree based structure. This can hint to the use of a graph of captures and passes, with each node composed of sets of plays to be played in any order.

3.3.4 All-Moves-As-First

The currently best computer Go programs use an improvement on MCTS called RAVE, which is the logical next step from AMAF.

In MCTS, at the end of a playout, we start the back propagation of the playout outcome, updating the statistics (wins and losses) pertaining to the specific transitions that were part of the sequence of plays simulated. It is easy to imagine that, while this will quickly cover the possible states and transitions in the first few plies of the game tree, most deeper transitions will go seldom visited.

With AMAF we keep a different group of statistics of the transitions of all the sequences that followed *after the current state* that were played by the current player, therefore disregarding the state that actually originated them. This strategy works under the assumption that, while plays played at different turns may have drastically different outcomes, they still are biased to the expected outcome from Monte Carlo simulations.

Furthermore the statistics are also kept from the random playout phase, where a lot of transitions, if not all, are visited. This means we can potentially gather outcomes for all $19x19$ transitions and update all the states in the simulation sequence. Algorithmically we are using the same, perhaps recursive, method of traversing the tree from MCTS, and being able to update nodes without the extra work of looking up transpositions or recording the relationship between states.

Since captures may happen, the same board position can be played at by both players. Petr Baudiš [Bau11] suggested only updating the AMAF information for the player that first played there, since the value of this information decreases with the distance from the state.

Having recorded the wins and losses for transitions from children moves, we can adjust our selection policy to guide the search not only based on the Monte Carlo outcomes but also AMAF outcomes. As introduced previously the UCT1 selection

policy is as follows, where $Q_{s,a}$ is the average of the outcome of the playouts following transition a from state s, and $c_{n,m}$ the exploration vs exploitation bias parameter of UCB.

$$I_s^{UCT1}(t+1) = \underset{a \in A(s)}{\operatorname{argmax}}(Q_{s,a}(t) + c_{N_s(t),N_{s,a}(t)}) \tag{3.7}$$

In the adopted AMAF variant, α-AMAF, we now have a weighted average between the MCTS outcomes and the ones obtained via AMAF propagation, $Q_{s,a}^{AMAF}$, derived in the same fashion. We call this weighted average $Q_{s,a}^{\alpha-AMAF}$:

$$Q_{s,a}^{\alpha-AMAF} = \alpha \times Q_{s,a}^{AMAF} + (1-\alpha) \times Q_{s,a} \tag{3.8}$$

The constant α is the bias between the MC and AMAF outcomes and can be arrived at empirically. We are, however, attributing the same weight to both sources of knowledge (MC and AMAF) regardless of how well explored the state is. RAVE corrects this by dynamically selecting the value of α.

3.3.5 Rapid Action Value Estimation

In RAVE, the objective is to select a factor $\beta_{s,a} = \alpha$ for each state transition based on our confidence on the amount of information gathered. A state that has been visited a lot should have enough information for us to trust the playout outcomes. A state that has seldom been visited however has little information if not for the shared AMAF statistics.

What we refer here as RAVE is called in some sources MC-RAVE, calling RAVE the simple application of α-AMAF, with AMAF not differentiating outcomes obtained from MC playouts from AMAF from AMAF sampling.

$\beta_{s,a}$ was derived by Gelly and Silver [GS11] to be, under the assumption that the game is even:

$$\beta_{s,a} = \frac{N_{s,a}^{AMAF}}{N_{s,a} + N_{s,a}^{AMAF} + 4 \times N_{s,a} \times N_{s,a}^{AMAF} \times b^2} \tag{3.9}$$

With b now being a constant not dependent on the state or transition. In Matilda the value of b was arrived at empirically. Some testing can be seen in Figure 3.3 for 9x9 boards. All things staying the same we need to update our UCT formula to reflect the use of the α-AMAF outcome average instead of just playout outcomes.

AMAF contributions are taken into consideration regardless of the depth they are found, relative to the state being updated. This is naturally not optimal, since the impact of a move made the next turn is more likely to be accurate than one made one hundred turns later.

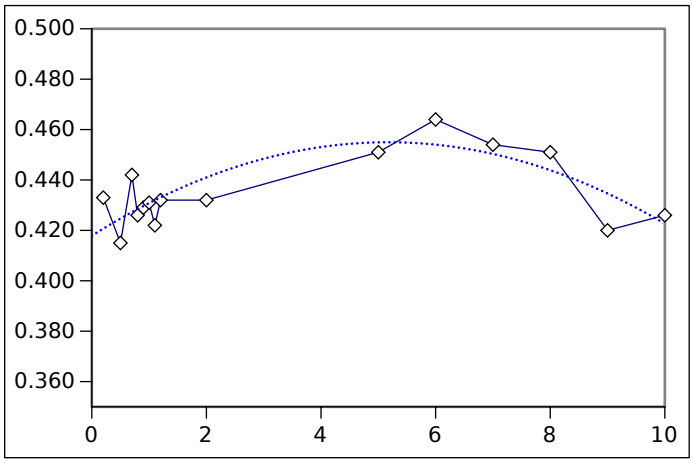


Figure 3.3: Win rate by MSE b constant

Some programs deal with this by limiting the contribution of AMAF traversions to a certain maximum depth; or until a capture occurs. Matilda instead uses a method of contribution smoothing. Originally the AMAF quality $Q_{s,a}^{AMAF}$ is the average of the outcomes gathered via AMAF (first played by the player in a subsequent state to the current one). In Matilda Equation 3.12 is used for each update instead, favoring closer AMAF samples.

$$c = 1 - tanh(d/D) \tag{3.10}$$

$$N_{s,a}^{AMAF} \leftarrow N_{s,a}^{AMAF} + 1 \tag{3.11}$$

$$Q_{s,a}^{AMAF} \leftarrow Q_{s,a}^{AMAF} + c \times \frac{z - Q_{s,a}^{AMAF}}{N_{s,a}^{AMAF}} \tag{3.12}$$

Where d is the depth the state was found relative to the state with the AMAF value being updated, and D is an empirically tuned confidence factor – of the confidence of AMAF contributions depending on their distance to a state. c is therefore the confidence on outcome z. The impact of this modification, according to constant D, is shown in Table 3.2 to be not yet evident in tests run in 9x9. Note the default behavior is equivalent to $D = \infty$.

D	Win rate	Games
8	41.8%	3427
10	43.1%	3984
11	42.2%	3540
12	42.5%	3736
14	41.8%	3427
15	42.2%	3246

Table 3.2: Win rate by D constant

3.3.6 Further MCTS improvements

Heuristic MC-RAVE

When using domain knowledge to initiate the state information, for instance as progressive bias, the AMAF statistics should also be initiated. When these statistics are initialized from the MC statistics they are called by Gelly and Silver Heuristic MC-RAVE [GS11]. It is argued that Heuristic MC-RAVE may replace the use of an exploration-exploitation bias altogether (thus the acronym MC instead of UCT), and this is done in programs like MoGo and Pachi. Empirically, we still obtained better results using both the UCB term and RAVE.

The MC statistics can be initiated by a function of measure of simulation equivalence: the number of wins or losses suggested to be equivalent to the confidence on the feature quality. These functions are usually learned for a great number of simple, tactical features – such as a play being a capture, or the number of liberties the resulting group has.

Virtual loss

When MCTS is parallelized with many worker threads sharing the game tree, and the MC and AMAF qualities are only updated after the playout phase, it often suffers from different threads following the exact same transitions before their quality having been updated. A technique to prevent this and instead promote exploration of another branches consists in adding a virtual loss to each state prior to continuing the simulation. This virtual loss is later corrected if the playout was actually won.

Using transition qualities and visit counts instead of wins and losses, we need to modify the state update method from [GS11] to incorporate the virtual loss. This is done by performing Equations 3.13 and 3.14 before continuing the search. When returning from the playout during the propagation of the simulation outcome we

execute Equation 3.15. Notice the current value of $N_{s,a}$ is used.

$$N_{s,a} \leftarrow N_{s,a} + 1 \tag{3.13}$$

$$Q_{s,a} \leftarrow Q_{s,a} - \frac{Q_{s,a}}{N_{s,a}} \tag{3.14}$$

$$Q_{s,a} \leftarrow Q_{s,a} + \frac{1}{N_{s,a}} \tag{3.15}$$

Horizon effect

If memory is short for the capability of the program to expand new states in the current system, a strategy of delayed state expansion can also be used. It consists in only expanding newly arrived at states after a certain amount of visits. If the memory available is expected to run out in the middle of a single turn, this is an attractive alternative to interrupting the search or falling into what has been coined the horizon effect.

The horizon effect is an anomaly of sample based searches that do not have the ability to refine the search past a certain point [Bau11]. Even though a sequence of plays on average has a certain probability of positive outcome, it could be easily refuted if we direct the search to the most promising plays. If we cannot then we are essentially reinforcing our biases. The danger in doing this is in tree search poisoning, where an easily refutable position returns a high rate of success and erroneously diverts the simulation of the rest of the game tree.

3.3.7 Play criticality

First described by Rémi Coulom and Seth Pellegrino, the notion of criticality refers to biasing the exploration towards board intersections whose ownership correlates highly with winning the match. This introduces the upkeep of a new group of statistics, now on position ownership. Up to now we were keeping for each play/intersection the MC quality and number of visits, and the same for AMAF. If we add information on the ownership of intersections at the end of the playouts, and on the relationship between ownership of the intersection and winning the match, we can measure its criticality based on their covariance.

Petr Baudiš introduced the application of criticality for progressive bias guiding [Bau11] by incorporating it in the formula of the RAVE bias. First Coulom's criticality formula was rewritten to reduce the information necessary:

$$C_{Pachi}(s,a) = P_{s,a}^{winown} - (2 \times P_{s,a}^{plrown} \times P_{s,a}^{plrwin} - P_{s,a}^{plrown} - P_{s,a}^{plrwin} + 1) \tag{3.16}$$

$C_{Pachi}(s,a)$ is the criticality of intersection a from state s. $P_{s,a}^{winown}$ is the probability of the simulation winner owning the intersection. $P_{s,a}^{plrown}$ the probability of the player owning the intersection, and $P_{s,a}^{plrwin}$ the probability of the player winning the simulation. You'll notice $P_{s,a}^{plrwin}$ is equivalent to $Q_{s,a}$ from UCT.

The modified RAVE formula from [Bau11] then complements $\beta_{s,a}$:

$$N_{s,a}^{CRIT} = |C_{pachi}| \times N_{s,a}^{AMAF} \tag{3.17}$$

$$N_{s,a}^{AMAF+CRIT} = N_{s,a}^{AMAF} + N_{s,a}^{CRIT} \tag{3.18}$$

$$\beta_{s,a} = \frac{N_{s,a}^{AMAF+CRIT}}{N_{s,a} + N_{s,a}^{AMAF+CRIT} + 4 \times N_{s,a} \times N_{s,a}^{AMAF+CRIT} \times b^2} \tag{3.19}$$

And on the transition quality calculation it affects the AMAF quality as well. We are therefore inflating the quality of critical intersections to be owned by adding virtual wins, and adding losses to stable intersections. We use wins/visits instead of precalculated qualities for AMAF, and techniques based on biasing through AMAF, because of the performance cost of calculating weighted averages. $W_{s,a}$ is therefore the observed number of wins from following transition a from state s.

$$W_{s,a}^{CRIT} = \begin{cases} N_{s,a}^{CRIT} & \text{for } C_{Pachi}(s,a) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.20}$$

$$Q_{s,a}^{AMAF+CRIT} = \frac{W_{s,a}^{AMAF} + W_{s,a}^{CRIT}}{N_{s,a}^{AMAF+CRIT}} \tag{3.21}$$

$$Q_{s,a}^{RAVE} = \beta_{s,a} \times Q_{s,a}^{AMAF+CRIT} + (1 - \beta_{s,a}) \times Q_{s,a} \tag{3.22}$$

The use of criticality by Petr Baudiš was also bound by a minimum number of visits, chosen empirically [Bau11].

3.3.8 Play effectivity

Also by Petr Baudiš the idea of play effectivity is explored, as the likelihood of a play being successful measured by the stone remaining on the board and making its neighbor intersections also belong to the player – its *local value* [Bau11].

Apparently for performance reasons using play effectivity as a biasing method is not performed in Pachi. In Matilda the primitives developed for state manipulation produce little overhead for effectivity biasing, and it was therefore experimented with.

Of the methods investigated for incorporating play effectivity a similar one to the previous method for criticality was found best. We modify the AMAF statistics and consequently the RAVE biasing and quality of the transitions via AMAF. The definition of local value from Pachi was lightly modified to:

$$lvalue_{s,a} = \frac{1}{3} \times P_{s,a}^{plrown} + \frac{2}{3N} \times \sum_{n=1}^{N} P_{s,a_n}^{plrown} \tag{3.23}$$

With $P_{s,a}^{plrown}$ the probability of the player owning intersection a and the end of the game from state s, and P_{s,a_n}^{plrown} the probability of owning neighbor intersection n. N is the total number of neighbor intersections (plays near the border of the board have less neighbors).

The AMAF values can then be updated in a similar way to the use of criticality:

$$N_{s,a}^{E} = c \times |lvalue_{s,a} - \frac{1}{2}| \times N_{s,a}^{AMAF} \tag{3.24}$$

$$N_{s,a}^{AMAF+CRIT+E} = N_{s,a}^{AMAF+CRIT} + N_{s,a}^{E} \tag{3.25}$$

$$Q_{s,a}^{AMAF+CRIT+E} = \begin{cases} \dfrac{W_{s,a}^{AMAF} + W_{s,a}^{CRIT} + N_{s,a}^{E}}{N_{s,a}^{AMAF+CRIT+E}} & \text{for } lvalue_a > 0 \\ \dfrac{W_{s,a}^{AMAF} + W_{s,a}^{CRIT}}{N_{s,a}^{AMAF+CRIT+E}} & \text{otherwise} \end{cases} \tag{3.26}$$

With c some constant for tuning the contribution to RAVE. Alternatively RAVE schedule constant b can be tuned instead.

After experimenting with several definitions of point local value and integration with RAVE we were unable to produce positive results in 9x9. The above description produced the best results observed. To notice is that the accuracy of play effectivity is lessened by the fact that real MC simulations may not end with the board full, and the detection of eyes for the purpose of $lvalue$ calculation was not performed. For convenience our experiments also used the same limit on minimum number of MC samples for effectivity and criticality, but there is no basis for their relationship.

3.3.9 Last Good Reply with Forgetting

Last-Good-Reply with Forgetting (LGRF) is an improvement for MCTS that works surprisingly well [BD10]. A Last-Good-Reply policy (LGR) follows the last reply of the opponent that brought her a win after the current play selected: if in state s_1 we opt for play a_1, and the last time we followed a_1 we lost, then we have kept a good

reply a_2 by the opponent that can be followed. We can extend this to include more than the last transition as discriminant for reply a_2.

The act of forgetting (LGRF) then adds that if a reply a_2 was actually poor, it is cleared, and the default transition selection policy is used instead when we later follow transition s_1, a_1. Exploration is not harmed since for a sequence that brought victory to one player, the other one's plays are forgotten.

Let's consider an example applied in Go: for $\delta(s_i, a_j)$ meaning transition j played in state i by the black (B) or white (W) player δ.

In a simulation with plays $B(s_1, a_1), W(s_2, a_2), B(s_3, a_3), W(s_4, a_4)$ resulting in a victory for white, transition $B(s_1, a_1)$ would save or update a_2 as best reply, and $B(s_3, a_3)$ would save a_4. All other replies would be left unchanged in simple LGR. In LGRF, $W(s_2, a_2)$ and $W(s_4, a_4)$ would clear their replies (if they have any).

Besides being inexpensive by avoiding performing transition selection computations in some states, LGRF experimentally achieves good results with a fixed number of simulations. It also adds little memory overhead since for each play we only store one reply.

3.3.10 Dynamic komi

The MCTS algorithm has a big weakness related to extreme situations: when the current state of the match favors one player in particular, the great majority of playouts will either be very positive or very negative. Since the outcome of a playout is only in terms of winning or losing, a naive MCTS implementation might give up on uphill battles and ease up on favoring positions. Strong players often do not make big risks if they know of a safe way to victory; but MCTS programs may take this one step too far – being satisfied with a 0.5 points victory – and a small error in its estimates can turn the game around. This is surprisingly prevalent in MCTS programs.

There are a few possible solutions to this. Typical alternatives to just wins and losses are taking into consideration the score difference or playout depth. Another solution can be a form of situational compensation, that attempts to even the simulation outcomes by observing past outcomes. One of the ways to accomplish this is with a shift of the board evaluation function; modifying the reward distribution. This method was called dynamic *komi* because the *komi* is a constant in the typical board evaluation function that can be easily adjusted.

This method was first discussed in the computer-go mailing list, and elaborated upon in Pachi [Bau11]. Most important was the conclusion that mixing MC and AMAF values from before and after shifting the *komi* did not introduce significant inconsistencies.

The idea is to, given the distribution of scores using the current *komi*, change the *komi* to favor the underdog. After each change the *komi* is kept as is for a set number

of simulations to give the observer time to stabilize the new score distribution. This method was called Value-based Situational Compensation (VBSC). It defines a range for *komi* stability. Values outside that range cause a linear increase or decrease of the *komi*.

We've experimented with different thresholds for passing, *komi* offset limits and methods of preventing score flapping (single changes of the *komi* can change the win rate drastically in the endgame) yet could not observe a strength improvement in even games in 9x9 boards with 3 seconds time limit per match. Dynamic *komi* is notorious for being difficult to tune, and not always applicable. In the future we plan to test in larger boards and time settings, as well as in handicap games. Stone handicap games constitute the situations identified where the use of dynamic *komi* is supposedly more advantageous.

3.3.11 Domain knowledge

Up to this point we've shown techniques that attempt to make better use of the time available by directing the search to the most promising plays. Most of these techniques were related to the transition selection phase of MCTS. They, however, didn't make use of any domain knowledge.

One of the ways heuristics in Go can be used to influence MCTS is by initializing newly expanded UCT nodes with prior values – as introduced already as Heuristic MC-RAVE. Just like a human player would be tempted to look at the most obvious plays first, even heuristics with little accuracy can contribute to the search.

If our expert knowledge is advanced enough then we can prune out entire branches of the game tree, which is extremely advantageous. Common examples are prohibiting altogether plays in own proper eyes[1] and bad *self-ataris*[2]. Both of these can occasionally be good plays, especially *self-ataris*, and tactical analysis is often needed. For performance reasons the same pruning often can't be made in the playout phase of MCTS.

To further restrict the branching factor, plays far from a stone may also be disqualified. An exception has to be made, of course, of starting plays. Passes may also be disqualified either altogether (while there are still legal plays) or until the only plays are liberties of groups in *seki*.

Some of the heuristics commonly used in the initialization of states with progressive bias are shown bellow.

[1]The safest eye definition, i.e. intersections that are surrounded by four adjacent player stones, plus at least three in the diagonals. If close to the border then it must be instead completely surrounded, since both players can use the border to their advantage.

[2]Many *self-ataris* are poor plays, but they can also be *throw-ins*, filling eye space of dead groups and used to allow the creation of *bulky five nakade*.

1. Even-game heuristic – assumes the players are even in strength (attributes an even quality rating for all legal plays).

2. Edge heuristic – plays in the edge and the corners of the board are discouraged.

3. *Nakade* heuristic – *nakade* plays are encouraged.

4. Avoid capture heuristic – plays that save a friendly group are encouraged.

5. Capture heuristic – plays that capture an opponent group are encouraged.

6. Pattern based heuristics – patterns are tested centered in the candidate play. A precompiled database is used.

7. *Joseki* heuristics – *joseki* plays are encouraged. A precompiled database or tactical solver is used.

8. *Tesuji* heuristics – *tesuji*[3] plays are encouraged. A precompiled database is used.

9. Random playouts – play a number of playouts for each prospective play. This may be used to estimate ownership – focusing the exploration in more contested parts of the board besides providing more MC outcomes sooner, delaying UCT state expansion.

10. Grandparent heuristic – information from the transitions in the grandparent node is inherited; under the assumption that the same transition for the same player in a similar situation will have a similar outcome. Where this heuristic falls short, however, is in the fact that for the game state to have been just visited, it means we haven't explored the grandparent very well either; which means we are mostly just propagating prior values across the tree. This heuristic works best if we are delaying UCT state expansion, either by the random playouts heuristic or using a minimum of MCTS visits before expanding new tree states.

The use of prior values is usually implemented as progressive bias, though it could potentially be used as criteria for progressive widening. While this constitutes move selection biasing, complete pruning can also be done when the features overwhelmingly discourage a play.

[3] *Tesuji* are well-studied, short, local sequences of good play that may not be obvious to see but lead to a capture or strategic advantage.

3.3.12 Playout phase

Up until now our playouts consisted in just random play – i.e. uniformly selecting a legal play that was not inside an own eye; and playing until the end of the match or stopping early with a mercy threshold or maximum play depth. Discouraging playing in own eyes is what many authors call *light* playouts. Conversely the term *heavy* is used for more tactically rich playouts.

The use of a mercy threshold consists in ending a playout early if the difference in match outlook exceeds a certain amount. The outlook is usually given by the difference in number of stones plus *komi* (including dynamic *komi* offset). Playouts stopped by mercy threshold can return their estimated score immediately. Furthermore matches stopped by reaching the maximum match depth may also be counted as valid playouts. It is necessary for playouts to have a maximum depth when not testing for repetitions.

Improving the strength of a computer Go program by improving the default policy strength is a dark art. Increases in playout strength do not necessarily correlate with overall program strength; it appears that minimizing the biases is much more important. Even so, some techniques have been useful over the years for non-random playouts, that prioritize for instance plays close to the previous play. The matching of 3x3 patterns has also become a popular heuristic.

In this regard two approaches would become widespread for implementing a random playout policy. One approach would be to learn the weights of different patterns by observing the selected plays from real game records; attempting to mimic the players selection. With a probability of selection of each pattern we can perform a probability distribution selection in the policy.

Minorization-Maximization algorithm

To include many sources of information – 3x3 shape and tactical information – it quickly becomes difficult to train each pattern with all this information simultaneously, since many patterns will not have enough samples to be representative.

A solution to this problem is in dividing and training different sources of information – features – separately and then combining them for estimating the quality of the candidate play. Equation 3.27 exemplifies the probability of selecting a play that has feature values 1, 3 and 4, against all other combinations of features present at the state being evaluated.

$$P_{1,3,4} = \frac{\gamma_1 \times \gamma_3 \times \gamma_4}{\gamma_1 \times \gamma_2 + \gamma_1 \times \gamma_3 \times \gamma_4 + \gamma_3 \times \gamma_4 \times \gamma_5 + \ldots} \tag{3.27}$$

Each candidate play is a team of feature values. The same feature can only appear once (or not at all) and assume one value; but different teams may share feature

values. This constitutes a Bradley-Terry model which has been subject to a few algorithms for the learning of a default policy for Go, like Simulation Balancing [HCL10] and Minorization-Maximization (MM) [Cou07].

With MM we learn a model for the weight of parameter values by Bayes inference. After each run – where we observed many competitions (game states) – we update the involved feature values i:

$$\gamma_i \leftarrow \frac{W_i}{\sum_{j=1}^{N} \frac{C_{ij}}{E_j}} \tag{3.28}$$

With W_i the wins of the feature value (the number of cases where it appeared and was selected), N the number of competitions where i was present, C_{ij} the combined strength of the participants of is team and Ej the sum of the strengths of the participants at competition j.

Another approach to heavy playout policies consists in programmatically encoding the information of tactical features, and their weights. Usually they consist in a few tactical checks, like if a play is a capture or safe[4], and matching in the neighborhood of the last play a series of handcrafted shapes. Play selection is performed uniformly random among the candidate plays. The features that are 3x3 shapes usually only cover some of the most important cases, such as *hane*[5] and *cutting*[6]. This was popularized by MoGo [GWMT06].

To further speed up the playouts feature checkpoints are also used, where less priorital plays are only considered if the previous tactical checks are not satisfied. If no plays are selected across all checkpoints a random legal play is chosen.

In practice this solution has proven very effective, at least in small boards, even though the solution is simple and quick to implement. The downside is that it requires expert knowledge in codifying the handcrafted playouts.

Evolutions and combinations of these two approaches exist, such as using MoGo style playouts with trained 3x3 pattern qualities, or stochastically skipping some feature checkpoints in MoGo style playouts.

Playouts with a lot more tactical evaluation and larger patterns are also often called super-heavy playouts, and have been successfully applied in some programs. The playout policy is also often used for prior values of UCT itself, either by itself or with extra consideration depending on how heavy it is.

Instead of guiding the playout phase, we could also adopt a strategy of removing the playout phase altogether, replacing it with an algorithm that can accurately estimate the outcome of the match. The difficulty is in finding such an algorithm that is

[4]In the context of playout policies a play is *safe* if the resulting group has at least two liberties.

[5]A play where the stone *reaches around* the other.

[6]A play where the apparent contiguity of the stones is interrupted, because they were not strongly connected (by adjacency).

a good classifier overall (with no weaknesses to particular sub-problems) and is fast to compute.

3.4 Artificial neural networks

ANN have been used in the last decades as a simple solution for discovering the patterns present in some complex data. ANN are systems inspired by the biological neural networks, consisting of a graph-like structure with vertices for neurons and directed arcs for synapses. In the most simple variants, the system features a number of input units and output neurons, with possibly more neurons between these, connecting them; and through the successive application of some transformation functions, the energy signal input in the input units is mapped to a response codified in the output of the outer neurons [Hay98].

The family of ANN used in Matilda is that of the Multilayer Perceptrons (MLP); probably the most thoroughly studied example of ANN.

For an ANN to be trained with supervised learning it is important that it can map the desired response for its input in its structure, to be able to generalize the problem (adapt to examples of the problem that it has never been subjected to) and to account for the noise in the data set. A data set in this context is simply a set of observed combinations of states and responses in the problem the ANN tries to model. A popular method of supervised learning is through an error correction algorithm. In MLP this usually means propagating the observed error at the outputs through every neuron, correcting the weights of each synapse.

Using ANN for board wide move selection in Go is challenging:

1. 19x19 Go is an extremely complex game, to be able to learn to play the game at a reasonable strength the size of the network and the number of examples in the data set must be very large.

2. To play Go well the consistency of the quality of play is also very important. Machine learning algorithms tend to privilege situations that occur more often, potentially leaving a Go playing ANN with serious lacunae.

3. Training as introduced above also requires a data set, which may be impossible to obtain. While there are freely available compilations of game records for 19x19 boards with Japanese rules, other board sizes and rule sets are extremely underrepresented.

This work does not attempt to produce a Go playing ANN, instead it aims to produce a quality predicting network for state evaluation, or urgency classifier. A quality

predicting network attempts to classify the quality of each available play, and our objective is that it is accurate enough that it can bias a MCTS, reducing the average regret. It may be terribly wrong in a subset of the problems, but we are looking for an on average positive contribution, that is then verified by a more thorough algorithm (like MCTS). This is a more humble objective than actually having the ANN play the game directly, and therefore should be possible with a more modest sized network and hardware requirements.

The idea of combining these MCTS or state space searches with ANN for the purpose of solving more difficult problems that what can be solved by one of them individually is not new. It has been attempted throughout the years, and is seeing a revival with the new found popularity of CNN.

One of the attempts at using simpler ANN for the purpose of playing Go was in 1998 made by the authors of the SANE method (1996). The SANE method attempted to be (yet another) solution to problems where supervised learning cannot be applied for lack of an effective evaluation function.

In this section the production of MLP via two families of algorithms is presented. The first subsections will explain the genetic evolution algorithm used and the structure of a Go evaluating ANN. Next is presented the solution for MLP with supervised learning with error signal backpropagation.

3.4.1 Genetic evolution of neural networks

As learning algorithms, ANN usually feature the repetitive application of an evaluation phase, followed by a a reinforcement phase. In the evaluation phase an ANN is exposed to an outside influence on its input synapses, calculates its impact throughout the net and outputs what output it may have in its output synapses. In the reinforcement phase the network is modified, usually only in the weight given to each synapse, in an attempt to move its observed output to the expected (desired) output.

If the network output cannot be accurately evaluated then it is very difficult to direct the structure of the network towards the ideal structure. This can be the case in Go, where a move by itself can be very hard to grade, given the long reaching consequences it may have in the later game. If we use an external algorithm to grade every single move then the process will be very slow; if instead we use a set of compiled examples then the network strength at playing Go will be limited by the strength of the examples. The training set also has to be very large to teach Go on a 19x19 board and the training set has to be representative of the many sub-problems present in the game, which is also difficult to guarantee.

Because of this it may be impracticable to apply the usual learning algorithms of ANN to problems such as Go. If we want to use this computer generated solution

in the form of an ANN, and are unable to have it learn the desired behavior the normal way, we can use other algorithms, like genetic evolutionary ones, that should eventually converge to an acceptable solution.

Genetic evolutionary algorithm are, again, decades old algorithms that take inspiration from the natural sciences, in this case from the process of evolution by natural selection. A genetic algorithm works on a population of individuals, or chromosomes, evaluating the fitness of each individual to the problem in question, and then promoting the mating of the best fit individuals. If the population size is limited, the offspring of these individuals replace the worst fit.

3.4.2 Go playing neural networks

Before describing the evolutionary method used it is important to identify the structure of our network. All networks in this work are two-layer perceptrons. They all feature three input units per board intersection, that codify one of eight things: own stone, opponent stone, illegal to play, empty and one liberty after play (self-atari), two liberties, three liberties, four and finally five or more.

An illegal classification strictly refers only to suicides and *ko* violations. Tactically poor plays like playing in own eyes are not discouraged directly. The rationale behind this was that MCTS will already be heavily biased based on such knowledge; we want to observe the impact of the smallest of networks on top of it.

The network neurons are not subjected to any bias parameters and use the hyperbolic activation function:

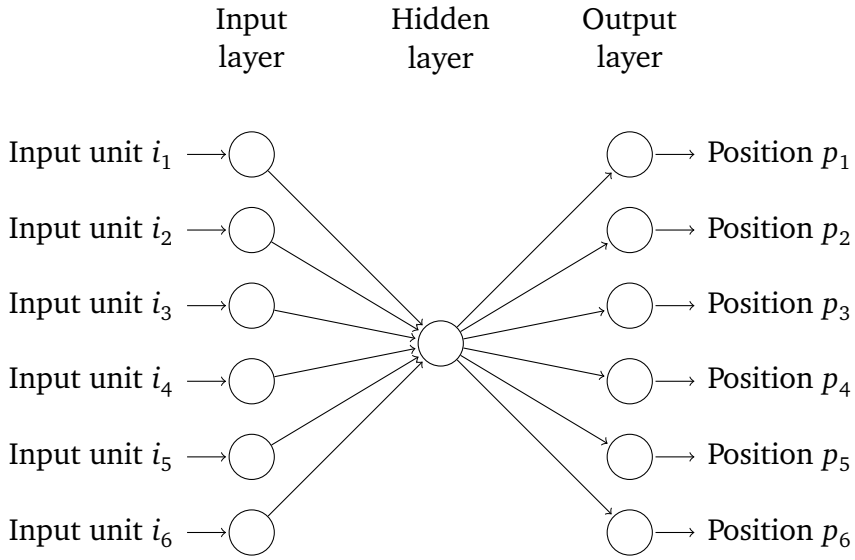
$$f(x) = a \times tanh(bx) \tag{3.29}$$

With $a = 1.7159$ and $b = 2/3$, and output layer target values $\{-1, 1\}$. One output neuron is used per board intersection. This allows the network to output an estimate of quality for every possible play, instead of only the best play. In selecting the best play the neuron whose output is least distant from 1 is selected. There is no codification for passing, instead a constant cutoff point was set where all outputs bellow which were considered worse than passing. This cutoff point is used only when the network is put to play by itself.

An ANN such as this is an exercise on whether the simplest of networks still offers value to a MCTS based program; not whether a more feature rich, deep network is better. At this point in time this is hard to dispute, with the recent results on the use of CNN.

In the hidden layer the genetically evolved networks and the trained networks differ. The SANE method performs genetic operations with the entire hidden layer as its chromosome. Each neuron also has a few connections to the input and output layers. In this work the number of 8 input connections and 6 output connections

is used. Please note that a typical neuron only has one output with multiple inputs subjected to weights. In the SANE method however the connections between the hidden neurons and the output neuron are defined at the hidden neurons. Because of this these neurons will be called simply nodes from now on. These nodes are represented in the figure bellow, showing a node fed by six input units and contributing to six output neurons.



There is no locality for the connections, that is to say, they can connect to positions very far away in the corresponding Go board. The original SANE method also didn't use a constant amount of connections to the input and output layers. A connection could change layers via mutation, even if it meant the node wasn't connected to one layer at all. This was the only detail of this algorithm that this work parted ways with, enforcing a constant number of unique connections to both layers.

The number of nodes in the hidden layer is a subject of experimentation. In ANN more neurons don't necessarily mean better results, and the authors of the SANE method made no hypothesis of a good number of nodes for larger boards.

3.4.3 Evolving neural networks

The above structure for an ANN can be represented in the form of a genetic algorithm chromosome as a fixed length array of definitions of nodes of the hidden layer. Each chromosome is composed therefore of the connections from the input layer to that node and from that node to an output layer node, and associated weights. Since our chromosome is composed of similar parts the operation of crossover – used to generate offsprings of the current population in a genetic algorithm – is possible.

Having defined the representation of our chromosome to be evolved in a genetic algorithm, we can take another look at the SANE method. Richards et al. further SANE by also defining a different genetic algorithm, which presents a symbiotic relationship between the evolution of two populations [RMM97]. When applied to Go, this meant treating each candidate network formed from the node population, as another individual; this time of a network population. Network mutation is defined as choosing one of the composing nodes at random, and randomizing its connections and weights.

When evolving networks there are a lot of similar nodes. If we only evolve the networks without any kind of node sharing, the result will be a lot of wasted effort, since we will only be removing inefficient nodes one at a time when a network is crossed over twice correctly or that specific node mutated. If we share the nodes by the evolved networks but don't have them evolve, we are limiting our genetic material renewal to the mutation operation. By also evolving both populations at the same time we are changing the nodes in all the networks that include them, based on the supposition that a very poor quality node will also be of lower quality in all other networks it is inserted in, and we will be replacing non participant nodes on the basis of a high correlation between participation and contribution to playing Go.

This means the operation of crossover of the network population contributes to the fitness and consequentially the evolution of the node population, by speeding the renewal of its genetic material.

Having said this, the evolutionary algorithm with two populations is described succinctly in pseudocode in Algorithm 1.

This algorithm features the skeleton SANE method, with a final piece that tests a certain stopping criteria like defeating an external opponent. The calculation of the fitness of the networks themselves was tested as three different variants. The first variant, **sane-opt-rnd**, initially used for testing the correctness of the implementation itself, calculates the fitness of a network by playing five matches against a quasi random opponent; the network plays white. The opponent is random except for not filling own eyes and avoiding playing at the corners. The second variant, **sane-opt-gtp**, is similar but the opponent is randomly played (for each turn) either by the previous random opponent, or by an external program via GTP. The opponent used was GNU Go 3.8. The third and final variant, **sane-tournament**, pits the networks against each other to generate their fitnesses. Each network plays at least five games starting first and against randomly picked opponents (besides itself). This means on average each network plays ten matches.

A tournament based approach is reasonable since we are only trying to isolate the best networks in the evolution process; and it already provides a smooth difficulty curve which is important to facilitate the networks evolution. In the **sane-opt-gtp**

```
Result: Go-playing network
randomize node population;
randomize network pop. from node pop.;
while true do
    calculate network fitnesses;
    calculate node fitnesses from network fitnesses;
    sort network and node populations;
    if best network satisfies stopping criteria then
        return best network;
    else
        crossover best nodes;
        crossover best networks;
        two-phase mutation of offspring networks;
        mutation of offspring nodes;
    end
end
```

Algorithm 1: Co-evolution of networks and nodes for Go

this curve is represented by the gradual change from random play to GNU Go play selection. The algorithm starts from 100% random play selection, gradually changes to 100% GNU Go play with level 0, and from then on would increase the level of GNU Go.

For all the variants the algorithms disallow *superkos* where possible (GNU Go, the GTP opponent used, allows *superkos* by default so it wasn't enforced in its turn) and if a match does extend beyond reason, both players are penalized.

The unified stopping criteria gives a certain minimum quality of play expected, which is useful for comparing the fitness variants. In the solution chosen, the best network by fitness is pitted against an increasingly harder combination of random play and GNU Go play. The program stops when a network is capable of winning every match. This means we can stop testing early if an early match is lost, which is useful because more random matches are also much faster to compute. The slow nature of pitting the network against an external algorithm is lessened by running it only once per generation, of every few generations.

Using ANN to play entire matches, the current structure dictates the exact response of the network, and therefore if two ANN enter in a *superko* cycle (for instance during the fitness calculation) they will be stuck until the maximum search depth, which is the worst scenario in terms of performance. This actually happens every few hundreds of matches among young ANN. By also enforcing the *superko* rule, the repetitive plays are not played and the match can continue.

Experiments

This network formulation is admittedly insufficient for competitive Go. Although it is simple enough to obtain good results in small board sizes, the complexity appears to rise tremendously the larger the board. More advanced, modern approaches, usually attempt to develop more local networks, and have them applied or generalized to bigger boards; this usually is much more cost effective than attempting to tackle 19x19 boards head on. Secondly, the rate of increase in the SANE method parameters, like the population size and neurons by network, is also not well understood. This requires extra experimentation which is time consuming because of the already time costly evolution. Finally a third problem deals with the ability to defeat an external opponent. Regardless of this being done in the fitness calculation stage or in the stopping criteria stage, the external opponent is probably much stronger than anything an ANN this small can hope to be, and a method of slowly increasing the difficulty to aid the evolution is non-existent. Adding randomness to the play selection is a poor substitute because it produces noise in the relationship between network fitness and match outcomes, which in turn delays the convergence of the population to its desired quality of play.

For each of the three fitness variants of the SANE method an input codification of 2 bits and 3 bits was tried. The difference in quality was marginal. The **sane-opt-rnd** after 1300 generations showed no improvement in defeating its random opponent. The maximum win rate observed was 86%. The **sane-competitive** version did not generate champion networks capable of defeating 90% random / 10% GNU Go consistently and also seemed to stop improving or do so very slowly. the **sane-opt-gtp** was less thoroughly tested than the others two variants because of the large amount of time needed to compute the fitness against GNU Go; after 32 generations it was also in the quality level of only winning against 90% random / 10% GNU Go players. Besides the difference in number of input units per board position, some other parameters were tried, like a larger number of neurons in the neuron pool and per network blueprint. The values ranged from 2250 to 3750 hidden layer neurons per blueprint and 12750 to 21000 total neurons in the population.

In all three variants the best network of one generation was rarely present in the next, which suggests the fitness functions were poor indicators of quality. This was expected, since we in the end are trying to improve on the quality of individual moves by the overall result of a match.

Although the networks generated are very poor overall players, that is not to say whether or not it is a result of a few poor plays; it is important to understand whether the network is improving as a whole as an urgency classifier. To better judge this, after each n generations the champion network was also subjected to a test run. In this test run the network is evaluated with hundreds of thousands of combinations of inputs and expected outputs, to gauge statistics like the median rank of the desired

play. The results are presented at the Results section of the Implementation chapter, contrasting them with the trained MLP networks.

3.4.4 Training multilayer perceptrons

In this method the network is trained one example from the data set at a time: first a forward phase is performed, where the problem state is codified and input at the input units. Next the signal is fedforward until it reaches the output units, which are again checked against another codification for the output of the system. Then, in the reinforcement phase the desired output and the observed output are compared and the error signal generated. This error signal is then sent backwards, adjusting the synaptic weights of the network based on their individual contributions to said error.

There is a lot of literature about training MLP with back-propagation of the error, and this work will not attempt to reinvent the wheel, or go in depth about the math behind the algorithm. In fact this work uses a very simple, generic implementation of the MLP and its training, that given the parameters used should be easy to replicate.

In our MLP the structure of the network is fixed, with only the weights fluctuating. The codification of the inputs and outputs is the same as the one used with the SANE method, with three input units and one output neuron per board intersection. It is the hidden layer and connections that are different. The input layer codifies 3 bits per board intersection; if we want to maintain spacial locality we can say they are three *maps* offering different views over the same layered board position. In our hidden layer we use a single map of size 19x19. This structure is illustrated in Figure 3.4 for 5x5 boards – to the left is a Go board and to the right each square represents a neuron. Every neuron in the output layer connects to all neurons of the hidden layer that are at least a certain maximum distance (d) away. Similarly every neuron in the hidden layer is connected to every input unit in all three maps that is closer than d. For the distance a Manhattan formula is used where the input map of the unit is not taken into consideration. The use of this distance means each neuron has a perceptive field. Since there are two layers of these neurons, we can say that the output layer neurons even have a focal field, where they can perceive more about things closer to them – because they have many overlapping connections – than farther away, where the perceptive fields are stretched and connections more sparse.

The use of the term map should not be confused with its meaning in CNN.

With a very small p it is impossible to learn whole board strategies. Having small values of p also increases the generalization capability of the network, besides having the obvious advantage of being faster to compute. Larger values of p ($p > 7$) were found to be optimal in 19x19 in the experiments performed. Note how for one corner

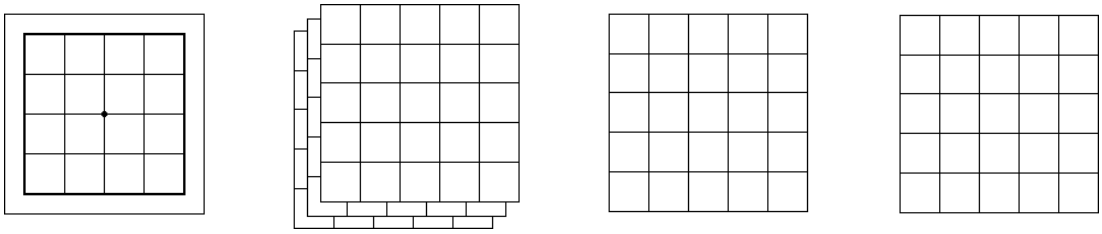


Figure 3.4: 5x5 board and input, hidden and outputs layers

intersection to be able to influence the opposing corner, p must be at least 19. With $p = 10$ each of the 722 neurons has an average of 291 connections.

For training the weights were initialized randomly from $[-2.4/\omega, 2.4/\omega]$ \$[-0.001, 0.001]$. ω corresponds to the number of input synapses of each neuron, and is therefore neuron specific depending on where it is located in the network.

Upon experimenting it became apparent that the network easily entered into a state of saturation, virtue of having many outputs with only one active each time. This also introduced high fluctuation on the network when high learning rates were used.

MLP training typically features either constant learning rates or adaptive learning rates. The use of adaptive learning rates allows faster learning by adaptation of the learning rate so it increases when a high rate can be used, and decreases when a lower rate is needed – because we've passed a minima of the error curve, as detected by the signal of the weight correction [Alm97]. Adaptive learning rates were experimented with for this problem with no success.

Unfortunately in this application of MLP training the neurons are very seldom expected to activate and as such the learning rates would accelerate quickly, causing them to saturate the activation function. In this work saturation was such a problem that we used a constant low learning rate of 0.002, after experimenting with values between 0.5 and 0.0005. Interestingly this value is close to $1/(19 \times 19)$.

Another issue related to the saturation of the network is that the trained network has a lot of difficulty in activating the output neurons output to the target value 1. On experiments with purportedly small training sets it was observed that only with the network overfit did it start to yield outputs close to the target value for the best play. This means that if we are interested in selecting the best plays, we have to either discover the average threshold which is network dependent, or we have to sort the outputs by energy first and then select a fixed number of plays. Another alternative could be to use non-symmetric target values for the output – a kind of *output normalization* as is sometimes used on the inputs.

Given that the board presents natural symmetries, the reader might ask herself whether some method of weight sharing or cooperative learning was used. When applied to fields like pattern recognition, where the data also exhibits similar sym-

metries and even shift invariance (which is important for learning individual Go patterns), CNN often employ these methods. These were identified as possible improvements but due to time constraints, and the low impact of such an improvement (since learning is performed offline) they were not explored. In tasks that require much longer training times such as training CNN these are invaluable techniques.

The results of the training of ANN, and the evolution of ANN using SANE, given as the accuracy of the network to identify the play present in the data set, is shown in the Results section of the Implementation chapter. It is followed by the results of the application of the resulting networks as part of a MCTS algorithm. A description of the data set and verification methodology can also be found there.

3.4.5 Use as prior values heuristic

As explored before, an MCTS can be improved by biasing different parts of the algorithm, and limiting the branching factor of others. In this work we experimented with using a product of ANN computation as an heuristic for prior values of newly expanded UCT states. This solution has been done before in a number of publications on the application of CNN to Go. Maddison et al. use CNN to provide prior values a-posteriori [MHSS14] (optimizing batch processing) and Yuandong and Yan block the state expansion waiting for the processing of the CNN for prior values [TZ15].

When using the two-layered MLP as a prior values heuristic in this work, the board configuration is first fed the input layer of the network. Upon the forward pass, the output energy of the network is sorted by distance to target value (1). Then the legal plays are divided into three groups based on their output signals: the plays in indexes $[0, l/4[$, $[l/4, l/2[$ and the rest; with l the number of legal plays not disqualified by tactical evaluation.

A play, depending on the group it is placed in, has the corresponding statistics initiated differently to promote the exploration of the most promising plays first. The top 25% plays are promoted while the bottom 50% are demoted. This heuristic is only used when the number of legal plays is over a minimum threshold. When the board is very full many plays will be about solving problems of life and death, which this network should not be capable of.

3.5 Time allotting

Up to this point the previous techniques have mostly concerned the evaluation of single board states as to decide on the next play. In real life a computer Go software requires a context sensitive layer.

As introduced with the solution architecture, a computer Go program is often used with a game coordinator, or controller. One of its tasks in ensuring the rules

are followed, which includes time keeping. For time keeping it contains internally two time counters alternatively counting down; ending the match early if one of the counters runs out. This controller program informs the Go playing program of the time available for the entire match, not for each turn. How best to use the time available is a decision often left to the context sensitive layer of the program.

Some computer Go program *think* for what their programmers believe is a short amount of time, as not to bore its human opponent, because they lack more sophisticated time allotting mechanisms. Others distribute the time available uniformly throughout the game, estimating how long it should take. This distribution is usually very conservative so that the opponent can't win simply by playing a very long match.

This behavior may be the most suitable for casual play; but for competitive play may introduce a window of opportunity. In this work we propose a number of time control strategies that if placed in charge of a computer Go state evaluator should produce a stronger quality of play throughout a whole match. The only requirement this introduces is that the execution time of the evaluator is either predictable or its algorithms interruptible at will.

We shall call this extra piece of software a time control module. Whereas before the game coordinator would directly ask the evaluator for its opinion of a certain game state, disregarding the game context of said state, now we have a module that attempts to pace itself for the whole match and take advantage of unsuspecting opponents. A time control module can be more or less complex depending on three factors:

1. The variety of systems of time control used.

2. The support for playing against human players as well as computer programs. There are a number of special considerations for both. Being able to automatically distinguish human from computer players midgame would also be valuable.

3. The algorithms that play the game and how well can their execution be controlled at will, plus the ability to reconcile internal information with human requests like that of undoing the last move.

Being bound by the convention set by the GTP, most programs support the Canadian *byo-yomi* time system. It consists in the specification of three parameters: a main playing time (also known as absolute) and the *byo-yomi*, which includes a number of stones that have to be played in a certain amount of time (*byo-yomi* period). The *byo-yomi* (if present) starts after the absolute time runs out. When all *byo-yomi* stones of the period have been played the *byo-yomi* time is reset for the

next batch of stones. This time system can also be used to support other simpler time systems, like an absolute time per match (sudden death) or fixed time per turn.

Canadian *byo-yomi* system is a variant of Japanese *byo-yomi*, which uses instead multiple *byo-yomi* periods usually of a single stone. Japanese *byo-yomi* is most commonly used among humans. Both methods are supported in Matilda.

In Matilda the allotted time per turn (tt) is given by Equation 3.30. t_b and t_a are the time available for *byo-yomi* and absolute time parts. s_b is the number of *byo-yomi* stones and s_e an estimate of the length of the current match.

$$tt = \text{argmax}(\frac{t_b}{s_b}, \frac{t_a}{s_e/2}) \tag{3.30}$$

$$s_e = \text{argmax}(B_e, S) \tag{3.31}$$

Equation 3.31 is used for the estimate s_e of the length of a match for each player, with S as the board side size and B_e as the number of empty intersections. It is usually more useful than just the number of turns that have elapsed, since games can have varying amounts of captures. This yields a potentially lower than usual game length for 19x19 Go (180 instead of 200), the reason for this is we want to allot more time in the early and midgame of the match than in the endgame, where the branching factor is smaller. The estimate s_e starts low and is then corrected throughout the match if it lasts longer than expected while still in absolute time. It is divided by 2 because each player will play half of the moves.

This is the generic time allotment mechanism used. Other programs also take into consideration the certainty of the algorithms behind them, for instance allotting more time if the MCTS simulations were inconclusive. Doing that, by virtue of the collected statistics, would also be impacted by dynamic *komi* offsetting. MCTS can also be stopped early if the game outlook is very one-sided. This makes it harder for an opponent to attempt to win a lost game by staying in the game.

Another simple improvement to be used with MCTS pertains to the initial population of the UCT+T structures. If a computer program switches from using an openings book to using MCTS, the state tree will initially be empty; whereas at later turns it will already have much information to better guide the search. Because of this the first play using MCTS may be of much weaker quality. To compensate for this its execution can be awarded a bonus amount of time, for tree population.

Adversaries to perfect use of the available time are the difficulty in ending an MCTS at precisely the time allotted, and compensating for network latency. To make matters worse, most game coordinators only inform the program of the available time and time system used in the beginning of the match. If it was also transmitted between plays the program would be better equipped to assess the time already spent, and correct its internal clock values. The GTP also does not include a command for transmitting the expected network latency.

To counter this last issue, Matilda can optionally collect the amount of time elapsed between commands received to estimate the round-trip latency of the communication. This technique relies on the controller program to wait for the responses between sending more commands. If this is not done, it cannot be used reliably, and a constant lag correction is used instead.

3.5.1 Against humans

Based upon the above progressive linear time allotting Equations 3.30 and 3.31 (page 51) we can then further tweak the available time per turn for a psychological advantage (against human players only therefore). Some possible techniques include:

1. Performing time based subliminal cues to influence human play.

2. Identification of game situations that are easier for computers and harder for human beings, and vice-versa.

Please notice this section deals a lot with the psychological and it has no basis but my own experience in playing competitive board games. One such observed experience is that human players can be consistently led to change their rhythm. Among two players, at the beginning of a match there is usually a degree of distrust, caution and anticipation. After the opening stage where playing is mostly automatic, the time used to play leaves a heavy impact on both players. A player that considers herself the strongest will usually tend to play faster than her opponent; likewise a player that sees herself in a bad position will feel pressured to play faster to emulate the stronger player. If one is not careful with her pacing she can be led to increase their rhythm past their comfort zone, and commit uncharacteristic mistakes.

This is something that a time control module can abuse when playing against human opponents. By carefully playing ever so slightly faster, throughout a match that lasts for hundreds of plays, the software can slowly make the human player play faster. This is also to the computers advantage since the game complexity goes down the fuller the board is. On the upside this strategy, besides requiring the knowledge of whether the opponent is a human being, should also be less effective the longer the match, because the human sensibility for the time used per turn is limited.

Another technique related with timing cues is playing one turn unexpectedly slow, or fast. This suggests that a particular play was either more complex than usual, more obvious, or the communication is lagging (if played online). If the play was simple and the program used more than usual time the human player will be left suspicious and overcautious. If it was difficult and played quickly the player will be left alarmed and rethinking the expected sequences of play. Either way the

Simulations	Win rate	ELO difference	Games
1000	50.3%	2	435
2000	69.5%	143	488
4000	82.1%	265	431
8000	92%	424	512
16000	94%	478	468
32000	96.6%	581	468

Table 3.3: Self play with varying number of simulations

computer player has the advantage because humans are usually very visual – that is to say, they benefit greatly from being able to see the current state of the board; and are hampered by thinking many plays ahead. For the computer player however, spending the normal time thinking of its play and then spending an extra amount of time thinking of the response to the most obvious human response is no more difficult. This requires the computer program to be able to continue the MCTS in the background while in the opponents turn. If the human player is unaffected then the quick replies in complex positions may prove costly, however.

Another family of possible improvements deal with identifying the problems human and computer problems have the most difficulty with, and attempting to abuse them. Machine learning can be applied offline to human game records to correlate game situations with blunders committed, or the contrast between playing styles and qualities, such as contact play, stone connectivity and *tenuki* reluctance.

If a game situation can be classified in the difference in complexity for humans and programs, that can also be used as MCTS bias.

To better understand the effect of the time available on reinforcement learning by MCTS, Table 3.3 shows the result of self play with an increasing number of simulations per turn. The opponent is always the version using 1000 simulations. The matches were run in 9x9 with the players alternating colors. Both the win rates and ELO differences are shown to better represent the increase in strength. It can be seen that the increase is not linear with number of simulations.

Chapter 4

Implementation

During this work a computer Go program was produced that uses many of the previously explored techniques, named Matilda. It is distributed as permissive free software. Matilda is directed for competitive play in BSD, Linux and Mac OS X systems with shared-memory architectures. Its only requirement is the use of an OpenMP 3.0 supporting C99 compiler, like *gcc* or *clang*. It also benefits from being used in 64-bit systems, although it was written to be immediately portable to any architecture.

OpenMP is a shared-memory API for facilitating parallel programming [CJP].

Matilda was implemented with safety, minimalism and speed in mind. The focus on safety aims at it being ran and left unattended for long periods of times – without memory leaks or potential inconsistencies – which is important for leaving Matilda playing online without supervision. The focus on minimalism makes it as small a codebase as possible; removing unused code and over-engineered traps that are difficult to maintain. The focus on speed is a consequence of the competitive purpose of the program. It requires a balance to be found between solution abstraction and the performance hit that from it may arise. Most notably was the decision to have the size of the Go board set at compile time, instead of settable via GTP.

Matilda supports odd board side sizes between 5x5 and 21x21. Boards larger than 21x21 would require more memory to store group liberty counts; boards larger than 25x25 are also not supported by the GTP. More important than the board size limits is the fact that Matilda makes use of many external files generated from outside processes. Since those files depend on the availability of game records it is inadvisable to use Matilda with board sizes other than 9x9, 13x13 and of course – 19x19. Both odd and even *komi* are also supported – this makes draws possible. Drawn outcomes are treated the same as losses in the MCTS algorithm (for both players), except that the criticality statistics are not updated.

Only the Chinese rule set with positional *superkos* is supported, which disallows multiple stone suicides and uses area scoring. Many computer Go programs also only

support Chinese rules and then *simulate* other rule sets. Enforcing other methods of *superko* (situational – used in Japanese rules), as well as changing the scoring function to only count territory are simple modifications. The inclusion of the number of captured stones in the scoring function – necessary for territory scoring, used in the Japanese rules – is however more difficult. Specially when transpositions are concerned. We can instead shift the evaluation function scores one point in favor of the opponent. Final scoring between Japanese and Chinese rules usually only differs by one point; this sub-optimal correction should introduce little difference in strength. This or other methods of Japanese rules simulation are as of yet not implemented in Matilda.

4.1 Organization

The project is entirely implemented in C and organized with a shared code base, that includes almost all the functionality of Matilda. On top of it a number of programs are built, one of which the actual Matilda executable for Go playing. Other programs built are used for the compilation of opening books and data sets, for feature extraction from game records, ANN training and unitary testing. This is represented in Figure 4.1.

Other programs were also present in the past, for the genetic evolution of ANN using the SANE method, for the generation of 7x7 rhombus patterns, feature extraction with Minorization-Maximization, etc. These were removed from the main codebase when experiments with them were not pursued further.

4.1.1 GTP and SGF support

Matilda has almost complete GTP version 2 draft 2 support. As is customary, it also adds private extensions to the protocol. This section will succinctly enumerate the limitations of GTP commands that presently do not have full support; and then the private commands added in Matilda.

1. boardsize – the protocol specifies this command must not fail; yet it fails in Matilda if the requested board size does not correspond to the board size the application was compiled for. No fix is expected for this issue.

2. final_status_list – the protocol is ambiguous on whether the command may fail with the option *seki*; which happens in Matilda.

Matilda also supports two private commands for use when connected to the KGS Go Server; seven commands used in GNU Go; two commands to connect with GoMill

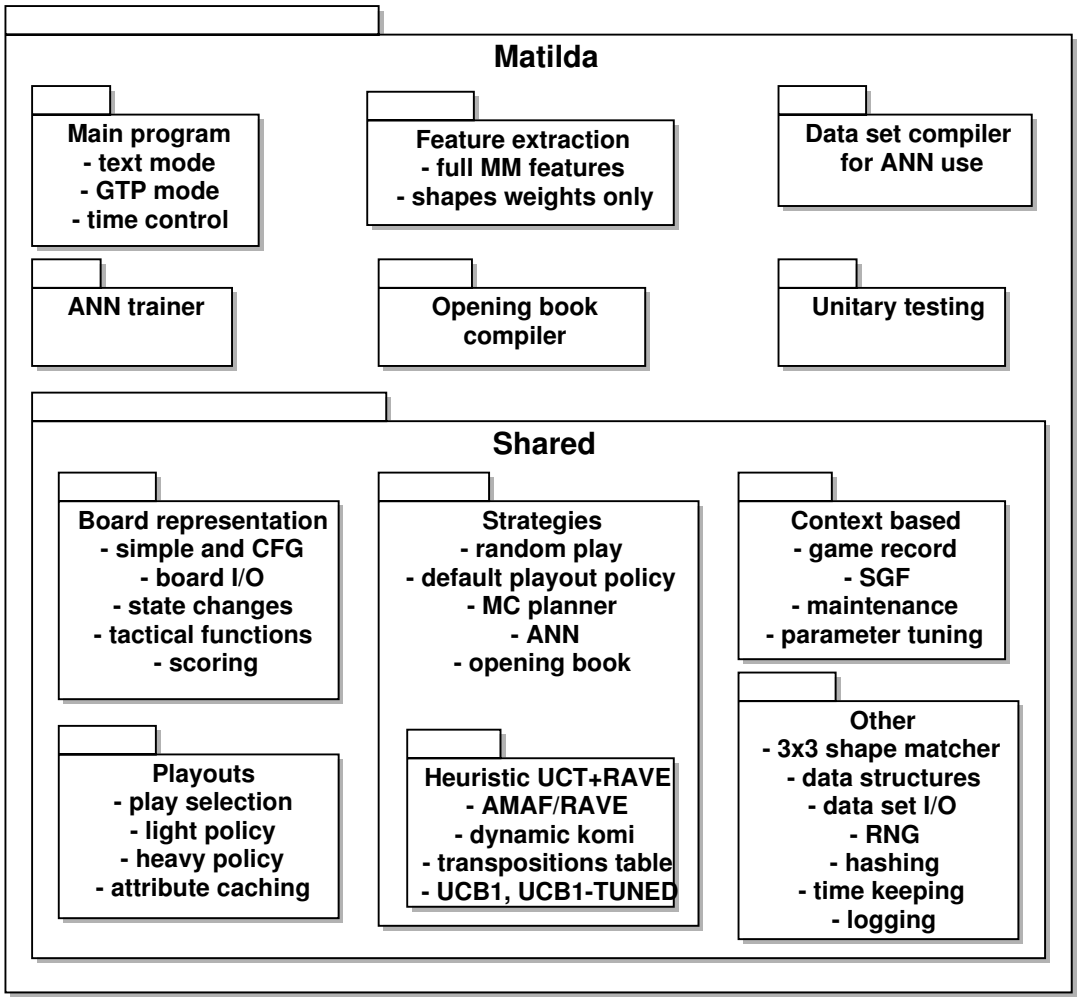


Figure 4.1: Project overview by visibility and concern

– a computer Go tool suite; and six commands specific to Matilda. Two of these last commands allow changing the active strategy, invoked with requests to play.

Most programs don't offer multiple playing strategies, for the obvious reason that one of them will be the strongest, and receiver of the most attention. This is also true in Matilda. Strategies other than Heuristic UCT-RAVE as described in this document, are implemented and can be played against, but generally offer much poorer results: the ANN used almost do not pass until there are no legal plays left, the simple Monte Carlo planner is only a decent player in very small boards, and so on. Most strategies are kept for testing rather than with actual competitive play in mind. For testing they are useful since they do not use a transpositions table and are fast to compute. As such they do not interfere with the execution of UCT+T and its maintenance; allowing self-play with just one instance of Matilda.

Matilda further supports three more private commands for playing Frisbee Go. Frisbee Go is a recently invented variant of Go where play is non-deterministic [Alt15]. Since it is so recent no other free software supports playing Frisbee Go, which makes it difficult to estimate the strength of Matilda at this point in time.

In the past Matilda supported GTP for both client and server applications. This was necessary because the genetic evolution programs required communication with remote GTP-speaking programs. When this functionality was removed Matilda was free to only fulfill the role of GTP server, and the client-side support was removed from the main codebase.

4.2 Book openings

A book of openings is one direct application of domain knowledge. This knowledge in particular was observed, either by a human or computer program, and collected in some way. It codifies rules of $state \rightarrow play$ pairs; sometimes with multiple response plays, with the purpose of quickly providing a strong reply to the most common opening sequences. It uses the simplest form of pattern matching, matching a whole board state to a unique response play. Using the whole board as the pattern means it quickly becomes computationally prohibitive to store and compare a large number of openings. To reuse our rules they are applied invariable of playing player or board symmetry.

We use the term *reduced state* to mean a state representation that has been transformed to be the unequivocal representative of game states that can be transformed in the same state via color changes, board flips and rotations. Similarly *state reduction* is used to mean the act of transforming a game state in its reduced state.

Support for the opening books of the program Fuego is widespread; and they are also supported by Matilda. Fuego opening books are text files with one line per rule, with the state in vector form followed by equally good response plays. This makes

the book more accessible to editing by human beings.

Matilda uses its own format for opening books, but they can be generated automatically from both SGF collections and Fuego-style books. By default rules from Fuego-style books are given higher priority, but it is important to support the compilation from SGF collections to have more versatility in the number of board sizes supported. SGF files, even if not played by human players, can be generated from computer play and help improve the strength of Matilda in uncommon board sizes.

The openings collection is a straightforward, sequential process. A large collection of professional[1] games is searched and analyzed. From it, all board states and corresponding plays are recorded for the first n turns.

Some response plays are more common than others, and the frequency of each appearing in the game collection is recorded. This frequency is used as probability to form a Markov process where the weights are the probability of each of those plays, starting from the initial blank state. Games started with handicap stones are therefore not considered. In Matilda, for this purpose, the game result is not taken into consideration when extracting states and their responses. An opening book can therefore be thought of as containing an ordered sequence of $\{\{i_0, i_1, ...\}\{j_0, j_1, ...\}\} \rightarrow \{(p_0, k_0), (p_1, k_1), (p_2, k_2)\}$ rules, with the *is* and *js* meaning intersections filled by the playing player and opponent, respectively, and the *p,k* pairs the probability of playing at a certain position k. In Matilda the residual probability of playing something other than the top two plays is added to the best play of the two.

When generating the openings book the state in reduced form plus its CRC32 hash are stored together with the two response plays and their probabilities of activation. When playing Matilda includes a hash table indexed by CRC32 hash with lists for buckets. Furthermore when Matilda arrives at a state that cannot be answered just from the opening book, its use is turned off for the rest of the match.

After the collection process has completed, the computer Go software only needs to compare its current reduced position to the saved states in the hash table, and if one of them matches, follow one of the transitions with the associated probability. Having more than one possible transition brings variety to the play which is pleasant for human opponents.

The restrictions on the collection of game states and transitions currently used impose Chinese rules, a minimum of $S + 1$ turns per match, and use plays only from the first S turns. Only states with at least three appearances are used. These values were tailored to the data set used and the needs of Matilda; plus to be reinforced by Fuego-style books as said above. S equals the size of the side of the board.

The openings book for 19x19 was generated from 63861 games which included

[1]The term *professional* is here used loosely, because professional players seldom play in public, making it difficult to collect a sizable sample from that level of play.

537333 unique plays and 68 passes; and from Fuego-style books that contained 28611 unique rules. In total 54541 rules were exported (511403 did not meet the three appearances requirement). The games were played between KGS Go Server players of 6d or better rank. The use of rules extracted from game records did not improve significantly on the strength of the program when compared with using just the rules from Fuego-style books, for the board sizes for which they were available.

4.2.1 State hashing

CRC32 hashing is used in Matilda for full state hashes only. The freely available CRC32 implementation by Gary S. Brown was imported from the XNU kernel source code, made available by Apple Inc. It is licensed as public domain. It was lightly modified to reflect a use in user mode (without kernel imports) and to adhere to the code conventions of the rest of the solution.

Where advantageous, Zobrist hashing was used instead. A Zobrist hash is often used for game state hashes in games like chess and Go. It is a non-cryptographic hash that can represent a game state with a small number of bitwise exclusive or operations over a similar (previous) state; which is particularly powerful in board representations of Go where a play usually only modifies a single intersection. Zobrist hashing is more efficient in the MCTS algorithm and transpositions table, where it can take advantage of the little change between game states. In the comparison of states in reduced form, where hashes have to be created from scratch, CRC32 is used instead. Matilda is capable of generating on-demand collections of 64-bit values with perfect bit distribution to be used as codification tables for Zobrist hashing.

Aside from hashing game positions, smaller, local configurations can also benefit from hashing. A modified form of Zobrist hashing is used for these as well, which consists in the codification of only a neighborhood of a position, invariant of its location in the board.

4.2.2 Joseki book

The previous generation of opening books creates potentially very deep sequences (19 turns for 19x19). Typically by that time the opening stage is over and we are observing a *joseki*; thus computer Go programs often include dedicated *joseki* books.

These *joseki* books consist in a compilation of sequences of play over a specific portion of the board that are believed to be of equal quality to both players. These are usually applied in the corners but it is not unusual to have them extend to the sides. In such cases the openings book per se is smaller, leaving the engagements for the *joseki* book.

Joseki representations only cover parts of the board to increase the probability of rule matching. *Joseki* can consist of sequences of several tens of plays, and are little

influenced by stones in some parts of the board. To represent them with full-board representations is therefore wasteful.

Being deeper in the game tree, extracting *joseki* automatically from game records with confidence is also harder. Fortunately, there are compilations of *joseki* already made for Go in numerous forms, from physical to digital. *Joseki* are often shown in full boards or showing one of the sides. This is not coincidental. The evolution of a *joseki* usually means the players will exchange influence over certain directions of the board. If to one direction the wrong player already has a very strong presence the *joseki* may be completely misread, with the stones encircling themselves.

Because of this a *joseki* cannot be a simple rule with a partial state representation covering a corner or side, but include information, to the extent that is deemed reasonable with the available resources, that hints to the influence of the players in different directions from the *joseki*.

Joseki compilation and evaluation is not present in Matilda yet. When added it should prove invaluable for 19x19 Go.

4.3 Monte Carlo tree search

4.3.1 Move selection

Matilda uses prior values as progressive bias, UCB1-TUNED bias, LGRF, RAVE and criticality in move selection. We call this Heuristic UCT-RAVE for brevity.

When a new game state is found all its legal plays are discovered. To reduce the branching factor a few tactical checks are performed that prohibit playing in own eyes and simple eye shapes, producing ladders, playing in liberties of groups in *seki* and producing bad *self-ataris* (*throw-ins*[2] and filling opponent eye space are allowed). Plays near *nakade* are also prohibited (the non-vital points), as well as in safe shapes that can become eyes, and in long eye shapes of a minimum length. To further reduce the branching of the search when killing opponent groups some fillings of eye shape in *miai* are only left with one possible play. A notion of area of influence was also experimented with, that discouraged the exploration of zones of the board that are already very likely to belong to the player, but it was very crude and is not currently being used. All plays further than four intersections from a stone are prohibited (with the exception of opening plays). This last restriction is aimed at larger boards.

Besides the fast tactical solver for simple ladders – directed at the heavy playouts, two more exhaustive solvers are used in the state expansion of UCT. These discover if a group of stones can be attacked and killed, or made safe, and the plays involved.

[2]Sacrificing a stone to ensure the opponent can't secure an eye.

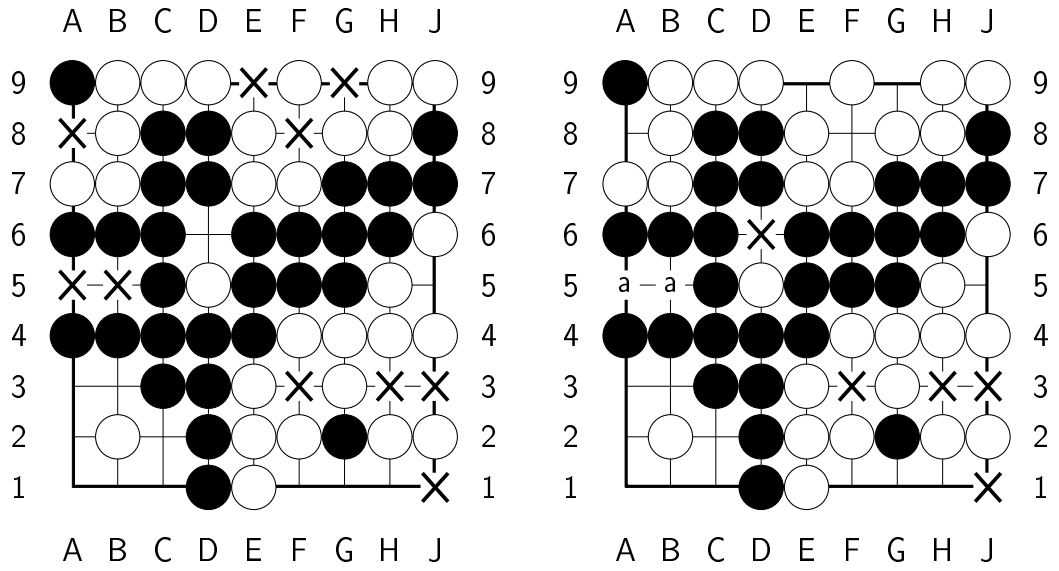


Figure 4.2: Move pruning by black (left) and white (right)

Move pruning is exemplified in Figure 4.2. Plays at intersections marked with an X are prohibited for black to play (left board) and white (right board). It includes disqualifications by the suicide rule. One of intersections **a** is disqualified at random (they are a low importance *miai*). More advanced tactical evaluations that notice, for instance, that the large black group has at least two stable eye shapes and therefore cannot be killed; thus allowing prohibiting plays in both **a** intersections, are not performed.

If a legal play is not disqualified by the above restrictions, it can be either encouraged or discouraged with prior values. The following prior values heuristics are used:

1. Edge heuristic.

2. Even-game heuristic.

3. *Nakade* heuristic.

4. Avoid producing *kos* when another *ko* is not active. The reasoning is in attempting to save *ko* threats.

5. Avoid *self-ataris*.

6. Extension heuristic – favor extension plays that increase the minimum number of liberties of a neighbor friendly group.

7. *Atari* heuristic – favor safe plays that put an opponent group in *atari*.

8. Capture heuristic – discriminated by number of stones captured.

9. Favor safe plays that match the patterns used for the playout policy. These are applied in the entire board instead of just around the last play.

10. Favor plays in the neighborhood of the last. The definition of neighborhood includes the liberties of adjacent groups.

11. Favor plays in the 3rd line and discourage plays in the 2nd and 1st lines[3], if without neighbor stones. This heuristic is only used in larger boards.

12. Use the trained MLP to favor and discourage some of the plays.

Most of these heuristics are in some way present in other programs. The use of large handcrafted patterns, 4x4 and 5x5, that suggested multiple plays was also used for some time in Matilda. With time the most important use cases were identified (defending and attacking eye shape, and reducing the search branching) and the use of these patterns was replaced by the above programmatic checks. The efficient application of patterns of these sizes was very memory costly.

Move grouping

The last experiments performed in Matilda pertain to move grouping. The idea is in not considering the full branching options in poorly visited states, instead grouping plays believed to be similar in outcome.

In Matilda the technique used separates move group statistics from move (play) statistics. When a state is first visited its possible plays are grouped in move groups. In the selection phase of MCTS, UCT selection is performed first among the move groups, and then among the moves of the selected group.

In the propagation phase move group statistics are updated as normal, potentially updating MC and AMAF information several times. If the number of visits reaches certain thresholds, the move group is further divided into two, until the group consists of only one play.

This technique speeds up selection, since in most states only part of the transitions are evaluated. Solutions to move grouping have to ponder on these problems:

[3]Counting from the edge of the board. The 1st line is the border.

1. Play grouping – intuitively grouping plays that are more similar should produce better results.

2. Move group selection – if the UCT selection of move groups is simpler than the selection of plays, the exploration of very expanded states will be hurt.

Saito et al. perform grouping based on simple feature groups [SWUH07] (proximity to last move and to border) but it is sensible to expand the criteria to more interesting features. Childs et al. experiment with overlapping groups [CBK08], which make possible the mixing of non-exclusive features. Good results were obtained with the authors noting that the grouping criteria is very important.

In Matilda we've experimented with random grouping, grouping based on MC quality and grouping based on MC visits. Group selection has been tried using both UCB1 and UCB1-TUNED, with and without RAVE. All results have been very poor; we hope to further research this topic in the future, with more interesting grouping criteria.

4.3.2 UCT+T structure

The details of how the information necessary for MCTS is stored and managed have been omitted up to this point, but a competitive Go program has to place a great deal of attention in the efficiency of the structures and their maintenance – tournament games can be played with long thinking times, and it is important for a program to be able to use the time available correctly.

This problematic can be divided into a number of decisions the developer has to ponder on. Some of these are not independent of the others.

1. Whether to store statistics (wins/losses) on each search tree state or transition.

2. What state information to keep; and if it allows testing for *superko*.

3. Whether to use a tree-based structure, hash table, or something else.

4. How to perform structure maintenance with safety.

5. Whether to detect transpositions, and how.

On storing statistics on transitions

Most MCTS formulations apply to game states. The propagation of playout outcomes affects the win rate at each game state (its state quality) and a transposition occurs when from two states we arrive at the same state. When this occurs and we arrive

at that state, it is already populated with some playout outcomes. It can be said that two sequences of play become one from that state forward.

This formulation, however, makes very inefficient the phase of selection. On selecting the transition from a game state we need to lookup the qualities of all the destinations. If instead we store statistics on transitions instead of states, we benefit from cache locality on this selection. The upside is that we waste memory on yet-to-explore transitions (which are in every leaf game tree node). Two sequences of play now only *unite* after arriving at the same state; since the parents of that state have different qualities – transition qualities – for selecting transitions that arrive at the same state.

State information

Before discussing the detection of transpositions it is necessary to identify the information needed as representative of each game state. if we only store the contents of the board we are not detecting *kos*. Luckily to detect *kos* we only need to add the information on the last capture of a single stone, if any has happened in the last turn.

Detecting *superkos*, however, is much more difficult. Detecting *superkos* requires testing the contents of the board with all the previous states of the game. If we support transpositions, then there are parts of the game tree that are known – the plays played already in the current game; and parts that are variable – the plays played in MCTS that may contain alternative sequences of play that arrive at the same transposition.

Therefore with transpositions, even the best efforts to catch and disqualify *superko*-leading plays in MCTS are not perfect. Most programs do not attempt to tackle *superkos* in MCTS, or do so without detecting transpositions.

Data structure

Having decided on our storage of statistics and state information, we still need to decide on a data structure to hold this information.

When traversing a MCTS game tree we need to quickly select the next state to follow. We can improve this by storing statistics on game transitions instead of states themselves, and we can advance the state by either playing out the game on the Go board, or just following state references.

If we choose to advance the simulation by playing out the game, we need to find if the resulting state is a transposition of a previously found state. With a hash table the process of finding if a state already exists and finding a transposition are one and the same. With a tree-based approach there is an implicit path to a state, and

following a transition is unrelated to finding if the resulting state is a transposition of another.

Because of this, if transpositions play a big role for the problem at hand – i.e. if there is a lot of information to be reused by discovering them – then a hash table based structure is recommended. If they are not that important we can use a tree-based structure and take advantage of faster state transition following.

This problem is made less clear when considering state maintenance. With a hash table approach we essentially have weak links between states. We can free game states knowing full well that if they were to be actually useful they will be remade when a lookup fails to find them. When using hard links (memory references) in a tree-based approach we are impossibilitated from freeing deep nodes: only shallow nodes that have no references to them can be freed.

Structure maintenance

This brings us to the problems of game tree pruning with safety. There are at least three moments when we may want to perform pruning:

1. The game has ended; all memory can be freed.

2. The turn has ended. Upon choosing a play we can free the states that are not descendants of the resulting state.

3. We have run out of memory; we may opt to free part of the game states that we deem less useful.

The first moment is trivial. The second moment is possible with both tree based and hash table structures.

With tree based structures we can traverse the subtree from the state that is now the current state of the game, and mark their nodes. Afterwards we can free all states not previously marked. We require however a method of traversing all states independent of their tree organization.

With hash table structures the most efficient method is in recording the maximum depth each state is found at, and freeing all states bellow depth 2 – for two player games. We are freeing the states of the current player and the opponents because we will be starting a new search after their reply.

The third moment poses the most problems. First we need to be able to estimate the usefulness of a state. Mere depth it is found in the game tree is a poor indicator, since we want to search promising sequences of play deeper. If we attribute usefulness to probability of being traversed, we can free the states that have been traversed least recently.

Such a pruning technique will probably cause many of the recently freed states to be explored again since they were probably in the edges. It is also very difficult to perform this pruning solution with memory safety when using hard links. Because of this, this pruning moment is usually not considered. Programs instead simply stop the search that turn when running out of memory.

Programs that insist on continuing the search but without expanding and adding new states to the tree; essentially only performing more random playouts, are often victims to the horizon effect.

Transpositions detection

Transpositions can be of four types:

1. Permuted plays – we arrive at the same state by following the same set of plays but in a different order.
2. Passing – we arrive at the same state after the opponent passes, if both players share the same game tree.
3. Repetitions – a previous capture made it possible for a *ko* or *superko* violation, and we're not preventing these.
4. Board transposition – two states are shown to be equivalent when performing color flips or matrix transpositions on the board.

Permuted plays and transpositions by passing are transparently detected when using a hash table, but it requires performing lookups at every transition. Using a tree-based structure a common, partial, solution is searching the relatives of the current state for transpositions. This method catches most of the transpositions but is also not very efficient.

Ignoring transpositions, as uncommon as they are, may also bring other benefits such as using separate data structures for each player, thus speeding node lookup when using hash tables.

Implementation in Matilda

Matilda uses a compromise between the options above. It is justified by some general observations:

1. Memory is cheap. Although matches may take a long time, it is very difficult for them to use more memory than currently available for consumer grade electronics. Systems used for tournament play are also considerably more powerful. If running out of memory in long matches, a possible solution adopted by

many programs is in only performing state expansion after a minimum number of visits. Matilda also makes use of this.

2. Maintenance on demand is expensive. It is particularly expensive when considered that it may not bring any advantage, and instead provoke an horizon effect.

3. Transpositions are rare, but they still happen. Transpositions by permuted play are particularly important, although with AMAF/RAVE at least some information is shared.

With this in mind Matilda uses a state representation without direct support for *superko* testing, that stores statistics relative to the transitions instead of the game states, and uses both a tree structure and a hash table. The tree structure is formed of hard links between states, and the hash table is maintained on the side to discover the states with which to hard link. This way the transpositions table is only subjected to lookups the first time a transition is traversed.

The MCTS algorithm is performed iteratively, instead of recursively, and a stack of state memory pointers is saved and used to detect situational *triple ko* and *quadruple ko* violations. Situational is more strict than the positional definition used in Chinese rules – requiring the same player to play for it to be a repetition; and *triple* and *quadruple kos* are the shortest of *superkos* – repetitions to the states four and six plays back, respectively. *Superkos* are very rare, so this is mostly useful to better solve life and death in the endgame as cost effectively as possible. It is also important to prevent *superkos* when using virtual loss – to avoid erroneously updating the same transition several times. Context-wise Matilda prohibits positional *superkos* instead of situational, and all types of *superko*, not just the ones mentioned above.

When a *superko* is detected, the simulation is scored a loss to the player who played the illegal play. This discourages the *superko* producing sequence without prohibiting the repeating state altogether, since it might be valid through a transposition.

To ensure memory safety Matilda doesn't perform maintenance on-demand. If the program is misparametrized for the system and time control settings, and runs out of time in the middle of its turn, the MCTS is stopped early. This avoids triggering the horizon effect. Some programs are more advanced and can detect the occurrence of the horizon effect mid-search, and search a few plies deeper from the offending state.

The hash table used is also separated in two, one for each player. This means there is no confusion on the legal plays of each player and that lookups are also faster. Transpositions by passing, or *superko* repetitions with different players are therefore not detected.

This solution makes a compromise between the transpositions detection of using hash tables, and the speed of following tree hard links, which scales better with board size.

Maintenance between turns consists in freeing the states not found in the subtree with the current game state as root; which gives us flexibility in performing maintenance even if Matilda is put to play against itself. Using only a hash table and freeing states by maximum depth found is a poorer solution since a great number of states are not freed many turns past their need. This method of maintenance, without maintenance on-demand, has a very negative impact on program strength. In terms of performance there is little (but positive) advantage to a tree and hash table versus hash table only structure.

The hash tables are indexed by Zobrist hash and verified by board contents plus last eaten stone (for *ko* detection). This means transpositions through matrix transposition are also not detected. The impact of this is small since a game state, past the very beginning of the match, becomes very unlikely to match other states found in the search. Coupled with the fact that considering transpositions of this type would require state reductions and full hashes at every turn, instead of Zobrist hash updates, made this an easy decision to make.

The hash tables also use simple linked lists for buckets.

4.3.3 State representation

When making a high performance computer program for traversing state spaces, a lot of time goes into the structures and primitives for their transition – ensuring the execution is as fast as possible while fulfilling the minimum requirements for playing Go and satisfying tactical analyses.

When presenting the information kept for UCT+T we specified the need for the description of the board contents and the information necessary for the detection of *kos*.

The generic, multi-purpose board representation in Matilda is shown bellow, consisting of a description of the contents of the board in terms of stone color, and information on the last play played and stone captured if the last play caused exactly one stone to be captured.

1. Board contents for every intersection, i.e. if occupied and by a stone of which color.

2. Intersection last played (or illegal if last play was a pass).

3. Intersection last eaten a single stone (or illegal if a single stone was not eaten).

The last eaten position is used to detect *ko* violations without a full board comparison. A play at the last eaten position is a *ko* violation or suicide if it doesn't result in a multiple stone capture. The last played position is useful for heuristics that require a measure of proximity between plays.

This simple representation is enough for most things, but for the UCT algorithm, internally, more information is necessary. Next is shown the structure of a transpositions table node, that as expected includes gathered statistics from the playouts and via AMAF; with other fields for maintenance purposes.

1. Board contents (black stone, empty stone or empty)
2. Intersection last eaten a single stone (or illegal)
3. Zobrist hash
4. Total state visits for MCTS
5. Information on legal transitions for the player

And the transition information includes the statistics necessary for RAVE, criticality, effectivity, LGRF, and hard links for the formation of the game tree, for every playable transition by the player the node belongs to:

1. Board play coordinates.
2. Coordinates for LGRF.
3. Statistics for UCT, AMAF/RAVE, criticality and effectivity biasing.
4. Identifier of destination state to avoid hash table lookup.

This transition information is of course subject to the game state and playing player (which is identified by the hash table the node is stored in).

Matilda implements this using one-dimensional arrays. 1D arrays for board representations are popular in Go, because it is possible to codify extra buffer zones, marked as *out of bounds*, and not have to explicitly test if an intersection in inside the board. This is illustrated in Figure 4.3. Every legal intersection (clear) is surrounded in the 3x3 neighborhood by a legal or illegal marked intersection (gray) if the matrix is laid one dimensionally from top to bottom, left to right.

This may make for more efficient code and is described by Müller [Mü02a]. Matilda does not implement buffer zones, just 1D arrays. This was because it was felt that these techniques would perform better for *expand-and-apply* strategies – where from a certain intersection we want to discover intersections that share some attribute. Since Matilda from the start avoided these strategies because they were

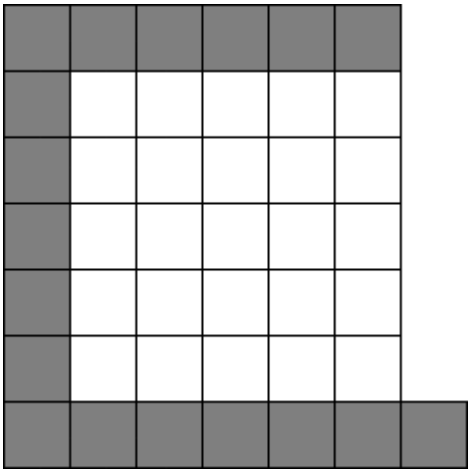


Figure 4.3: 2D representation of 5x5 1D layout

less efficient, adding the extra complexity and memory burden of buffer zones was believed not to be worth the effort later on. It is possible that in the future, techniques are implemented that would make this board representation more attractive.

A description of Go state with just stones on a board is enough for light use, but it is not very efficient for techniques that require liberty counts or *atari* testing. In computer Go, a commonly used abstraction is that of the Common Fate Graph (CFG). CFG are suitable for problems that can be simplified by uniting constituting parts because they have a common fate from that point forward, like stones in a group in Go. A CFG is often represented as an undirected graph where the nodes are groups of connected stones and the arcs immediate connections between enemy groups. Sometimes empty intersections are also represented, which makes group captures very efficient (the inversion of a group label, color to empty). CFG are colored, with nodes of the same color being united into the same node, sharing arcs.

In Matilda the following items are used, that make a CFG without empty groups. It is also complemented by fields that are kept up to date for performance reasons, like a list of all empty intersections.

1. Board contents.
2. Position last played or illegal (if passed).
3. Position last eaten a single stone.
4. Zobrist hashes of neighbors of each intersection.
5. List on empty intersections.
6. Number of neighbors of each color in the 4 and 8-intersection neighborhood.

7. Group information.

A group in Go is formed when stones of the same color touch, and is represented in Matilda with at least:

1. Color/Owner.

2. Number of real liberties and their bitmap.

3. Number of stones and their coordinates.

4. Number of neighbor groups and their references.

These structures are used to efficiently tell if groups are in *atari*, and provide information on liberties and stones captured for the various tactical functions. Real liberty counts are used, in contrast with pseudo liberties. Pseudo liberties refers to a counting method that some programs use. It allows quickly answering whether a group is in *atari*, but cannot be used to provide a trusted value of liberty counts, and is also impracticable to answer on the state of a stone after playing (without simulating the actual play and captures). Programs that use pseudo liberties usually feature faster playouts and heuristics that don't rely on exact liberty counts.

It may seem inefficient to have a bitmap of the positions that are liberties, given that on average most groups will have a very low number of them. It is, however, made efficient by the use of a memoization table of bits for a byte-sized value and the ability to use simple bitwise operations when uniting groups of stones.

Since the playouts occupy much of the time spent by an MCTS Go program, there have been many attempts to provide fast playouts and still have the necessary information for the most demanding heuristics. In Matilda an array of all the stones belonging to a group is also used. It improves on the speed of group captures.

While this solution is simple, it is still insufficient for the most costly operation of our playouts: simulating a play and counting the resulting group liberties. When we simulate a play that captures something, and that capture makes liberties for another neighbor group of the placed stone, we would need to have the number of stones in contact between the groups. Fortunately, there are two observations to be made:

1. A capture of a single stone group that touches a neighbor friendly group is only a single liberty to the resulting group.

2. A capture of a multiple stone group that touches a neighbor friendly group contributes **at least** two liberties to the resulting group.

As such, even without label information on the stones in neighborship, we can often guarantee that a group has at least two liberties, and is therefore safe to play at. The safety of a play is the most often test made – real liberty counts are less often needed in Matilda.

We also don't make use of the notion of empty group, which makes captures potentially slower, but simplifies placing stones and dealing with liberties. In essence stone groups and empty groups are very different in how the fate of their intersections is not similar.

Following the minimalist approach of Matilda, both the simpler board structure and the CFG are implemented. This allows using the best tool for each task and ensuring the good behavior of both implementations via unitary testing. The game representation is also complemented by a cache of play information in the playouts of MCTS. The reuse of this information allows much faster heavy playouts. This cache is unique for each player and is simply an array of bit masks, that contains the following information.

```
dirty
play_is_legal
play_is_put_in_atari
play_is_safe
play_captures
play_matches_pattern
```

The bit fields should be self-explanatory. This information is kept up to date for every legal position of the playout, with the efficient recalculation only of the potentially affected plays.

4.3.4 Heavy playouts

The use of heavy playouts has already been introduced in the playout phase section. Our aim is not to produce strictly more high-level play simulations, but more realistic simulations – that better represent the outlook after a certain point in the game tree (where we started the playout). Because of this the use of more advanced domain knowledge is not necessarily useful, or useful enough to deserve its performance cost. The playout phase is where a MCTS program spends the vast majority of its time. It is essential to find a good balance between value of information and time spent.

Of the two families of methods introduced prior, probability distribution selection based on trained features, for the purpose of guiding heavy playouts, seems to be the most promising method right now. Having said that, both methods have been implemented in Matilda and the use of handcrafted patterns has outperformed the other.

Handcrafted policy

The program MoGo is not open source, but a description of its patterns was published [GWMT06] and was transcribed to the pattern code used in Matilda. Patterns from the program Michi, which are slightly different from MoGo's, were also tested. The code used to represent patterns is a subset of the one popularized by GNU Go, which should make the patterns immediately recognizable by computer Go programmers.

While in the playout phase, each player plays until both players pass in a row, their difference in stones is too large (mercy threshold) or they've run past the expected amount of turns. When this happens it is usually the sign of a *superko* at the end of the game. Regardless of the method of stoppage, a playout outcome is always returned. Matilda doesn't award *no-results*, following the idea that even a worse than normal playout is biased towards the real outcome distribution.

When selecting each play, plays that are in own eyes, simple eye shapes, *ko* or suicide violations and easily refutable *self-ataris* are disqualified.

Play selection is thus performed by testing the following conditions. The first condition that is satisfied yields one play randomly from the set of plays that satisfied it, with equal probability.

1. Play a saving play (make a neighbor friendly group in *atari* become safe) in the neighborhood of the last play. Making it safe can be either by extending the group one stone, or by capturing the threatening opponent group.

2. Play a *nakade* anywhere on the board.

3. Play safely at a board intersection nearby the last play, if it matches a 3x3 handcrafted shape. The shapes have different weights for play selection (probability distribution is used).

4. Play a capturing play anywhere in the board.

5. Play a random legal play, biased to selecting a play in an intersection without neighbors.

These simple rules require some tactical analysis, namely whether a play is safe, enumerating the liberties of a group, its neighbors, etc. As shown before a cache is used to reuse play attributes efficiently. For extra performance the 3x3 shapes are first expanded when read by Matilda. The shapes as written by a human are formed with generic symbols, such as a position being matched for *own stone or empty*. These rules need to be substituted. On startup they are expanded until we obtain real, possible configurations. They are also rotated and flipped and saved

duplicated – one hash table per player. This allows the process of reading a board position and its pattern match lookup to be very efficient.

To avoid having to transpose a part of the board, with attention to its boundaries, to generate a pattern representative; Zobrist hashes of the 3x3 neighborhood of each position are maintained in the board representation. This was introduced before and is a technique popularized by GNU Go and other programs. In Matilda it is faster, but not a critical improvement, unlike attribute caching or the use of a CFG.

Trained policy

The training of feature qualities for a playout policy was performed for Matilda but was not shown to be more effective as of yet than the handcrafted policy. For training feature qualities both a simple frequency method – where the quality is the result of the observed likelihood of the pattern being selected in states where it appears – and Rémi Coulom's MM algorithm [Cou07] were implemented.

It is still too soon to select the list of features to be used in Matilda, because up to now testing has been made with a constant number of playouts, disregarding the feature time cost, but Matilda is currently using the following feature list. All features have a single possible value except for the trained 3x3 shapes.

1. Contiguity to the last play (in the eight intersections around it).
2. Play is a non-capturing *self-atari*.
3. Play is a capture.
4. Play is safe and contiguous to a friendly group in *atari*.
5. 3x3 shapes centered on prospective play.

Training was performed over a set of 2465 even-matches from the KGS collection, of the years 2001 and 2002, with no restrictions on rule set or *komi*, containing 479994 game states (not necessarily unique).

Currently Matilda is using a MoGo-style playout policy with probability distribution only if the play is selected from the matching of 3x3 shapes. The weights are trained with Minorization-Maximization and consulted for every expanded 3x3 shape. If the weight of a shape in particular is very close to zero, that shape is discarded.

Since Matilda does not impede *superkos* in the MCTS – both in the tree and in the default policy – it is possible for the program to fall into infinite cycles. This is naturally undesirable, and the simplest solution is to use a maximum depth cut-off point – where tree traversions or playouts that take longer are stopped and the outcome of the simulation returned as is. This solution however can produce an

anomaly similar to the horizon effect: if the cutoff depth is constant, and we are performing simulations from the same root state that can produce a *superko*, and if we are unlucky and the *superko* situation can influence the simulation outcome – then by stopping early we are awarding one outcome overwhelmingly; whereas if the *superko* continued further it could benefit the other player. The correct behavior – if we can't detect and avoid *superkos* – would be to null the *superko* involved stones and return an outcome irrespective of it.

This is what is indirectly accomplished in Matilda. In the playout phase the simulation max depth cutoff point is not a constant value, to allow randomness in *superko* resolution. Repetitions by *superko* take several turns and this method distributes the moment when the repetition is stopped so it doesn't always favor one player.

4.4 Tallying the score

Tallying the score – as in the ability to evaluate the final score of a match – can be a challenging problem for a computer Go player. Be reminded that the score in Go is usually the result of an agreement between the players; and computer programs are not known to be flexible. The score is usually tallied near the end game, either because the match ended or because a search algorithm (like MCTS) needs the outcome of a playout, for instance. This second scoring necessity doesn't require a perfectly precise score of the match – more often than not a faster algorithm that can give a sensible estimate is preferred.

On the other hand, the official match score has to be provided as accurately as possible. One possible way of tallying the score is to find out the stones that are unconditionally alive, i.e. the stones that can't possibly be captured by the opponent if not for own mistake, no matter how many turns in a row the opponent plays. Having the groups of alive stones, one may be tempted to count all the intersections inside those groups as belonging to the stones player.

This method of counting territory is suggested in the Tromp-Taylor rules. These do not form a rule set by themselves (they are more of a reformulation of Chinese and New Zealand rules), but provide some insight in how to resolve disputes in scoring, and simplifying the game of Go overall. This is particularly useful for computer players.

The difficulty in scoring lies when an opponent resigns (as often human players do) while the board is still very empty. By the scoring solution proposed above only the stones unconditionally alive would be useful to delimit area, which may produce extremely erroneous results in the early and mid stages of the game.

Unfortunately there is no elegant solution to this in Matilda. After a resignation the only scoring that can be made is by *imagining* what the game would look like if it continued, being played by perfect players. If the player that resigned indeed

was right to resign she should be unable to score a win. Some elaborate attempts at tackling this are present in the programs *Explorer* and *Steenvreter*, that use either an heuristic analysis or a trained MLP, respectively. Most programs however estimate the final score in these situations by assuming each stone has an influence area, and that disputed intersections more influenced by one of the players belong to that player. This approach cannot solve disputed areas with no nearby stones, and special considerations for groups in *seki* are needed.

The Matilda implementation contains four distinct methods of scoring, detailed here. The first, fastest and more prone to err, simply counts all the stones for either side. The second, besides the stones placed on the board, also counts proper eyes. Naturally these two are useful in providing very fast estimates, since they almost don't tally territory, which is essential in both Japanese and Chinese scoring.

The third implementation, more time costly, identifies for every group of empty intersections, the colors of the stones adjacent. If only one variety of stones is adjacent then the territory is tallied as that color. This scoring method correctly deals with encircled territory but doesn't recognize dead groups of stones, and by not removing them, ignores trivially disputed territory. This method is the exact implementation of area scoring, assuming the players will play through and kill dead groups of stones before ending the match.

The final implementation performs a MCTS from the current board; not to select the best transition for the current player, but to estimate the final state of the game when well played. Afterwards each intersection assumes a stone of the player that is most likely to occupy it at the end of the simulations. Very disputed intersections are marked empty. Finally over this new board disposition area scoring is performed. This method is much more accurate in estimating scores when the board is still partially empty, but is also much slower by having to perform hundreds of thousands of simulations. Because of this it is only used when requested via GTP or when writing SGF files for instance; in everything else in Matilda faster score estimates are preferred.

An alternative approach by simulation is used in GNU Go where the game is actually played to the end, turn by turn. This approach can be much more time consuming when the board is still close to empty.

Figure 4.4 shows an example game state (top left) with white to play. It was evaluated using the second, third and fourth methods (top right, bottom left and bottom right boards). The stones on the board mean the intersection was counted for that stone's player. Empty intersections benefit no player. The final method correctly identifies that in the bottom right the black stone **a** was *nakade*, killing the white group and eventually solving the *seki* at the left. At the top it identifies that white can capture the three black stones, securing liberties to then kill the top right group, if it first plays **b**, ensuring black cannot make two eyes.

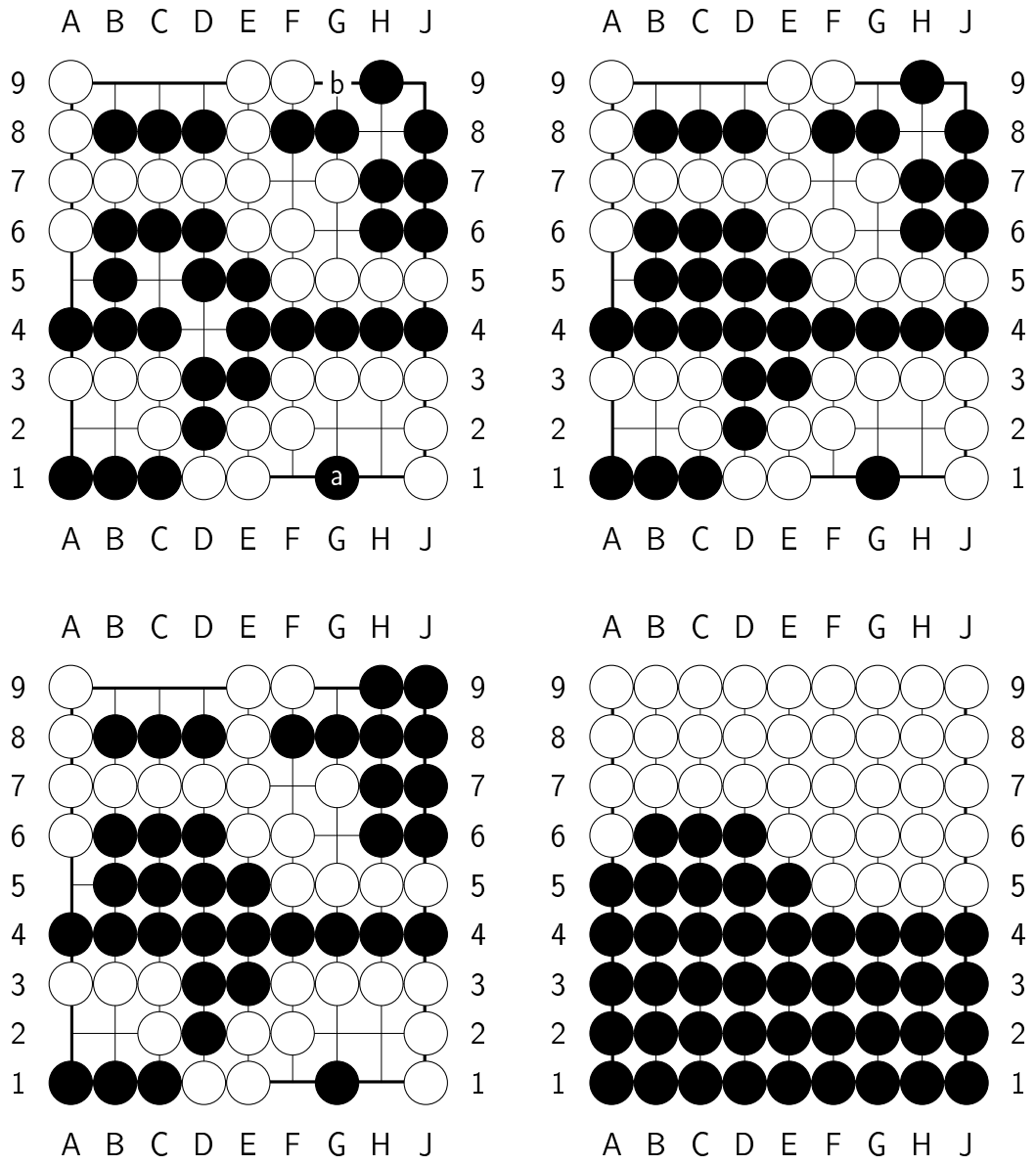


Figure 4.4: Scoring examples

Although none of the implementations formally identify liberties of groups in *seki*, the final implementation quite robustly deals with *seki* and capture races which are typically hard problems to solve. It is still lacking in more complex positions with multiple *ko* threats, and long *superkos* (the quick simulations do not test them) and of course is still weaker an estimator the emptier the board is.

4.5 Results

4.5.1 ANN for state evaluation

On tuning the weight distance p used in training MLP for state evaluation, it was found that maximum distances as short as 6 are not unreasonable. Choosing a small distance provides both benefits for the generalization ability of the network as well as the number of connections, and therefore the time needed to compute the network. More important than finding the optimal value for training the network, it is important to take its computation time into consideration and tune the distance as part of the final MCTS algorithm.

In Figure 4.5 are shown the median ranks of the best play, for networks trained with maximum receptive distances between 6 and 10, over 10 epochs. All tests shown are for 19x19. All networks use the same constant learning rate of 0.002.

The median rank consists in the median rank of the output energy at the coordinate of the selected play, in the example from the data set. It is a better indicator than the average because it is common for the majority of poor plays to be clustered in the minimum output value possible. If several plays are tied the median rank used is that of the middle point of their positions.

Figure 4.6 shows the accuracy of the network when trained with $p = 9$. The accuracy is the probability of the network identifying exactly the selected play. This value cannot be close to 100% without overfitting since in Go there are many plays that are equally strong in the same situation, and the examples are also not perfect. The use of CNN has been able to reach accuracies over 50% in similar data sets, when trained to predict the professional play. The accuracy of the 2-layered MLP used in Matilda is extremely small, but they will be used for general guiding, with a focus on urgency instead of exact best play.

Figure 4.7 (page 81) shows the selection probability for the same network trained with $p = 9$. The selection probability is the probability of the play selected in the example being in the top 25% legal plays (ignoring *ko*). This groups varies with size and play legality, being therefore example dependent. In actual MCTS the number of candidate plays will be lower – since repetitions and tactical information is taken into consideration – increasing the accuracy of the ANN. This is the best indicator of adequacy of the ANN for Matilda before actual gameplay testing.

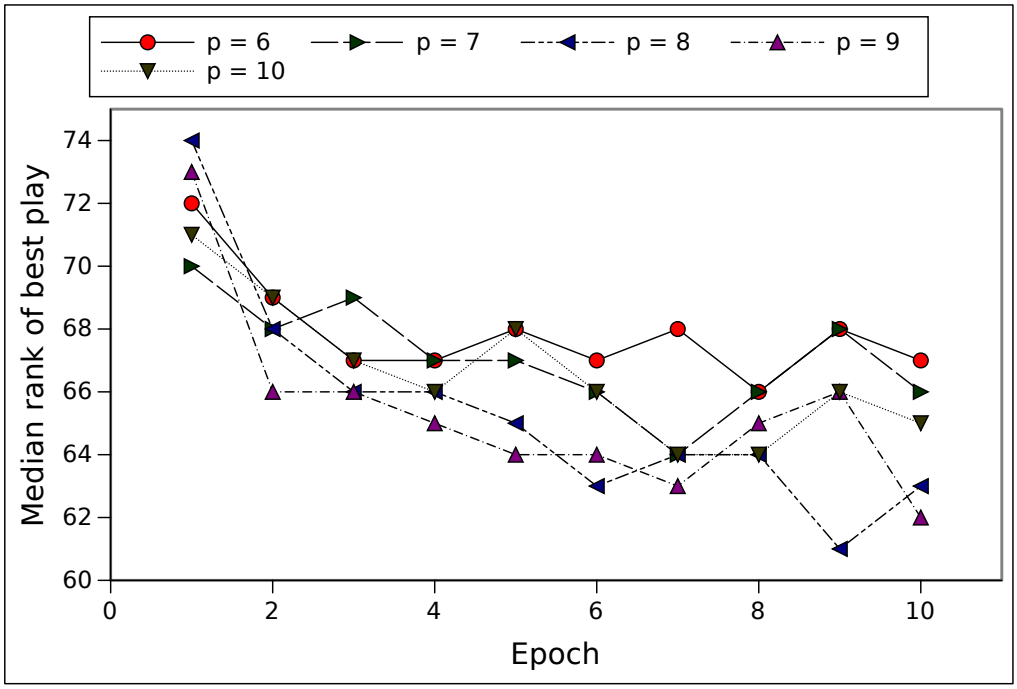


Figure 4.5: Median rank of best play by distance

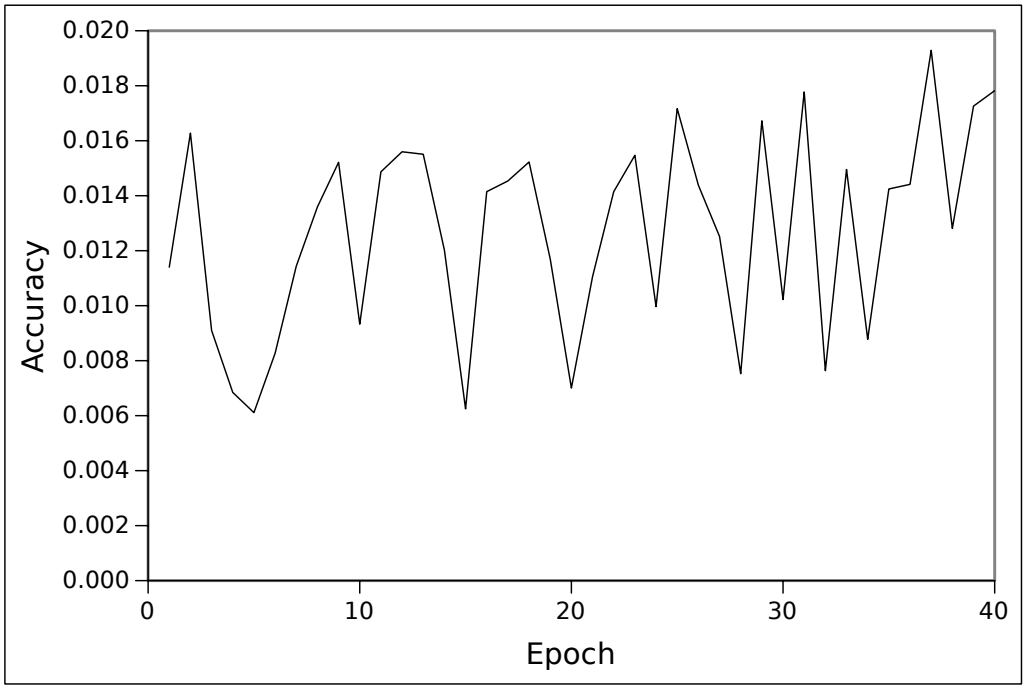


Figure 4.6: Accuracy at selecting best play

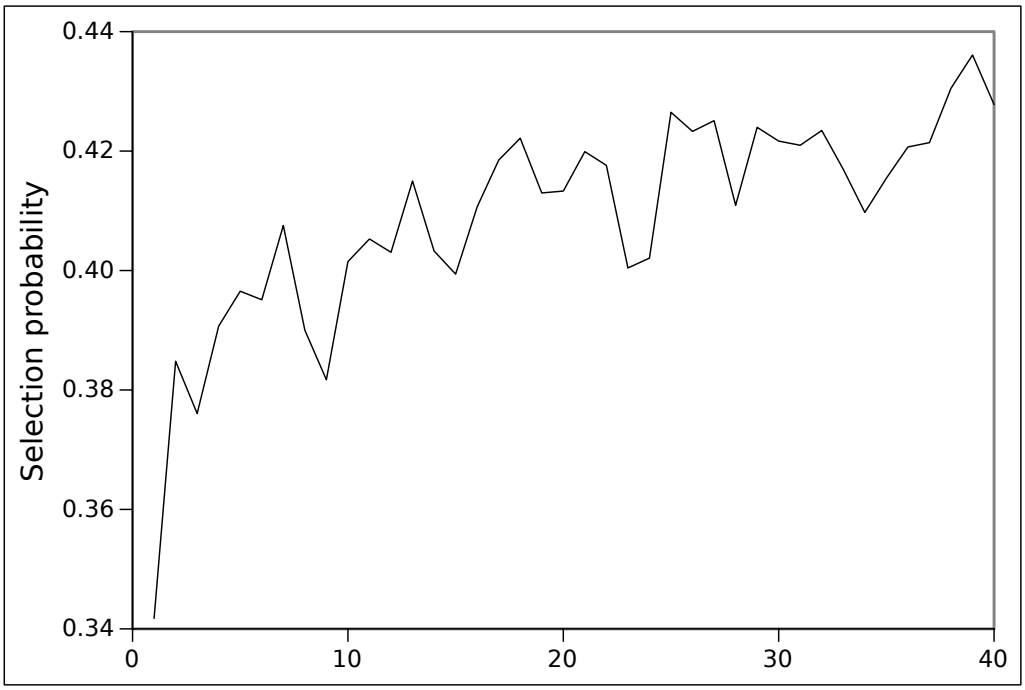


Figure 4.7: Probability of finding best play in top 25% legal

Comparison with SANE

To be able to compare the different methods of producing Go evaluation neural networks, the evolved networks via SANE were also subject to testing against a test set. The results shown in Figure 4.8 (page 82) clearly show how the networks fail to improve even after long periods of time. By our estimates a minimum of 5000 generations should be needed to evolve the networks by SANE, though by the 3000 generations shown here some improvement was already expected.

The networks trained via the SANE method are so poor that they didn't evolve to consistently understand that one of the input bits codified a playable intersection, which is evident by the median rank of the best play being higher than 122 (average of legal plays of the examples in the data set).

At first look it can be thought that computing the MLP can be made more efficient by, after a play, only computing the parts of the network that have been modified. This is actually not possible, given the nature of the MCTS algorithm. In MCTS when a new state is created, and initiated with the estimate from the ANN, a playout is performed and the search ended. The next search is unlikely to stop at exactly the descendant of the previous state, and as such the ANN codifications between the two states cannot be relied to be similar. To perform this optimization a *burst* modification to the algorithm could be made, where the search doesn't stop at the

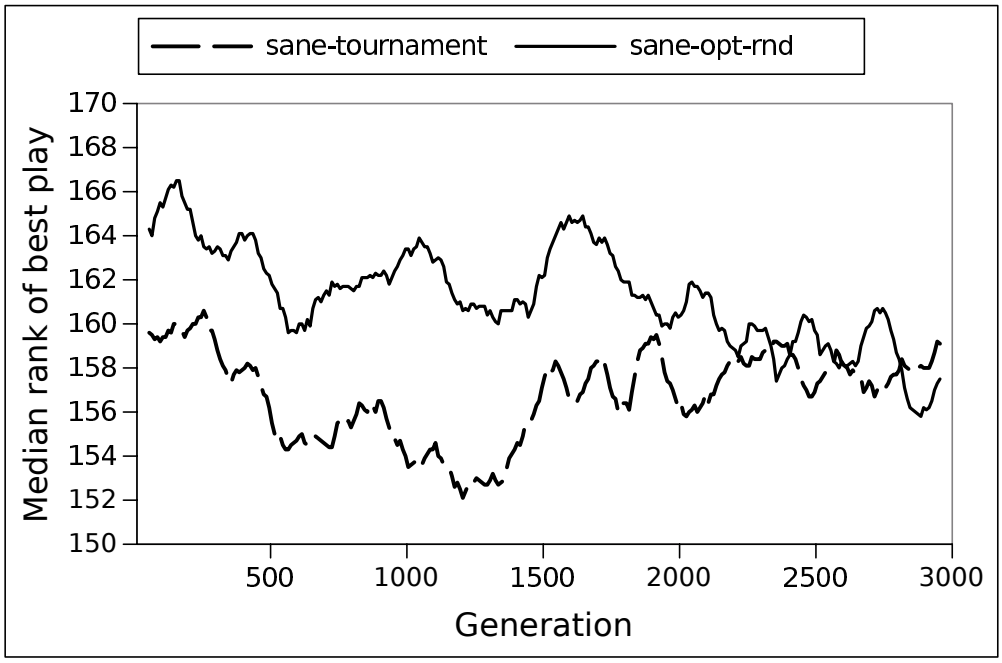


Figure 4.8: Median rank of best play with SANE method

Board size	Win rate	Games	Avg turns	Avg time
9x9	91.6%	136	88.2	271.6s
13x13	33.3%	216	149	456.5s

Table 4.1: GNU Go 3.8 (B) vs Matilda

first new state found.

4.5.2 Strength at playing Go

Matilda was tested for its strength with and without the trained MLP as prior values heuristics. In table 4.1 the strength of Matilda without ANN priors can be seen from playing against an external program. It was run against GNU Go 3.8 level 10 (default difficulty) in 9x9 and 13x13 boards, with 5 and 10 minutes per match respectively – sudden death. *Komi* was set at 7.5 stones.

The table above shows the win rate for Matilda when playing as white (starting second), the number of games played, the average turns per match and the average time Matilda used. The matches last longer than usual because GNU Go doesn't resign when losing.

The board sizes were chosen for being the most popular (minus 19x19, see bel-

low). The time settings were chosen for being the time controls used in the Computer Go Server (CGOS). The opponent was chosen to be GNU Go for several reasons:

1. It is relatively strong (even if not champion level anymore) while still playing very fast. This is important for running benchmarks fast. On the CGOS GNU Go 3.8 is ranked above 1800 ELO.

2. It is free software, well documented and amply used in previous Go research as a comparison opponent.

3. It has many configuration options that are useful, like being asked to capture all stones instead of passing in the end of the game, to simplify scoring.

For reproducibility, GNU Go 3.8 was compiled using the default settings, and ran with the options to activate Chinese rules with positional *superko*, as well as allowing resignations (so the matches are ran faster).

These results were obtained using opening books imported from Fuego and Pachi, Heuristic UCT-RAVE with criticality, LGRF, virtual loss; without effectivity, grandparent knowledge or the use of dynamic *komi*. Thinking in the opponents time is enabled, although GNU Go plays very fast for this to have a significant impact.

As can be seen, Matilda is currently much stronger in smaller boards than in larger ones – so much so that the benchmark for 19x19 was omitted; the win rate would be too close to 0% to be representative. A number of possible improvements are shown in the further work section of the conclusion. The comparison with the use of ANN for prior values should be read with a grain of salt, since the impact of a technique benefits from the absence or unsuitability of the others. The use of ANN priors should be less beneficial in programs already better prepared for larger boards.

In this work, the ANN calculations do not make use of GPU yet. Because of this, and because it is expected that an ANN to be used in Go would make use of GPU, the comparison with using ANN priors was performed with self-play and with a constant number of simulations per turn. The games were played with 10000 simulations per turn maximum, 7.5 *komi* and with the players alternating colors. No state expansion delay is used (and thus 10000 simulations equals 10000 playouts, except if a *superko* is detected).

Table 4.2 shows the win rate for the version using ANN priors. The average time relative pertains to the time used in comparison to the faster version without ANN – the performance impact of ANN priors without GPU computation. Take into consideration that tests with a fixed number of playouts per turn and in high concurrency have their performance suffer when the playouts take longer, because the impact of the virtual loss is felt more.

Board size	Win rate	Games	Avg turns	Avg time relative
9x9	49.8%	257	48.6	1.07
13x13	51.8%	863	78.6	1.24
19x19	52.4%	124	147	1.37

Table 4.2: Win rate with ANN priors (self play)

With the time and resources available it was also impossible to tune all parameters, heuristic weights, constants, etc. Our best efforts were employed but tuning was mostly performed for 9x9. Tuning in 13x13 was performed almost only on heuristics only active for larger boards, such as preferring plays away from the sides and without neighbor stones. Parameter tuning for 19x19 wasn't performed at all, including the use of ANN priors.

The strength change by the use of ANN priors is barely positive without any complex domain knowledge needing to be computed. For a strength impact this small this solution is probably inadequate for domain knowledge rich programs. Also as expected the impact does not correlate linearly with the board size.

4.5.3 Data set used

In this work, collections of game records were used for the extraction of knowledge. From the KGS Go Server game records between 6d players, or with one of them at least 7d[4], were obtained from u-go.net on the 18th of May, 2015.

Another collection of professional Japanese matches, mostly from the 20th century, was also explored. Both of these collections are commonly used for training in computer Go, but this one ended up not being used since it did not contain games with Chinese rules. The availability of game records for 19x19 was large enough for our uses and we could be very selective on the game settings used. For Matilda only games with Chinese rules, reasonable *komi* and no handicaps were used, which were less than 1% of the total games. The inclusion of *komi* also disqualifies older games played before it's invention.

When creating a data set for supervised learning using ANN every $\{state, play\}$ pair was extracted if the play was not a pass, the turn is not in the last few turns of the match, the number of stones is bellow the threshold for using the ANN as prior knowledge heuristic, the match was not started with handicap stones and finally the match lasted a reasonable minimum number of turns.

To the unique states the most popular play is selected and the state is codified

[4]These rankings are specific to the server and have not been actually awarded by a human organization.

with information that will later be useful for the codification of our ANN inputs, such as the number of liberties after playing. Each state was also rotated and flipped to create more examples and to better train the network, since it doesn't use shared weights.

When training ANN the data set was divided into a training set (9/10ths) and a test set (1/10th). The training set was shuffled at the beginning of each epoch. In testing SANE evolved networks only the test set was used.

For board sizes other than 19x19 the game collections were made by pitting GNU Go against itself, generating large amounts of reasonably strong play in a short time. The strength of the play present in the data set is the more important the greater the complexity the network can learn; so this method is not suitable for more robust ANN structures.

4.6 Parallelization

This work made extensive use of multi-processor directives to better take advantage of modern shared memory multi-processor architectures. First, the critical areas were identified, as being the computation units where the software spent most of its time. In the genetic evolution program this area pertained to the calculation of the network fitnesses. In the Go playing software the most time consuming part is related to the MCTS algorithm.

For portability, simplicity and performance, OpenMP was opted for instead of POSIX threads or other similar solutions.

The UCT+T algorithm has received considerable attention for parallelization, both for shared memory and cluster environments. Chaslot et al. identify four methods of parallelization [CWH08]:

1. Leaf – upon reaching a leaf node we perform several playouts in parallel. We wait for their completion to propagate all outcomes at once.

2. Root – each worker builds its own search tree in parallel without information sharing. When the time available ends the play is selected based on the sum of all individual searches.

3. Tree with global exclusion – all workers share the same search tree but only one can traverse it as a time. The other threads are free to be running playouts (which have no side effects).

4. Tree with local exclusion – all workers share the same search tree with the minimal exclusion necessary to ensure the consistency of the structures and state information.

The first method is perhaps the simplest, but wastes more time on the synchronization of the worker threads since we have to wait for the completion of the last playout; and the elapsed time may vary greatly. Root parallelization excels in cluster environments since almost no information sharing is necessary. Tree parallelization allows keeping a great number of worker threads active in shared memory. If local exclusion is used the time locked is smaller but we may need to implement some method of virtual loss, to promote the exploration of different sequences of play.

In experiments performed by Chaslot et al. [CWH08] the best results for 13x13 were achieved with root parallelization, which is unintuitive for us. In 9x9 the results instead slightly favored tree parallelization with local exclusion. We believe this may have to do with the application of virtual loss. By either adding or not a virtual loss we are either allowing the over simulation of a few transitions or the unnecessary exploration of others, respectively. We do not have an empirically achieved middle ground, since the bias (one loss) is an atomic value.

We have not conducted experiments on this subject. Matilda implements tree parallelization with local exclusion and a virtual loss. If support for clusters is later added it will probably be in the form of root parallelization, with tree parallelization at each node.

For cluster environments Open MPI was also used, which is an implementation of MPI – an API for message passing between computers registered as part of a cluster of computer systems.

Only the SANE method ANN evolver was implemented to take advantage of MPI, because of the time required to calculate the fitness of the networks when playing against an external opponent. Eventually the difficulty in finding an available cluster, and the use of alternative fitness calculation methods that are less time consuming (**sane-opt-rnd** and **sane-competitive** instead of **sane-opt-gtp**), meant the cluster solution was no longer essential, and was removed.

4.7 Testing

In testing competitive computer Go software four approaches are commonly used.

Perhaps the most obvious is play-testing, where the program competes against external opponents to observe changes in strength between versions. The greater the number of matches and the variety of the opponents the better. Competitions against previous versions of itself can also be done. This approach is the only reliable way of testing changes that may affect many parts of the software and unfortunately is also very slow.

To perform play-testing, Matilda first used its own game coordinator (now removed) and then twogtp, which is part of GoGui – a free software graphical interface

for computer Go programs using GTP. The board graphics in this document were also exported to LaTeX with GoGui, and rendered with the package psgo.

With twogtp hundreds of thousands of matches were played versus itself, GNU Go 3.8 and Pachi in the last months, for the purpose of parameter tuning and testing whole game-playing strength. This was done mostly on 9x9. While most tuning was performed manually – trying to find a good combination of variable values – automatic methods were also employed. One would, through brute force, test a range of values for a few variables at a time. This was very slow and had problems will local optima, and was soon replaced. Next automatic tuning was performed using a genetic algorithm. It evolved a population of program configurations, using the program win rate against GNU Go as fitness criteria. With this method dozens of variables are configured simultaneously.

Not satisfied with the time the optimization process was taking – because of the lengthy fitness calculation – a method of parameter optimization by UCT was also implemented. It builds a tree, traversed using UCB1 and First Play Urgency (FPU), and updated with the outcome of each match against GNU Go. Each node is a variable value with its MC quality and number of visits. We haven't tested enough to assert this method as more efficient, however it has the advantage of approximating first the variables closer to the top of the tree. This means we can speed up the process by gradually removing the best trained variables and focusing on the others. The use of FPU is unfortunate – since we do not have obvious heuristics for black box tuning; however by specifying an initial guess, we can use a normal distribution as heuristic prior values. Information sharing can also be done by storing the average outcomes of all variable values, invariant of the path to them.

Both genetic evolutionary algorithms and UCT have been used in the past to parametrize computer Go programs [CWSvdH08, Cou12].

Because of time constraints the application was tuned with as little as 0.3 seconds per match, and constants dependent on search length may be misparametrized, like the RAVE MSE b constant or minimum visits for the use of criticality.

Regression testing is an alternative approach to play testing, that saves time by having the program solve individual problems compiled previously. This approach is best at quickly identifying how program changes have changed the success at answering specific sub-problems of Go, but may not have the sensitivity to discern small changes in strength. Testing this way has not been performed in Matilda yet, although the technical requirements are met: being able to load SGF files at a specific point and being asked to evaluate the current position via GTP.

A third approach is the use of unitary testing. It is mostly useful with ensuring the correctness of the algorithms and data structures after big changes. A fourth approach is in runtime sanity checks performed when the program is compiled for debugging instead of release. While unitary testing helps detect inconsistencies in

very low level and contained code, runtime sanity checks help ensure the information processes of the program fall into their expected behavior. When compiling for debugging the final executables are also made to include the information necessary for profiling. For this purpose the free software valgrind was used.

4.8 Use demonstration

Matilda currently possesses two user interfaces. The text mode interface is directed at people that want to use it locally without any external software. It is very limited in functionality and is not very useful outside of casual play. The GTP interface is directed at having the program be connected to a game controller, and offers a more complete array of options. It can also be used locally for debugging, since the protocol is text-based.

In the past Matilda also included its own graphical mode for X11 systems, but it was obsolete given the availability of GTP-speaking graphical client programs for Go. Programs like GoGui provide many private extensions to GTP, that allow visualizing more information about the state evaluation that just the selected play – which was the initial motivation for Matilda having its own graphical interface.

The output of the text programs was made to resemble the one used in GNU Go. An example of local play with text mode is shown in Listing 4.1. The coordinate system used can also be switched between Japanese and European styles.

Debug messages are written to the standard error file descriptor and log file, and are shown when GTP mode is used locally. Listing 4.2 shows GTP mode with informational messages mixed in.

Listing 4.1: Example of text mode use

```
Matilda 2016—03 running in text mode. In this mode the
    options are limited and no time limit is enforced. To run
     using GTP add the flag —gtp. Playing with Chinese rules
    with 7.5 komi; game is over after two passes or a
    resignation.


White (O): matilda
Black (X): human

   A B C D E F G H J
 9 . . . . . . . . . 9
 8 . . . . . . . . . 8
 7 . . + . . . + . . 7
```

```
 6 . . . . . . . . . 6
 5 . . . . + . . . . 5
 4 . . . . . . . . . 4
 3 . . + . . . + . . 3
 2 . . . . . . . . . 2
 1 . . . . . . . . . 1
   A B C D E F G H J

(Type the board position, like D6, or undo/pass/resign/score
   /quit)
Your turn (X): e5

White (O): matilda
Black (X): human
Plays (1): Be5

   A B C D E F G H J
 9 . . . . . . . . . 9
 8 . . . . . . . . . 8
 7 . . + . . . + . . 7
 6 . . . . . . . . . 6
 5 . . . .(X). . . . 5
 4 . . . . . . . . . 4
 3 . . + . . . + . . 3
 2 . . . . . . . . . 2
 1 . . . . . . . . . 1
   A B C D E F G H J

Last played E5

Computer thinking ...
```

Listing 4.2: Example of GTP mode use

```
0.000: Matilda now running over GTP
genmove b
4.138: time to play: 15.000s
4.138: mcts: time to play doubled to initialize tree
4.141: pat3: imported 22 with weights and generated 3042
   patterns
34.181: mcts: search finished (sims=651217, depth=18, wr
   =0.49)
```

```
= E5

34.407: mcts: freed 98068 states (460 MiB)
genmove b
36.506: time to play: 14.248s
50.754: mcts: search finished (sims=302790, depth=17, wr
   =0.63)
= E7

50.873: mcts: freed 820959 states (3852 MiB)
```

Chapter 5

Conclusion

5.1 Commentary

Matilda, as a computer Go player, attempts to play high quality Go (and succeeds in small boards), but it still has a long way to go in 19x19. It was created first as a means to understand the problems pertaining to computer Go and explore some improvements; and only secondly as a realistic attempt at an as strong as can be player. A strong computer Go program takes many years to develop, slowly inching towards greater strength.

While there is not a lot of free computer Go software, this dissertation could also have been built around another program, like Fuego, avoiding a lot of, ultimately, unnecessary work. This was opted against because it was important to build the whole solution to learn from experience the problems and solutions present in modern computer Go software, and from there being able to think of improving upon them. While what is presented here is only a part of computer Go development, we hope it serves as a good foundation for later work on Matilda or other projects.

If it were possible to go back in time, one of the best recommendations to make would have been to use parameter optimization methods from the start. The amount of time spent on parameter tuning, even when automatic, is tremendous. In producing Matilda over 600000 matches have been played against GNU Go for tuning.

Another recommendation would have been to have opted for a comparison of MCTS with the use of *heavy* – or deep – neural networks, like CNN, instead of only *lighter* ANN. The exploration of the SANE method, while the rest of the computer Go community invested in CNN, was in retrospect a prediction blunder.

On aiming to produce a strong computer Go program, many worthy problems were only glanced over. The problematics of accurate scoring – for instance – where whole-game simulation is performed instead of formally identifying the life and

death status of the stones. Other areas of research pertaining to Go are not just interesting *because they're there*, but also because they can bring meaning to MCTS based decisions – by helping a human user to understand the decisions made, even if not the process by which it arrived at them.

This work also made use of implementation surveys of computer Go free software, mainly Fuego and Pachi, and commentary from computer Go researchers from the computer-go mailing list, to which the author is indebted.

5.2 Future work

As evidenced in the Results section of the Implementation chapter, Matilda struggles much more on large boards than in smaller ones. The reasons for this can be grouped as follows:

1. Lack of strategic evaluation. Most computer Go programs feature rich strategic and tactical functions that can recognize and take advantage of more situations than the very few supported in Matilda. Without these *shortcuts* it is up to the internal representation of MCTS to solve many difficult sub-problems. Concepts like areas of influence, Martin Müller's definition of zone [Mü02b], player framework, and specific exhaustive solvers for capture races are necessary to better understand subtle whole-board problems. Databases of *joseki*, *tesuji* and understanding *good* and *bad shape* are all absent. The CFG representation also lacks a more high level abstraction of groups of stones not yet but probably connected, and allowing reasoning over the safety of groups.

2. We haven't obtained good results using trained playout policies – via MM. MoGo handcrafted policies are known to produce good sequences of local moves, and since they were tailored for strength at 9x9, they may be unsuited for larger boards. Also keeping track of key points and *ko threats* may be necessary to produce more accurate playouts.

3. Insufficient tuning. More testing is necessary; specially in larger board sizes. As seen in other programs, the impact of RAVE should be very positive and even replace the use of the UCB term in the selection phase. In Matilda the impact is positive but not that impressive, which reflects in the poorer results in larger boards. The reason for this discrepancy may be the misparametrization of the Heuristic UCT-RAVE prior values. These priors are still being learned and have not been tuned for 19x19.

Table 5.1 makes the case for the use of RAVE not being enough to replace the UCB term. The games were played against GNU Go level 0 with 5000 simulations

Disabled	Win rate	Games
nothing	58.4%	358
UCB term	26.4%	307
RAVE	54.4%	320
Criticality	52.9%	780
LGRF	48.4%	440
prior values	44%	450
heavy playouts	7.1%	326

Table 5.1: Impact of removing different improvements

Disabled	Win rate	Games
nothing	81.1%	201
RAVE	77.5%	209

Table 5.2: Impact of removing RAVE with 100000 simulations per turn

per turn on 9x9. As can be seen, the impact of RAVE is small with this few number of simulations; the tests are repeated in Table 5.2 for RAVE with 100000 simulations per turn. In any number of simulations, the impact of RAVE was expected to be more positive. This is probably because RAVE (and AMAF smoothing constant D) – which is an improvement aimed at larger boards – was also only tuned in 9x9. Figure 3.3 (page 30) shows the impact of b is small in 9x9.

The use of ANN has shown that, with little domain knowledge given, a small MLP does not introduce a significant advantage. Its advantage at identifying good looking – promising – moves when the board is very empty is already in part overshadowed by more efficient domain knowledge heuristics, or trained features. Methods like MM are able to reach accuracies of over 40% over professional records [Cou07] (these results have not been reproduced). Having a lower bound on the contribution of a network, it is interesting to find next the upper bound – the point at which using a larger CNN no longer contributes to the program.

Matilda may also have been too optimized for speed instead of playing strength. Looking back to Table 3.3 (page 53), the diminishing returns may justify preferring more costly – accurate heuristics, when playing with longer thinking times. Parameter optimization also has to contemplate the impact on different total playing times.

Several small improvements from a performance point of view, but important for a users experience, are also scheduled to be implemented. These include simulating other rule sets (particularly Japanese), playing in online servers other than KGS Go Server, analyzing previous matches, better *seki* resolution, improved score estimates

in unsettled game positions and supporting Fischer clock and other non-*byo-yomi* based time systems.

When Matilda was started the use of CNN was an emerging idea; by the time this document was finished CNN had already revolutionized computer Go and produced the first victory against a professional human player in an even game. This increasingly means that a practical decision will eventually have to be made on the future of Matilda: whether to support a greater variety of computer systems by not implementing CNN, relying more on rules and reasoning; or follow the competitive route and start experimenting with CNN to maximize playing strength. CNN seem poised to become ubiquitous in Go in the next years.

Chapter 6

References

[AFK02] Peter Auer, Paul Fischer, and Jyrki Kivinen. Finite-time analysis of the multiarmed bandit problem. In *Machine Learning*, 2002.

[All94] Louis Victor Allis. Searching for Solutions in Games and Artificial Intelligence, 1994.

[Alm97] L. B. Almeida. *Handbook of Neural Computation*, chapter Multilayer perceptrons. 1997.

[Alt15] Ingo Althöfer. Frisbee Go Simulation, 2015. `http://computer-go.org/pipermail/computer-go/2015-November/008172.html` [Online; accessed 6-May-2016].

[Bau11] Petr Baudiš. *MCTS with Information sharing*. PhD thesis, Faculty of Mathematics and Physics of the Charles University in Prague, 2011.

[BD10] Hendrik Baier and Peter D. Drake. The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte-Carlo Go, 2010.

[BH03] Bruno Bouzy and Bernard Helmstetter. Monte Carlo Go Developments. In *Advances in Computer Games Conference ACG-10*, pages 159–174, 2003.

[Bru93] Bernd Bruegmann. Gobble, 1993. `http://www.cgl.ucsf.edu/go/Programs/Gobble.html` [Online; accessed 12-February-2016].

[CBK08] Benjamin E. Childs, James H. Brodeur, and Levente Kocsis. Transpositions and Move Groups in Monte Carlo Tree Search, 2008.

[CJP] Barbara Chapman, Grabrielle Jost, and Ruud Van Der Pas. Using OpenMP: portable shared memory parallel programming.

[Cou07] Rémi Coulom. Computing Elo Ratings of Move Patterns in the Game of Go. In *Proceedings of the Computers Games Workshop 2007*, pages 113–124, 2007.

[Cou12] Rémi Coulom. *Advances in Computer Games: 13th International Conference, ACG 2011, Tilburg, The Netherlands, November 20-22, 2011, Revised Selected Papers*, chapter CLOP: Confident Local Optimization for Noisy Black-Box Parameter Tuning, pages 146–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. `http://dx.doi.org/10.1007/978-3-642-31866-5_13`.

[CS14] Christopher Clark and Amos J. Storkey. Teaching Deep Convolutional Neural Networks to Play Go. *CoRR*, abs/1412.3409, 2014.

[CWH08] Guillaume M. J-B. Chaslot, Mark H. M. Win, and H. Jaap Van Den Herik. Parallel Monte-Carlo Tree Search. In *6th International Conference, Computers and Games 2008*, pages 60–71, 2008.

[CWSvdH08] Guillaume M. J-B. Chaslot, Mark H. M. Winands, István Szita, and H. Jaap van den Herik. Cross-Entropy for Monte-Carlo Tree Search. In *ICJA Journal 31*, pages 145–156, 2008.

[Dah99] Fredrik A. Dahl. Honte, a Go-Playing Program Using Neural Nets. In *Workshop on Machine learning in Game Playing*, pages 205–223. Nova Science Publishers, 1999.

[Enz03] M. Enzenberger. Evaluation in Go by a neural network using soft segmentation. In *10th Advances in Computer Games conference*, pages 97–108. Kluwer Academic Publishers, 2003.

[GS11] Sylvain Gelly and David Silver. Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go, 2011.

[GWMT06] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modication of UCT with patterns in Monte-Carlo Go, 2006.

[Hay98] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2 edition, 1998.

[HCL10] Shih-Chieh Huang, Rémi Coulom, and Shun-Shii Lin. Monte-Carlo Simulation Balancing in Practise, 2010.

[MHSS14] Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move Evaluation in Go Using Deep Convolutional Neural Networks. *CoRR*, abs/1412.6564, 2014.

[Mü02a] Martin Müller. Computer Go, 2002.

[Mü02b] Martin Müller. Counting the score: Position evaluation in computer Go. In *Computers and games: 4th International Conference, CG 2004, Volume 3846 of Lecture notes in computer science*, pages 273–284. Springer, 2002.

[RMM97] Norman Richards, David E. Moriarty, and Risto Miikkulainen. Evolving Neural Networks to Play Go. *Applied Intelligence*, 8:85–96, 1997.

[Sea16] David Silver and Aja Huang et al. Mastering the Game of Go with Deep Neural Networks and Tree Search, 2016.

[SM07] D. Silver and M. Müler. Reinforcement Learning of Local Shape in the Game of Go. In *20th International Joint Conference on Artificial Intelligence*, pages 1053–1058, 2007.

[SWUH07] Jahn-Takeshi Saito, Mark H. M. Winands, Jos W. H. M. Uiterwijk, and H. Jaap Van Den Herik. Grouping Nodes for Monte-Carlo Tree Search. In *BNAIC 2007*, pages 276–283, 2007.

[TF07] John Tromp and Gunnar Farnebäck. Combinatorics of Go, 2007.

[Tro16] John Tromp. Number of legal Go positions, 2016. `http://tromp.github.io/go/legal.html` [Online; accessed 22-January-2016].

[TT95] John Tromp and Bill Taylor. Tromp-Taylor Concise Rules of Go, 1995. `http://www.cs.cmu.edu/~wjh/go/tmp/rules/TrompTaylor.html` [Online; accessed 5-February-2016].

[TZ15] Yuandong Tian and Yan Zhu. Better Computer Go Player with Neural Network and Long-term Prediction. *CoRR*, abs/1511.06410, 2015.

[vdW04] Erik van der Werf. *AI techniques for the game of Go*. PhD thesis, Universitaire Pers Maastricht, 2004.

[Zob70] Albert Zobrist. *Feature Extraction and Representation for Pattern Recognition and the Game of Go*. PhD thesis, University of Wisconsin, 1970.

Index