

Move Prediction in the Game of Go

A Thesis presented

by

Brett Alexander Harrison

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 1, 2010

Abstract

As a direct result of artificial intelligence research, computers are now expert players in a variety of popular games, including Checkers, Chess, Othello (Reversi), and Backgammon. Yet one game continues to elude the efforts of computer scientists: Go, also known as Igo in Japan, Weiqi in China, and Baduk in Korea. Due in part to the strategic complexity of Go and the sheer number of moves available to each player, most typical game-playing algorithms have failed to be effective with Go. Even state-of-the-art computer Go programs are weaker than high-ranking amateurs. Thus Go provides the perfect framework for developing and testing new ideas in a variety of fields in computer science, including algorithms, computational game theory, and machine learning.

In this thesis, we explore the problem of move prediction in the game of Go. The move prediction problem asks how we can build a program which trains on Go games in order to predict the moves of professional and high-ranking amateur players in other Go games. An accurate move predictor could serve as a powerful component of Go-playing programs, since it can be used to reduce the branching factor in game tree search and can be used as an effective move ordering heuristic. Our first main contribution to this field is the creation of a novel move prediction system, based on a naive Bayes model, which builds upon the work of several previous move prediction systems. Our move prediction system achieves competitive results in terms of move prediction accuracy when tested on professional games and high-ranking amateur games. Our system is simple, fast to train, and easy to implement. Our second main contribution is that we describe in detail the process of implementing the framework for our move prediction system, such that future researchers can quickly reproduce our results and test new ideas using our framework.

Contents

1	Computer Go: The Ultimate AI Challenge	3
1.1	History	4
1.2	Rules	4
1.3	Why is Go hard?	6
1.4	Recent Computer Go Programs	7
1.4.1	Goal Search	8
1.4.2	Heuristic Evaluation Functions	8
1.4.3	Monte-Carlo Tree Search	8
1.4.4	Table Lookups	9
1.5	Other Computer Go Techniques	9
1.5.1	Neural Networks	9
1.5.2	Temporal Difference and Reinforcement Learning	9
1.5.3	Cellular Automata	9
1.6	Recent Work in Move Prediction	10
2	Methods for Move Prediction	12
2.1	The Move Prediction Problem	13
2.2	General Move Prediction Algorithm	14
2.3	Feature Extraction	15
2.3.1	Capture type	16
2.3.2	Capture, number of stones	18
2.3.3	Extension	18
2.3.4	Self-atari	18
2.3.5	Atari	18
2.3.6	Distance to border	18
2.3.7	Distance to previous move	19
2.3.8	Distance to move before previous move	19
2.3.9	Monte-Carlo owner	19
2.3.10	Jumps and Knight's moves	20
2.3.11	Pattern match	21
2.4	Machine Learning Techniques for Scoring Feature Vectors	21
2.4.1	Naive Bayes	21
2.4.2	Bradley-Terry Move Ranking	23

2.5	Testing Move Prediction Accuracy	26
2.5.1	Data	26
2.5.2	Results	26
2.6	Future Research	30
3	Implementing a Framework for Move Prediction	31
3.1	The Go Board	31
3.1.1	Go board data structure	32
3.1.2	Chains	33
3.1.3	Legal moves	36
3.1.4	Processing moves	38
3.1.5	Move history and undo	38
3.2	Monte-Carlo Playouts	41
3.2.1	Simplified Go board	41
3.2.2	Selecting random moves	43
3.2.3	Computing the Monte-Carlo owner feature for legal moves	44
3.2.4	Performance	44
3.3	Pattern Database	44
3.3.1	Pattern representation	44
3.3.2	Pattern symmetries	45
3.3.3	Counting pattern frequencies	46
3.3.4	Pattern extraction	46
3.3.5	Performance	46
3.4	Concluding Remarks	47
	Acknowledgments	48
	Bibliography	48
A	Rules of Go	52
A.1	Capture	52
A.2	Suicide	53
A.3	Ko	53
A.4	Life and Death	54
A.5	Scoring	55
B	Go Terminology	56
C	Additional Go Resources	63

Chapter 1

Computer Go: The Ultimate AI Challenge

“...the tactic of the soldier, the exactness of the mathematician, the imagination of the artist, the inspiration of the poet, the calm of the philosopher, and the greatest intelligence.”

Zhang Yunqi, 围棋的发现 (Discovering weiqi), Internal document of the Chinese Weiqi Institute, 1991.

Computer Go has been, and is still, an extraordinary challenge for computer scientists. Due to the strategic complexity of Go and the sheer number of moves available to each player, Go provides the perfect test framework for cutting-edge artificial intelligence research. We choose to focus on one aspect of computer Go, the move prediction problem. Given a database of Go games, the problem of move prediction asks how a computer program can train on a training set of games and subsequently be able to predict moves in a test set of games. Research in this field of study can be applied directly to creating general Go-playing programs. An accurate move predictor can be used to reduce the branching factor of game tree search, and can also be used as an effective move ordering heuristic. Move predictors have been successfully employed in some of the best computer Go programs, such as CrazyStone [10] [12] [11].

In this thesis, we make two main contributions to the field of move prediction in the game of Go. Our first main contribution to this field is the creation of a novel move prediction system, based on a naive Bayes model, which builds upon the work of several previous move prediction systems. Our move prediction system achieves competitive results in terms of move prediction accuracy when tested on professional games and high-ranking amateur games. Our system is simple, fast to train, and easy to implement. Our second main contribution is that we describe in detail the process of implementing the framework for our move prediction system, such that future researchers can quickly reproduce our results and test new ideas using our framework.

We begin by giving an introduction to the game of Go and relevant work in computer

Go. In Sections 1.1 and 1.2, we provide a brief history of Go and overview of the rules of Go. In Section 1.3, we compare Go to other similar games and explain why Go is a hard problem for computers. In Sections 1.4 and 1.5, we discuss the best computer Go programs in recent years and the techniques that they employ, as well as several other less successful artificial intelligence techniques that have been used in computer Go. In Section 1.6, we survey recent work in the field of move prediction in the game of Go.

1.1 History

The game of Go, also known as Igo in Japan, Weiqi in China, and Baduk in Korea, is over three thousand years old, first mentioned in Chinese works as early as 1000 B.C.E. In ancient China, Go was an art to be mastered by the gentlemen of society, along with poetry, calligraphy, and painting. Legends are told of wars that were settled over games of Go instead of wasting lives on the battlefield. By the time of the Tang (618-906 C.E.) and Song (960-1126 C.E.) dynasties, books about the rules of Go and Go strategy were published and widely distributed. Go became wildly popular in China, and many players developed fantastic skill in the game [37].

Around 735 A.D., Go spread to Japan. Most likely, an envoy to China brought the game back with him to Japan after learning the game in China. Another legend tells of a Go game in 738 A.D. between two noblemen, in which one nobleman killed his opponent with his sword after losing the game. Over the next few hundred years, the game slowly spread throughout Japan. Go became a spectator sport in the court of the emperor, and soon several prominent Go experts emerged as the *de facto* Go masters of Japan. These Go players were allowed to set up schools for Go instruction, and became full-time Go players and teachers [41].

Go is now a national pastime in Japan. But Go is not played only in Japan and China; The International Go Federation reports members in seventy-one countries, and the popularity of Go is steadily rising in the United States and Europe. Millions of players worldwide enjoy playing and studying the game of Go [19] [28].

1.2 Rules

The game of Go is played on a 19×19 grid (Figure 1.1), although it is common for beginners to play on smaller board sizes, including 7×7 , 9×9 , and 13×13 . There are two players, Black and White, who take turns placing stones of their own color on the intersections of the board. Black always plays first, and each player may place at most one stone per turn. Each player's objective is to maximize the amount of territory that player controls at the end of the game plus the number of enemy stones that player has captured and taken as prisoners. A player can capture an enemy group of stones by completely surrounding the group, removing all of its "liberties," which are adjacent empty intersections, after which the captured stones are removed from the board. Players may pass when they decide that they

can make no move which will enlarge their territory or reduce their opponent's territory. When both players pass in succession, the game ends. The two players decide which stones are "alive," and hence remain on the board, and which stones are "dead," and are hence removed from the board and taken as prisoners. Each player's score is calculated from the amount of territory that player controls minus the number of that player's own stones taken as prisoners by the opposing player. The player with the highest score wins.

For readers unfamiliar with the rules and terminology of Go, we have provided a detailed discussion of the rules of Go in Appendix A, as well as a dictionary of commonly-used Go terminology in Appendix B. We will make extensive use of the terms in Appendix B throughout the paper.

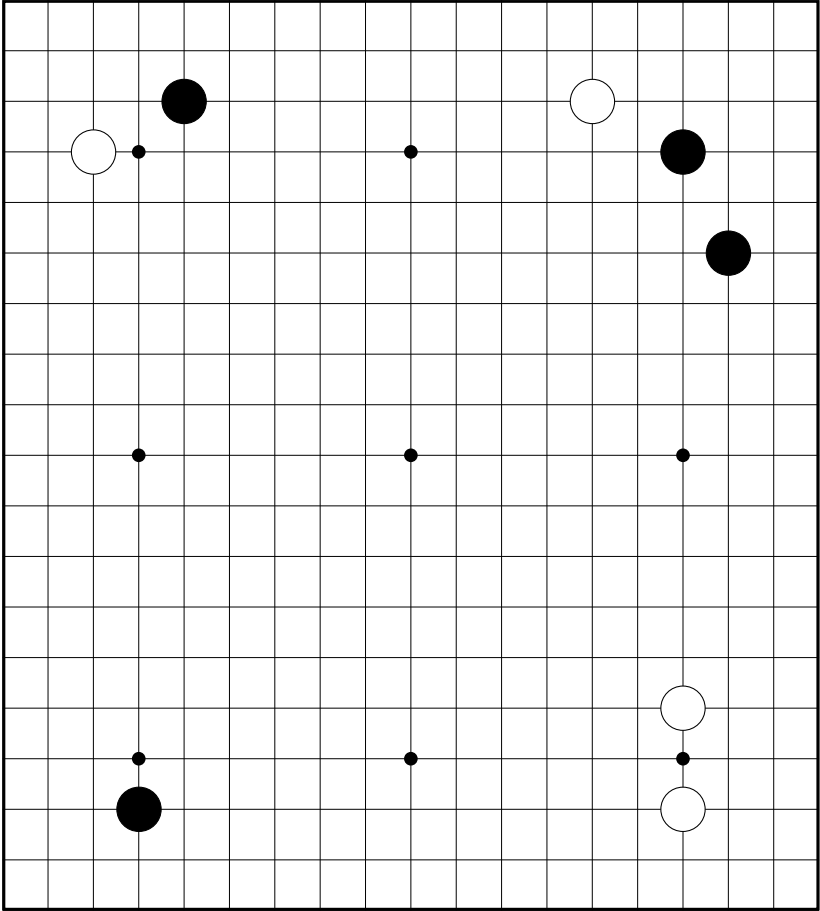


Figure 1.1: The standard 19×19 Go board, after several opening moves

1.3 Why is Go hard?

Games have been a main focus of computer science research since the invention of the modern computer. For many popular games, including Checkers, Chess, Othello (Reversi), Backgammon, and Go-moku, algorithms have been developed that enable computers to play these games. These algorithms rival human ability, and in some cases, even surpass it. For Checkers and Go-moku¹, researchers have created provably unbeatable programs [35] [1]. World-champion level computer players have been developed for Chess [7], Othello [5], and Backgammon [39]. But the best computer Go players compete at the mid-kyu level. (For an explanation of the Go ranking system, see the definitions of Kyu and Dan in Appendix B.)

From a game theoretic standpoint, the game of Go possesses many of the same characteristics as Chess, Checkers, and Othello. In particular, Go has the following properties:

Zero-sum Every gain for white is a loss for black, and vice-versa. There is no possibility for cooperation in Go.

Deterministic There is no randomness involved in Go. Games such as Backgammon are *non-deterministic* if they involve elements of randomness, such as rolling a die.

Two-player This is the simplest kind of multiplayer game. Games such as Poker may involve more than two players.

Perfect-information Both players can see the entire board at all times. Games such as Poker are *imperfect-information*, because players have their own private information which is not available to other players.

So why then is Go so much harder than its siblings in the family of two-player, zero-sum, deterministic games? There is no definite answer, but there are several likely reasons:

Game complexity When measured by any reasonable metric, the game complexity of Go is enormous. There are two standard choices for measuring game complexity:

1. The *state-space complexity* of a game is the number of possible configurations of the board. In other words, it is the number of different legal states of the game reachable from the beginning of the game. The state-space complexity of 19×19 Go is $\approx 10^{171}$, compared to $\approx 10^{50}$ for Chess and $\approx 10^{18}$ for Checkers.
2. The *game tree size* is the number of leaves in the fully enumerated game tree of the game. In other words, it is the number of different games that can be played. The game tree size of 19×19 Go is $\approx 10^{360}$, compared to $\approx 10^{123}$ for Chess and $\approx 10^{54}$ for Checkers.

We summarize these facts in Table 1.1 [1].

Since the complexity of Go is so large, standard algorithms, such as depth-limited minimax search with alpha-beta pruning, are ineffective, since even reaching shallow depths of the game tree requires colossal amounts of time and space resources.

¹Go-moku is a game of five-in-a-row played with the same board and stones as Go.

Game	Board size	State-space	Game tree size
Checkers	8×8	10^{18}	10^{54}
Chess	8×8	10^{50}	10^{123}
Go	19×19	10^{171}	10^{360}

Table 1.1: Game complexity of Checkers, Chess, and Go

Board evaluation Games such as Chess, Checkers, Othello, and Go-moku admit computationally efficient *heuristic evaluation functions*, functions which assign a real number to a state of the board which indicates how “good” or “bad” the board is for a player. Go has not been found to admit any simple and effective evaluation function. One reason for this phenomenon is that even single stones can have substantial effects on the game as the game progresses, and furthermore perturbing the position of a single stone to a new position one space away can completely change an outcome. An effective evaluation function would require deep insight to be able to evaluate these effects.

Local and global positions The outcome of a game is the result of complex interactions between global positions of chains of stones which may determine overall trends in territory, and local positions of stones that determine how localized battles for territory proceed. Even at the amateur level, the outcome of a game of Go is not robust to small perturbations in stone positions, since such perturbations can greatly affect life and death on the board. For this reason, standard pattern recognition algorithms in machine learning cannot be naively applied to Go.

Simple ruleset and large board The rules of Go can be explained to a child in one sitting. However, one may spend his or her entire lifetime improving and mastering Go playing ability. Since the ruleset is quite simple with very few restrictions on stone placement, and since the size of the board is so large in comparison to other games, Go strategy can become immensely complex. In some sense, humans have the innate ability to use much more creativity in adopting and creating Go strategy than computers. It is not unusual to hear Go professionals and instructors compare Go to art.

1.4 Recent Computer Go Programs

The best-performing computer Go programs in recent times have been GNU Go [32], Go++ [34], Crazy Stone [10] [12], Many Faces of Go [22] [21], and MoGo [45]. These programs have become prominent in the computer Go scene as winners in computer Go tournaments such as the KGS Computer Go Tournament and as commercial programs available for purchase. According to their creators, these programs perform at various rankings between 9-kyu and 2-kyu, although the exact rank of these programs is subject to dispute. However, there has definitely been progress and success, due to recent advances in Monte-Carlo Go. Recently, the MoGo computer program defeated an 8-dan professional by 1.5 points in a 9-stone

game [20], which places the strength of MoGo somewhere in the single-digit kyu level (see the definition of Handicap in Appendix B).

These programs all use some combination of the techniques that we describe in the next subsections.

1.4.1 Goal Search

Instead of searching for max-min nodes, goal search looks for nodes in the game tree which accomplish specific tasks. In Go, these tasks include establishing life, death, eyes, connections, cuts, safety of territory, and captures. As one of the earliest concepts in computer Go, goal search is an over-simplified approach to alpha-beta search that has not by itself proved effective. Human knowledge of Go strategy incorporates more subtle complexity than simple boolean combinations of high-level tasks [30].

1.4.2 Heuristic Evaluation Functions

Despite the difficulty of creating heuristic evaluation functions, the best programs still use basic heuristics to direct search. For example, stones that are in atari may lead the evaluator to value highly moves that save the stones. As a similar example, the evaluator may value moves which result in the capture of opponent's stones and save the program's own stones. Evaluation functions in the best programs are usually quite complex, involving estimations of life, death, and territory. Go researchers are also exploring algorithms for learning evaluation functions using machine learning techniques. For many of the best programs, these evaluation functions have proved useful as move ordering heuristics which cut down the effective branching factor of alpha-beta search [30] [42] [31] [14].

1.4.3 Monte-Carlo Tree Search

Monte-Carlo tree search has been one of the most effective strategies for playing computer Go, and has shown great success with the programs Crazy Stone and MoGo. Monte-Carlo tree search is a game tree search that uses a Monte-Carlo heuristic evaluation function to evaluate potential moves. This evaluation plays out thousands of random games from potential move nodes in the game tree. The move chosen is the move which generates the best set of random games. Surprisingly, this random evaluation function performs quite well. The intuition is that the heuristic favors moves which make connections and increase the liberties of groups, which is often a desired outcome for the player. Moreover, the Monte-Carlo evaluation function is cheaper to compute than most current board evaluation functions for Go, especially towards the middle and end of a typical game in which a winner can be determined in fewer random moves. The problem with Monte-Carlo tree search is that it can easily overlook crucial moves, and in general tends to play very non-traditional moves [24] [12] [13] [16].

The game of Go involves a constant tradeoff between exploration (establishing presence in unclaimed parts of the board) and exploitation (pursuing local battles for territory). To

approach the exploration-exploitation problem, MoGo and most current Monte-Carlo tree search programs use an algorithm called UCT (Upper bound Confidence for Tree), which was originally developed for the multi-armed bandit problem, to guide the tree search. Many other UCT-based Go programs have been developed in recent years since the combined success of UCT and Monte-Carlo tree search [13] [6] [45].

1.4.4 Table Lookups

All of the most successful Go programs depend on storing tables of *joseki* (sequences of moves that result in fair outcomes for both black and white), *fuseki* (sequences of moves in the first few dozen moves of the game), and end-game maneuvers in their programs. The obvious disadvantage of such a technique is that it cannot possibly store all the patterns and situations that may occur in a game. Professionals will often remark that just memorizing joseki is insufficient; knowing exactly the situations in which to use particular joseki is essential to making use of joseki at all.

1.5 Other Computer Go Techniques

1.5.1 Neural Networks

Artificial neural networks have been used in several computer Go programs, including NeuroGo and GoNN. In some cases, neural networks are used to learn evaluation functions, while in others, neural networks are employed for global and local pattern recognition. Due to the large board size and relatively few move restrictions in Go, many artificial neurons and connections are required to learn an evaluation function of any reasonable complexity. In addition, many layers are required to capture the complex interactions of stones in different parts of the board [18] [33].

1.5.2 Temporal Difference and Reinforcement Learning

Temporal difference (TD) learning and reinforcement learning proved to be successful game playing techniques for other games, including Backgammon. The large state space of Go prevents these techniques from being effective when applied directly to the game of Go. [17] [15] [42].

1.5.3 Cellular Automata

There has been some research in the applications of cellular automata to Go, due to the high resemblance of cellular automata to evolving Go board positions. In one paper, authors tried learning cellular automata rules from expert games. The number and type of different rules grew exponentially large, and so this type of learning was heavily space intensive, and moreover did not perform well in practice. While there has not been much success in this area, there is much ground yet to be explored [23].

1.6 Recent Work in Move Prediction

Move prediction, the main subject of this paper, is a supervised learning problem: Given a database of Go games, the problem of move prediction asks how a computer program can train on a training set of games and subsequently be able to predict moves in a test set of games. Move prediction is currently a relatively unexplored field. However, there is great potential for move prediction to become an essential component of Go programs. The main applications of move prediction in game-tree search are branching factor reduction and move ordering. Suppose that an accurate move predictor can predict all professional moves within its top 30 predictions. Then by limiting a game-tree search to the set of the 30 most likely moves to be played at each step in the game, the move predictor can reduce the branching factor of search from about 250 to 30, a tremendous improvement. Alternatively, a Go program that uses search-based algorithms can use the move ranking abilities of a move predictor as an effective move ordering heuristic. Move predictors have been successfully employed in some of the best computer Go programs, such as CrazyStone [10] [12] [11].

There are several related papers that have appeared in the last decade that have attempted to solve the move prediction problem. In [43], the authors focus on the problem of local move prediction, which asks if a program can distinguish between correct moves and all other moves within a small neighborhood around each correct move. Their method involves first extracting pattern-based and tactical features from moves on the board, and then running dimensionality reduction algorithms on the extracted feature data. Next, they train a multi-layer perceptron on the lower-dimensional data to rank moves based on the likelihood that they will be played. The training moves consist of the actual moves played which are labeled with a positive class for training the network, and random moves within a local neighborhood of each played move which are labeled with a negative class for training the network. They train their network on 200,000 moves (100,000 correct-move, random-move pairs) from amateur games played on the Pandanet Internet Go Server (IGS), which we approximate is about 400 games in total. Then they test their predictor on 52 professional games. In local neighborhoods the predictor ranks 48% of the professional moves first. On the full board the predictor ranks 25% of the correct moves first, 45% within the best three, and 80% in the top 20.

In [38], the authors focus on using pattern matching to solve the move prediction problem. They extract patterns of different shapes and sizes from a database of 181,000 professional games, extracting a total of 12,000,000 patterns. They also extract several types of tactical features to aid in prediction. With these patterns, they train on all 181,000 games (approximately 250,000,000 moves) with a Bayesian ranking model in order to train their move predictor. On a test set of 450 professional games, their ranking system ranks 34% of the professional moves first, 66% in the top 5, 76% in the top 10, and 86% in the top 20.

In [11], the author presents a novel method for using Bradley-Terry competition models to train a move predictor. Patterns and other features are treated as participants in competitions. Each competition is between the actual move played in a game, which is represented as the winning team of features, and all other legal moves at that point in the game that were not played, which are represented as the losing teams of features. From the

results of these competitions, we can estimate strength parameters for each feature which maximize the likelihood of the results. The strength of any move is simply the product of the strengths of the features associated with that move, and the predictor ranks possible moves according to their strengths. A move predictor using this method is trained on a set of 652 high-ranking amateur games (131,939 moves) on the K Go Server (KGS), with an extracted pattern database of 16,780 patterns. On a test set of 551 high-ranking amateur games (115,832 moves), their ranking system ranks 34.9% of all correct moves first.

Chapter 2

Methods for Move Prediction

“The ancient Japanese considered the Go board to be a microcosm of the universe. Although, when it is empty, it appears to be simple and ordered, the possibilities of gameplay are endless. They say no two Go games have ever been alike, just like snowflakes. So, the Go board actually represents an extremely complex and chaotic universe. And that’s the truth of our world, Max. It can’t be easily summed up with math. There is no simple pattern.”

“But as a Go game progresses, the possibilities become smaller and smaller. The board does take on order. Soon, all the moves are predictable.”

“So, so?”

“So maybe, even though we’re not sophisticated enough to be aware of it, there is a pattern, an order underlying every Go game.”

Darren Aronofsky, Pi, 1998.

In this chapter, we describe algorithms for move prediction in the game of Go. In Section 2.1, we formally define the problem. In Section 2.2, we offer a novel generalization of previous approaches to the move prediction problem, which is extensible to both new feature definitions and using new learning techniques for scoring feature vectors. In Section 2.3, we explain the types of features we extracted from our Go game data sets. In Section 2.4, we describe two different move predictors that train on extracted feature vectors, built with two different machine learning techniques. The first move predictor is our novel naive Bayes move predictor, while the second move predictor is a Bradley-Terry move predictor based on the work of [11]. Since the Bradley-Terry move predictor is currently the best-performing move predictor in the literature, we choose to implement it and reproduce the results of [11] in order to provide a benchmark for comparison. In Section 2.5, we describe the performance of our naive Bayes move predictor and the Bradley-Terry move predictor when trained and tested on professional and high-ranking amateur games. In Section 2.6, we offer directions for future research.

2.1 The Move Prediction Problem

Informally, the move prediction problem asks how well a computer can predict moves in a game of Go. Given a training set of games and a test set of games, can we build a program which trains on the training set in order to predict which moves are played in the test set?

Previous papers in the field of Go move prediction only define the move prediction problem informally. We offer a more rigorous mathematical definition of the move prediction problem. In this chapter, we assume that all Go games are played on 19×19 boards. First, we define a *move* as a representation of a move in a Go game. For all moves m ,

$$m \in \{\text{BLACK}, \text{WHITE}\} \times \{1, \dots, 19\} \times \{1, \dots, 19\}.$$

For example, the move $m = (\text{BLACK}, 16, 3)$ places a black stone at the intersection with x -coordinate 16 and y -coordinate 3. Next, we define a *board state* S as a mapping from coordinates of intersections on the board to the possible states of that intersection, i.e.

$$S : \{1, \dots, 19\} \times \{1, \dots, 19\} \rightarrow \{\text{BLACK}, \text{WHITE}, \text{EMPTY}\}.$$

For example, if S the current state of the board after $m = (\text{BLACK}, 16, 3)$ is played, then $S(16, 3) = \text{BLACK}$.

Now we define a *game*, denoted $G = (T, \mathbf{m}, \mathbf{S}, \mathbf{L})$, as a representation of a game of Go, where

- T is an integer which represents the number of moves played in G ,
- $\mathbf{m} = (m_1^G, \dots, m_T^G)$ is an ordered list where for $t \in \{1, \dots, T\}$, m_t^G is move number t in game G ,
- $\mathbf{S} = (S_0^G, \dots, S_{T-1}^G)$ is an ordered list where for $t \in \{1, \dots, T\}$, S_{t-1}^G is the board state just before m_t^G is played,
- $\mathbf{L} = (L_0^G, \dots, L_{T-1}^G)$ is an ordered list where for $t \in \{1, \dots, T\}$, L_{t-1}^G is the set of legal moves available just before move m_t^G is played, and so $m_t^G \in L_{t-1}^G$. We denote legal moves in L_{t-1}^G as $\ell_k^{G,t-1}$ for $k \in \{1, \dots, |L_{t-1}^G|\}$.

For convenience, we define corresponding functions $T(G) = T$, $\mathbf{m}(G) = \mathbf{m}$, $\mathbf{S}(G) = \mathbf{S}$, and $\mathbf{L}(G) = \mathbf{L}$.

We define a *move predictor* \mathcal{P} as a function which takes as input a set of training games $\mathbf{G} = \{G_1, \dots, G_n\}$, a board state S , and a move m , and outputs a score $\sigma \in \mathbb{R}$ which corresponds to the likelihood that m will be played in a game with current board state S , given that the predictor has trained on \mathbf{G} . For convenience, we let $\mathcal{P}_{\mathbf{G}}$ denote the predictor \mathcal{P} trained on \mathbf{G} , so that we can write $\sigma = \mathcal{P}_{\mathbf{G}}(S, m)$ for board states S and moves m .

Finally, given a move predictor \mathcal{P} and two sets of games \mathbf{G}_1 and \mathbf{G}_2 , we define the *accuracy* of \mathcal{P} given \mathbf{G}_1 and \mathbf{G}_2 , denoted $\Pi(\mathcal{P}, \mathbf{G}_1, \mathbf{G}_2)$, as the probability that, given a game $G \in \mathbf{G}_2$ chosen uniformly at random from \mathbf{G}_2 and a move number t chosen uniformly

at random from $\{1, \dots, T(G)\}$, the move predictor $\mathcal{P}_{\mathbf{G}_1}$ scores move m_t^G the highest out of all moves in $L_{t-1}^G \in \mathbf{L}(G)$, i.e. for all $k \in \{1, \dots, |L_{t-1}^G|\}$, $\mathcal{P}_{\mathbf{G}_1}(S_{t-1}^G, m_t^G) \geq \mathcal{P}_{\mathbf{G}_1}(S_{t-1}^G, \ell_k^{G,t-1})$. We can estimate the accuracy empirically by counting the proportion of moves in \mathbf{G}_2 that are scored the highest out of the corresponding set of legal moves:

$$\begin{aligned} \Pi(\mathcal{P}, \mathbf{G}_1, \mathbf{G}_2) &\approx \frac{\sum_{G \in \mathbf{G}_2} \sum_{t=1}^{T(G)} \chi(m_t^G, \ell_{k^*}^{G,t-1})}{\sum_{G \in \mathbf{G}_2} T(G)}, \\ k^* &= \arg \max_{k \in \{1, \dots, |L_{t-1}^G|\}} \mathcal{P}_{\mathbf{G}_1}(S_{t-1}^G, \ell_k^{G,t-1}), \\ \chi(m_t^G, \ell_{k^*}^{G,t-1}) &= \begin{cases} 1 & m_t^G = \ell_{k^*}^{G,t-1} \\ 0 & m_t^G \neq \ell_{k^*}^{G,t-1} \end{cases}. \end{aligned}$$

We can now define the *move prediction problem*: Find a move predictor \mathcal{P} which maximizes $\Pi(\mathcal{P}, \mathbf{G}_1, \mathbf{G}_2)$ for any two non-intersecting sets of games \mathbf{G}_1 and \mathbf{G}_2 .

2.2 General Move Prediction Algorithm

How can we design a move predictor \mathcal{P} ? How can we train a machine to correctly estimate the likelihood that a move will be played? One possibility is to train directly on the states of the board in each game. Unfortunately, given that there are $361 = 19 \cdot 19$ intersections on the board which can take one of three possible intersection states (BLACK, WHITE, or EMPTY), there could be as many as $3^{361} \approx 10^{172}$ board states.¹ In a given training set, most board states will almost never be seen more than once, and so it would be difficult for any learning algorithm to have enough sample data to learn a reasonable mapping from board states to move scores.

Another problem with training directly on board states is that if a perfect mapping from board states to move scores existed, this mapping would necessarily be highly discontinuous. Consider for example the two diagrams in Figure 2.1. These two board states are identical except the stone marked \triangle differs by one space. The outcomes for White in these two diagrams are completely opposite; in the left diagram, White's group is unconditionally alive, i.e. it cannot be captured by Black, while in the right diagram, White's group is unconditionally dead, i.e. Black will inevitably capture White's group. (For more information on Life and Death, see Section A.4.)

Building a successful move prediction algorithm requires extraction of higher-level features from board states and moves. This allows a learning algorithm to train on a lower-dimensional data set where the feature values often repeat, even though the board states and moves may be in completely different physical locations in the board. For example, an algorithm may extract a binary feature which is turned on if the current move will capture any opponent stones given the current board state. As another example, an algorithm may

¹Given symmetries and the impossibility of certain board states, the true upper bound is smaller than 10^{172} , but only by a few orders of magnitude.

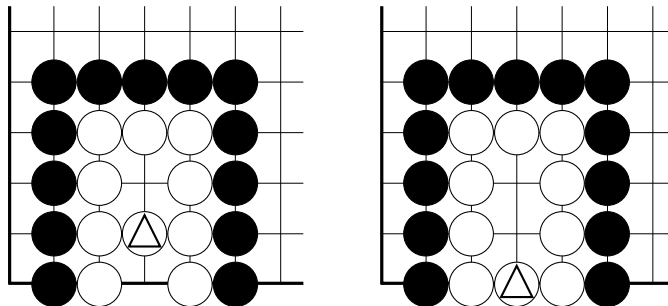


Figure 2.1: Perturbing the location of the marked stone by one space changes the status of White’s group from alive to dead.

extract a feature which encodes the Manhattan distance between the current move and the previous move. These are just two examples of simple features which are inexpensive to compute but collectively transform the states and moves in a training set into a much more manageable data set, on which a suitable move predictor can be trained. Section 2.3 provides more information about feature extraction, specifically describing the features used in our move predictor.

We now generalize previous move predictors by offering a general move prediction algorithm which trains on extracted features (Algorithm 1). Our general algorithm is extensible to both adding new feature types and using new learning techniques for scoring feature vectors. This algorithm relies on a method called `LEARNSCORINGFUNCTION` which takes as input the sets of feature vectors associated with all legal moves, along with their classifications as either 1, i.e. the move is actually played, or 0, i.e. the move is a possible legal move but is not played, and produces a function which assigns a score to any input feature vector, representing the likelihood that the corresponding move will be played. We will describe several choices for `LEARNSCORINGFUNCTION` in Section 2.4. As described above, this algorithm also relies on a method `EXTRACTFEATUREVECTOR` which, given the current board state and a possible legal move, computes the corresponding feature vector.

2.3 Feature Extraction

The goal of feature extraction is to convert a move in a Go game played at a particular board state into a lower-dimensional feature vector which describes the important features of that move. We implemented the extraction of the features listed in the following subsections. These features build upon previous work in feature extraction for move prediction, in particular the work of [11], [43], and [38]. We add one new feature to this list, which captures jumps and knight’s moves (see Section 2.3.10). For definitions of Go terminology, we refer the reader to Appendix B.

Algorithm 1 General move prediction algorithm

```
D  $\leftarrow \emptyset$ 
for all  $G \in \mathbf{G}$  do
  for  $t = 1$  to  $T(G)$  do
     $D \leftarrow \emptyset$ 
    for  $k = 1$  to  $|L_{t-1}^G|$  do
       $F \leftarrow \text{EXTRACTFEATUREVECTOR}(S_{t-1}^G, \ell_k^{G,t-1})$ 
      if  $\ell_k^{G,t-1} = m_t$  then
         $D = D \cup (F, 1)$ 
      else
         $D = D \cup (F, 0)$ 
      end if
    end for
  D  $\leftarrow \mathbf{D} \cup D$ 
end for
 $\mathcal{P}_{\mathbf{G}} \leftarrow \text{LEARNSCORINGFUNCTION}(\mathbf{D})$ 
return  $\mathcal{P}_{\mathbf{G}}$ 
```

2.3.1 Capture type

This feature captures the various types of captures that can occur. The different capture types are illustrated in Figure 2.2. The possible values are:

- 0 No capture occurred.
- 1 The capture prevents the player's own capture. This occurs when a chain of stones belonging to the current player has been placed in atari (i.e. has one liberty left and hence is in danger of being captured on the next move), but the current move captures stones, and as a result of the capture the player's chain that was in atari is no longer in atari.
- 2 The move captures a stone whose placement in the previous move captured stones of the current player.
- 3 The capture prevents a connection from the captured stones to another chain of enemy stones.
- 4 The move captures a chain of stones that are not currently in a ladder (see the definition of Ladder in Appendix B).
- 5 The move captures a chain of stones that are currently in a ladder.

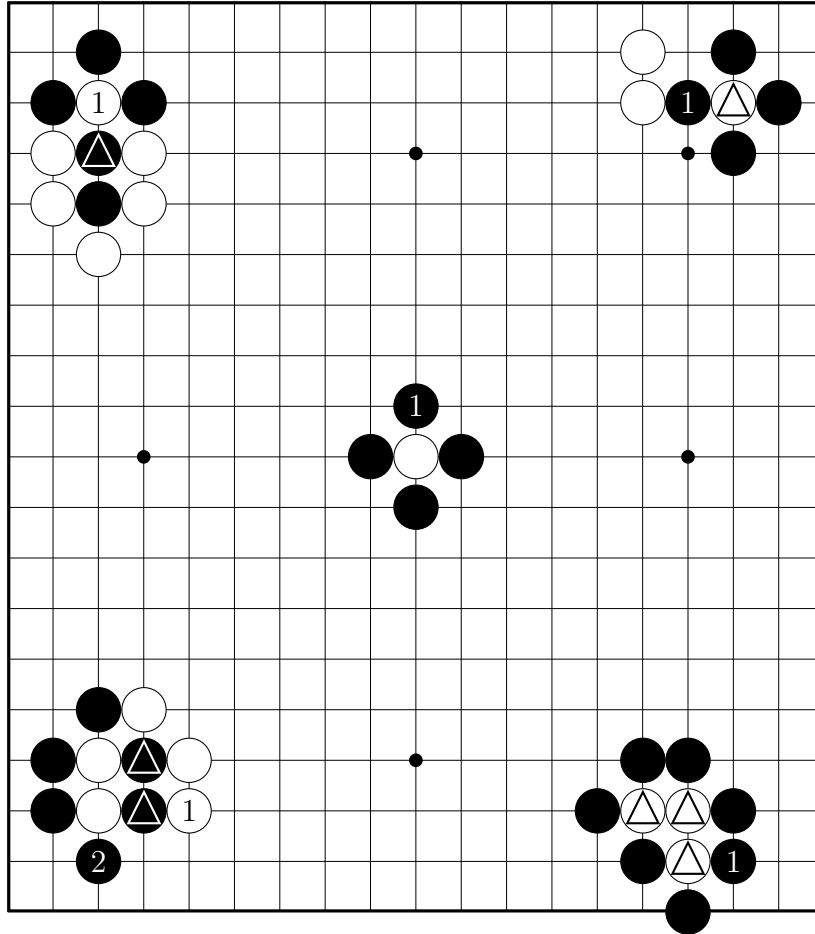


Figure 2.2: Capture type feature. *Capture type 1* (bottom left): After White 1, the marked black stones are in atari. After the capture at Black 2, the marked black stones are no longer in atari. *Capture type 2* (top left): White 1 captures two black stones. Black 2, played at the previous location of the marked black stone, recaptures White 1. *Capture type 3* (top right): Black 1 captures the marked White stone, preventing White from making an extension to the nearby group by playing at Black 1. *Capture type 4* (center): Black 1 captures one white stone, which is not in a ladder. *Capture type 5* (bottom right): Black 1 captures the three marked white stones, which are currently in a ladder, i.e. they cannot avoid capture.

2.3.2 Capture, number of stones

This feature represents the number of stones n that are captured as a result of the current move. If $0 \leq n \leq 6$, then the feature takes value n . Otherwise, if $n > 6$, then the feature takes value 7. We add this feature with the intuition that capturing larger groups is more urgent than capturing smaller groups.

2.3.3 Extension

This feature represents several different types of extensions from atari. The possible values are:

- 0 No extension occurred.
- 1 The move is an extension of a chain in atari that is not currently in a ladder.
- 2 The move is an extension of a chain in atari that is currently in a ladder.

2.3.4 Self-atari

This is a binary feature which has value 1 if the current move places the current player's own chain in atari. The possible values are:

- 0 The move does not place any chain owned by the current player in atari.
- 1 The move places a chain owned by the current player in atari.

2.3.5 Atari

This feature represents several different types of atari, which are moves that reduce an enemy chain to one liberty. the possible values are:

- 0 The move does not place any enemy chain in atari.
- 1 The move places an enemy chain in atari, and as a result of the move, that enemy chain is in a ladder.
- 2 The move places an enemy chain in atari while there is a ko point on the board.
- 3 The move places an enemy chain in atari and does not fall into any of the categories above.

2.3.6 Distance to border

This feature measures the distance from the move to the border of the board, which is defined to be the minimum distance from the coordinate of the move to the closest point on each of the four board edges. The feature takes value d if the distance to the border is d and $d \leq 6$. Otherwise, if $d > 6$, the feature takes value 7.

2.3.7 Distance to previous move

This feature measures the distance from the current move to the previous move. If the current move has coordinates (x_1, y_1) and the previous move has coordinates (x_2, y_2) , the distance between the moves is calculated as

$$d = |x_1 - x_2| + |y_1 - y_2| + \max(|x_1 - x_2|, |y_1 - y_2|).^2$$

The feature takes value d if the distance between the moves is d and $d \leq 16$. Otherwise, if $d > 16$, the feature takes value 17. Note that if the current move is the first move of the game, we set this feature to 17 by default.

2.3.8 Distance to move before previous move

This feature measures the distance from the current move to the move before the previous move. If the current move has coordinates (x_1, y_1) and the move before the previous move has coordinates (x_2, y_2) , the distance between the moves is calculated as

$$d = |x_1 - x_2| + |y_1 - y_2| + \max(|x_1 - x_2|, |y_1 - y_2|).$$

The feature takes value d if the distance between the moves is d and $d \leq 16$. Otherwise, if $d > 16$, the feature takes value 17. Note that if the current move is the first or second move of the game, we set this feature to 17 by default.

2.3.9 Monte-Carlo owner

This feature is an innovation of [11]. Given the current board state, we play 63 *Monte-Carlo* *playouts*. In a Monte-Carlo playout, an entire random game is played out from the current board state, where each move is chosen uniformly at random from the legal moves available to that player, except for moves which fill simple eyes (see the definition of Simple Eye in Appendix B). The playout finishes when there are no available moves left for either player. Then for the given move, we count the number of times c that the current player owns the coordinate of the move at the end of the 63 games. We then set the value of this feature to $n = \lfloor c/8 \rfloor$, so this feature takes on values 0 through 7, depending on how many times the player owns the corresponding coordinate at the end of the 63 Monte-Carlo playouts.

The intuition with this feature is that its value gives a heuristic estimate of the probability that a player will own a position at the end of the game. If the probability is very high, then the player likely already controls that point at the current board state, so playing at that point may be a waste of a move. If the probability is very low, then the enemy likely already controls that point at the current board state, so playing at that point may be also

²The added $\max(|x_1 - x_2|, |y_1 - y_2|)$ term allows us to distinguish between certain moves which may have the same Manhattan distance from the previous move but have different spatial relationship with respect to the previous move. For example, 2-space knight's moves and 3-space jumps are distinguished with this method of calculating distance.

be a waste of a move. If the probability is close to 0.5, then that point may be a key move which decides who will claim that position at the end of the game.

2.3.10 Jumps and Knight's moves

We encode different size jumps and knight's moves with this feature. These kinds of moves are typical in the game of Go, and are often played to support existing stones and build territorial frameworks on the board. We define a move as an n -space jump if there is another stone owned by the same player that is n intersections away in one of the four cardinal directions (north, south, east, west) from the move, and if there are no enemy stones along the line between the two stones. We define a move as an n -space knight's move if there is another stone owned by the same player that is $n + 1$ intersections away in one of the four cardinal directions and one intersection away in an orthogonal direction from the move, and if there are no enemy stones in the rectangle with opposite corners positions at the two stones. If this move is a jump or knight's move from at more than one stone, then this feature is set to the jump or knight's move that is closest in Manhattan distance from the nearby stone. The possible values are:

- 0 The move is not an n -space jump or n -space knight's move away from any other stone owned by the same player, for $0 \leq n \leq 4$.
- 1 The move is a 0-space jump, i.e. an adjacent connection.
- 2 The move is a 1-space jump.
- 3 The move is a 2-space jump.
- 4 The move is a 3-space jump.
- 5 The move is a 4-space jump.
- 6 The move is a 0-space knight's move, i.e. a diagonal play or "shoulder hit".
- 7 The move is a 1-space knight's move (same as a knight's move in Chess).
- 8 The move is a 2-space knight's move.
- 9 The move is a 3-space knight's move.
- 10 The move is a 4-space knight's move.

2.3.11 Pattern match

This is a very effective feature for move prediction. We extract all the 3×3 , 5×5 , 7×7 , and 9×9 patterns centered at every move in a set of about 25,000 games.³ We discard patterns that appear with frequency below a certain threshold. This feature’s value is set to the index of the largest pattern matched in the database, or a value of 0 if no pattern is matched.

2.4 Machine Learning Techniques for Scoring Feature Vectors

How can we train on vectors comprised of the above features in order to score moves? In this section, we give two possible machine learning techniques that can be fitted to this problem.

2.4.1 Naive Bayes

In this section, we describe our novel method for scoring feature vectors, which we use to produce a move predictor that is fast to train, fast to score unseen move instances, and easy to implement. Our simple method for scoring feature vectors uses a naive Bayes model to estimate the probability that a move will be played. Let f_1, \dots, f_n denote features for a move and let $\vec{v} = (v_1, \dots, v_n)$ denote a vector of feature values. Suppose we have a function \mathcal{C} which maps each feature vector to a binary classification. Then by Bayes’ rule, the probability that a move m with extracted feature vector (v_1, \dots, v_n) has classification 1 is given by

$$\mathbb{P}(\mathcal{C}(\vec{v}) = 1 \mid f_1 = v_1, \dots, f_n = v_n) = \frac{\mathbb{P}(\mathcal{C}(\vec{v}) = 1) \cdot \mathbb{P}(f_1 = v_1, \dots, f_n = v_n \mid \mathcal{C}(\vec{v}) = 1)}{\mathbb{P}(f_1 = v_1, \dots, f_n = v_n)}.$$

In a naive Bayes model, we assume that each feature is conditionally independent of all other features given the class. With the naive Bayes assumption, the model becomes

$$\begin{aligned} \mathbb{P}(\mathcal{C}(\vec{v}) = 1 \mid f_1 = v_1, \dots, f_n = v_n) &= \frac{\mathbb{P}(\mathcal{C}(\vec{v}) = 1) \cdot \mathbb{P}(f_1 = v_1, \dots, f_n = v_n \mid \mathcal{C}(\vec{v}) = 1)}{\sum_{c \in \{0,1\}} \mathbb{P}(\mathcal{C}(\vec{v}) = c) \cdot \mathbb{P}(f_1 = v_1, \dots, f_n = v_n \mid \mathcal{C}(\vec{v}) = c)} \\ &= \frac{\mathbb{P}(\mathcal{C}(\vec{v}) = 1) \cdot \prod_{i=1}^n \mathbb{P}(f_i = v_i \mid \mathcal{C}(\vec{v}) = 1)}{\sum_{c \in \{0,1\}} \mathbb{P}(\mathcal{C}(\vec{v}) = c) \cdot \prod_{i=1}^n \mathbb{P}(f_i = v_i \mid \mathcal{C}(\vec{v}) = c)}. \end{aligned} \quad (2.1)$$

For a move m and board state S , we define a binary classification function \mathcal{C} such that if $\vec{v} = (v_1, \dots, v_n)$ is the feature vector extracted from m and S in game G , then $\mathcal{C}(\vec{v}) = 1$ if $m \in \mathbf{m}(G)$, and $\mathcal{C}(\vec{v}) = 0$ otherwise. We define the *naive Bayes move predictor* to be

³We actually extract two different sets of patterns, one from a set of about 25,000 professional games and another from a set of about 25,000 high-ranking amateur games.

Algorithm 1 with `LEARNSCORINGFUNCTION(D)` defined as the method which estimates the naive Bayes model parameters and returns the move predictor $\mathcal{P}_{\mathbf{G}}$, where for move m and board state S ,

$$\mathcal{P}_{\mathbf{G}}(S, m) = \mathbb{P}(\mathcal{C}(\vec{v}) = 1 \mid f_1 = v_1, \dots, f_n = v_n), \quad (2.2)$$

where

$$\vec{v} = (v_1, \dots, v_n) = \text{EXTRACTFEATUREVECTOR}(S, m). \quad (2.3)$$

We show the definition of `LEARNSCORINGFUNCTION(D)` in Algorithm 2. For convenience, we denote the number of feature values for feature f_i as $|f_i|$, and we denote the j th possible feature value for feature f_i as v_i^j . In practice, we initially set all parameters to some small $\epsilon > 0$ to avoid problems that arise from multiplying or dividing by zero, such as when a feature value is seen in a test set but is not present in the training set.

Note that the typical use of naive Bayes models is to classify feature vectors, i.e. to find c which maximizes

$$\mathbb{P}(\mathcal{C}(\vec{v}) = c \mid f_1 = v_1, \dots, f_n = v_n).$$

Our novel idea is to use a naive Bayes model to find which feature vector is most likely to have a class of 1, i.e. to find $\vec{v} = (v_1, \dots, v_n)$ which maximizes

$$\mathbb{P}(\mathcal{C}(\vec{v}) = 1 \mid f_1 = v_1, \dots, f_n = v_n).$$

Algorithm 2 `LEARNSCORINGFUNCTION(D)` for naive Bayes move predictor

```

Initialize  $N_C[]$ ,  $N_F[][][]$ 
for all  $(\vec{v} = (v_1, \dots, v_n), c) \in \mathbf{D}$  do
     $N_C[c] \leftarrow N_C[c] + 1$ 
    for  $i = 1$  to  $n$  do
         $N_F[i][v_i][c] \leftarrow N_F[i][v_i][c] + 1$ 
    end for
end for
 $\mathbb{P}(\mathcal{C}(\vec{v}) = 0) \leftarrow N_C[0] / (N_C[0] + N_C[1])$ 
 $\mathbb{P}(\mathcal{C}(\vec{v}) = 1) \leftarrow N_C[1] / (N_C[0] + N_C[1])$ 
for  $i = 1$  to  $n$  do
    for  $j = 1$  to  $|f_i|$  do
         $\mathbb{P}(f_i = v_i^j \mid \mathcal{C}(\vec{v}) = 0) \leftarrow N_F[i][v_i^j][0] / \sum_{k=1}^{|f_i|} N_F[i][v_i^k][0]$ 
         $\mathbb{P}(f_i = v_i^j \mid \mathcal{C}(\vec{v}) = 1) \leftarrow N_F[i][v_i^j][1] / \sum_{k=1}^{|f_i|} N_F[i][v_i^k][1]$ 
    end for
end for
return  $\mathcal{P}_{\mathbf{G}}$ , as defined by Equations 2.1, 2.2, and 2.3

```

2.4.2 Bradley-Terry Move Ranking

The Model

Bradley-Terry models are simple but powerful methods for modeling the skill levels of participants in a competition. In [11], the author uses methods for optimizing generalized Bradley-Terry models in order to create an effective move predictor. The optimization of generalized Bradley-Terry models is described in detail in [27]. We reproduce the move predictor of [11] to provide a comparison for our naive Bayes move predictor.

In a Bradley-Terry model, each participant in a competition is associated with a strength parameter γ_i . The probability that player i defeats player j in a pairwise competition is defined as

$$\mathbb{P}(i \text{ beats } j \mid \gamma_i, \gamma_j) = \frac{\gamma_i}{\gamma_i + \gamma_j}.$$

In a general Bradley-Terry model, participants may compete as members of one or more teams in a competition, and that competition may occur between multiple teams. If n teams $\mathcal{T}_1, \dots, \mathcal{T}_n$ compete, the probability that team \mathcal{T}_k wins the competition is defined as

$$\mathbb{P}(\mathcal{T}_k \text{ wins} \mid \gamma) = \frac{\prod_{i \in \mathcal{T}_k} \gamma_i}{\sum_{\ell=1}^n \prod_{i \in \mathcal{T}_\ell} \gamma_i}.$$

For example, if $\mathcal{T}_1 = \{1, 3, 5\}$, $\mathcal{T}_2 = \{1, 2, 3\}$, and $\mathcal{T}_3 = \{4, 6\}$ compete, the probability that \mathcal{T}_1 wins is

$$\mathbb{P}(\mathcal{T}_1 \text{ wins} \mid \gamma_1, \dots, \gamma_6) = \frac{\gamma_1 \gamma_3 \gamma_5}{\gamma_1 \gamma_3 \gamma_5 + \gamma_1 \gamma_2 \gamma_3 + \gamma_4 \gamma_6}.$$

We can translate the move prediction problem into a generalized Bradley-Terry model. Suppose that for every feature f_i , every possible feature value v_i^j for $j \in \{1, \dots, |f_i|\}$ is associated with a strength parameter γ_i^j . For notational convenience, given a feature vector $\vec{v} = (v_1^{j_1}, \dots, v_n^{j_n})$ we write $\gamma_i^{j_i}$ as $\gamma[f_i(\vec{v})]$. At a given point in the game, there are a set of available moves $L = \{\ell_1, \dots, \ell_K\}$ where each ℓ_k corresponds to a feature vector \vec{v}_k . Then we model the probability that ℓ_k is played over all other moves as

$$\mathbb{P}(\ell_k \text{ is played} \mid \gamma) = \frac{\prod_{i=1}^n \gamma[f_i(\vec{v}_k)]}{\sum_{k'=1}^K \prod_{i=1}^n \gamma[f_i(\vec{v}_{k'})]}.$$

In other words, we model every move decision as a competition between teams of feature values, where the strength of a move is defined as the product of the strengths of its corresponding feature values, and where the probability that a move is chosen is the ratio of that move's strength to the sum of all moves' strengths.

Estimating Strength Parameters

Bradley-Terry models are most useful for estimating strength parameters from a set of competition results. Given a set of competition results \mathbf{R} involving n participants, we want to find strength parameters $\gamma = (\gamma_1, \dots, \gamma_n)$ that maximize the likelihood of the strength

parameters given the results, i.e. that maximize $\mathbb{P}(\gamma | \mathbf{R})$. By Bayes' rule, this equivalent to finding γ which maximizes $\mathbb{P}(\mathbf{R} | \gamma) \cdot \mathbb{P}(\gamma) / \mathbb{P}(\mathbf{R})$. The term $\mathbb{P}(\mathbf{R})$ is a normalizing constant that does not affect the maximization of $\mathbb{P}(\gamma | \mathbf{R})$, since $\mathbb{P}(\mathbf{R})$ is not dependent on γ . The term $\mathbb{P}(\gamma)$ is a prior distribution over the strength parameters. For convenience, the prior $\mathbb{P}(\gamma)$ is written as $\mathbb{P}(\mathbf{R}' | \gamma)$ where \mathbf{R}' are virtual results that determine the prior. For example, in [11], \mathbf{R}' is set to give every participant one virtual win and one virtual loss in a competition against a virtual opponent with fixed strength parameter 1.0. In this way, we can write $\mathbb{P}(\gamma | \mathbf{R}) \propto \mathbb{P}(\mathbf{R}', \mathbf{R} | \gamma)$, and so maximizing $\mathbb{P}(\gamma | \mathbf{R})$ is equivalent to maximizing $\mathbb{P}(\mathbf{R}', \mathbf{R} | \gamma)$.

To make the finding of γ that optimizes $\mathbb{P}(\mathbf{R}', \mathbf{R} | \gamma)$ more tractable, we assume that the results of each competition are conditionally independent of the results of all other competitions. So we have that

$$\mathbb{P}(\gamma | \mathbf{R}) \propto \prod_{R \in \mathbf{R}' \cup \mathbf{R}} \mathbb{P}(R | \gamma).$$

For the case of move prediction, this is a fair assumption. Although move choice in general depends on previous moves, all relevant information about the current state of the board and how the current move interacts with previous moves should be encoded in the features.

For every competition between n teams $\mathcal{T}_1, \dots, \mathcal{T}_n$, the probability that \mathcal{T}_k wins is given by

$$\mathbb{P}(\mathcal{T}_k \text{ wins} | \gamma) = \frac{\prod_{i \in \mathcal{T}_k} \gamma_i}{\sum_{\ell=1}^n \prod_{i \in \mathcal{T}_\ell} \gamma_i}.$$

We can rewrite $\mathbb{P}(\mathcal{T}_k \text{ wins} | \gamma)$ in terms of γ_i , for every participant i , as

$$\mathbb{P}(\mathcal{T}_k \text{ wins} | \gamma) = \frac{A_i \gamma_i + B_i}{C_i \gamma_i + D_i},$$

where A_i , B_i , C_i , and D_i are factors that do not depend on γ_i . For example, if $\mathcal{T}_1 = \{1, 3, 5\}$, $\mathcal{T}_2 = \{1, 2, 3\}$, and $\mathcal{T}_3 = \{4, 6\}$ compete, the probability that \mathcal{T}_1 wins is

$$\mathbb{P}(\mathcal{T}_1 \text{ wins} | \gamma) = \frac{\gamma_1 \gamma_3 \gamma_5}{\gamma_1 \gamma_3 \gamma_5 + \gamma_1 \gamma_2 \gamma_3 + \gamma_4 \gamma_6}.$$

We can rewrite this equation in terms of γ_1 as

$$\mathbb{P}(\mathcal{T}_1 \text{ wins} | \gamma) = \frac{A_1 \gamma_1 + B_1}{C_1 \gamma_1 + D_1},$$

where $A_1 = \gamma_3 \gamma_5$, $B_1 = 0$, $C_1 = \gamma_3 \gamma_5 + \gamma_2 \gamma_3$, $D_1 = \gamma_4 \gamma_6$. In [27] and [11], it is shown that one can maximize $\mathbb{P}(\mathbf{R}', \mathbf{R} | \gamma)$ for a set of results $\mathbf{R}' \cup \mathbf{R} = (R_1, \dots, R_J)$ by iteratively choosing

a parameter γ_i and updating it according to the formula,

$$\gamma_i \leftarrow \frac{W_i}{\sum_{j=1}^J \frac{C_{ij}}{E_j}}, \quad (2.4)$$

where $\mathbb{P}(R_j | \gamma) = (A_{ij}\gamma_i + B_{ij}) / (C_{ij}\gamma_i + D_{ij})$, $W_i = |\{j \mid A_{ij} \neq 0\}|$, and $E_i = C_{ij}\gamma_i + D_{ij}$. The term $\mathbb{P}(R_j | \gamma)$ is the j th competition result, written in terms of γ_i . The term W_i is the number of times that participant i is a member of a winning team in $\mathbf{R} \cup \mathbf{R}'$. The term E_j is the sum of the strengths of the teams competing in R_j .

In practice, we can choose the order of which parameters to update by gradient descent. We can keep track of the most recent change in the log likelihood of the results $\mathbf{R} \cup \mathbf{R}'$ given the parameters γ resulting from each update of γ_i . We can then choose which γ_i to update next by choosing γ_i which when previously updated resulted in the biggest change in log likelihood. We can initialize all changes in log likelihood to be some large value.

In addition, while the algorithm suggests that we should re-process all competition results every time a strength parameter is updated, in practice we can simultaneously update a set of mutually exclusive strength parameters with one pass through the data [27]. By construction, the feature values of a single feature type are mutually exclusive, i.e. no two feature values of the same feature type can be set at the same time. Finally, the constant E_j for each result R_j can be precomputed and reused in every update.

Move Prediction

Let $\mathcal{T}(S, m)$ denote the team of feature values extracted from a move m played on board state S . To train a Bradley-Terry model on a training set of Go games \mathbf{G} , for each $G \in \mathbf{G}$ and for each $t \in 1, \dots, T(G)$ we extract a competition result R_t^G in which $\mathcal{T}(S_{t-1}^G, m_t^G)$ beats $\mathcal{T}(S_{t-1}^G, \ell_k^{G,t-1})$ for each $k \in \{1, \dots, |L_{t-1}^G|\}$. Then we iteratively choose the feature type with the largest previous change in log likelihood and update the strength parameters for all feature values in that feature type according to Equation 2.4. We stop updating the parameters once the largest change in log likelihood is less than 0.001. Also, as in [11], we set the prior $\mathbb{P}(\gamma)$ implicitly by adding one virtual win and one virtual loss for each feature value against a single virtual opponent with fixed strength parameter 1.0.

We define the *Bradley-Terry move predictor* to be Algorithm 1 with LEARNSCORING-FUNCTION(\mathbf{D}) defined as the method which estimates the Bradley-Terry strength parameters and returns the move predictor $\mathcal{P}_{\mathbf{G}}$, where for move m and board state S ,

$$\mathcal{P}_{\mathbf{G}}(S, m) = \prod_{i \in \mathcal{T}(S, m)} \gamma_i.$$

2.5 Testing Move Prediction Accuracy

2.5.1 Data

We use two different data sets:

- **PRO**: a collection of 24,524 professional games played between 1998 and 2009, obtained from the GoGoD (Games of Go on Disk) database [26].
- **KGS**: a collection of 25,993 amateur games played between 2007 and 2009, which were played on the K Go Server (KGS) [36] and have been archived in [25]. In all games in this collection, at least one player has amateur rank of at least 7 dan.

For each data set, we extract a separate database of patterns. The databases consist of 3×3 patterns that appear over 1000 times in each data set, and 5×5 , 7×7 , and 9×9 patterns that appear over 500 times in each data set. We call these two pattern databases **PROPATTERNS** and **KGSPATTERNS**. Note that we decreased the frequency threshold for the larger patterns in order to include more large patterns in the database.

For training, we randomly selected 500 games from both **PRO** and **KGS**. We call these two training sets **PROTRAIN** and **KGSTRAIN**. **PROTRAIN** contains 109,492 moves, while **KGSTRAIN** contains 103,416 moves. For testing, we randomly selected 100 games from both **PRO** and **KGS** (which were checked to make sure the games were not also in the training sets). We call these two test sets **PROTEST** and **KGSTEST**. **PROTEST** contains 21,777 moves, while **KGSTEST** contains 19,004 moves.

2.5.2 Results

We first extracted feature vectors from all training and test sets. Feature extraction takes 13.2 seconds per game on average. Approximately 70% of the time needed for feature extraction is spent calculating the Monte-Carlo owner heuristic. We then trained a Bradley-Terry move predictor and a naive Bayes move predictor on each of the two training sets. The Bradley-Terry move predictor takes approximately 78 minutes to train on 500 games, while the naive Bayes move predictor takes approximately 5 minutes to train on 500 games.

For each predictor and each test set, we tested the percentage of correct moves in the test set within the top n predictions of the predictor (ranked by score), for $n \in \{1, \dots, 100\}$. Results are shown in Figures 2.3 and 2.4, where NB stands for naive Bayes and BT stands for Bradley-Terry. So “NB PRO” denotes a naive Bayes predictor that is trained on **PROPATTERNS** and **PROTRAIN**, and is tested on **PROTEST**. In Figure 2.6, we demonstrate our implementation of the Bradley-Terry move predictor with GoGui, an open source graphical user interface for Go programs.⁴ Green squares on intersections indicate that moves are unlikely to be played on those intersections given the current position, whereas squares that are more red indicate moves that are more likely to be played. The magenta square represents

⁴<http://gogui.sourceforge.net/>

the top prediction by the predictor. Note that in the example in Figure 2.6, the predictor accurately follows common opening patterns.

n	NB PRO	NB KGS	BT PRO	BT KGS
1	26.24%	31.28%	30.09%	34.59%
2	38.39%	44.41%	42.20%	47.98%
3	45.44%	51.57%	49.79%	55.81%
4	50.43%	56.62%	54.60%	60.91%
5	53.96%	60.43%	58.45%	64.62%
10	64.62%	70.84%	69.33%	75.27%
20	75.88%	81.01%	80.29%	84.97%
30	82.40%	86.20%	86.04%	89.37%
40	86.43%	89.58%	89.60%	92.35%
50	89.30%	91.89%	92.18%	94.41%
100	96.77%	97.44%	97.75%	98.32%

Figure 2.3: Cumulative move prediction accuracy

The results show that our naive Bayes move predictor performs at least as well as the move predictor in [43]. Unlike the move predictor in [43], our naive Bayes move predictor requires no dimensionality reduction techniques and no training of neural networks, and hence our method is both simpler and faster. In addition, unlike neural network models, naive Bayes models do not suffer from the “curse of dimensionality,” and so our model is robust to adding new features types. Furthermore, the number of parameters in the naive Bayes model is linear in the number of feature dimensions, so the time required to train the naive Bayes model scales well with adding new feature types. Finally, neural networks often get caught in local optima which yield suboptimal results, whereas naive Bayes models are completely deterministic and always yield the correct model parameters.

We have successfully reproduced the results of [11] with our implementation of the Bradley-Terry move predictor, which uses two additional features (the feature for number of stones captured and the feature for jump/knight’s moves) but significantly fewer patterns (about 16,700 patterns in [11] versus about 1,400 patterns in our database). We have two possible explanations for why our implementation of the Bradley-Terry move predictor performs as well as the Bradley-Terry predictor in [11]. First, it is possible that the additional patterns in the database of [11] are not matched often enough, so the corresponding strength parameters are too close to 1.0 to significantly change predictions. Second, it is possible that the two additional features that we include capture strategic elements present in the patterns that are not included in our database.

The Bradley-Terry move predictor performs strictly better than our naive Bayes move predictor, as shown in Figures 2.3 and 2.4. The added predictive power of the Bradley-Terry move predictor confirms the advantage of using Bradley-Terry models over naive Bayes models. In a Bradley-Terry model, the strength of a feature is updated not only according to

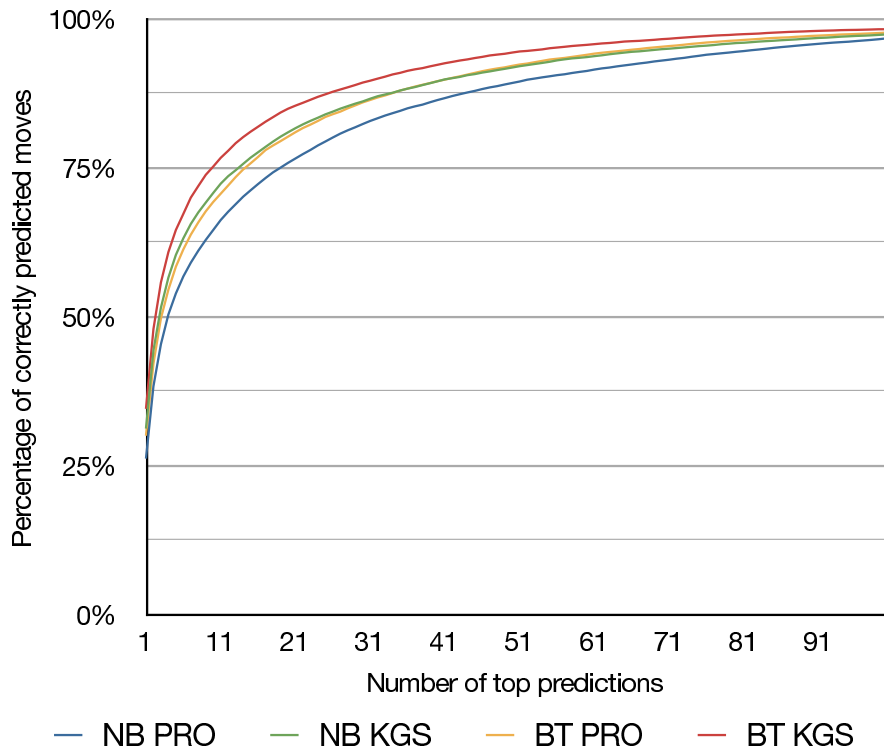


Figure 2.4: Cumulative move prediction accuracy

whether that feature is played in the actual move, but also according to what other features are present in the same move. For example, if a played move matches a rare pattern but also is a capture, then assuming the capture feature has a high strength parameter, the strength of the rare pattern will not be increased greatly since most of the “win” of that move will be attributed to the strength of the capture feature. Naive Bayes models do not capture these kinds of interactions between features.

Given that the Bradley-Terry move predictor is never more than 5% more accurate than our naive Bayes move predictor, and given that our move predictor takes an order of magnitude less time to train, we conclude that our naive Bayes move predictor is a viable alternative to the Bradley-Terry move predictor. Our naive Bayes move predictor is also simpler and easier to implement. From a practical standpoint, our naive Bayes move predictor has a significant advantage over the Bradley-Terry move predictor: it can be implemented as an online learning method. The Bradley-Terry move predictor requires all of the feature vectors to be loaded into memory before training, since the update function for strength parameters requires computation for all competition results. For our naive Bayes method, when processing a single move, once the frequency counts for features and classes have been updated, that move can be discarded. Hence, unlike the Bradley-Terry system, our naive Bayes system is not restricted by memory constraints. Also, during actual play, we can process new feature vectors and immediately incorporate the new instances into the model in an online fashion,

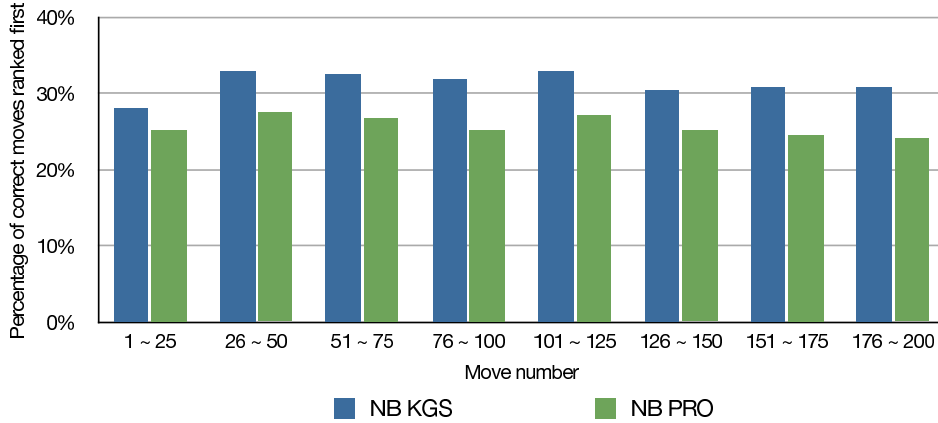


Figure 2.5: Move prediction accuracy by move number

so our model can quickly adapt to current play. Our model does not require complete re-training each time a new instance is added to the data set. This suggests that our naive Bayes model may be a more practical choice for incorporating move prediction into an actual computer Go player.

An additional notable finding of our experiments that is not mentioned in previous work is that there is a significant difference between the move predictors’ accuracy on professional games and amateur games. This suggests that, for move predictors which train on basic tactical features and pattern matches, amateur games are more predictable than professional games. Professional Go players are known to be creative and innovative in their gameplay, often discovering new *joseki* (sequences that yield equal outcomes for both players) and *tesuji* (the term for clever or skillful maneuvers that yield best play in a local position). This also suggests that the move predictor in [38] may perform better on professional games than [11], since the former was tested on professional games while the latter was only tested on amateur games.

Finally, we tested our naive Bayes move predictor’s accuracy at different stages of the game. Figure 2.5 shows the prediction accuracy for moves 1 through 25, 25 through 50, etc. On both the KGS and PRO datasets, the move prediction accuracy does not change significantly as the game progresses. This result is similar to the results of the Bradley-Terry move predictor in [11]. For comparison, we note that for the move predictor in [38], move prediction accuracy generally decreases as the game progresses. This suggests that our naive Bayes move predictor and the Bradley-Terry move predictor can be used more effectively for branching factor reduction and as a move ordering heuristic, since the move prediction accuracy will be approximately the same at all depths of the game tree.

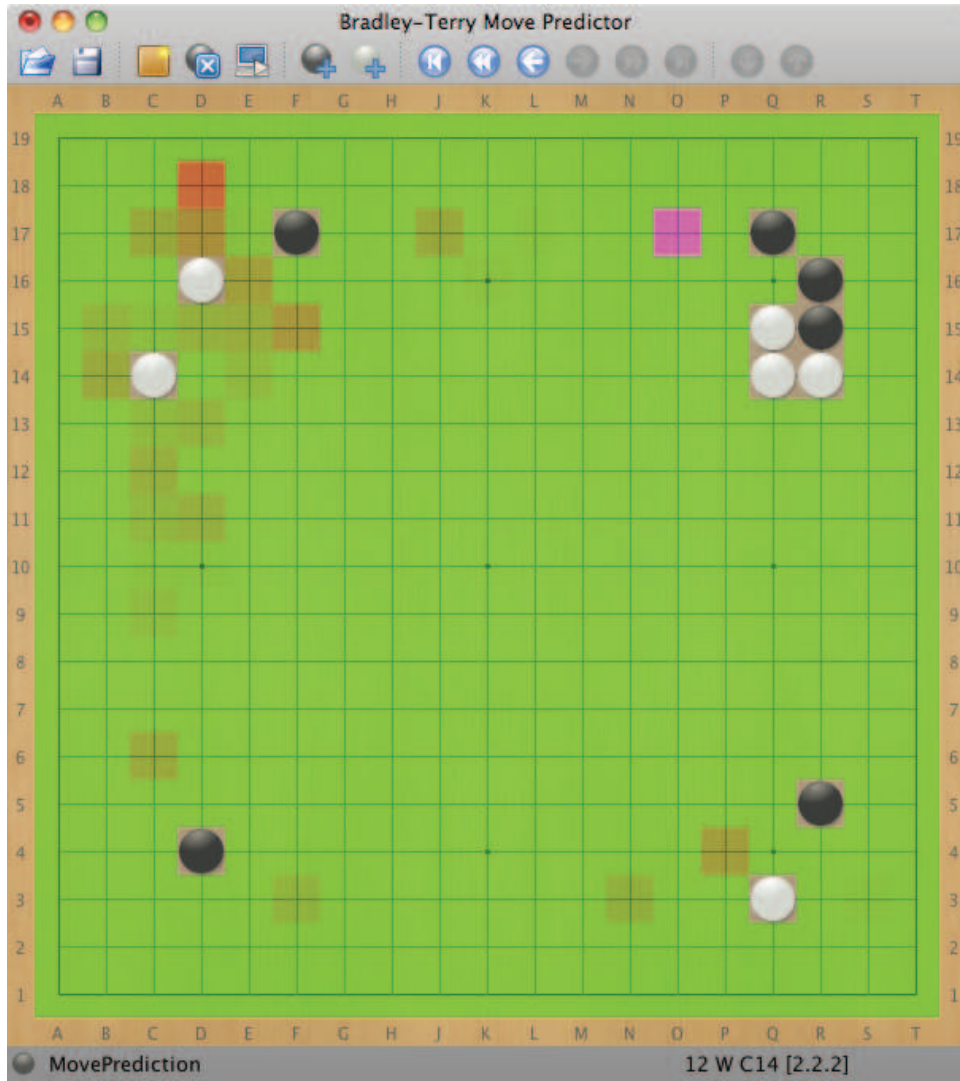


Figure 2.6: Demonstration of Bradley-Terry move predictor with GoGui

2.6 Future Research

We offer two main directions for future research. The first direction aims at finding better learning methods for scoring and ranking feature vectors. For example, methods such as decision tree learning which do not assume independence of the feature vectors may yield better move prediction accuracy. The second direction aims at extracting new feature types to provide more information about each moves. There are many moves that are tactically very different but have almost identical feature vectors given the current set of feature types. There may be ways to automatically extract relevant features from the board. On the other hand, discussion with professional Go players and analysis of how they predict moves may yield insight into how to more accurately extract the important features of moves.

Chapter 3

Implementing a Framework for Move Prediction

“In theory, there is no difference between theory and practice. But, in practice, there is.”

Lawrence “Yogi” Berra

There are numerous implementation choices that need to be made in constructing a move predictor, or any other framework for Go-related programs. Unfortunately, detailed specifications of how to build such a framework are difficult to find, as they are generally absent from academic literature. In this chapter, we provide methods for constructing a Go framework that can be used to train and test move prediction algorithms, but that is generally applicable to most Go-related problems. In Section 3.1, we describe how to represent the Go board and implement various board features, such as detecting legal moves, storing chains, and counting liberties. In Section 3.2, we describe how to implement fast Monte-Carlo playouts, which we use to compute the Monte-Carlo owner feature in our move predictor. In Section 3.3, we describe how to construct a database of frequently played patterns from a set of games. We use C++ as the sample language, although the methods are extensible to any common imperative language. In Section 3.4, we offer concluding remarks.

3.1 The Go Board

When designing a Go board implementation, there is a tradeoff between speed of play (how fast the board can process a move) and speed of analysis (how fast the board can answer queries about chains, liberty counts, and other features such as those described in Section 2.3.1). Since we need to perform feature extraction for a large number of moves, we choose to implement a board that enables fast analysis.

3.1.1 Go board data structure

A standard data structure for an $s \times s$ board is an $s \times s$ array of an `enum` type which has three possible values: `Black`, `White`, and `Empty`. However, in practice, it is faster to implement the board as a one-dimensional array with a one-dimensional coordinate system. For speed of edge detection, it standard to represent the board with a one-dimensional array of size $(s+2) \cdot (s+1) + 1$, where each value in the array is an `enum` with four possible values: `Black`, `White`, and `Empty`, and `Wall`.

To illustrate the structure of this board, Figure 3.1 shows the layout of an empty 7×7 board representation, shown in two dimensions for convenience, where “.” represents an empty intersection and “#” represents the wall of the board. Adding the wall coordinates in this way ensures that the neighbors of any non-wall point in each of the possible eight directions (north, northeast, etc.) have values in the representation. With this representation, there is no need for conversion of the one-dimensional points to two-dimensional coordinates in order to detect if a neighboring point is off the board.

# # # # # # # # # #	0	1	2	3	4	5	6	7
#	8	9	10	11	12	13	14	15
#	16	17	18	19	20	21	22	23
#	24	25	26	27	28	29	30	31
#	32	33	34	35	36	37	38	39
#	40	41	42	43	44	45	46	47
#	48	49	50	51	52	53	54	55
#	56	57	58	59	60	61	62	63
# # # # # # # # # #	64	65	66	67	68	69	70	71
#	72							

Figure 3.1: Representation of a 7×7 board. *Left:* The empty board is shown, where “.” represents an empty intersection and “#” represents the wall. *Right:* The one-dimensional index of each intersection is shown.

We can calculate the neighbors of a point using four functions N, S, E, W which take a point p as input and return the neighboring point to the north, south, east, and west, respectively. These are computed as follows:

- $N(p) = p - (s + 1)$,
- $S(p) = p + (s + 1)$,
- $E(p) = p + 1$,
- $W(p) = p - 1$.

We also store several other useful pieces of data in the board structure:

- **chain_reps**: an integer array of size $(s + 2) \cdot (s + 1) + 1$ which maps all occupied points p to a representative point q of the chain containing p , and which maps every empty and wall point to 0. In other words, if points p_1 , p_2 , and p_3 all belong to the same chain, then the representatives for each point will be the same q , where $q \in \{p_1, p_2, p_3\}$.
- **chains**: an array of chain data structures which maps all chain representatives to its corresponding chain data structure.
- **black_prisoners**, **white_prisoners**: integers that keep track of the number of stones captured from each player.
- **ko_point**: an integer which stores the current ko point if it exists, and 0 otherwise.
- **move_history_list**: a list of move history data structures which store information about all past moves played. Each data structure stores the player, the point played, the ko point before the move was played, and the directions of capture if the move resulted in any captures.
- **depth**: the number of moves that have been played on the board.

We show the basic header for this data structure in Code 3.1.

3.1.2 Chains

Chains are connected groups of adjacent stones which are owned by the same player. Chains share liberties, and many features (such as capture and atari) rely on counting chain liberties. To facilitate chain analysis, we represent chains as a data structure and incrementally update all chains on the board with each move.

We represent chains internally with six data members:

- **points**: an array of integers storing all points in the chain
- **num_points**: the current number of points in the chain
- **points_indices**: an array of size $(s + 2) \cdot (s + 1) + 1$ which maps each point p in the chain to i such that `points[i] = p`, and maps all other points to -1 .
- **liberties**, **num_liberties**, **liberties_indices**: analogous data members for keeping track of liberties.

We show the basic header for this data structure in Code 3.2

With this representation in place, we can perform the following chain operations in constant time:

- **addPoint**: adds the specified point to the chain, if the point is not already in the chain.

Code 3.1 Go board data structure

```
struct Board
{
    // State of a board intersection
    enum State { BLACK, WHITE, EMPTY, WALL };

    // Size parameters for a 19x19 board
    const int SIZE = 19;
    const int BOARD_SIZE = (SIZE+2)*(SIZE+1)+1;

    // Max number of previous moves to store
    const int MAX_HISTORY = 600;

    // Arrays for storing states, chains, and chain representatives
    State states[BOARD_SIZE];
    Chain chains[BOARD_SIZE];
    int chain_reps[BOARD_SIZE];

    // Current ko point if exists, -1 otherwise
    int ko_point;

    // Number of stones captured from each player
    int black_prisoners;
    int white_prisoners;

    // Move history list
    MoveHistory move_history_list[MAX_HISTORY];
    int depth;

    // Function declarations

    ...
}
```

Code 3.2 Chain data structure

```
struct Chain
{
    static const int SIZE = 19;
    static const int BOARD_SIZE = (19+2)*(19+1)+1;

    // SIZE*SIZE is very loose upper bound on the number of
    // points and liberties that a chain can have
    static const int MAX_POINTS = SIZE*SIZE;
    static const int MAX_LIBERTIES = SIZE*SIZE;

    // Data members for keeping track of points
    int points[MAX_POINTS];
    int num_points;
    int points_indices[BOARD_SIZE];

    // Data members for keeping track of liberties
    int liberties[MAX_LIBERTIES];
    int num_liberties;
    int liberties_indices[BOARD_SIZE];

    // Function declarations

    ...
};
```

- `removePoint`: removes the specified point from the chain, if the point is actually in the chain.
- `hasPoint`: checks if the specified point is in the chain.
- `addLiberty`, `removeLiberty`, `hasLiberty`: analogous functions for liberties.

To add a point to the chain, we simply add the point to the end of the `points`, set the appropriate index for the newly added point in `point_indices`, and increment `num_points`. To remove a point from the chain, we swap the point and the last chain point in both `points` and `point_indices`. Then we remove the point and decrement `num_points`. Checking whether a point is in the chain is equivalent to checking that its index is not -1 . We show the `addPoint`, `removePoint`, and `hasPoint` functions in Code 3.3. The functions for liberties are analogous.

The incremental updating of chains is explained below.

3.1.3 Legal moves

With most Go rulesets, a move in Go is legal if the following three conditions hold:¹

1. The move is played in an empty intersection that is within the bounds of the board.
2. The move is not played at a ko point.
3. The move is not suicide, i.e. the move does not cause the immediate capture of itself.

The first two conditions are simple, and require only checking a board state and equality with the ko point. The third condition is slightly more involved. A move is *not* suicide if at least one of the following three conditions hold:

1. The move is adjacent to an empty intersection.
2. The move is adjacent to a chain owned by the same player, which has at least two liberties (since the placement of the move will remove one of those liberties).
3. The move is adjacent to a chain owned by the other player, which has exactly one liberty (since the placement of the move will capture that enemy chain).

With the chains on the board up-to-date, these checks are also easy to compute.

¹For simplicity, we ignore superko, which occurs extremely rarely in games.

Code 3.3 Chain functions

// Add point to chain. Assumes that num_points < MAX_POINTS

```
void Chain::addPoint(int point)
{
    // if point is in chain, do nothing
    if (points_indices[point] != -1)
        return;

    points[num_points] = point;
    points_indices[point] = num_points;
    num_points++;
}
```

// Remove point from chain

```
void Chain::removePoint(int point)
{
    // if point is not in chain, do nothing
    if (points_indices[point] == -1)
        return;

    // swap last point with current point
    int index = points_indices[point];
    int end_point = points[num_points-1];
    points[index] = end_point;
    points_indices[end_point] = index;

    // remove point
    points[num_points-1] = 0;
    points_indices[point] = -1;
    num_points--;
}
```

// Check if chain has point

```
bool Chain::hasPoint(int point)
{
    return (points_indices[point] != -1);
}
```

3.1.4 Processing moves

Processing a legal move must accomplish several different tasks, including joining adjacent chains, removing captured chains, updating liberty counts, and updating the ko point.

When a move is played, we first create a new chain which contains only the move point. Then we can process the move by inspecting the neighbors of the move, and taking the appropriate action given the state of each neighbor. If a neighbor state is empty, then we add that neighbor as a liberty in the new chain. If a neighbor point has a stone owned by the same player, then we join the chain containing that stone to the new chain. We can join two chains by adding all points and liberties of the second chain to the first chain. After the chains are joined, we must iterate through the points of the chain and update the liberties and chain representatives appropriately. Finally, if a neighbor point has a point owned by the enemy, then we count the number of liberties in the chain containing the neighbor point. If the number of liberties is greater than 1, then the move simply causes the removal of the move point from that chain's liberties. Otherwise, if the number of liberties is exactly 1, then a capture results, and we remove the neighbor chain from the board. Note that when a chain is removed from the board, the neighboring chains will gain liberties, so we have to update the liberties of those neighboring chains. If the number of stones captured is exactly 1, then that captured point becomes the ko point. Otherwise, we clear the ko point. We complete the function by incrementing the counter for the number of moves played.

We present the pseudocode for processing moves in Algorithm 3.

3.1.5 Move history and undo

When extracting features from moves, it is often necessary to modify the board state. For example, to determine if a move is a capture that prevents the player's own capture, we process the move and determine if a capture occurred, and if so then we determine whether any chain adjacent to the captured chain was in atari before the move was processed. But we often compute features for moves that are never actually played, so we need to undo all modifications after feature extraction is complete.

The easy way of implementing undo is to copy the entire board state before each feature extraction, and have the feature extraction operate on the board copy. However, copying the entire board state can be expensive, both in terms of time and memory usage. By maintaining a history of moves played, and by making small modifications to the `PROCESSMOVE` function (Algorithm 3), we can implement incremental undo functionality.

We define a move history data structure which stores four pieces of information:

- **player**: the player who moved.
- **point**: the point played in the move.
- **ko_point**: the ko point before the move was played.

Algorithm 3 PROCESSMOVE($p, player$)

Initialize chain c with p
 $captured \leftarrow []$
for all neighbors n of p **do**
 if STATE(n) = EMPTY **then**
 ADDLIBERTY(c, n)
 else if STATE(n) = $player$ **then**
 $c \leftarrow JOINCHAINS(c, GETCHAIN(n))$
 UPDATELIBERTIESANDCHAINREPS(c)
 else if STATE(n) = OPPOSITE($player$) **then**
 $nc \leftarrow GETCHAIN(n)$
 if NUMLIBERTIES(nc) = 1 **then**
 REMOVEFROMBOARD(nc)
 UPDATEPRISONERS($nc, player$)
 PUSH(POINTS(nc), $captured$)
 for all chains nnc neighboring nc **do**
 UPDATELIBERTIES(nnc)
 end for
 else {NUMLIBERTIES(nc) > 1}
 REMOVELIBERTY(nc, p)
 end if
 else {STATE(n) = WALL}
 {do nothing}
 end if
end for
if $|captured| = 1$ **then**
 $ko_point \leftarrow captured[0]$
else
 $ko_point \leftarrow 0$
end if
 $depth \leftarrow depth + 1$

- `capture_directions`: a size 4 array of true/false values that maps each of the four cardinal directions to true if the move captured stones in that direction, and false otherwise.

We show the move history structure definition in Code 3.4.

Code 3.4 Move history data structure

```

struct MoveHistory
{
    // Enum representing four cardinal directions
    enum Direction { NORTH, EAST,
                    SOUTH, WEST };

    // Move information
    Player player;
    int point;

    // Ko point before move was played
    int ko_point;

    // capture_directions[d] = true if and only if
    // a capture occurred in the direction d from point
    bool capture_directions[4];

    // Function declarations ...
};

```

Now the `PROCESSMOVE` function requires only a few extra steps to maintain the move history list. First, when a move is processed, we create a new move history structure which contains the current point, the current player, and the current ko point. Then, when inspecting the neighbors in each direction, if that neighbor is to be captured, then we add the corresponding direction to the move history structure by setting the appropriate value in the `capture_directions` array to true.

We are ready to describe the `UNDO` function. When the function is called, we first pop the latest move history from the move history list. We set the state of the last-played point to `EMPTY`, and temporarily set the ko point to 0. Next, we inspect each neighbor, as we did in the `PROCESSMOVE` function. If that neighbor is a part of an enemy chain, we add the last point back to the liberties of that chain. If that neighbor is owned by the player who moved last, then we recursively search the neighbors of that point in order to reconstruct the chain containing that neighbor. This is necessary because if the last move connected two chains,

then calling `undo` must split those two chains. Finally, for every capture direction in the move history, we flood fill enemy stones in that direction, i.e. we fill up all contiguous empty intersections in the direction with enemy stones. In order to keep the chain information updated correctly, we can call our `ProcessMove` function to add the enemy stones in the previous capture locations. We complete the `UNDO` function by restoring the ko point and decrementing the counter for number of moves played.

3.2 Monte-Carlo Playouts

In this section, we pay special attention to computing the Monte-Carlo owner feature.

3.2.1 Simplified Go board

In order to compute a large number of random playouts in a feasible amount of time, we now need move processing to be as fast as possible. For this purpose, we implement a new board data structure that is used only for Monte-Carlo playouts.

The main idea is that, as mentioned above, incrementally creating and updating chains on the board enables fast analysis at the cost of slowing down playouts. This decrease in playout speed is not so noticeable in feature detection, but since the speed of these playouts is the limiting factor for computing the Monte-Carlo owner feature, we want a different board implementation that minimizes the time required for each `PROCESSMOVE` step.

To accomplish this goal, instead of keeping an array of chains and chain representatives, we keep an array of chain neighbors, called `next_point_in_chain`, which has size $(s+2) \cdot (s+1) + 1$. For a point p , if the board state is `EMPTY` at p , then `next_point_in_chain[p] = 0`, but if the board state is either `BLACK` or `WHITE`, then `next_point_in_chain[p] = q` where q is the next point in the chain containing p (and if p is the only point in its chain, then $q = p$). The `next_point_in_chain` array will be maintained such that the sequence

$$p, \text{next_point_in_chain}[p], \text{next_point_in_chain}[\text{next_point_in_chain}[p]], \dots$$

iterates through all points in the chain containing p before returning back to p . In other words, `next_point_in_chain` forms implicit circular linked lists of stones that belong to the same chain. We can loop through the stones in the chain with a loop such as the one in Algorithm 4, which gives the pseudocode for counting the number of liberties in a chain, and which is explained in the next paragraph.

To count the liberties of a chain without counting any liberty more than once, we maintain another array of size $(s+2) \cdot (s+1) + 1$ called `liberty_marks`, in which each point is mapped to an integer “mark”. Each point is initially mapped to 0, and the current mark `current_liberty_mark` is also initially set to 0. Every time we count the number of liberties of a chain (Algorithm 4), we increase `current_liberty_mark` by 1, and every time we find a liberty that is unmarked, we add the liberty to the liberty count and mark it by setting the liberty’s value in `liberty_marks` to `current_liberty_mark`. In addition, since processing

a move only requires knowledge of whether a chain has 0, 1, or ≥ 2 liberties, if we reach ≥ 2 liberties in the liberty count, we return the liberty count immediately. On average, this makes the iterative liberty counting procedure much faster.

Algorithm 4 NUMLIBERTIES(p), where p is a non-empty, non-wall point on the board

```

 $\ell \leftarrow 0$ 
current_liberty_mark = current_liberty_mark + 1
 $q \leftarrow p$ 
repeat
  for all neighbors  $n$  of  $q$  do
    if STATE( $n$ ) = EMPTY and liberty_marks  $\neq$  current_liberty_mark then
       $\ell \leftarrow \ell + 1$ 
      liberty_marks[ $n$ ]  $\leftarrow$  current_liberty_mark
    end if
  end for
  if  $\ell \geq 2$  then
    return  $\ell$ 
  end if
   $q \leftarrow$  next_point_in_chain[ $q$ ]
until  $q = p$ 
return  $\ell$ 

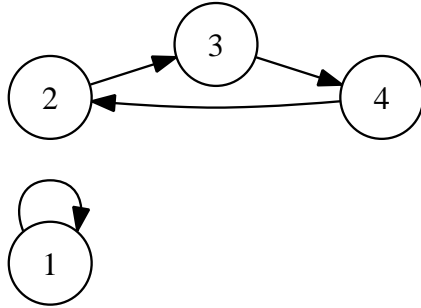
```

In the original definition of PROCESSMOVE, we required a JOINCHAINS operation which added all points and liberties of one chain to another. Since this implementation represents chains as implicit linked lists, and since we only count liberties as needed, we instead want JOINCHAINS to adjust the value of next_point_in_chain for the points in each chain such that all the points now form a circular linked list representing the entire joined chain.

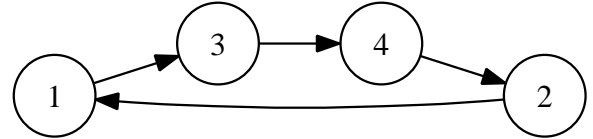
We can join chains using a few simple operations. First, when a move at point p is first processed, we set next_point_in_chain[p] $\leftarrow p$. For the first neighbor point n that is owned by the same player, we join p with the chain containing n by setting next_point_in_chain[p] \leftarrow next_point_in_chain[n] and next_point_in_chain[n] $\leftarrow p$. Figure 3.2 illustrates the joining of the point 1 with the chain containing adjacent point 2.

If the move joins two or more chains, then for each adjacent chain after the first, we set next_point_in_chain[n'] \leftarrow next_point_in_chain[n] and next_point_in_chain[n] $\leftarrow p$, where n' is the most recently joined neighbor. Figure 3.3 illustrates the joining of the chain containing point 1 in the right side of Figure 3.2 with another adjacent chain containing point 5. In Figure 3.3, $p = 1$, $n' = 2$, and $n = 5$.

There is one more step necessary to ensure correct joining of adjacent chains: we need to make sure that the same chain is not joined twice. This can be implemented with an array chain_marks and a corresponding integer mark current_chain_mark. Every time we process a move, we increment current_chain_mark, and for every adjacent point to be joined, if that point is not already marked, then we mark every point in the adjacent chain and join the chain with the methods above.

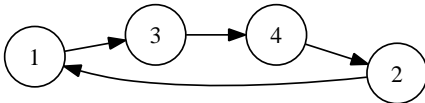
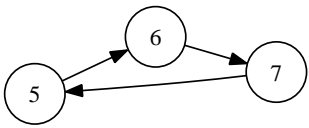


Point 1 adjacent to chain containing 2, 3, 4

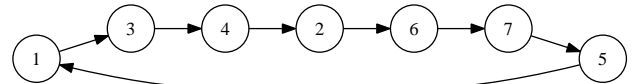


Point 1 joined with chain containing 2, 3, 4

Figure 3.2: Before and after joining point 1 with the chain containing adjacent point 2



Chain containing point 1 adjacent to chain containing 5, 6, 7



Chain containing point 1 joined with chain containing 5, 6, 7

Figure 3.3: Before and after joining the chain containing point 1 with the chain containing adjacent point 5, after point 1 has already been joined with the chain contained point 2

The `PROCESSMOVE` and `CHECKLEGAL` functions for the new board implementation are analogous to the versions defined above, with the modifications mentioned above. However, we do not provide undo functionality. Each Monte-Carlo playout (on an empty board) requires between 400 and 600 moves to reach the end of the game. Copying the board once and playing out each move on the copy is significantly faster than calling `undo` for each move played in the simulation.

3.2.2 Selecting random moves

The new functionality that we need to add to our Monte-Carlo board is the ability to quickly select a move uniformly at random from available legal moves (except moves that fill simple eyes). For this purpose, we keep a list of empty points on the board. This list is implemented as a similar data structure to the point and liberty lists in the definition of the chain data structure (Code 3.2 and Code 3.3), so that the empty point list can perform addition and removal of points in constant time.

Whenever we want to select a random move, we first copy the empty point list, and then until we find a valid move (i.e. the move is both legal and does not fill a simple eye), we select a random point from the empty point list copy, and check if it is a valid move for the

current player. If the move is valid, we return the move. Otherwise, we remove the point from the empty point list copy and repeat. If the list becomes empty but we still have not found a valid move, then there are no valid moves left for that player.

3.2.3 Computing the Monte-Carlo owner feature for legal moves

In order to speed up the computation of the Monte-Carlo owner feature, given a current board state S and a set of legal moves L for which we want to compute the feature, we do not run $63 \cdot |L|$ separate playouts, i.e. 63 different playouts for each move in L . Instead, we run only 63 playouts in total, and at the end of each playout determine the owner at each point in L . Given that there are an average of 250 legal moves at any point in the game, this change speeds up computation by a factor of about 250. The increase in speed is worth the bias that this change introduces into the simulation.

3.2.4 Performance

Our system computes 1,500 complete Monte-Carlo playouts per second on an empty 19×19 board. This is slow compared to Go programs which were built specifically for quickly simulating Monte-Carlo playouts, such as MoGo [45], but such programs have more complicated board implementations, are heavily optimized, and are often multi-threaded. Our Monte-Carlo simulator is easy to implement and fast enough to compute the Monte-Carlo owner heuristic in a feasible amount of time.

3.3 Pattern Database

In this section, we describe how to build a database of frequently played patterns in a set of Go games.

3.3.1 Pattern representation

As with the Go board, we could represent $n \times n$ patterns as $n \times n$ arrays of board states. However, array comparison is slow, and in general operations on entire arrays require iterating through all elements of the array.

We choose to implement patterns as pairs of 128-bit integers,² which we call `BLACK_PATTERN` and `WHITE_PATTERN`. Our pattern representation supports square pattern sizes up to 11×11 . We map a two-dimensional coordinate system onto each $n \times n$ pattern, where $(0, 0)$ is the upper left corner of the pattern and $(n - 1, n - 1)$ is the lower right corner of the pattern. Given a pattern coordinate (i, j) , let $q = i \cdot n + j$. The following rules determine the state of the pattern at (i, j) :

²We implemented pattern extraction on 64-bit architecture, so we implemented a 128-bit integer type which stores two 64-bit integers internally. On a 32-bit architecture, one can implement a 128-bit integer type which stores four 32-bit integers internally.

- If the q th bits of `BLACK_PATTERN` and `WHITE_PATTERN` are both unset, then the state of the pattern at (i, j) is `EMPTY`.
- If the q th bit of `BLACK_PATTERN` is set, and the q th bit of `WHITE_PATTERN` is unset, then the state of the pattern at (i, j) is `BLACK`.
- If the q th bit of `BLACK_PATTERN` is unset, and the q th bit of `WHITE_PATTERN` is set, then the state of the pattern at (i, j) is `WHITE`.
- If the q th bits of `BLACK_PATTERN` and `WHITE_PATTERN` are both set, then the state of the pattern at (i, j) is `WALL`.

We also store a `BOUNDING_BOX_SIZE` with each pattern, which dictates the size of the pattern. With the `BOUNDING_BOX_SIZE`, we can distinguish between two empty patterns of different sizes. Now we can determine the equality of two patterns with by testing if both `BLACK_PATTERNS` are equal, and if both `WHITE_PATTERNS` are equal, and if both `BOUNDING_BOX_SIZES` are equal. This is significantly faster than testing the equality of two $n \times n$ arrays, which requires n^2 comparisons.

Note that we can set the q th bit of a 128-bit integer x by setting $x \leftarrow x \mid (1 \ll q)$, where \ll is the left-bitshift operation.

3.3.2 Pattern symmetries

When extracting patterns, we need to adjust for symmetries. I.e., we want to match two patterns if they are the same modulo any rotation, reflection, or interchanging of black and white. Let I denote the identity, let R_x denote rotation by x degrees, let H denote horizontal reflection, and let F denote flipping the colors on the board. Then there are sixteen possible pattern symmetries: $I, R_{90}, R_{180}, R_{270}, H, H \circ R_{90}, H \circ R_{180}, H \circ R_{270}$, and each of the proceeding eight composed with F . During the extraction process, we can eliminate the need to check for symmetries composed with F by setting the states of the pattern such that the move is always played by Black. In other words, if a move is played by Black, then the pattern around the point played is exactly as appears on the board, and if a move is played by white, then the pattern around the point has opposite color stones as the pattern around the point as it appears on the board.

To test the symmetry of two patterns, we test if the two patterns' respective *canonical patterns* are equal. We define the canonical pattern C_ρ of a pattern ρ as follows:

$$C_\rho = \max \{ I \circ \rho, R_{90} \circ \rho, R_{180} \circ \rho, R_{270} \circ \rho, H \circ \rho, H \circ R_{90} \circ \rho, H \circ R_{180} \circ \rho, H \circ R_{270} \circ \rho \},$$

where \max is defined as the maximum over the 256-bit integer values attained by computing $(\text{BLACK_PATTERN} \ll 128) \mid \text{WHITE_PATTERN}$.

3.3.3 Counting pattern frequencies

We want to construct a database of patterns that appear at least κ times in our dataset for some threshold κ . If we store every unique pattern in the data set (unique up to symmetry) along with a frequency counter, then we will quickly run out of memory on standard hardware. This is because, for example, there can be as many as 49^4 different configurations of a 7×7 section of the board. Although the true upper bound is much smaller due to symmetries and impossible board configurations, the number of possible board configurations is still far too great to store every pattern that is found.

To successfully count pattern frequencies in a way that is both time and space efficient, we use a *count-min filter*, also called a *count-min sketch* [9]. The main idea is that we can use a constant-size hash table T and a hash function h which maps a pattern to a position in the hash table. Each position in the table stores the number of times a pattern was observed that hashed to that location. So every time a pattern ρ is observed, we can increment $T[h(\rho)]$.

A potential problem is that there may be *collisions*, i.e. two different patterns may hash to the same location in the table. So to reduce the probability of collisions, a count-min filter uses multiple tables T_1, \dots, T_k with corresponding hash functions h_1, \dots, h_k . Every time a pattern ρ is observed, we increment $T_i[h_i(\rho)]$ for $i \in \{1, \dots, k\}$. To find the number of times a pattern ρ has been seen so far, we simply find $\min_{1 \leq i \leq k} T_i[h_i(\rho)]$. With only a few hashing operations and lookups required, this method is extremely fast, and uses a constant amount of memory. To decrease the chances of collision, one can increase the number of tables and hash functions used. In practice, we set $k = 5$ and use tables of size 2^{18} .

We define a function for hashing a pattern to a 32-bit unsigned integer in Code 3.5, where `RANDOM_SEED` is a random non-negative integer less than the table size, which is 2^{18} in our case. There are k different `RANDOM_SEEDS` that are precomputed ahead of time, one for each table. The `RANDOM_SEEDS` ensure that each hash function is pairwise independent.

3.3.4 Pattern extraction

Finally, we briefly describe the pattern extraction process. Given a set of Go games $\mathbf{G} = \{G_1, \dots, G_n\}$, for every game G_i we play all the moves in $\mathbf{m}(G_i)$. Every time we play a move, we find the canonical pattern centered at the point where the move was played. In practice, we extract patterns of size 3×3 , 5×5 , 7×7 , and 9×9 . We increment the count of each pattern in a count-min filter as described above. When the count of a pattern first reaches the given threshold κ , we append the 128-bit integers `BLACK_PATTERN`, `WHITE_PATTERN` and the integer `BOUNDING_BOX_SIZE` to a database file. When the extraction process is complete, we have a database file which we can read in later for purposes of pattern matching in feature extraction.

3.3.5 Performance

Our system extracts all 3×3 , 5×5 , 7×7 , and 9×9 patterns from a dataset of 25,993 games containing 5,132,089 moves in 12 minutes and 43 seconds.

Code 3.5 Pattern hash function

```
unsigned int hash(Pattern pattern, unsigned int random_seed)
{
    unsigned int prime = 13;
    unsigned int result = 1;
    result = prime * result + (unsigned int)(pattern.black_pattern);
    result = prime * result + (unsigned int)(pattern.black_pattern >> 32);
    result = prime * result + (unsigned int)(pattern.black_pattern >> 64);
    result = prime * result + (unsigned int)(pattern.black_pattern >> 96);
    result = prime * result + (unsigned int)(pattern.white_pattern);
    result = prime * result + (unsigned int)(pattern.white_pattern >> 32);
    result = prime * result + (unsigned int)(pattern.white_pattern >> 64);
    result = prime * result + (unsigned int)(pattern.white_pattern >> 96);
    result = prime * result + (unsigned int)(pattern.bounding_box_size);

    // return the computed hash modulo the random seed
    return (result % random_seed);
}
```

3.4 Concluding Remarks

In this chapter, we have described in detail the process of implementing a framework for move prediction. The board implementations can also be used in other Go-related research areas, or can be used simply to play the game. It is our hope that with this contribution, future research in the field of computer Go can focus on generating and testing new ideas, instead of reproducing previous results.

Our implementation methods are by no means the only ways to implement a Go framework. We encourage future researchers to create more efficient data structures and algorithms for move playout, feature detection, and pattern extraction. In the spirit of this thesis, we encourage these researchers to report their efforts and publish their implementation decisions, so that this field can continue to be propelled forward.

Acknowledgments

I would like to thank my advisor Yiling Chen for her guidance and expert knowledge. I would like to express my deep gratitude to David Wu, who offered his ingenuity, insight, and advice on many aspects of this project, in addition to giving me lessons on how to play Go. I would also like to thank Haoqi Zhang, Samuel Galler, and Lee Seligman for reading drafts of this paper and providing invaluable comments and advice. Finally, I would like to thank my parents Rhonda and Aaron Harrison, my brother David Harrison, my girlfriend Lauren Kaye, and my friends and roommates for their constant love and support.

Bibliography

- [1] L. Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, The Netherlands, 1994.
- [2] American Go Association. Welcome to the american go association. <http://www.usgo.org>.
- [3] Richard Bozulich. *One Thousand and One Life-and-Death Problems*, volume 2 of *Mastering the Basics*. Kiseido, 2002.
- [4] Jay Burmeister and Janet Wiles. Accessing go and computer go resources on the internet. In *to appear in Proceedings of the Second Game Programming Workshop in Japan*, September 1995.
- [5] Michael Buro. Logistello's homepage. <http://www.cs.ualberta.ca/~mburo/log.html>.
- [6] Tristan Cazenave and Nicolas Jouandeau. On the parallelization of uct. *Proceedings of the Computer Games Workshop*, Jan 2007.
- [7] ChessBase. Chessbase.com - chess news - kramnik vs deep fritz: Computer wins match by 4:2. <http://www.chessbase.com/newsdetail.asp?newsid=3524>.
- [8] William S. Cobb. *The Book of Go*. Sterling, New York, 2002.
- [9] G Cormode and S Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
- [10] Rémi Coulom. Crazy stone. <http://remi.coulom.free.fr/CrazyStone/>.
- [11] Rémi Coulom. Computing elo ratings of move patterns in the game of go. *Computer Games Workshop*, Jan 2007.
- [12] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *Lecture Notes in Computer Science*, 4630:72, 2007.
- [13] Rémi Coulom. Monte-carlo tree search in crazy stone. *Proceedings of Game Programming Workshop 2007*, 2007.

- [14] Peter Drake and Steve Uurtamo. Move ordering vs heavy playouts: Where should heuristics be applied in monte carlo go. *Proceedings of the 3rd North American Game-On Conference*, Jan 2007.
- [15] David Silver et al. Reinforcement learning of local shape in the game of go. *20th International Joint Conference on Artificial Intelligence*, Jan 2007.
- [16] Haruhiro Yoshimoto et al. Monte carlo go has a way to go. *Proceedings of the National Conference on Artificial Intelligence*, Jan 2006.
- [17] Nicol Schraudolph et al. Learning to evaluate go positions via temporal difference methods. *Studies In Fuzziness And Soft Computing*, Jan 2001.
- [18] Paul Donnelly et al. Evolving go playing strategy in neural networks. *AISB Workshop in Evolutionary Computing*, Jan 1994.
- [19] International Go Federation. The international go federation.
<http://www.intergofed.org/>.
- [20] United States Go Federation. Computer beats pro at u.s. go congress.
<http://www.usgo.org/index.php?id=4602>.
- [21] David Fotland. David fotland's many faces of go.
<http://www.smart-games.com/manyfaces.html>.
- [22] David Fotland. Knowledge representation in the many faces of go.
<http://www.smart-games.com/knowpap.txt>.
- [23] Dao-Xiong Gong and Xiao-Gang Ruan. Using cellular automata as heuristic of computer go. In *Proceedings of the 4th World Congress on Intelligent Control and Automation*, volume 3, 2002.
- [24] M Guillaume, M Winands, and H Van den Herik. Parallel monte-carlo tree search. *Computers and Games: 6th International Conference*, Jan 2008.
- [25] Ulrich Görtz. Sgf game records. <http://www.u-go.net/gamerecords/>.
- [26] T Mark Hall. Gogod encyclopaedia and database. <http://www.gogod.co.uk/>.
- [27] DR Hunter. Mm algorithms for generalized bradley-terry models. *The Annals of Statistics*, 32(1):384–406, 2004.
- [28] Kaoru Iwamoto. *Go for Beginners*. Random House, Inc., New York, 1976.
- [29] Toshiro Kageyama. *Lesons in the Fundamentals of Go*. Kiseido, 1979.
- [30] Martin Müller. Computer go. *Artificial Intelligence*, 134(1):145–179, 2002.

- [31] Xiaozhen Niu. *Recognizing safe territories and stones in computer Go*. PhD thesis, University of Alberta, 2005.
- [32] GNU Project. Gnu go - gnu project - free software foundation.
<http://www.gnu.org/software/gnugo/>.
- [33] Mohammed Raonak-Uz-Zaman. *Applications of neural networks in Computer Go*. PhD thesis, Texas Tech University, 1998.
- [34] Michael Reiss. Go++ faq. <http://www.goplusplus.com/go4ppfaq.htm>.
- [35] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers Is Solved. *Science*, 317(5844):1518–1522, 2007.
- [36] Bill Shubert. Kgs go server. <http://www.gokgs.com>.
- [37] Arthur Smith. *The Game of Go: The National Game of Japan*. Moffat, Yard & Company, New York, 1908.
- [38] D Stern, R Herbrich, and T Graepel. Bayesian pattern ranking for move prediction in the game of go. In *In Proceedings of the 23rd international conference on Machine Learning*, Jan 2006.
- [39] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3), March 1995.
- [40] Jav van der Steen. Gobase.org - go games, go information and go study tools.
<http://www.gobase.org>.
- [41] Jav van der Steen. Gobase.org - history of go.
<http://www.gobase.org/reading/history/>.
- [42] Erik van der Werf. *AI techniques for the game of Go*. PhD thesis, Universiteit Maastricht, The Netherlands, 2004.
- [43] Erik van der Werf et al. Local move prediction in go. *Lecture Notes in Computer Science*, Jan 2003.
- [44] Rob von Zeijst and Richard Bozulich. *All About Ko*, volume 6 of *Mastering the Basics*. Kiseido, 2007.
- [45] Yizao Wang and Sylvain Gelly. Modifications of uct and sequence-like simulations for monte-carlo go. *IEEE Symposium on Computational Intelligence and Games, 2007. CIG 2007*, pages 175–182, 2007.

Appendix A

Rules of Go

The rules described in this section are the general rules of Go. Note that there are slight variations to these rules. The precise statement of the rules may vary depending on the ruleset.

A.1 Capture

Stones remain on the board as long as they have *liberties*. In [28], the author uses the following analogy: Consider the single white stone in Figure A.1. Compare this stone to a man at the intersection of streets in a city. The man can only move in one of four directions along intersecting streets, either up, down, left, or right. If this man is a fugitive running from the police, as long as he can advance in a single direction, he can escape. If all four directions are blocked, the man will inevitably be captured. Analogously in Go, empty intersections that are adjacent to a stone are called that stone's *liberties*. In Figure A.1, the white stone has only one liberty, at the intersection marked \times . If Black plays a stone at the intersection marked \times , the white stone is *captured* and removed from the board. Until the end of the game, Black keeps the newly captured white stone as its *prisoner*.

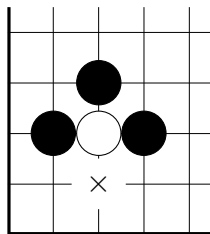


Figure A.1: White's single stone has one liberty.

Groups of adjacent stones are called *chains*. Chains share liberties, so that a player cannot capture a single stone in a chain without capturing the entire chain. In Figure A.2, White has one chain with two liberties, at the intersections marked \times . If Black plays at

either intersection marked \times , White's chain will have only one liberty remaining, and hence is in danger of being captured by Black's next move. A stone or chain that has only one liberty is said to be in *atari*.

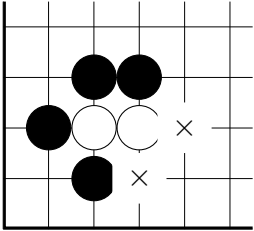


Figure A.2: White's chain has two liberties.

A.2 Suicide

In most rulesets of Go, *suicide* is illegal; in other words, a player cannot place a stone whose placement results in the immediate capture of that player's stones. In Figure A.3, White cannot play at the intersection marked \times , since that would result in the immediate capture of White's stone placed at \times . However, White *can* play at the intersection marked \square , since this play would first result in the capture of the black stone marked \blacktriangle , giving the recently placed white stone one liberty.

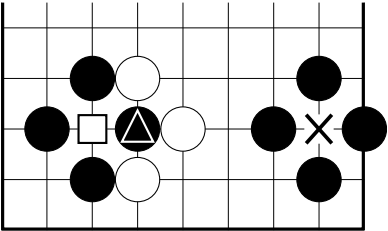



Figure A.3: Placing a white stone at \times is suicide.

A.3 Ko

Let us inspect further the situation in Figure A.3. White can play at the intersection marked \square , resulting in the capture of the black stone marked \blacktriangle . In theory, Black could place another stone where \blacktriangle used to be, resulting in the capture of the white stone at \times . This situation is called *ko*, and the cycle could repeat indefinitely. To prevent this infinite cycle from occurring, the *rule of ko* states that if one player captures in a *ko*, the other player

cannot immediately recapture. In our example, black could not immediately play where  used to be, and would have to wait at least one turn before attempting to capture White's stone at \times . Stated differently, whenever exactly one stone is captured on the board, the previous location of the captured stone becomes the *ko point* until after the next move is played, when either a new ko point is set (if there is another capture of one stone) or there is no ko point set. Playing at a ko point is illegal according to the rule of ko.

There are many subtleties to ko, as well as additional ko situations that require careful treatment beyond the basic rule of ko. For a complete exposition of ko, see [44].

A.4 Life and Death

The concepts of *life* and *death* are crucial to the understanding and scoring of Go. Stones are *alive* if they can avoid capture and are *dead* if they cannot avoid capture. For example, White's stones in Figure A.4 are alive. As long as the intersections marked \times remain empty, Black cannot capture White's stones; it would be suicide for Black to play at either intersection marked \times .

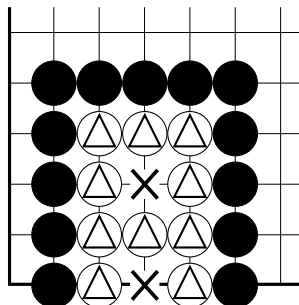


Figure A.4: White's stones are alive.

In Figure A.5, White's stones are dead. Black need only place a stone at the intersection marked \times to capture White's two stones, while it would be suicide for White to play at \times . If Black does not play at \times before the end of the game, then at the end of the game, White's dead stones are removed from the board and taken as prisoners, regardless of the fact that the stones remained on the board.

Generally speaking, stones are alive if they form two or more *eyes*, and they are dead if they form one or zero *eyes*. An eye is a group of empty intersections that is surrounded by stones of a single color. An eye must provide one sure internal liberty. If no sure internal liberty is provided, the group of empty intersections is called a *false eye*. In Figure A.4, White's stones have two eyes at the intersections marked \times , and hence White's stones are alive. In Figure A.5, White's stones have no eyes, and hence White's stones are dead. For a situation with false eyes, see the explanation for *False Eye* in Appendix B. We call an eye a *simple eye* if it has exactly one empty intersection.

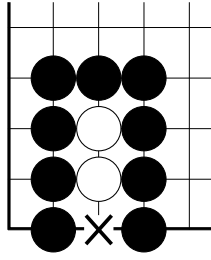


Figure A.5: White's stones are dead.

There are exceptions to the mantra that stones are alive if they form two or more eyes. These rare exceptions are called *seki*, or mutual life. Seki occurs when a player's stones have one or zero eyes, but the opposing player will not attack the stones since such an attack would result in the capture of the opposing player's own stones. The situation in Figure A.6 is an example of seki; neither player will play in the intersections marked \times since such a play would place that player's own stones in atari.

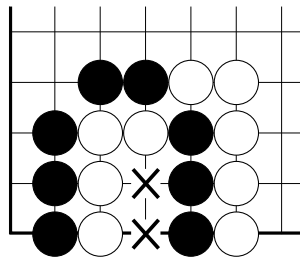


Figure A.6: An example of seki.

For more explanation of and exercises in life and death, see [3].

A.5 Scoring

When both players pass in succession, the game ends. Dead stones are removed from the board and taken as prisoners. Each player counts the total *territory* that player controls. A player's territory is comprised of empty intersections that are surrounded only by that player's stones. *Neutral territory*, which is territory that is not completely surrounded by either player, is not counted towards either player's score.

A player's score is equal to the amount of territory that player controls minus the number of prisoners taken by the opposing player. In addition, since White is disadvantaged by playing second, White is awarded a predetermined amount of additional points at the end of the game. This addition to white's score is called *komi*, and is usually equal to 5.5 or 6.5. Komi is chosen to be a non-integer to avoid ties. The player with the highest score at the end of the game is declared the winner.

Appendix B

Go Terminology

In this section, we provide a list of commonly used Go terms. These terms are useful both for understanding the rules of Go and for describing computer Go algorithms.

Adjacent Two stones, two intersections, or a stone and an intersection are *adjacent* if they are directly connected by a single line segment on the Go board, with no other stone or empty intersection in between. In Figure B.1, the black stones marked \blacktriangle are adjacent, while the unmarked black stones \bullet are not adjacent.

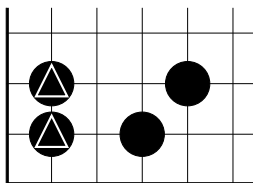


Figure B.1: Marked black stones are adjacent.

Alive Stones are considered *alive* when either the stones cannot be captured or when the stones' owner can still prevent their capture. Generally speaking, a chain with at least two eyes cannot be captured. In Figure B.2, the white stones marked \triangle have two eyes and are alive.

Atari A stone or chain is in *atari* if it has only one remaining liberty. A stone or chain that is in atari risks immediate capture on next turn. In Figure B.3, all white stones marked \triangle are in atari.

Capture A chain is *captured* when it has no liberties. Captured stones are removed from the board and kept as prisoners by the player who captured them. In Figure B.4, if black plays at the intersection marked \times , then Black will capture White's chain.

Chain A *chain* is a set of adjacent stones. The stones in a chain share liberties, and no stone in the chain can be captured unless the entire chain is captured. In Figure B.5,

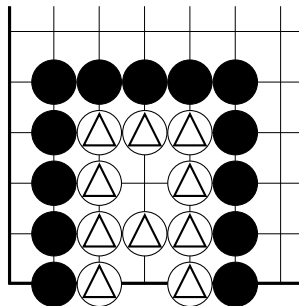


Figure B.2: White's stones are alive.

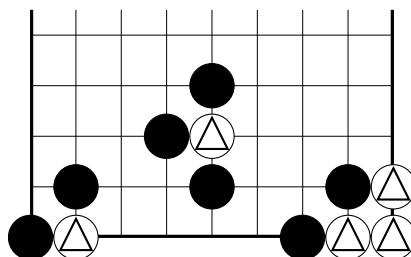






Figure B.3: Marked white stones are in atari.

Black has two chains on the board. Note that because the black stones marked  and  are not adjacent, Black's two groups do not form one large group.

Dan The *dan* rank is an advanced rank. The lowest dan rank is 1-dan (also called shodan), while the highest rank is 7-dan. Each increase in dan rank roughly corresponds to an increase in strength of one handicap stone. There is also a professional dan ranking, which ranges from professional 1-dan to professional 9-dan. The difference between any two professional dan-ranked players is usually very small, with a difference in strength no more than two or three handicap stones even between a professional 1-dan player and a professional 9-dan player.

Dead Stones are considered *dead* when the stones' owner cannot prevent their capture, even if the game ends before capture. Generally speaking, a chain with no potential to make at least two eyes cannot avoid being captured. Dead groups are taken as prisoners at the end of the game. In Figure B.6, the white stones marked  are dead. If this position remains at the end of the game, the white stones marked  are removed and taken by Black as prisoners.

Eye A group of empty intersections that is surrounded by stones of a single color is called an eye. An eye must provide one sure internal liberty. In Figure B.7, White's stones have two eyes at the intersections marked \times .

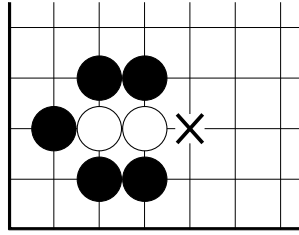


Figure B.4: White's chain has two liberties.

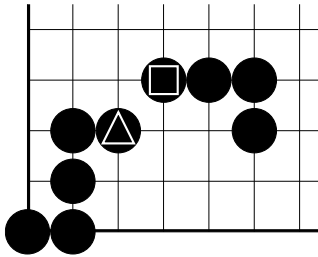


Figure B.5: Black has two chains on the board.

False Eye A group of empty intersections that is surrounded by stones of a single color, but that does not provide a sure internal liberty. In Figure B.8, the intersection marked \times is a false eye. If black moves at \times , then the white stones marked \triangle are captured, leaving the remaining white group with only one eye. Thus all white stones in Figure B.8 are dead.

Handicap When two players with different ranks play each other in Go, *handicap* stones are placed on the board for the weaker player in order to make the chances of winning roughly equal for both players. The number of handicap stones placed on the board in an amateur game is generally the difference in amateur kyu/dan rank. For example, if a 20-kyu player plays against a 16-kyu player, the 20-kyu player will place four handicap stones on the board at the start of the game. There are smaller differences in strength between players with professional dan ranks, so that only a few stones would be placed on the board to make a game between a 1-dan professional and a 9-dan professional an even match. In handicap games, komi is usually set to 0.5.

Ko The term *ko* refers in general to a cycling series of captures. For example, in Figure B.9, if Black plays ① at the intersection marked \times , Black captures the white stone marked \triangle . Then White could play ② where \triangle used to be, capturing ①. This cycle could repeat infinitely. To prevent such a cycle, the *rule of ko* states that if one player captures in a ko, the other player cannot immediately recapture. In this example, after black captures White \triangle with ①, White could not place ② where \triangle used to be. White could attempt recapture with ④ at the earliest, provided the opportunity

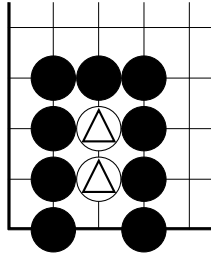


Figure B.6: White's stones are dead.

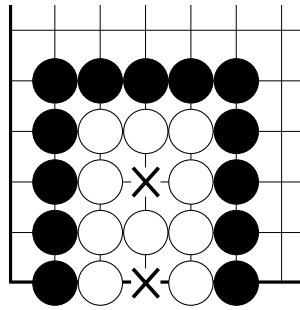


Figure B.7: White's stones have two eyes.

still exists.

Komi In Go, Black always plays first. Since White is disadvantaged by playing second, a predetermined amount is added to White's score at the end of the game. This addition to White's score is called *komi*. The value of komi is usually 5.5 or 6.5 for a standard 19 by 19 game. Komi is usually chosen to be a non-integer to prevent ties. In handicap games, komi is usually set to 0.5.

Kyu The *kyu* rank is an amateur rank. There is no agreed-upon lowest kyu rank, but 30-kyu generally indicates a beginner who has just learned the rules. The highest kyu rank is 1-kyu, which is only one rank below amateur 1-dan. Each decrease in kyu rank roughly corresponds to an increase in strength of one handicap stone.

Ladder A *ladder* is a forcing sequence that results in the capture of the opponent's stones. More rigorously, we define ladders with the following recursive definition. Without loss of generality, White's stones are in a ladder if (1) the stones are in atari, (2) it is White's turn to move, and (3) after White moves, Black can play a move which either captures the white stones in question or places them in a ladder. For example, in Figure B.10, White's stones are in a ladder, since the sequence (1) to (8) forces the capture.

Liberty Free intersections that are adjacent to a chain are called *liberties* of that chain. When a chain has only one liberty, the chain is in atari. A chain is captured when it

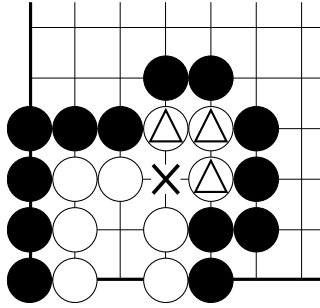


Figure B.8: White's stones have one eye and one false eye.

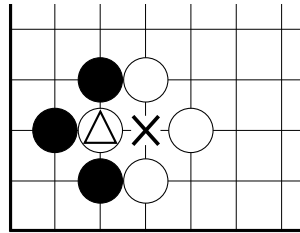


Figure B.9: An example of Ko.

has no liberties. In Figure B.11, White's chain has two liberties, at the intersections marked \times .

Moku The Japanese word for intersection is *moku*. Score is counted in terms of moku.

Neutral Territory is considered *neutral* if it does not belong to either player. In Figure B.12, the territory marked \times is neutral territory.

Prisoner Stones that are captured during the game or dead stones that are removed at the end of the game are called *prisoners* and are kept by the player who captures/removes them. A player's score at the end of the game is negatively affected by the number of prisoners taken by the opposing player.

Score A player's *score* is equal to the amount of territory that player controls minus the number of prisoners the opposing player took (plus komi if it applies). A player's stones that are still on the board do not count toward that player's score. The player with the highest score at the end of the game wins.

Simple Eye Without loss of generality, an empty intersection is a *simple eye* for Black if the intersection is adjacent only to Black stones and if one of the following three conditions holds:

1. The empty intersection is not on the edge of the board, and Black owns at least three of the four immediate corners.

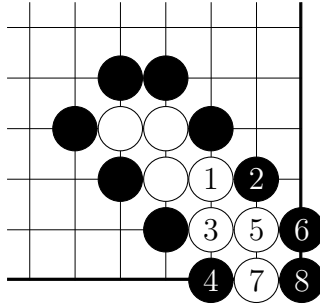


Figure B.10: White's chain is in a ladder. The sequence White 1 to Black 8 forces the capture.

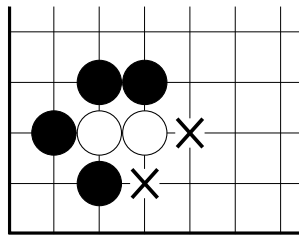


Figure B.11: White's chain has two liberties.

2. The empty intersection is on the edge of the board but is not on the corner of the board, and Black owns at least two of the three immediate corners.
3. The empty intersection is on the corner of the board and Black owns the only immediate corner.

In other words, a simple eye is an eye comprised of only one intersection that is not a false eye.

Suicide A *suicide* is a move by a player which results in the immediate capture of one or more of that player's stones, after possible capture of enemy stones is considered. In most Go rulesets, suicide is illegal. Note that in Figure B.13, placing a black stone at the intersection marked \times is not suicide, since it first results in the capture of the white stone marked \triangle .

Territory At the end of the game, the empty intersections that a player surrounds and undisputedly controls is called that player's *territory*. Along with captured stones and komi, the number of intersections inside a player's territory constitutes that player's score.

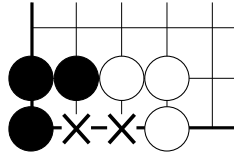


Figure B.12: Neutral territory belongs to neither player.

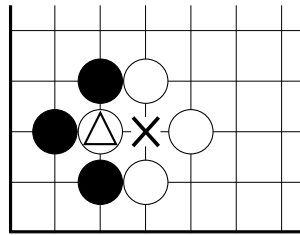


Figure B.13: Black's play at \times is not suicide.

Appendix C

Additional Go Resources

For an easy introduction to rules of Go, see [8]. For a more traditional introduction to Go and Go strategy, see [28]. For lessons in strategy and playing techniques in Go, see [29]. For more information on the history of Go, see [37] and [41]. For information about Go on the Internet, see [40] and [2]. For a comprehensive list of Internet Go resources, see [4].