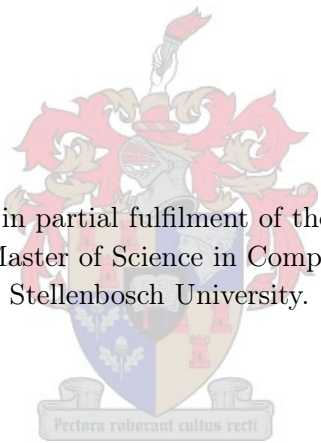
Decision Forests for Computer Go Feature Learning

Francois van Niekerk



Thesis presented in partial fulfilment of the requirements of
the degree of Master of Science in Computer Science at
Stellenbosch University.

Supervisor: Dr. Steve Kroon

March 2014

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: February 23, 2014

Abstract

In computer Go, moves are typically selected with the aid of a tree search algorithm. Monte-Carlo tree search (MCTS) is currently the dominant algorithm in computer Go. It has been shown that the inclusion of domain knowledge in MCTS is able to vastly improve the strength of MCTS engines. A successful approach to representing domain knowledge in computer Go is the use of appropriately weighted tactical features and pattern features, which are comprised of a number of hand-crafted heuristics and a collection of patterns respectively. However, tactical features are hand-crafted specifically for Go, and pattern features are Go-specific, making it unclear how they can be easily transferred to other domains.

As such, this work proposes a new approach to representing domain knowledge, decision tree features. These features evaluate a state-action pair by descending a decision tree, with queries recursively partitioning the state-action pair input space, and returning a weight corresponding to the partition element represented by the resultant leaf node. In this work, decision tree features are applied to computer Go, in order to determine their feasibility in comparison to state-of-the-art use of tactical and pattern features. In this application of decision tree features, each query in the decision tree descent path refines information about the board position surrounding a candidate move.

The results of this work showed that a feature instance with decision tree features is a feasible alternative to the state-of-the-art use of tactical and pattern features in computer Go, in terms of move prediction and playing strength, even though computer Go is a relatively well-developed research area. A move prediction rate of 35.9% was achieved with tactical and decision tree features, and they showed comparable performance to the state of the art when integrated into an MCTS engine with progressive widening.

We conclude that the decision tree feature approach shows potential as a method for automatically extracting domain knowledge in new domains. These features can be used to evaluate state-action pairs for guiding search-based techniques, such as MCTS, or for action-prediction tasks.

Uittreksel

In rekenaar Go, word skuiwe gewoonlik geselekteer met behulp van 'n boomsoektogalgoritme. Monte-Carlo boomsoektog (MCTS) is tans die dominante algoritme in rekenaar Go. Dit is bekend dat die insluiting van gebiedskennis in MCTS in staat is om die krag van MCTS enjins aansienlik te verbeter. 'n Suksesvolle benadering tot die voorstelling van gebiedskennis in rekenaar Go is taktiek- en patroonkenmerke met geskikte gewigte. Hierdie behels 'n aantal handgemaakte heuristieke en 'n versameling van patrone onderskeidelik. Omdat taktiekkenmerke spesifiek vir Go met die hand gemaak is, en dat patroonkenmerke Go-spesifiek is, is dit nie duidelik hoe hulle maklik oorgedra kan word na ander velde toe nie.

Hierdie werk stel dus 'n nuwe verteenwoordiging van gebiedskennis voor, naamlik besluitboomkenmerke. Hierdie kenmerke evalueer 'n toestand-aksie paar deur rekursief die toevoerruimte van toestand-aksie pare te verdeel deur middel van die keuses in die besluitboom, en dan die gewig terug te keer wat ooreenstem met die verdelingselement wat die ooreenstemmende blaarnodus verteenwoordig. In hierdie werk, is besluitboomkenmerke geëvalueer op rekenaar Go, om hul lewensvatbaarheid in vergelyking met veldleidende gebruik van taktiek- en patroonkenmerke te bepaal. In hierdie toepassing van besluitboomkenmerke, verfyn elke navraag in die pad na onder van die besluitboom inligting oor die posisie rondom 'n kandidaatskuif.

Die resultate van hierdie werk het getoon dat 'n kenmerkentiteit met besluitboomkenmerke 'n haalbare alternatief is vir die veldleidende gebruik van taktiek- en patroonkenmerke in rekenaar Go in terme van skuifvoorspelling as ook speelkrag, ondanks die feit dat rekenaar Go 'n relatief goedontwikkelde navorsingsgebied is. 'n Skuifvoorspellingskoers van 35.9% is behaal met taktiek- en besluitboomkenmerke, en hulle het vergelykbaar met veldleidende tegnieke presteer wanneer hulle in 'n MCTS enjin met progressiewe uitbreiding geïntegreer is.

Ons lei af dat ons voorgestelde besluitboomkenmerke potensiaal toon as 'n metode vir die outomaties onttrek van gebiedskennis in nuwe velde. Hierdie eienskappe kan gebruik word om toestand-aksie pare te evalueer vir die leiding van soektog-gebaseerde tegnieke, soos MCTS, of vir aksie-voorspelling.

Acknowledgements

I would like to express my sincere gratitude to the following people for their support throughout this work:

- My supervisor, Dr. Steve Kroon, for your extensive guidance and support, that frequently went beyond the call of duty.
- The MIH Media Lab, for the generous financial support and excellent research environment.
- My fellow lab colleagues, especially Hilgard Bell, Dirk Brand, Leon van Niekerk and the other members of the gaming research group, for your advice and support inside and outside of the lab.
- My friends and family, for your loving help.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

AIG	Augmented intersection graph
ASG	Augmented stone graph
AUC	Area under the curve
BSG	Basic seki graph
BTM	Bradley-Terry model
CFG	Common fate graph
DS	Descent-Split
EDS	Entropy Descent-Split
ELS	Entropy-Loss-Split
EWS	Entropy Win-Split
GBTM	Generalized Bradley-Terry model
IG	Intersection graph
KGS	KGS Go Server
LGRF	Last good reply with forgetting
LS	Loss-Split
MCTS	Monte-Carlo tree search
MLE	Mean log-evidence
NDS	Naive Descent-Split
RAVE	Rapid action value estimation
RL	Reinforcement learning
SG	Stone graph
SS	Symmetric-Separate
UCB	Upper confidence bounds
VLW	Variance of the natural logarithm of the weights
WE	Winrate-Entropy
WLS	Win-Loss-Separate
WRS	Winrate-Split
WS	Win-Split
WSS	Weighted Symmetric-Separate
WWE	Weighted Winrate-Entropy
WWLS	Weighted Win-Loss-Separate

Terminology

atari	When a chain has only one liberty.
augmented graph	A stone or intersection graph with an auxiliary node.
auxiliary node	Additional node used to indicate the candidate move.
capture	Remove a chain because it has zero liberties.
CFG distance	Length of the shortest path in a CFG.
chain	Region of black or white stones.
circular distance	Distance metric.
decision tree feature	The approach proposed in this work.
descent statistics	Recorded statistics used for query selection.
discovered graph	Representation of the ordered list of predicates.
feature instance	Set of features with trained weights.
feature levels	Set of mutually-exclusive options a feature can assume.
frontier	MCTS tree nodes with unexplored children.
GnuGo	Traditional computer Go engine.
ko	Go rule forbidding the repetition of previous positions.
komi	Points to offset the advantage of moving first in Go.
liberties	Adjacent empty intersections of a chain.
Mogo	MCTS computer Go engine.
Oakfoam	MCTS computer Go implementation used as a base.
Pachi	MCTS computer Go engine.
playout	MCTS simulation.
progressive bias	Method for using domain knowledge in MCTS.
progressive widening	Method for using domain knowledge in MCTS.
pseudo-liberties	Variation of liberties used for computational efficiency.
quality criterion	Component of the query selection policy.
query language	Set of queries used in a decision tree feature.
query selection policy	Policy for selecting decision tree queries.
query system	State-action pair representation and query language.
region	Orthogonally-adjacent intersections of the same type.
selection policy	Policy for node selection during MCTS tree descent.
seki	Stable Go situation where neither player should move.
simulation policy	Policy for move selection during an MCTS simulation.
suitability condition	Component of the query selection policy.
team	Set of individuals in the GBTM.
test	Evaluation of a set of feature instances.

Chapter 1

Introduction

Go is a sequential two-player board game of perfect information [1]. It is well-known for its great tactical and strategic depth, despite its simple, elegant rules. Partially due to this emergent complexity, Go has been played for thousands of years and undergone extensive study by professional players and scholars of the game. Efforts to analyze Go theoretically have led to advancements in the field of combinatorial game theory (CGT) [2, 3, 4], and decades of research into developing strong computer players for the game have resulted in a number of new artificial intelligence (AI) techniques for games [5, 6]. However, Go still remains a notable challenge for AI, with top human players still being considerably stronger than the best computer Go engines [6, 7].

In order to select strong moves, computer Go engines typically search a game tree containing legal moves from the current position and their followups, with some form of evaluation applied to the tree leaves [5, 6]. Traditionally, the minimax or negamax algorithms have been used to find the optimal move from the root of the tree [8]. An aspect of Go which makes computer Go particularly difficult is the huge branching factor of these trees — in a typical Go board position there are usually more than 100 legal moves whereas many other games have an order of magnitude fewer moves to consider [9].

One common approach to mitigating the branching factor of Go is the use of an ordering constructed over a position's legal moves, to selectively evaluate the position's node by only investigating certain children in the game tree selected using the ordering [6, 10, 11]. While there are various methods of encoding domain knowledge so that it can be used for move ordering, automated methods are typically preferred.

In order to efficiently select a move for play, various approaches to growing and evaluating the game tree are used. In traditional computer Go engines,

game trees were usually grown and positions evaluated using a large collection of hand-crafted domain knowledge, which is difficult to maintain and extend [12]. While these techniques have been able to achieve a moderate level of strength, they have reached a point where progress is difficult and has stalled [5].

Recently, traditional techniques have largely been replaced by more effective Monte-Carlo tree search (MCTS) approaches [6, 7]. MCTS combines stochastic Monte-Carlo simulations with game tree search principles, and is currently the *de facto* algorithm for computer Go [6]. The inclusion of domain knowledge has greatly improved the strength of MCTS engines [6, 13, 14]. MCTS is able to achieve a moderate level of playing strength with very limited domain knowledge: compared to traditional techniques, MCTS is able to achieve the same level of play with much less domain knowledge [6, 7].

There are a variety of methods for using domain knowledge in computer Go with MCTS; most of them are focused on improving the tree and/or simulation policies [6, 14]. While domain knowledge in the simulation policy often focuses on selecting moves to make the simulations better represent the true value of a position, domain knowledge in the tree policy is typically focused on mitigating the large branching factor or focusing effort on more promising moves [6]. In MCTS, the most successful approach to constructing a move ordering for branching factor reduction is feature extraction that encodes domain knowledge from a collection of high-level games, as presented in [11]. These features currently encode Go-specific pattern and tactical information for move evaluation, by analyzing the surrounding board intersections and a few simple tactics respectively. Due to their high level of accuracy for move prediction, the use of these features is able to greatly improve the playing strength of an MCTS computer Go engine by limiting the effective branching factor of the game tree.

1.1 Problem Statement

While the use of features for encoding domain knowledge in computer Go has been shown to be a powerful technique, current pattern features are specific to Go, and it is not yet clear how they can be applied to other domains where patterns are not readily available; also, in many other domains, tactics have not been hand-crafted yet. Furthermore, many important Go concepts are not represented by the current feature extraction approaches. Therefore, a more general automated method for extracting domain knowledge in the form of features is desirable, as hand-encoding domain knowledge is time-consuming and error-prone, and thus often not feasible for a new domain.

This work proposes decision tree features, a new and more flexible technique for extracting domain knowledge. These features evaluate a state-action pair using queries structured in a decision tree. These decision trees recursively partition the input space of state-action pairs as queries are encountered in a tree descent, and weights at the resultant leaf nodes are used for state-action pair evaluation. In this work, decision tree features are applied to Go, and the state-action pair corresponds to a Go position and candidate move. As such, the decision tree queries examine the area surrounding the candidate move. Due to their descriptive flexibility, these decision tree features will be able to encode concepts from Go domain knowledge that the automated components (pattern features) of the current approach cannot.

While decision tree features are more easily transferable to other domains, due to the lack of comparable results in these domains, this work will evaluate the performance of decision tree features at extracting domain knowledge for computer Go. The hypothesis is that these decision tree features will be able to extract domain knowledge with comparable performance to the current feature extraction, as measured according to move prediction and the playing strength of a computer Go engine. If this is the case, we conjecture that decision tree features will be a feasible alternative to tactical features in other domains.

1.2 Objectives

The objectives of this work are the following:

- Propose an approach using decision trees as features for domain knowledge extraction and state-action pair evaluation.
- Apply the proposed approach to the domain of Go.
- Implement a proof-of-concept feature system for Go that allows tactical, pattern, and decision tree features to be extracted, trained, and used in both move prediction tests and an MCTS computer Go engine.[1]
- Investigate the performance of the Go decision tree features by measuring the impact of various design choices (domain representation, query selection policy, tree size and forest size) on move prediction and playing strength.
- Determine the feasibility of decision tree features, as an alternative to state-of-the-art tactical and pattern features.

[1] Tactical and small (3x3) pattern features were initially present in the implementation.

1.3 Contributions

The work presented in this thesis[2] makes the following contributions:

- An approach to using decision trees as features is proposed, which is more general and flexible than the current tactical and pattern features.
- These decision tree features are applied to Go as part of an open-source MCTS engine, and the impact of various factors on their performance, in terms of move prediction and playing strength, is measured.
- It is shown that decision tree features are a feasible alternative to computer Go tactical and pattern features, both in terms of move prediction and playing strength with a fixed number of simulations per move.[3]

1.4 Outline

The remainder of this thesis is structured as follows: First, Chapter 2 introduces the necessary background information, including an overview of feature extraction and its use in computer Go, as well as the state of the art for computer Go move prediction. Chapter 3 presents the proposed approach to using decision trees as features and applies it to computer Go, and Chapter 4 discusses the implementation considerations for applying decision tree features to computer Go. Chapter 5 evaluates the implementation of the proposed approach and analyzes the results. Finally, Chapter 6 provides a conclusion and overview of the work and results presented in this thesis.

[2]Initial work was presented at the Workshop on Computer Games at the International Joint Conference on Artificial Intelligence [15].

[3]Decision tree features are only found to be comparable to tactical and pattern features with a fixed number of playouts per move. However, this work only explored the feasibility of decision tree features and as such, the implementation still has much scope for optimization.

Chapter 2

Background and Related Work

This chapter presents background information on various topics needed to provide context in the rest of this work. The current approaches to storing domain knowledge for computer Go will be illustrated, and their relevant shortcomings highlighted.

First, Section 2.1 presents a summary of the board game of Go. Second, Section 2.2 provides an overview of the field of computer Go. Then Section 2.3 outlines Monte-Carlo tree search (MCTS), the dominant computer Go algorithm, before Section 2.4 reviews the use of domain knowledge in computer Go. Section 2.5 describes some uses of graphs for computer Go. Section 2.6 discusses the generalized Bradley-Terry model (GBTM) and its use in modeling features, and training their weights, in computer Go. Finally, Section 2.7 outlines the traditional decision tree approach to classification.

2.1 The Game of Go

Go is a combinatorial board game, i.e. a two-player game with discrete sequential positions and perfect information [1, 3]. It is played on a board consisting of a rectangular grid of intersections, with 19x19 being the standard size. This work focuses on 19x19 boards, as data with other board sizes is limited. Figure 2.1 shows an example Go position on a 19x19 board.

The rules and essential concepts of Go that are used in this work can be summarized as follows: two players, black and white, alternate placing stones of their respective color on empty board intersections, or passing.[1] In this way, intersections have a status: black, white or empty. Orthogonally contiguous intersections with the same status form a *region*. A region of black or white stones is a *chain*, and the orthogonally adjacent empty intersections

[1]Pass moves are typically only played at the end of the game.

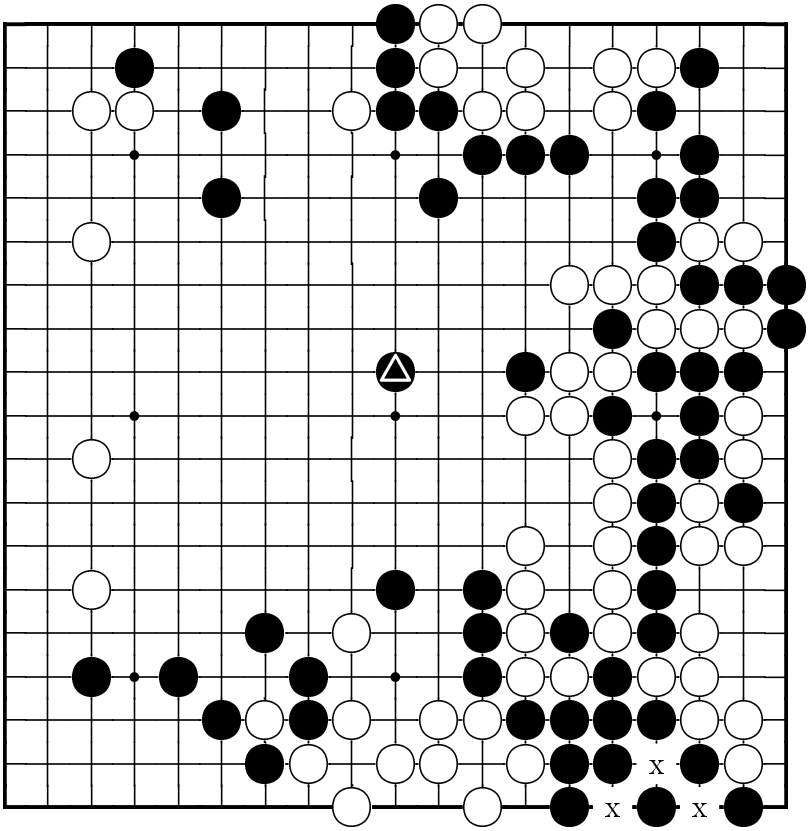


Figure 2.1: A famous Go game (Honinbo Shusaku vs. Gennan Inseki, 1846). Shusaku has just played his famous 'ear-reddening' move, indicated by △. All the empty intersections, except those marked with 'x,' are legal moves for white.

of a chain are its *liberties*. After each move, chains of the opposing color with zero liberties, are said to be *captured*, and are entirely removed from the board, with the corresponding intersections becoming empty. Chains that have only one remaining liberty are said to be in *atari*. Moves that would result in a chain of the same color with zero liberties (*suicide* moves), or moves that would result in a board position identical to a previous position (*ko* rule) are not allowed. The game ends after two consecutive pass moves (one by each player). The winner of the game is the player controlling the largest portion of the board.[2]

A typical Go position on a 19x19 board can easily have more than 100 legal moves [9]. Figure 2.1 shows an example Go board with a position from a famous game. In this example (which shows a critical point of the game) there are over 200 legal moves — all but the three empty intersections marked with 'x' are legal moves for white.

To improve the speed of playing moves in computer implementations, *pseudo-liberties* are sometimes used [16]. Pseudo-liberties are an approximation of normal liberties that can result in a significant reduction in required computational resources. The number of pseudo-liberties of a chain is the sum of the number of adjacent empty intersections of each stone in the chain, i.e. liberties that have more than one adjacent stone from the relevant chain are counted multiple times (once for each adjacent stone in the chain). An important attribute of pseudo-liberties is that it can easily be determined if a chain is in atari, by keeping track of the number of pseudo-liberties, the sum of their positional indices[3], and the sum of the squares of their positional indices. Refer to Appendix A for an implementation that uses pseudo-liberties.

2.2 Computer Go

Computer Go is the field of game AI concerned with playing the game of Go. Although there has been much progress in this field in recent years, top human Go players are currently considerably stronger than the strongest computer Go engines on a 19x19 board [6, 7]. Among recent computer Go results against a professional Go player, the best result is a single win on 19x19 with just four handicap stones, indicating that top humans are cur-

[2]An intersection is controlled by a player if it has the same color as that player, or if the empty region containing the intersection is surrounded (on all possible orthogonal intersections) by intersections of that player's color.

[3]A positional index of an intersection is a unique numerical identifier for the intersection, such as the intersection's row number plus the number of rows on the board times the intersection's column number.

rently close to four ranks stronger than the best computer Go engines [17] — four ranks difference corresponds to an expected winrate of at least 90% for the stronger player in an even game (with no handicap).

In order to select strong moves in a game, computer Go engines typically search a game tree. A game tree[4] is a tree consisting of game states as nodes, with actions represented by edges to other game states. In Go, the game state is the current board position[5] and the actions are player moves. To search for a move, a game tree is constructed, with the current position at the root of the tree and some form of evaluation applied to the tree leaves [5, 6].

Traditionally, computer Go has employed the minimax or negamax algorithms to find the optimal move from the root of the game tree, by propagating the evaluation values from the leaves to the root of the tree [5, 8]. Unfortunately, these algorithms rely on an evaluation function, which is notoriously difficult to design and implement for Go [12]. Furthermore, the large branching factor of Go (due to the large number of legal moves in a typical Go position) means that the overall size of the game tree, and the number of leaves, grows very quickly with the depth of the tree, limiting the search, and therefore the strength of the engines [5].

Traditional computer Go engines make extensive use of hand-crafted domain knowledge for the evaluation function and mitigation of the large branching factor [5, 12]. These approaches often attempt to mimic humans, using high-level concepts (such as groups, territory, influence, life and death) which can be difficult to precisely encode [5, 12].

Domain knowledge can be used to mitigate the branching factor in Go by constructing a move ordering over a position's legal moves, and then using this ordering to selectively evaluate the position's tree node by only investigating selected children in the game tree [6, 10, 11]. While there are various methods of encoding domain knowledge for move ordering, automated methods, such as pattern features described in Section 2.4.1, are typically preferred.

While these traditional computer Go techniques have been able to achieve a moderate level of strength, they have reached a point where progress is difficult and has stalled, largely due to the difficulty of extending large collections of hand-crafted domain knowledge, and the implicit limit on game tree depth due to the branching factor [5]. Recently these traditional techniques have

[4]Although game trees are called 'trees' for historical reasons, it can be more efficient to represent them as directed acyclic graphs, depending on the domain. In this work, the strict definition of trees is used.

[5]The game state technically also contains the move history, to determine illegal moves according to the ko rule [1].

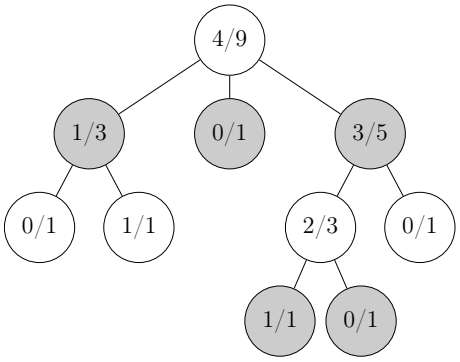


Figure 2.2: Example MCTS tree. Nodes show the number of playout wins over the total number of playouts through that node.[8] Shaded nodes indicate the opponent plays from this position.

largely been replaced by more effective Monte-Carlo tree search (MCTS) approaches [6, 7].

Section 2.3 next introduces the MCTS algorithm, then Section 2.4 discusses approaches to extracting and using domain knowledge for computer Go.

2.3 Monte-Carlo Tree Search

Monte-Carlo (MC) simulations are stochastic simulations of a model. Repetition of MC simulations can result in good estimates for problems without known deterministic solutions. In computer Go, MC simulations, usually referred to as *playouts*, are performed by selecting and playing moves according to a *simulation policy*, beginning from an initial board position.

A feasible simulation policy can be as simple as selecting random legal moves.[6] In a playout, moves are played until the game ends due to two consecutive passes. In this terminal position, it is easy to score the position and therefore determine the result of the playout: win or loss. This playout result can be considered a sample of the value of the initial playout position. Due to their simplicity, playouts can be performed very quickly (in the order of 1000 playouts per second per core on a 19x19 board[7]), to get a better idea of the value of the initial position.

[6] A simulation policy in computer Go will at least typically exclude clearly bad moves that 'fill eyes,' or passing while there are other legal moves that don't 'fill eyes.'

[7] A test of Oakfoam, a computer Go MCTS engine, with the MCTS implementation used in this work, resulted in 750 playouts per second from an empty 19x19 board.

Monte-Carlo tree search (MCTS) combines MC simulations with game tree search principles by performing playouts from game tree nodes [6]. Due to dramatic performance improvements over traditional techniques, MCTS is currently the *de facto* algorithm for computer Go [6, 7]. An MCTS tree's nodes store the statistics of the playouts (effectively the number of wins and losses) that start from any descendant of the node (including the node itself). Figure 2.2 shows an example MCTS tree.

MCTS constructs a game tree by iterating the following four steps: selection, expansion, simulation and backpropagation. Figures 2.3–2.6 illustrate the four steps, starting from the example tree in Figure 2.2.

During selection, the current MCTS tree is descended, from the root to a *frontier node*[9] by selecting nodes according to the *selection policy*. The upper confidence bounds (UCB) policy, a popular[10] selection policy, selects the child node i that has the largest *urgency* $U(i)$ at each descent step to implement a exploration-exploitation trade-off [6]. The urgency of node i is shown in Equation 2.1, where n_i is the number of playouts through node i, r_i is the winrate[11] of these playouts, N is the number of playouts through the parent of node i, and C is the *exploration constant*. The exploration constant C can be adjusted to balance exploring new nodes against the exploitation of moves that currently appear to be more favourable. In the default UCB policy, unexplored nodes have an urgency of infinity and are thus selected before explored nodes.

$$U(i) = r_i + C\sqrt{\frac{\ln N}{n_i}} \tag{2.1}$$

After selection has reached a frontier node, expansion takes place. In expansion, a new child node, the *expansion node*, is added to the frontier node found during selection. The simulation step, consisting of performing a playout starting from the expansion node, is then performed. Finally, backpropagation updates the expansion node and its ancestors with the result of the playout (i.e. win and visit counts are updated).

[8]Note that the playout results are from the perspective of the player to move from the root node (the game tree shown is a minimax tree, not a negamax tree), and the sum of playouts through children nodes are typically not equal to the playouts through the parent node, as the children are only added to the tree after a playout through the parent node has been performed.

[9]A frontier node is a node that has one or more unexplored legal children moves.

[10]In practice, the UCB policy is typically augmented with other techniques such as Rapid Action Value Estimation (RAVE) [6]. Refer to Section 4.6 for more details.

[11]The winrate of a node is from the perspective of the player that just moved at the root of the tree (such as in a negamax tree).

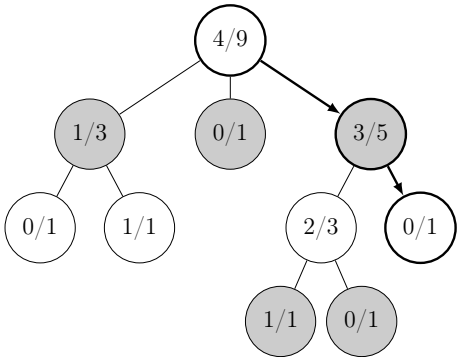


Figure 2.3: Selection step of an example MCTS iteration.

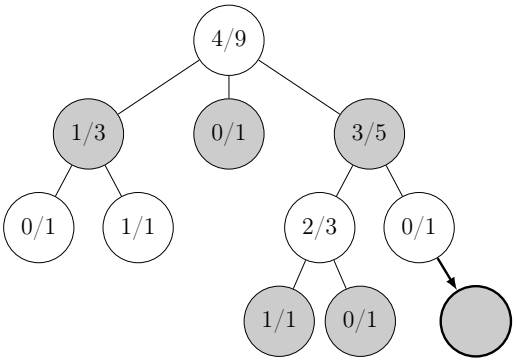


Figure 2.4: Expansion step of an example MCTS iteration.

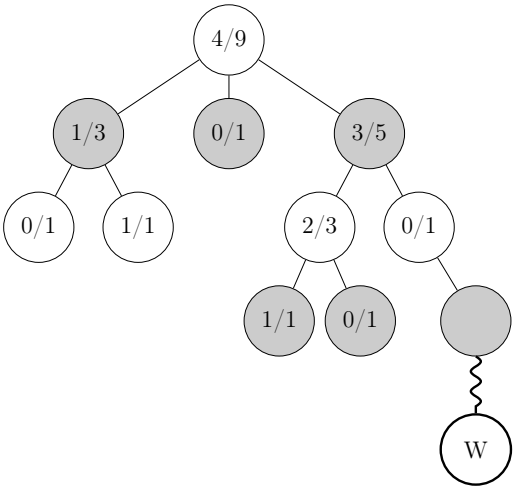


Figure 2.5: Simulation step of an example MCTS iteration, showing a playout that resulted in a win (W) for the player to move from the root node.

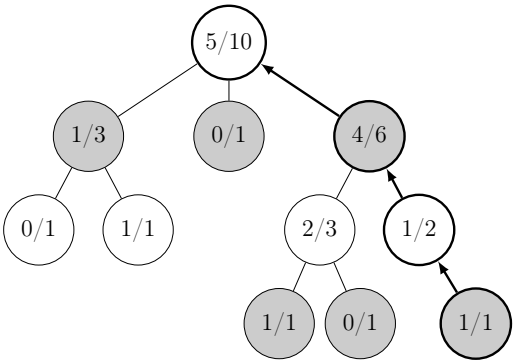


Figure 2.6: Backpropagation step of an example MCTS iteration.

A large number of these MCTS iterations, usually just referred to as playouts[12], are typically performed when searching for a move, to give a good evaluation of the candidate moves. Due to the nature of MCTS, the number of playouts performed is highly flexible — MCTS can make use of as much (or little) computational power and time as is available [18]. Results have confirmed that an increase in the number of playouts typically results in an increase in overall playing strength (with diminishing returns) [6, 18]. Furthermore, it has been shown that, under general conditions with infinite time and $C \geq \sqrt{2}$, MCTS will converge to optimal play [19].

While very simple selection and simulation policies can result in a moderate level of playing strength, it has been shown that incorporating domain knowledge into these policies can greatly improve the strength of computer Go MCTS engines [6, 13, 14]. Incorporating domain knowledge into the selection policy usually focuses on mitigating the large branching factor of the tree, and domain knowledge in the simulation policy usually focuses on making the playout results better represent the true value of a position. Section 2.4 examines the use of domain knowledge for computer Go.

2.4 Domain Knowledge for Computer Go

Domain knowledge is critical to traditional computer Go techniques [5, 12], and MCTS can also greatly benefit from its use [6, 13, 14]. Traditional techniques typically attempt to encode domain knowledge corresponding to high-level concepts used by humans (such as groups, territory, influence, life and death) [5, 12]. This approach tends to involve constructing a number of hand-crafted models of these different concepts, and their design and imple-

[12]A playout can also refer to just the simulation step, depending on the context.

mentation is difficult, time-consuming and error-prone. Since the onset of the dominance of MCTS approaches, this approach has not been directly used by many computer Go engines [6, 7]. While there have been various attempts to include automated methods for extracting domain knowledge in traditional computer Go approaches, none have been particularly successful [5].

MCTS approaches often make use of simpler approaches, including tactical heuristics, to guide the selection and simulation policies [6]. While these approaches are frequently hand-crafted, their scope is severely limited in comparison to those used by traditional techniques, making design and construction much easier [6, 20].

Section 2.4.1 introduces Go features, a successful method of encoding domain knowledge for MCTS approaches to computer Go that makes use of automated methods. Section 2.4.2 then describes strategies for incorporating domain knowledge into the MCTS selection policy, and Section 2.4.3 briefly discusses methods of including domain knowledge into the MCTS simulation policy.

2.4.1 Go Features

Go features are used to extract and encode domain knowledge for Go, in order to evaluate candidate moves[13] [11, 21]. Go features are traditionally divided into pattern and tactical features. Pattern features are simple encodings of the state of the board intersections surrounding the candidate move. Tactical features encode simple domain knowledge not present in the pattern features, such as whether a move captures a chain in atari. When evaluating a candidate move, each feature takes on one of a number of mutually exclusive *feature levels*. A candidate move is then described by a feature vector[14], with each vector component specifying which level the corresponding feature assumes for the candidate move.

While the value of a candidate move should theoretically not depend on the previous moves in a game (with some minor exceptions[15]), a number of tactical features typically used depend on the previous moves, especially the distance to the previous two moves [11, 22]. The inclusion of this history information makes a significant difference to performance. As such, this work typically uses this history information, with some tests in Section 5.10 that consider a history-agnostic set of tactical features.

[13]Features do not typically attempt to determine the legality of a move; it is therefore assumed that only legal moves are evaluated.

[14]Vector operations are not performed on this feature vector.

[15]The ko rule requires information about the game history.

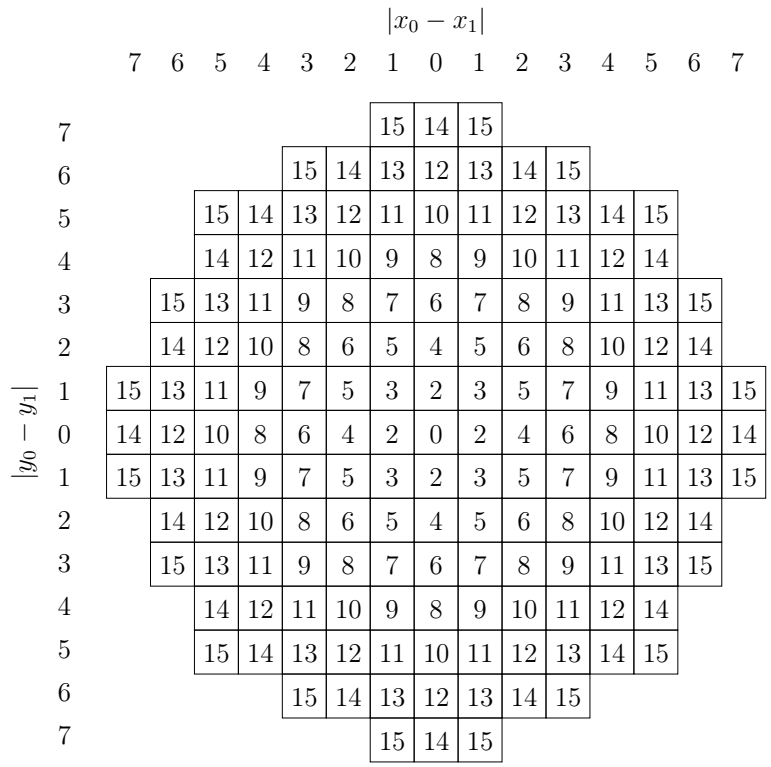


Figure 2.7: Visualization of the circular distance (δ°) measure. The δ° value that corresponds to a specific absolute difference in x and y coordinates is shown.

Patterns are typically represented by a single feature with many levels, with each level corresponding to a different pattern. The intersections included in a pattern are typically all those within a certain distance from the center of the pattern, and patterns are typically centered on a candidate move. A popular distance measure used for large patterns in Go, and in this work, is *circular distance* [11]: given two intersections with coordinates (x_0, y_0) and (x_1, y_1), the circular distance δ°, between the two points is defined by:

$$\delta^\circ = |x_0 - x_1| + |y_0 - y_1| + \max(|x_0 - x_1|, |y_0 - y_1|) \tag{2.2}$$

Figure 2.7 shows a visualization of the circular distance measure by showing the δ° value for a given absolute difference in x and y coordinates. In this work, frequently-occurring patterns including intersections within a circular distance of up to 15 are used.

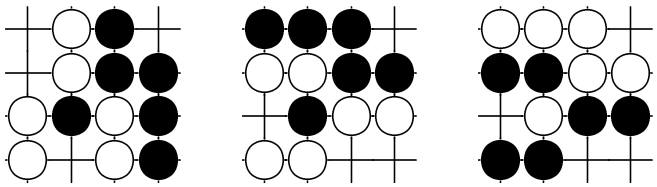


Figure 2.8: Three functionally-equivalent Go positions. From left to right, the second position is a rotation and reflection of the first position, while in the third position it is the opposite player to play in comparison with the other two positions.

Figure 2.8 shows three Go positions that are simple transformations of the same position. In order to ensure such transformed positions are treated identically in terms of domain knowledge, feature levels for patterns should be invariant to changes in rotation, reflection, and whose turn it is to play. Invariance to player turns is usually achieved by swapping stone colors as necessary, while the invariance requirements for rotation and reflection are met by considering the eight combinations of rotation and reflection and using the pattern with the lowest hash value.[16]

In order to make practical use of features, each level of each feature is assigned a trained weight, as discussed in Sections 2.6. Feature weights corresponding to the levels in each candidate move's feature vector are then combined to form a single weight for that move. These move weights can then be used for move prediction, or in the selection or simulation policies of MCTS, as described in Sections 2.4.2 and 2.4.3.

2.4.2 Progressive Strategies for MCTS

Progressive strategies for MCTS attempt to mitigate the game tree branching factor by incorporating domain knowledge into the selection policy of the tree, such that the effect is initially large (when the playout results are few and therefore noisy) and decays over time, as more playout results are included in the relevant part of the tree. Two successful progressive strategies for computer Go are *progressive bias* and *progressive widening*[17] [10, 11]. In the following descriptions, the domain knowledge value of node i is represented by $H(i)$, where a larger value of $H(i)$ indicates that it is more favourable according to the domain knowledge.

[16]Pattern hashes in this work are lossless hashes constructed with 2 bits for each intersection, to represent empty, black or white.

[17]The term 'progressive unpruning' is avoided in this work as it is ambiguously used in different literature [7, 10].

Progressive bias [10] modifies the UCB policy by adding a bias term, selecting the node i that maximizes

$$U_{\mathrm{PB}}(i) = r_i + C\sqrt{\frac{\ln N}{n_i}} + f(H(i), n_i). \tag{2.3}$$

In this equation, $f(x, n)$ incorporates domain knowledge as a prior value in a decaying manner. A typical $f(x, n)$ is x/n.

On the other hand, progressive widening limits the node selection during the selection step to a subset of the candidate children nodes [10, 11]. The nodes of the moves with the highest $H(i)$ values are included in this considered subset, and the size of the subset is slowly increased according to a schedule. A schedule that has been shown to be useful in practice is to add another node to the considered subset after an exponentially-increasing number of playouts have taken place through the parent node.

2.4.3 MCTS Simulation Policies

Domain knowledge has also been successfully incorporated into the MCTS simulation policy to improve the performance of MCTS engines [6, 10, 11, 23]. The two main approaches to including domain knowledge in the simulation policy are the heuristic-based and sample-based approaches.

In the heuristic-based approach, moves in the playouts are chosen using a number of heuristics. These heuristics are usually hand-crafted, and will typically select moves in close proximity to previous moves [6, 23, 20]. This approach was popularized by its use in MOGO, a successful computer Go MCTS engine [6, 23].

In the sample-based approach, playout moves are selected from a probability distribution over all the legal moves, and this distribution is constructed with the aid of domain knowledge (especially the features described in Section 2.4.1) [11]. This approach has been successfully used in CRAZY STONE, another successful computer Go MCTS engine [6, 11]. Simulation balancing has shown to be a successful method of training feature weights for use in this approach [24].

2.5 Common Fate Graphs

While simple Go patterns, as described in Section 2.4.1, are useful for encoding domain knowledge, representations that make use of graphs are also possible. This section examines the Common Fate Graph (CFG) representation. Alternative representations such as the Basic Seki Graph (BSG) [25]

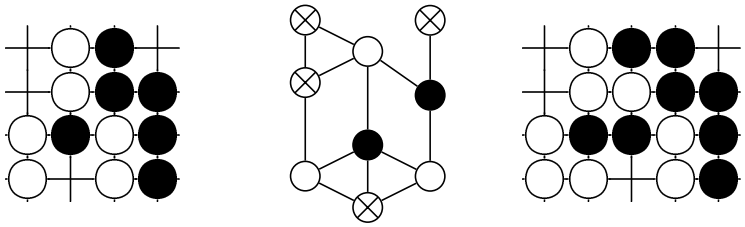


Figure 2.9: CFG representation of a Go board portions. Two normal Go board portions are shown on the left and right, with the common CFG representation of both portions shown in the center.

are also possible graph representations; however, the BSG cannot represent a complete Go board in the general case, and is therefore not considered in this work.

In the CFG representation of a Go board, each chain of stones and each empty intersection is represented by a single graph node, with edges between nodes with adjacent intersections [26]. In this representation, certain functionally-equivalent patterns become equal: Figure 2.9 illustrates the CFG representation of two Go board portions. Due to computational concerns, the practical use of CFGs in computer Go has been limited; however, one notable concept arising from this representation is the notion of the *CFG distance* between intersections: the number of CFG graph edges in the shortest path between the two intersections [7]. The compression and empty modifications introduced in Section 3.3 are inspired by the CFG, and the CFG distance is used in the tactical features listed in Section 5.3.

2.6 The Generalized Bradley-Terry Model

The Bradley-Terry model (BTM) can be used to model the outcome of a competition between two individuals [27]. In this probabilistic model, the skill of an individual i is represented by a positive value γ_i, with a larger value corresponding to a more skilled individual. The model then treats the outcome of a competition between two individuals as a Bernoulli random variable with

$$P(\text{individual } i \text{ beats individual } j) = \frac{\gamma_i}{\gamma_i + \gamma_j}. \tag{2.4}$$

The BTM can be generalized to allow for multiple *teams* of individuals [11, 28]. In the generalized Bradley-Terry model (GBTM) we consider, each competition is won by only one of the teams (all the other teams are

considered losers), and the skill of a team of individuals is represented by the product of the skills of the individuals. Given that I_t is the set of individuals of team t, the γ value of team t is $\gamma_t = \prod_{i \in I_t} \gamma_i$. The GBTM then models the outcome of a competition between a set of teams as a categorical (generalized Bernoulli) random variable:

$$P(\text{team } k \text{ wins in a competition between teams in } T) = \frac{\gamma_k}{\sum_{t \in T} \gamma_t} \quad (2.5)$$

So for example:

$$P(\text{1-2-3 wins against 2-4 and 1-5-6-7}) = \frac{\gamma_1\gamma_2\gamma_3}{\gamma_1\gamma_2\gamma_3 + \gamma_2\gamma_4 + \gamma_1\gamma_5\gamma_6\gamma_7} \quad (2.6)$$

The GBTM has been successfully used to model Go features [11]. In this scenario, candidate moves compete with the other legal moves of a position, in an attempt to be played; feature levels are represented by individuals, and candidate moves are represented by teams of these individuals. In this way, training data consisting of a number of board positions as competitions, with the actual move played in each position set as the winning team, can be used to train the parameters of the model (corresponding to the feature level weights) [11], using various techniques. Minorization-maximization (MM) is an algorithm used for training weights with the GBTM, and is used in this work. At a high level, its role in training weights is described in Section 2.6.1.

Alternative approaches not considered in this work include alternative feature models, such as the Thurstone-Mosteller model [21, 29], and alternative techniques for training weights, such as Loopy Bayesian Ranking, Bayesian Approximation Ranking, and Simulation Balancing [29, 24].

Table 2.1 lists a number of such algorithms and their notable published results in terms of move prediction performance. In terms of move prediction, simulation balancing is not suitable, and alternatives were inferior to the GBTM and MM at the beginning of this work (only the first two results from [21] and [11] were available at the start of this work), while newer results in [29] show that (when more training data is used) the performance of GBTM and MM remains very close to the best alternatives. During the later part of this work, Latent Factor Ranking, a new approach to modelling and training weights with improved move prediction performance was published [22]; however, this approach was not considered in this work due to time constraints.

Weight Training Algorithm	Prediction Rate	Source
Full Bayesian Ranking	34.0%	[21]
Minorization-maximization	34.9%	[11]
Bayesian Approximation Ranking	36.2%	[29]
Minorization-maximization	37.9%	[29]
Loopy Bayesian Ranking	38.0%	[29]
Latent Factor Ranking	40.9%	[22]

Table 2.1: Comparison of move prediction performance of tactical and pattern features, with various weight training algorithms, in terms of prediction rates. Only the first two results were available at the start of this work.

2.6.1 Minorization-Maximization

As discussed above, feature level weights correspond to skill (γ) values in a GBTM, which are initially unknown, and can in principle be estimated with the aid of a collection of training data. Let the training data be $\boldsymbol{D}$, a collection of GBTM competitions and their results. Given a vector of candidate skill values $\boldsymbol{\gamma}$, the likelihood of the training data $P(\boldsymbol{D}|\boldsymbol{\gamma})$ can be calculated for the GBTM. Then, suitable γ values can be obtained from the maximal likelihood estimate (MLE) of the training data: $\arg\max_{\boldsymbol{\gamma}} P(\boldsymbol{D}|\boldsymbol{\gamma})$.

The minorization-maximization (MM) algorithm is an iterative technique for approximating the MLE, and therefore finding suitable γ values [11, 28]. This is accomplished by repeatedly finding a surrogate function that minorizes the objective function $P(\boldsymbol{D}|\boldsymbol{\gamma})$ and ascending to the maximum of the surrogate function. The use of MM has shown to have good performance for determining GBTM γ values (and therefore feature level weights) for Go features [11, 29].

2.7 Decision Trees

Decision trees are tree structures with values at leaf nodes and queries at internal nodes, such that results of queries map to children nodes [30]. An input can be evaluated by descending the tree, with the result of evaluated queries determining the descent path. The value stored at the resultant leaf is then typically used as a predicted outcome for the input, or alternatively, a decision action to be taken in the state corresponding to the input.

It has been shown that constructing an optimal[18] binary decision tree is NP-complete [31]. Decision trees are therefore typically constructed by

[18] An optimal binary decision tree minimizes the expected number of queries in a descent.

selecting queries in a greedy manner: leaf nodes are iteratively expanded by selecting queries according to a local policy (i.e. a greedy strategy approximates the problem by consecutive local optimization), such as the Iterative Dichotomiser 3 (ID3) algorithm [32], or the improved C4.5 algorithm [33]. At a high level, ID3 and C4.5 use labelled training data to select queries that have minimal entropy of the distribution of this labelled data within their children nodes (i.e. the most information gain). In this way, a fully-constructed decision tree will tend to have homogeneous leaves (all the predictions stored corresponding to a leaf node are the same).[19]

Decision trees can be particularly sensitive to queries near the root of the tree. Decision forests, also known as random forests, can be used to construct a more robust model: this approach uses multiple decision trees that are grown from subsets of the input data to create an ensemble of decision trees. Such an ensemble of decision trees has been shown to yield more accurate classification than a single decision tree in many cases [34]. In Chapter 3, decision forests will be used to improve the accuracy of decision tree features.

2.8 Conclusion

In this chapter, a number of concepts were introduced and discussed. Key concepts that are used in the following chapters include: Go features, the CFG representation, the GBTM, and decision trees.

Chapter 3 will present decision tree features, which combine the concepts of features and decision trees. Decision tree features are intended to be a more general and flexible approach (in comparison to the tactical and pattern features introduced in this chapter) to encoding domain knowledge.

[19] To prevent potential over-fitting, the expansion of leaf nodes that are relatively close to homogeneous are often avoided.

Chapter 3

Decision Tree Features

This chapter proposes an approach using decision trees to extract and represent domain knowledge. *Decision tree features* attempt to evaluate and rank state-action pairs[1] using the extracted domain knowledge. Decision trees classically represent a function in a piece-wise manner, by recursively subdividing the input space based on some criteria, and storing simple functions (often constants) at the leaves. The decision trees used in this work similarly partition the input space in a hierarchical fashion, and assign a predicted value to each element of the final partition, corresponding to the leaves of the tree. To evaluate a state-action pair, the decision tree is descended, with each query result providing additional information about the evaluated state-action pair, and splitting the input space corresponding to the query node according to the information found; the leaf at the end of the descent path stores a weight for the evaluated state-action pair.

In order to improve robustness, an ensemble of decision trees, a decision forest, is proposed.[2] Each decision tree in the forest is considered an independent feature with each leaf node corresponding to a feature level (all the feature levels of a single decision tree are mutually exclusive). We conjecture that the use of a decision forest will reduce the overall sensitivity to our proposed query selection process in the trees, especially for queries near the root of the tree, which have a large impact on the overall structure of the trees.

In the application of the proposed approach to Go, decision tree features will evaluate candidate legal moves in a position by descending the decision tree, with query results during the descent collecting relevant information about the board position surrounding the candidate move. Each node in

[1]Note that for domains with deterministic outcomes, such as Go, evaluating states or state-action pairs can be considered equivalent.

[2]In most of this chapter, the approach for a single decision tree is discussed for clarity.

these decision trees thus represents a profile of the surrounding position, with the leaves representing the most detailed profiles.

The remainder of this chapter is structured as follows: Section 3.1 presents a description of the decision tree feature approach in a domain-agnostic setting. Section 3.2 considers the application of decision tree feature to the domain of Go. Section 3.3 describes the structure of the decision trees by presenting two classes of query systems for Go and a modification to ensure mutual exclusivity between the leaves. Section 3.4 describes how to construct decision trees by specifying a policy, with a number of quality criteria, for selecting the queries for the internal tree nodes. Section 3.5 discusses considerations for applying the decision tree features approach to other domains. Section 3.6 provides a brief summary and conclusion of this chapter.

3.1 Overview

A common problem is that of selecting an action to take in a given state. A popular approach to solving such problems is reinforcement learning (RL), which essentially constructs a mapping from states to actions. However, the state space of many domains (such as Go) is too large for RL without generalization. As such, features as described in Section 2.4.1 are used to evaluate state-action pairs by assigning each state-action pair a weight based on a number of potential features. These features can be hand-crafted or extracted using automated techniques. Decision tree features are an automated approach to extracting features with limited domain knowledge. A decision tree feature consists of a decision tree, and a forest of multiple features can be used to improve robustness.

To construct these decision trees, a domain-specific representation of a state-action pair is specified, and a *query language* is used for decision tree queries, allowing the state-action pair to be evaluated. To evaluate a candidate state-action pair, the relevant decision tree is descended, with query results refining the available information about the pair. As the decision tree is descended, an ordered list of predicates[3] is incrementally constructed at each node in the descent path, with each predicate being a combination of an ancestor query and its result. This list of predicates represents a profile of the information available at the current node. A weight is stored at each decision tree leaf and used as the evaluation of state-action pairs that result in a descent to that leaf.

[3]Note that these predicates are essentially just facts, and not to be confused with predicate logic.

The term *query system* is used for the combination of the state-action pair representation and query language; the query system is the representation of the relevant domain.

3.2 Application to Go

Decision tree features are comparable with tactical and pattern features used in Go, but require less expert knowledge and could therefore be more easily applied to domains other than Go — in comparison to tactical and pattern features, decision tree features are able to encode important Go concepts that pattern features cannot, and might require extensive effort with tactical features. Due to the availability of studies of the performance of tactical and pattern features in Go, in this work the feasibility of decision tree features is investigated by applying the approach to Go for comparison purposes.

In the application of decision tree features to Go, the state is the current board position, and an action is a legal move from the board position. The board position is represented as a graph, and used to construct an appropriate state-action pair for the evaluation of a candidate move and its surrounding position. The state-action pair is constructed by combining the graph representation (state) with an *auxiliary node* representing the candidate move (action), to form an *augmented graph* (state-action pair).

It may well be that different query systems will excel at extracting domain knowledge from different areas of Go, such as the opening or complex fights. As such, this work will present six query systems, based on two variations of the board representation. Furthermore, we conjecture that a combination of multiple query systems in a decision forest might outperform a single query system due to the overall reduced sensitivity of the decision forest.

When descending a decision tree for Go, the augmented graph, representing the state-action pair, is queried. The *discovered graph*, corresponding to the resultant ordered list of predicates, represents current knowledge about a portion of the augmented graph. The discovered graph begins as just the auxiliary node at the root of the decision tree and grows in size and specificity as the tree is descended. If the discovered graph were grown to its maximum size and detail, it would result in a graph equivalent to the augmented graph. Each decision tree leaf stores a weight evaluating candidate moves that result in a tree descent terminating at that leaf.

The resultant move evaluation weights can be used to generate an ordering of the legal moves in a position. The accuracy of this move ordering can then be evaluated and compared to variations of decision tree features or other types of features. This move ordering can be used to mitigate the branching

factor of the game tree, by directing the exploration at that position's node in the tree. This branching factor mitigation can be evaluated with an empirical playing strength test (this test is much more costly than a direct evaluation of the move ordering accuracy). Features can also be used to guide the playouts in MCTS (by selecting playout moves based on the result of a GBTM competition between the moves, using the feature weights), but this is not considered in this work.

3.3 Query Systems for Go

In this work, a single query system is used by each decision tree. A *query system* is the state-action representation and query language used by a decision tree, and could be considered the representation of the domain. This section presents two classes of query systems for the application of decision tree features to Go: the intersection graph and stone graph classes. Each class of query systems is based on a graph representation of the board position, but they differ in what the nodes of the graph represent. In their simplest forms, a node in an intersection graph corresponds to a board intersection, while a node in a stone graph corresponds to a stone on the board.

Variations within each class are created by applying up to two modifications to the simple graph representations. The chain compression modification represents chains of stones (contiguous regions of black or white stones) as single nodes, and the empty compression modification (only applicable to the intersection graph class) performs a similar change to empty intersections by merging contiguous regions of empty intersections. These modifications are illustrated in Figure 3.1, introduced and discussed further in Section 3.3.1.

In both classes of query systems, the number of liberties is an attribute of black and white nodes. Instead of the true number of liberties, a variation of pseudo-liberties [16], where chains in atari have their number of pseudo-liberties set to one, is used. This is done to simplify implementation and speed up execution, since we believe it is unlikely that the use of these modified pseudo-liberties (versus normal liberties) will have a large impact. Also note that the size and number of liberties of a node, in both classes, is of the entire region on the board containing the relevant intersection(s), and not just the intersection(s) represented by the node.[4]

[4]This is only relevant when the relevant modification has not been used (otherwise there is no difference), and allows the queries used by the simpler representations access to more information.

The query language for each class of query systems was designed such that queries are invariant to rotation and reflection; this ensures that rotations and reflections of a position are evaluated as equivalent positions. The query languages were also designed to be invariant to whose turn it is to play by swapping the colours of black and white stones when a move by white is evaluated; in this way, decision trees only evaluates moves by black.

Queries consist of a query type and a number of parameters with multiple possible outcomes. The parameters and outcomes depend on the query type. In the following descriptions of the query languages, parameters are specified in one of the following forms:

- **[value]** represents a scalar integer variable,
- **[A|B|C|...]** represents a number of mutually exclusive options, and
- **[A? B? C? ...]** represents any combination of the listed options.

Section 3.4 describes how queries are selected for internal decision tree nodes.

In order to compare two scalar values, in the query languages, in all possible ways ($<$, $\leq$, $=$, $\geq$, $>$ and $\neq$) a number of relational operators are required by relevant queries. However, due to the fact that only integer values are used in these query systems and the order of the set of query outcomes is insignificant, only two relational operators are required — in this work we typically use 'less than' and 'equals to.'

As described in Section 3.1, for move evaluation an auxiliary node (representing the candidate move) is added to the graph representation of the current position to form an augmented graph, and a discovered graph is grown from the augmented graph as the decision tree is descended. The discovered graph represents the information available at the current node in the descent path.[5] At the root of the tree, the discovered graph is initialized as just the auxiliary node. At each query, either a node is added to the discovered graph from the augmented graph (along with all its edges to nodes already in the discovered graph), or information about the existing nodes or edges in the discovered graph is refined. In order to refer to nodes in the discovered graph, they are numbered incrementally, in the order they are added, with the auxiliary node being node zero.

Sections 3.3.1 and 3.3.2 present the query system classes based on the intersection graph and stone graph board representations. Section 3.3.3 then describes a modification to ensure mutual exclusivity between tree leaves in both classes of query systems.

[5]The discovered graph is only a concept for the subset of the state-action pair implicitly contained within the ordered list of predicates.

3.3.1 Intersection Graph

For the class of intersection graph (IG) query systems, we typically use nodes to represent single intersections on the board. The chain and empty modification will optionally extend this by merging nodes corresponding to stones in the same chain or empty intersections in the same region. These graph representations are referred to as $\text{IG}_\emptyset$ (the intersection graph with no modifications), IG_C (the intersection graph with the chain modification), IG_E (the intersection graph with the empty modification), and IG_{CE} (the intersection graph with both modifications). This section presents the graph structure and query language for this class of query systems.

Graph Structure

The class of intersection graph board representations ($\text{IG}_\emptyset$, IG_C, IG_E and IG_{CE}) is as follows:

- There is a node corresponding to each stone ($\text{IG}_\emptyset$ and IG_E) or chain of stones (IG_C and IG_{CE}), and each empty intersection ($\text{IG}_\emptyset$ and IG_C) or maximal contiguous region of empty intersections (IG_E and IG_{CE}).
- There is an edge between each pair of nodes that have adjacent intersections.
- Each edge has a connectivity, which is defined as the number of pairs of adjacent intersections between the edge's end nodes.
- Each node has a status (black, white or empty) and a size (number of intersections in the containing region). Black and white nodes also have a number of liberties[6] of their corresponding chain.

The above definition of the class of intersection graph representations is designed to best represent scenarios in Go that contain many adjacent regions, such as life and death scenarios. We conjecture that in these scenarios, the various regions, their adjacency, and distance to each other, are the most important details. Although not an explicit part of the graph representation, every pair of nodes has a *graph distance*, which is the number of edges in the shortest path between the two nodes. The IG_C representation also corresponds to the well-known CFG representation [26].

The node that corresponds to the empty intersection of the potential move under consideration, is then labelled as the auxiliary node to form the

[6] As noted earlier, a variation of pseudo-liberties is used.

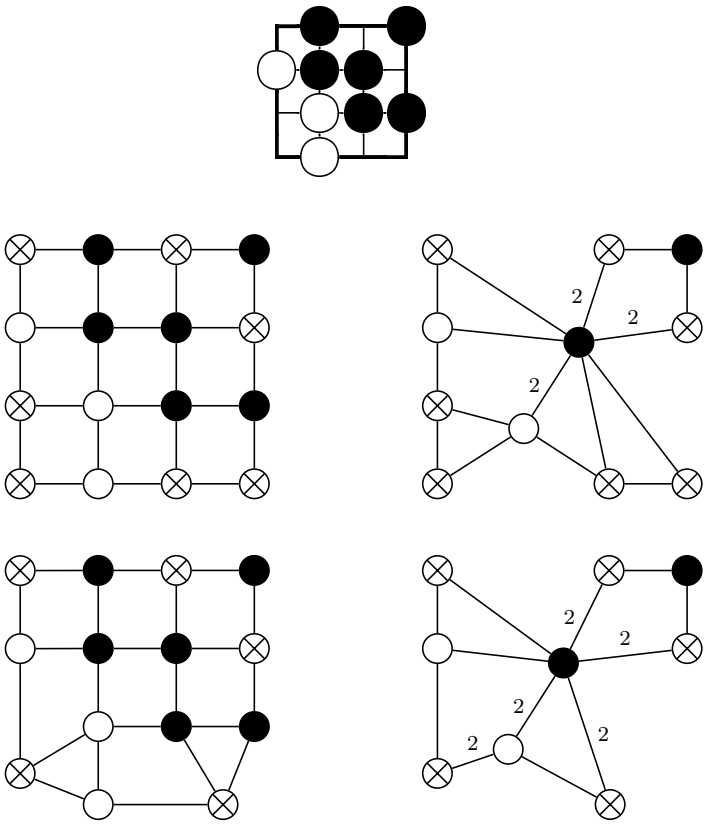


Figure 3.1: Example small Go board position with IG representations of the position below. The IG representations are $\text{IG}_\emptyset$ (top left), IG_C (top right), IG_E (bottom left) and IG_{CE} (bottom right). The connectivity of each edge is indicated next to the relevant edge, with edge weights of one omitted. Only the status of each node is indicated — other attributes are omitted.

class of augmented intersection graphs (AIG), with variations $\text{AIG}_\emptyset$, AIG_C, AIG_E and AIG_{CE}, from $\text{IG}_\emptyset$, IG_C, IG_E and IG_{CE} respectively.

Figure 3.1 shows an example small Go board position with the four IG representations of the position shown. For each edge in the representations, the connectivity of the edge is indicated.

Query Language

During a tree descent, the discovered graph is grown from the relevant AIG, beginning with the auxiliary node, using queries from the following query language consisting of three types of queries:

NEW: Is there a new **[black? white? empty?]** node adjacent (with a graph distance of one) to node **[x]**?
Look for a new node to add to the discovered graph from the AIG, and number the new node incrementally, if one is found. If multiple matching nodes are found, attempt to select a unique node according to the method described in Section 3.3.3. Separate children are added to the decision tree for each allowed status (i.e. black and/or white and/or empty), and none.

EDGE: Is the **[graph distance|connectivity]** between node **[x]** and node **[y] [less than|equal to] [val]**?
Query a single edge (connectivity) of the discovered graph, or the graph of edges (graph distance) of the AIG. Children are added to the decision tree for yes and no.

ATTR: Is the **[size|number of liberties]** of node **[x] [less than|equal to] [val]**?
Query a node of the discovered graph. Children are added to the decision tree for yes and no.

The above query language is designed to query the information that the graph representation is designed to contain: the attributes of the nearby regions, and their relation to each other.

Figure 3.2 shows a portion of an example decision tree using the $IG_\emptyset$ query system with a highlighted descent path. The leaf at the end of the descent path corresponds to a move that results in the capture (determined by the third query) of a single opposing chain (found by the first query) of undetermined size (as no query measured this). We also know that there is only a single adjacent opposing chain due to the outcome of the second query.

3.3.2 Stone Graph

For the class of stone graph (SG) query systems, we use nodes to represent a single stone on the board or a side of the board. The chain modification will optionally extend this by merging nodes corresponding to stones in the same chain. These graph representations are referred to as $SG_\emptyset$ (the stone graph with no modifications) and SG_C (the stone graph with the chain modification). In contrast with IG systems, SG systems do not explicitly represent the empty intersections on the board, and the empty modification is therefore not applicable. In addition, SG systems explicitly represent the sides, unlike

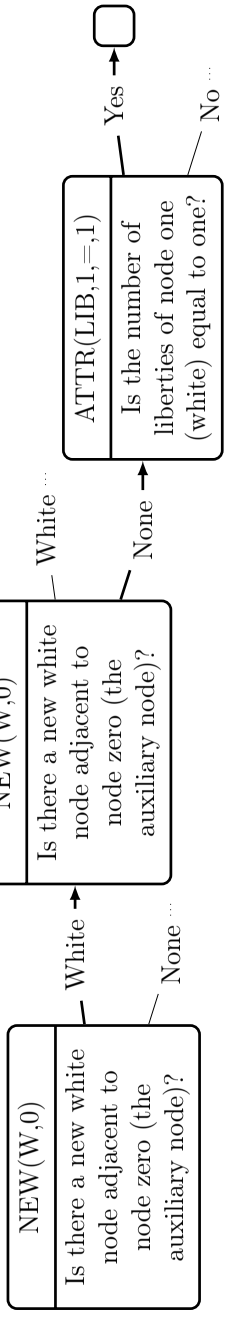


Figure 3.2: A portion of an example $IG_{\emptyset}$ decision tree showing a descent path. The leaf at the end of the descent path corresponds to a move that results in the capture of a single adjacent opposing chain of undetermined size.

IG systems. This section presents the graph structure and query language for this class of query systems.

Graph Structure

The class of stone graph board representations ($SG_\emptyset$ and SG_C) is as follows:

- There is a node corresponding to each stone ($SG_\emptyset$) or chain of stones (SG_C) on the board, and each of the four board sides.
- There is an edge between every pair of nodes, i.e. the graph is fully connected.
- The weight of each edge is the shortest Manhattan distance on the board between the stones or chains or board sides[7] represented by the edge's end nodes.
- Each node has a status (black, white or side). Black and white nodes also have a size (the number of stones) and number of liberties[8] of their corresponding chain.

The above definition of the class of stone graph representations is designed to best represent scenarios in Go that contain relatively few stones, such as the opening or middle game. We conjecture that in these scenarios, the stones and sides, and their relative positions, are the most important details, while the location of empty intersections can be indirectly inferred to some extent.

An auxiliary node, that represents the empty intersection for the potential move under consideration, is then added to each representation in the class of stone graphs to form the class of augmented stone graphs (ASG), with variations $ASG_\emptyset$ and ASG_C, from $SG_\emptyset$ and SG_C respectively. This node has edges to every other node in the ASG, and these edges have weights allocated as in the associated stone graph.

Figure 3.3 shows an example portion of a Go board position with the two SG representations of the position shown. For each edge in the representation, the weight of each edge is indicated.

[7] The distance between a stone on a side and that side is defined as zero, i.e. for purposes of measuring distance, a board side is the set of intersections on that edge.

[8] As noted earlier, a variation of pseudo-liberties is used.

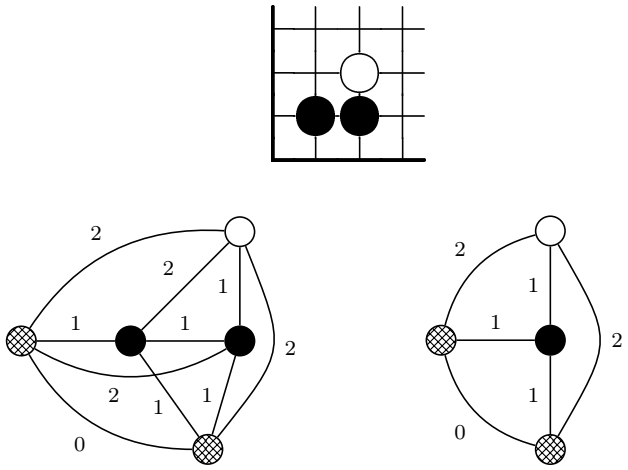


Figure 3.3: Example portion of a Go board position with SG representations of the position below. The SG representations are $SG_\emptyset$ on the left, and SG_C on the right. The weight of each edge is indicated next to the relevant edge. Only the status of each node is indicated — other attributes are omitted.

Query Language

During a tree descent, the discovered graph is grown from the relevant ASG, beginning with the auxiliary node, using queries from the following query language consisting of three types of queries:

NEW: Is there a new **[black? white? side?]** node with an edge weight to the auxiliary node less than or equal to **[distance]**?
Look for a new node to add to the discovered graph from the ASG, and number the new node incrementally, if one is found. If multiple matching nodes in the ASG are found, attempt to select a unique node according to the method described in Section 3.3.3. Separate children are added to the decision tree for each allowed status (i.e. black and/or white and/or empty), and none.

EDGE:[9] Is the edge weight between node **[x]** and node **[y] [less than|equal to] [val]**?
Query an edge of the discovered graph. Children are added to the decision tree for yes and no.

ATTR: Is the **[size|number of liberties]** of node **[x] [less than|equal to] [val]**?

[9]The keyword DIST is used in the implementation for legacy reasons.

Query a node of the discovered graph. Children are added to the decision tree for yes and no.

The above query language is designed to query the information that the graph representation is designed to contain: the attributes of the nearby stones and sides, and their relation to each other.

Figure 3.4 shows a portion of an example decision tree using the $SG_{\emptyset}$ query system with a highlighted descent path. The leaf at the end of the descent path corresponds to a move on the fourth line (which has a distance of three to the side) with no stones within a Manhattan distance of eight. Note that node zero is the auxiliary node and, in this case, node one is the closest side,[10] due to the outcome of the second query.

3.3.3 Resolving Multiple Descent Paths

In both of the query system classes, it is possible that a NEW decision tree query may not be able to identify a unique node in the augmented graph to add to the discovered graph. This is because queries are designed to be invariant to changes in rotation and reflection, and there might not be a way to differentiate between two or more nodes in the augmented graph (to add to the discovered graph) without breaking this invariance or making use of information not contained within the ordered list of predicates (explicitly or implicitly). Identifying a unique node is desirable because then the descent path does not have to branch, and the conceptual view that decision tree queries partition the input space can be maintained.

In the event that a unique node can not be identified, the decision tree feature approach considers a sequence of conditions (in addition to suitability conditions), in an attempt to maintain uniqueness. Each condition will retain those node(s) that best satisfy the condition and eliminate the others as potential matches for the query. These conditions are designed to enforce invariance to changes in rotation and reflection. The sequence of conditions used in this work is as follows:

1. Select the node(s) with the lowest edge weight to the auxiliary node (SG).
2. Select black nodes if any; otherwise select white nodes if any; otherwise select side (SG) or empty (IG) nodes.
3. Select the node(s) with the lowest edge weight (SG) or graph distance (IG) to nodes already in the discovered graph, in reverse order of discovery, i.e. decreasing node number in discovered graph.

[10] There cannot be a closer side due to the process described in Section 3.3.3.

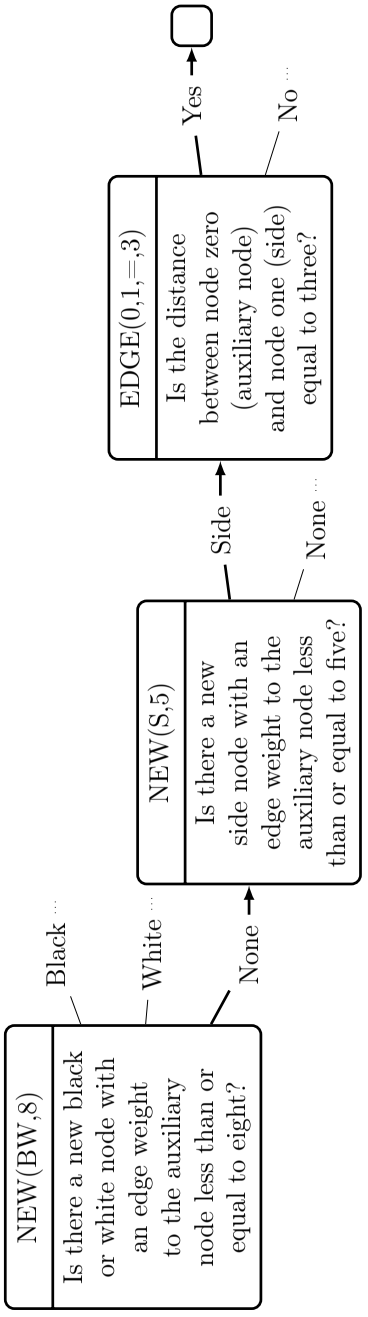


Figure 3.4: A portion of an example $SG_{\emptyset}$ decision tree showing a descent path. The leaf at the end of the descent path corresponds to a move on the fourth line with no stones within a Manhattan distance of eight.

4. Select the node(s) with highest connectivity to nodes already in the discovered graph, in reverse order of discovery (IG).

5. Select the node(s) with the most stones (SG) or intersections (IG) in its respective chain or region.

6. Select the node(s) whose respective chain has the most liberties (if nodes are not empty intersections or regions).

Note that for the last two conditions, the respective chain or region refers to the entire region on the board and not just the stone or intersection represented by the node. This is similar to the graph representations.

If the above conditions are not able to identify a unique node, then each of the possibilities are considered. This is accomplished by branching the tree descent and considering each branch in turn. For each branch, one of the remaining nodes from the augmented graph is added to the discovered graph and the tree descent continues. In this way, multiple leaf nodes will be reached.

While these conditions are not always able to find a unique node, empirical results showed that a single leaf node was reached in about 85% of tree descents for our data set (refer to Section 5.2 for details on the data set). It was therefore decided to branch and consider all the remaining nodes and their respective descent paths, but only return one of the final leaf nodes. The leaf node chosen is always the left-most node, i.e. the leaf node found first in an pre-order traversal of the tree. Preliminary investigations confirmed that this modification made negligible difference to move prediction accuracy, while providing a large reduction in training time and an increase in the size of training data set that could be handled. This modification was therefore used throughout the rest of this work without further mention. The improvement in training time is due to the leaf nodes of each decision tree becoming mutually exclusive, allowing decision trees to be treated as a single feature per decision tree as opposed to binary features for each decision tree leaf (refer to Section 4.3 for more details).

3.4 Query Selection

In order to construct decision trees for decision tree features, a method for selecting queries for decision tree nodes is required. Each potential query has two or more outcomes corresponding to children nodes, as defined by the query system in use. In traditional decision trees, the queries are typically chosen in a greedy manner due to computational issues (refer to Section 2.7).

In decision tree features, the labels for the data are not available. The weights could potentially be used, but they are only determined by training, which is a fairly lengthy process that takes place on a fully-grown tree. As such, traditional decision tree generation approaches are not suitable for this work. The remainder of this section presents an approach to selecting queries and constructing the decision trees for decision tree features, and the implementation considerations of using this approach to construct decision tree features are discussed in Chapter 4.

For decision tree features, a leaf node can be expanded by replacing it with a query and its possible outcomes, each of which is a new leaf node. When choosing a query to replace a leaf node (after enough data is captured), there is a large set of candidate queries that can be chosen from. The set of candidate queries depends on the information available in the ordered list of predicates at that point in the tree descent. In order to select a query, the *query selection policy*, as introduced in the remainder of this section, is used.

Since labelled data is not available during query selection, *descent statistics* that capture the descent properties of the tree are instead used to aid query selection. When enough data is captured, the candidate query that maximizes a certain *quality criterion* Q is chosen, provided a number of *suitability conditions* are satisfied. The relevant suitability conditions are discussed in Section 3.4.3, and depend on the quality criterion in use. The quality criteria are intended to use the descent statistics to guide tree growth. Two classes of quality criteria will be investigated in this work.

Section 3.4.1 describes the descent statistics used and how they are captured after tree descents. Section 3.4.2 presents a number of proposed quality criteria, grouped into two classes. Section 3.4.3 discusses the four suitability conditions used by these criteria.

3.4.1 Descent Statistics

Each state-action pair corresponds to a decision tree leaf (and thus a tree descent). Decision tree features can evaluate all the legal actions from a state by descending the decision tree for each action. As such, a number of positions from training data are extracted and the current decision tree is descended for each action in each position, with the descents of correct actions (those actions taken in the training data) being considered wins and the others losses. These descent statistics (number of wins and losses) are updated at the relevant leaf node: for each candidate query at the leaf node, the candidate query is evaluated and the relevant child outcome of the candidate query is updated. As discussed is Chapter 4, when enough statistics

have been captured, the descent statistics are used in the quality criterion to compare candidate queries.

For each candidate query q, the following statistics are recorded, or computed from other statistics:

- the number of wins (w_q);
- the number of losses (l_q);
- the total number of descents ($d_q = w_q + l_q$);
- the winrate ($r_q = w_q/d_q$);
- the set of children outcomes (C_q);
- the number of children ($\|C_q\|$);
- for each child c in C_q, the descent statistics w_c, l_c, d_c and r_c; and
- the subset of C_q with at least one descent ($C'_q = \{c \in C_q | d_c > 0\}$).

Note that in domains where there are n legal actions for a typical state, the winrate at the root of the decision tree will be close to $1/n$. As such, for domains where n is large (such as Go), the winrate of most nodes tends to be very low.

3.4.2 Quality Criteria

This section presents the split and separate classes of quality criteria. These quality criteria use the descent statistics defined in Section 3.4.1 to compare candidate queries for selection.

The split class of quality criteria attempts to choose queries that maximize the entropy of the distribution of the input space (i.e. the entropy of the distribution of the wins, losses or descents, depending on the quality criterion). The motivation is that in decision tree features a node represents a profile of domain knowledge; therefore, if queries are chosen to maximize the entropy of children outcomes, the leaf nodes of the decision tree should represent as much information as possible.

The separate class of quality criteria attempts to choose queries that separate the win and loss descents into homogeneous partitions, i.e. such that all outcomes have a winrate of 0 or 1. We conjecture that outcomes with a winrate close to 0 or 1 will tend to have extreme trained weights (closer to 0 or ∞), and the motivation is that that such trained weights should result in better performance. A potential issue is that queries with

high-winrate outcomes with only a few descents will tend to be favoured, and therefore very unbalanced decision trees may be constructed. Weighted variants that attempt to address this issue are therefore also proposed.

We conjecture that the separate class of quality criteria will perform better than the split class of quality criteria, due to the expected increase in evaluation accuracy from more extreme trained weights (closer to 0 or ∞).

The remainder of this section describes the various quality criteria. Certain technical details common to various criteria are discussed in Sections 3.4.3 and 4.2.

Split Class of Quality Criteria

In order to maximize the entropy of the distribution of descent statistics, the split class of quality criteria will be chosen to select queries that divide the portion of input space represented by a node into roughly equally-sized partitions. If all candidate queries only have two outcomes this is relatively straight-forward; however, this work also considers queries with potentially more than two outcomes. As such, a number of quality criteria are proposed: Naive Descent-Split, Descent-Split, Win-Split, Loss-Split, Entropy Descent-Split, Entropy Win-Split, Entropy Loss-Split and Winrate-Split.

The **Naive Descent-Split** (NDS) quality criterion is maximized in an attempt to split the descents into two equal parts by minimizing the deviation from an ideal even split. For query q, the number of descents are divided into d_n and the remainder $d_q - d_n$, where n is the number of descents to the 'none' outcome for NEW queries and the 'no' outcome for EDGE and ATTR queries. Equation 3.1 presents the selection criterion for NDS. NDS simplifies queries with more than two outcomes by dividing the query outcomes' descents into two groups in a domain-specific manner; the other criteria do not make this simplification and are therefore domain-agnostic. NDS was the first selection criterion implemented, and is included for comparison with published results [15].

$$Q_{\text{NDS}}(q) = -\left|0.5 - \frac{d_n}{d_q}\right| \tag{3.1}$$

The **Descent-Split** (DS), **Win-Split** (WS) and **Loss-Split** (LS) quality criteria are maximized in an attempt to split the descents, wins or losses into equal parts. In an ideal query q (from the perspective of the split class), each outcome would have an equal fraction ($1/\|C_q\|$) of the relevant statistic (descents, wins or losses). DS, WS and LS attempt to minimize the sum of the deviations from such ideal splits. Equations 3.2, 3.3 and 3.4 present the quality criteria for DS, WS and LS respectively. In comparison to NDS, we

believe DS is more accurately able to deal with queries that have more than two outcomes. If only queries with two outcomes are compared, NDS and DS are equivalent. We conjecture that these criteria will tend to select queries with fewer outcomes, due to the expected practical difficulty of minimizing the sum of the deviations over more outcomes. Due to the fact that the winrate is usually very low, we also conjecture that LS and DS will have very similar results.

$$Q_{\mathrm{DS}}(q) = -\sum_{c \in C_q} \left| \frac{1}{\|C_q\|} - \frac{d_c}{d_q} \right| \tag{3.2}$$

$$Q_{\mathrm{WS}}(q) = -\sum_{c \in C_q} \left| \frac{1}{\|C_q\|} - \frac{w_c}{w_q} \right| \tag{3.3}$$

$$Q_{\mathrm{LS}}(q) = -\sum_{c \in C_q} \left| \frac{1}{\|C_q\|} - \frac{l_c}{l_q} \right| \tag{3.4}$$

The **Entropy Descent-Split** (EDS), **Entropy Win-Split** (EWS) and **Entropy Loss-Split** (ELS) quality criteria are maximized in an attempt to maximize the entropy associated with the distribution of descents, wins or losses between children outcomes. For query q, the entropy of the distribution of descents is represented by $-\frac{d_c}{d_q} \log_2 \frac{d_c}{d_q}$ for outcome c (with similar forms for wins and losses).[11] Equations 3.5, 3.6 and 3.7 present the quality criteria for EDS, EWS and ELS respectively. We expect that these criteria will perform similarly to DS, WS and LS, but tend to favour queries with more outcomes as the maximum entropy of a query with k outcomes is $\log_2 k$.

$$Q_{\mathrm{EDS}}(q) = -\sum_{c \in C_q} \frac{d_c}{d_q} \log_2 \frac{d_c}{d_q} \tag{3.5}$$

$$Q_{\mathrm{EWS}}(q) = -\sum_{c \in C_q} \frac{w_c}{w_q} \log_2 \frac{w_c}{w_q} \tag{3.6}$$

$$Q_{\mathrm{ELS}}(q) = -\sum_{c \in C_q} \frac{l_c}{l_q} \log_2 \frac{l_c}{l_q} \tag{3.7}$$

The **Winrate-Split** (WRS) quality criterion is maximized in an attempt to split the descents into outcomes with equal winrates, by minimizing the difference between the winrate of each outcome and the parent's winrate. Equation 3.8 presents the quality criterion for WRS. Note that outcomes that have no descents, and therefore an undefined winrate, are ignored.

[11] Note that per convention we assume $0 \log 0 = 0$.

$$Q_{\text{WRS}}(q) = -\sum_{c \in C'_q} |r_c - r_q| \tag{3.8}$$

Separate Class of Quality Criteria

The separate class of quality criteria attempts to separate the wins and losses into homogeneous partitions. While a query that separates the wins and losses into perfectly homogeneous partitions is clearly ideal for this approach, it is much more likely that no ideal candidate query exists and it is unclear how to compare such non-ideal queries. Furthermore, it is possible that maximizing such quality criteria might result in the selection of queries that have a very small portion of the descents in one outcome (as it is practically easier for such an outcome to be homogeneous), and thus construct very unbalanced decision trees; therefore weighted variations of the criteria are also proposed. The proposed quality criteria of the separate class are: Win-Loss-Separate, Weighted Win-Loss-Separate, Symmetric-Separate, Weighted Symmetric-Separate, Winrate-Entropy and Weighted Winrate-Entropy.

The **Win-Loss-Separate** (WLS) and **Weighted Win-Loss-Separate** (WWLS) quality criteria are maximized in an attempt to separate the wins and losses into homogeneous outcomes. WLS attempts to do this by maximizing the sum of the differences, $|r_c - r_q|$, between the winrate of an outcome c and its parent query q. WWLS attempts to take the number of descents to each outcome into account by increasing the impact of outcomes with more descents. Outcomes that have no descents, and therefore an undefined winrate, are ignored. Equations 3.9 and 3.10 present the quality criteria for WLS and WWLS respectively. Note that Equation 3.9 is the negative of Equation 3.8. Also note that for WWLS, a larger d_c will more likely have a r_c closer to r_q, making this measure of deviation seem more meaningful. We conjecture that WWLS will perform better than WLS due to the weighting of outcomes.

$$Q_{\text{WLS}}(q) = \sum_{c \in C'_q} |r_c - r_q| \tag{3.9}$$

$$Q_{\text{WWLS}}(q) = \sum_{c \in C'_q} d_c |r_c - r_q| \tag{3.10}$$

The **Symmetric-Separate** (SS) and **Weighted Symmetric-Separate** (WSS) quality criteria are maximized in an attempt to separate the wins and losses into homogeneous outcomes while compensating for very low or high winrates. SS and WSS assume the winrate of each outcome should strive

towards an ideal winrate of 0 or 1 (which would indicate homogeneous outcomes). SS and WSS thus attempt to minimize the sum of the deviations, from the ideal, of the outcomes of query q, where the deviation for outcome c is the minimum of r_c/r_q and $(1 - r_c)/(1 - r_q)$. In this way, the interval from 0 to r_q and r_q to 1 are scaled such that potential deviations in opposite directions (towards 0 or 1) have a symmetrical maximum deviation. Outcomes that have no descents, and therefore an undefined winrate, are ignored. Equations 3.11 and 3.12 present the quality criteria for SS and WSS respectively. We conjecture that SS and WSS will perform better than WLS and WWLS because the winrates are usually very low in practice, and we expect the scaling of the winrate intervals (from 0 to r_q and r_q to 1) to amplify any indication of a good candidate query.

$$Q_{\text{SS}}(q) = -\sum_{c \in C'_q} \min\left\{\frac{r_c}{r_q}, \frac{1 - r_c}{1 - r_q}\right\} \tag{3.11}$$

$$Q_{\text{WSS}}(q) = -\sum_{c \in C'_q} d_c \cdot \min\left\{\frac{r_c}{r_q}, \frac{1 - r_c}{1 - r_q}\right\} \tag{3.12}$$

The **Winrate-Entropy** (WE) and **Weighted Winrate-Entropy** (WWE) quality criteria are maximized in an attempt to reduce the relative entropy[12] of the outcome winrates to an ideal query split with homogeneous outcomes. WE is designed to do this by partitioning the outcomes in C'_q into three sets: non-empty win (W) and loss (L) sets, and a (potentially empty) undetermined (U) set. The intuition is that the outcomes are either striving towards a winrate of 0 or 1, or they remain close to the parent winrate; as such, outcomes in the undetermined set are effectively ignored. The contribution of the outcomes to the relative entropy of the win (resp. loss) set is measured by comparing the winrate of each outcome c, to a desired winrate of 1 (resp. 0), resulting in $-\log_2 r_c$ (resp. $-\log_2(1 - r_c)$). Queries that tend towards homogeneous partitions are chosen by minimizing these terms over the win and loss sets. Equations 3.13 and 3.14 present the quality criteria for WE and WWE respectively. However, in both of these equations, all the terms that are being summed are negative; as such, the maximum will be found when W and L only contain a single outcome each. Equations 3.15 and 3.16 therefore present simplified forms of these quality criteria.

$$Q_{\text{WE}}(q) = \max\Big(\sum_{c \in W} \log_2 r_c + \sum_{c \in L} \log_2(1 - r_c)\Big) \tag{3.13}$$

[12]The relative entropy of distribution P in terms of distribution Q is $\sum_i P(i) \ln \frac{P(i)}{Q(i)}$.

$$Q_{\mathrm{WWE}}(q) = \max\Big(\sum_{c\in W} d_c \log_2 r_c + \sum_{c\in L} d_c \log_2(1-r_c)\Big) \tag{3.14}$$

$$Q_{\mathrm{WE}}(q) = \max_{c\in C'_q} \log_2 r_c + \max_{c\in C'_q} \log_2(1-r_c) \tag{3.15}$$

$$Q_{\mathrm{WWE}}(q) = \max_{c\in C'_q} d_c \log_2 r_c + \max_{c\in C'_q} d_c \log_2(1-r_c) \tag{3.16}$$

3.4.3 Suitability Conditions

In the two classes of query criteria, each quality criterion can have a number of associated suitability conditions. Four suitability conditions, S_1, S_2, S_3 and S_4, are presented here. All applicable suitability conditions of a quality criterion must be satisfied by a candidate query for the selection of that query.

S_1 specifies that the descents must go to at least two child outcomes, i.e. $d_c < d_q \ \forall\ c \in C_q$. This condition avoids the selection of queries that add no information (according to the descent statistics seen so far), since any potential new information is already contained within the ordered list of predicates. As such, S_1 is used by all quality criteria.

S_2 requires at least one win and one loss descent for a query, i.e. $w_q, l_q > 0$. S_2 prevents the separate class of criteria from adding more queries to separate an homogeneous partition. S_2 is thus used by all the separate class of quality criteria and also in WRS. Its use in WRS is to avoid adding queries when the winrate is either 0 or 1, and there can therefore be no deviation in the outcomes' winrates.

S_3 and S_4 are closely related to S_1, but adapted to the criteria that are only concerned with wins (S_3) or losses (S_4). As such, S_3 (resp. S_4) states that all the wins (resp. losses) must go to at least two child outcomes, i.e. $w_c < w_q \ \forall\ c \in C_q$ (resp. $l_c < l_q \ \forall\ c \in C_q$). S_3 is used by DS and EDS, while S_4 is used by LS and ELS.

Table 3.1 summarizes the various quality criteria and associated suitability conditions. For each quality criterion, all the associated suitability conditions that must be satisfied are indicated.

3.5 Other Domains

In order to apply decision tree features to a new domain, a domain-specific query system is required. The remainder of the decision tree feature ap-

Name	S_1	S_2	S_3	S_4	$Q(q)$
Naive-Descent-Split	X				$-\left\lvert 0.5 - \frac{d_n}{d_q} \right\rvert$
Descent-Split	X				$-\sum_{c \in C_q} \left\lvert \frac{1}{\lVert C_q \rVert} - \frac{d_c}{d_q} \right\rvert$
Win-Split	X		X		$-\sum_{c \in C_q} \left\lvert \frac{1}{\lVert C_q \rVert} - \frac{w_c}{w_q} \right\rvert$
Loss-Split	X			X	$-\sum_{c \in C_q} \left\lvert \frac{1}{\lVert C_q \rVert} - \frac{l_c}{l_q} \right\rvert$
Entropy-Descent-Split	X				$-\sum_{c \in C_q} \frac{d_c}{d_q} \log_2 \frac{d_c}{d_q}$
Entropy-Win-Split	X		X		$-\sum_{c \in C_q} \frac{w_c}{w_q} \log_2 \frac{w_c}{w_q}$
Entropy-Loss-Split	X			X	$-\sum_{c \in C_q} \frac{l_c}{l_q} \log_2 \frac{l_c}{l_q}$
Winrate-Split	X	X			$-\sum_{c \in C'_q} \lvert r_c - r_q \rvert$
Win-Loss-Separate	X	X			$\sum_{c \in C'_q} \lvert r_c - r_q \rvert$
Weighted Win-Loss-Separate	X	X			$\sum_{c \in C'_q} d_c \lvert r_c - r_q \rvert$
Symmetric-Separate	X	X			$-\sum_{c \in C'_q} \min \left\{ \frac{r_c}{r_q}, \frac{1-r_c}{1-r_q} \right\}$
Weighted-Symmetric-Separate	X	X			$-\sum_{c \in C'_q} d_c \cdot \min \left\{ \frac{r_c}{r_q}, \frac{1-r_c}{1-r_q} \right\}$
Winrate-Entropy	X	X			$\max_{c \in C'_q} \log_2 r_c + \max_{c \in C'_q} \log_2 (1 - r_c)$
Weighted-Winrate-Entropy	X	X			$\max_{c \in C'_q} d_c \log_2 r_c + \max_{c \in C'_q} d_c \log_2 (1 - r_c)$

Table 3.1: Summary of the quality criteria, with relevant suitability conditions indicated.

proach, including query selection[13] and weight training, is domain-agnostic. As such, decision tree features can be transferred to any domain where a query system can be constructed. Alternatives to decision tree features include hand-crafted features (like tactical features) and domain-specific features (such as pattern features), and these alternatives can be combined with decision tree features; however, these alternatives require more expert knowledge than decision tree features. We conjecture that a practical query system for decision tree features is realizable for a large number of domains with very little expert knowledge, especially game-related AI, and that decision tree features will be a feasible alternative in a number of these domains, in terms of performance.

There are a number of considerations for constructing a query system for a new domain: these include the ease of implementation, a method for efficiently obtaining and storing descent statistics for the candidate queries, and the required computational resources (both memory and processing power). For domains where RL has been considered, it might be feasible to combine the state representation used for RL with a query language. The ideal expressiveness of a query language is when the query system is able to (given enough computational resources) identify complete state-action pairs unambiguously.

Furthermore, training data for a new domain might be abundant (such as high-level games in a popular game-related domain such as computer Go). However, for domains where this is not the case, training data can potentially be generated with the use of a stochastic procedure that can generate a large collection of data quickly, or a procedure with very high computational requirements (too high to be feasible in a real-time scenario) that can generate training data of a high quality. Depending on the domain, it might also be possible to iteratively incorporate the trained features in one of the above procedures to generate better features overall (i.e. bootstrapping). As such, decision tree features can be applied to domains where training data is available, and training data can potentially be generated for other domains.

3.6 Conclusion

This chapter presented decision tree features, a new approach to extracting domain knowledge. Decision tree features make use of decision trees with a domain-specific query system to define the structure of the trees. A number of query systems for the game of Go are presented. For tree construction, a query selection policy, with a number of proposed quality criteria, was

[13] All of the proposed quality criteria are domain-agnostic, with the exception of NDS.

presented. Finally, applying decision tree features to other domains was discussed.

The next chapter will present the considerations of a practical implementation of decision tree features, as introduced in this chapter, to computer Go.

Chapter 4

System Implementation

Chapter 3 introduced the proposed decision tree features, and discussed their application to computer Go. This chapter presents the major implementation considerations and system pipeline for our implementation of decision tree features in Oakfoam, a computer Go engine.

Figure 4.1 illustrates the relationships between the different components required for constructing, using and testing decision tree features. The tree growth, weight training, and action evaluation components, together with their parameters, form the pipeline for a *feature instance*. In this pipeline, a feature instance is created by growing a decision forest, training weights for the tree leaves (together with any other features), and the use of the decision tree features (together with any other features) to evaluate actions. The output of some components can be reused in later pipeline executions to limit the difference between feature instances. After executing the pipeline, the feature instance, dependent on its settings and the training data, is the resulting set of features (possibly including decision tree, tactical and pattern features), with trained weights, that can be used for testing or engine usage. Chapter 5 will measure the impact of the various input parameters, and compare the feasibility of feature instances employing decision tree features to a state-of-the-art feature instance (consisting of tactical and pattern features).

Section 4.1 describes the selection of the training and testing data used by the system. Section 4.2 presents the process used to grow the decision forest, while Section 4.3 outlines the weight training process for the resulting forest and any other features. Section 4.4 explains how moves are evaluated using these features. Section 4.5 outlines how the testing data can be used to evaluate feature instance, and Section 4.6 describes the integration of the feature instances into Oakfoam.

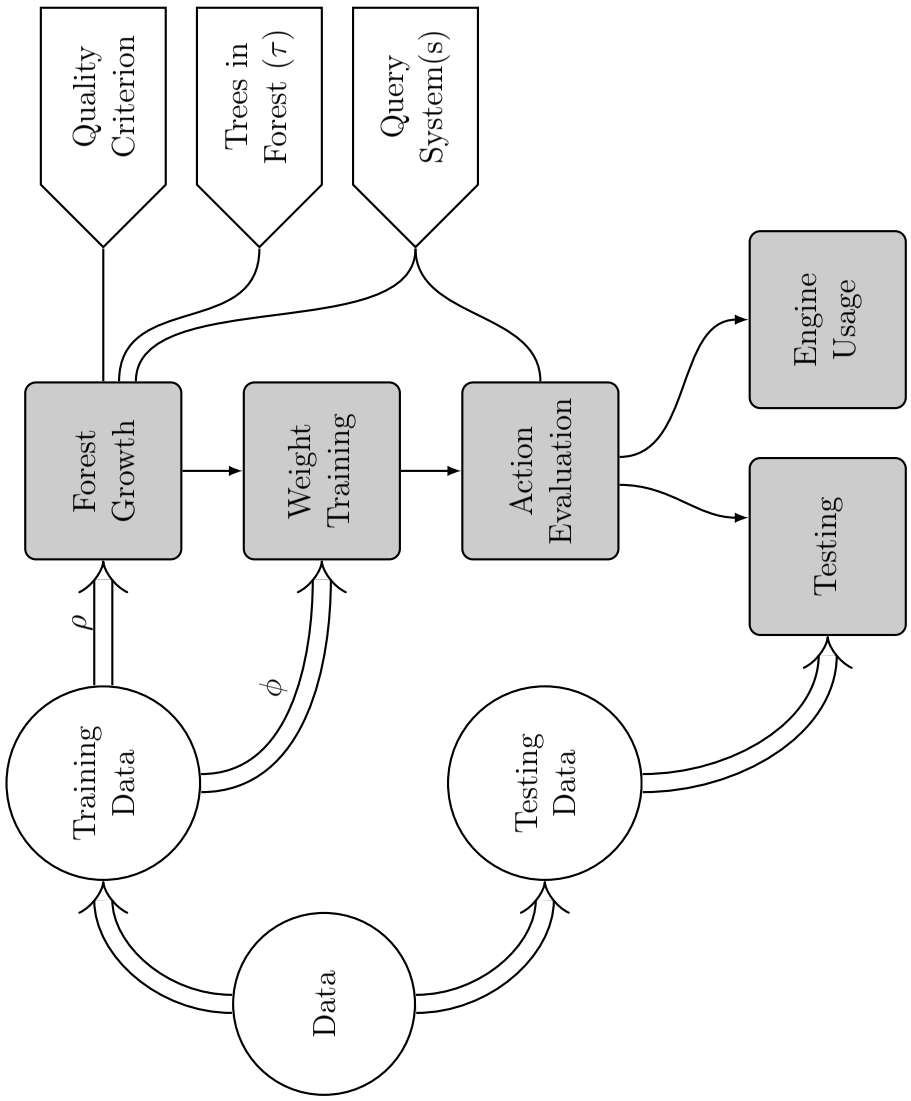


Figure 4.1: Diagram of components for decision tree feature construction, testing and usage.

4.1 Training and Testing Data

For training and testing, a collection of GBTM competitions is required. For Go, these competitions each consist of a board position and a resulting move. For this work, these competitions will be harvested from a collection of games. As such, the initial data set used for both training and testing is a large collection of games. These games should preferably be of a high level to improve the performance of the MCTS engine integration.

While there is a surplus of high-level training data readily available for Go, in other domains this might not be the case. In such situations, self-play might be able to generate useful training data, by instructing a stochastic engine to play a series of games against itself and focusing on the moves made by the winners of the games.[1]

Before a feature instance is created and tested, the data set is divided into training and testing data sets by randomly selecting games for each set.[2] In this way, the amount of training and testing data can be adjusted as needed. A typical division of data is: 50000 games for training (not all of which is actually used), and 1000 games for testing.

4.2 Forest Growth

As described from a domain-agnostic perspective in Section 3.4, decision tree features are constructed by growing a decision tree for each feature. As such, a decision forest can be constructed from multiple decision tree features. This decision forest is initialized as a root node per tree and grow as training data points are processed. In this section, the selection of training points will first be described, followed by the decision forest growing process (which involves descending the trees using the selected training points).

In this work, the parameter τ dictates the size of the decision forest, i.e. τ trees are grown at the same time. If multiple query systems are used in the decision forest, then the total number of trees is τ, and each query system is used by an equal portion of forest (in this work τ is always chosen so that this is possible).

The collection of GBTM competitions used to grow the forest are sampled from the training data. This is accomplished by selecting a number of games

[1] This process can be repeated after constructing a feature instance and incorporating it into the next repetition of the process (i.e. bootstrapping).

[2] Note that it is more usual to separating the data for training and testing once, before any testing has taken place. However, this was not done in this work to reduce the chance of over fitting the testing data. Testing in Chapter 5 also confirmed that the extra variance introduced by doing this is not of concern.

(ρ) from the training data in a random order, and then, for each position in each game, selecting the competition corresponding to that position (with the subsequent move as the competition's winning team) with a 10% probability for each tree in the forest. Due to limited computational resources, only a subset of the training data can be used. As such, the 10% probability is independently sampled for each tree in the forest, and ensures that a diverse set of competitions are sampled from the training data. This approach of processing games sequentially was chosen to simplify the tree growth: because the games can be read in sequentially, competitions do not need to be stored in a large database.[3]

For each team (and its corresponding move) in each competition chosen from the training data, the current decision tree (i.e. the tree that triggered the 10% probability, with this step repeated if multiple trees trigger) is descended and the descent statistics of the resultant leaf node are updated as described in Section 3.4.1. When a threshold number of descents through a leaf node have been recorded, that node is expanded by selecting a new query, according to the query selection policy. For the tests in this work, the threshold number of descents was fixed at 1000. When no suitable query can be found due to the suitability conditions, expansion is delayed for 10 descents.

In the process of tree growth, τ is the parameter that controls the number of trees in the decision forest, and ρ is the parameter that controls the amount of data used for tree growth, and therefore the approximate size of the trees. Note that τ and ρ together control the number of leaves in the entire decision forest, as each of the τ trees is grown with GBTM competitions sample from ρ games. The quality criterion is the parameter that controls the query selection, and therefore the shape[4] of the resulting decision tree.

4.3 Weight Training

In this work, tactical features are present in every feature instance (with the exception of the tests in Section 5.10). If pattern features are present, then the patterns of intersections within circular distance of 3 to 15 of a candidate move that occur at least 20 times in a collection of ξ games sampled from the

[3]In practice, positions would be stored instead of GBTM competitions, as generating each competition is somewhat costly. However, this is further complicated by the fact that some tactical features also depend on previous moves in the game, and not just the current position.

[4]While the split criteria should result in a balanced tree shape, the separate criteria might result in an unbalanced tree.

training data set, are used. ξ is the parameter used to control the number of patterns used in a feature instance.

Once a decision forest has been grown and/or patterns harvested, weights for the leaf nodes (which correspond to feature levels), and/or weights for all of the patterns must be determined. These weights are trained together with any other feature weights (such as other decision tree, tactical or pattern features). A collection of GBTM competitions are selected from the training data, by selecting a number of games (ϕ) randomly, and then selecting every position's corresponding competition with a 10% probability. Similarly to forest growth, computational resources are limited and therefore only a subset of the training data can be used; the 10% probability ensures that a diverse set of training data is sampled.

The sampled GBTM competitions are then used as input for the MM algorithm, with the output being suitable feature level weights, as described in Section 2.6.1. MM was chosen for weight training because it has been shown to have good performance, there are comparable results from previous work, and there is a freely available tool[5] [11]. These factors permit easier verification of the implementation and better comparison with tactical and pattern features for Go, due to fewer changes to the experimental setup [11, 29].

In this process, ϕ is the parameter that controls the amount of data used for weight training. While an increase of ϕ (resulting in more training data) will presumably result in more accurate weights, this increase will also result in the weight training process requiring more computational resources of time and memory. In this work, a lower ϕ was frequently chosen to reduce these computational resource requirements.

4.4 Action Evaluation

Once weights have been trained for the feature instance, the feature instance can be used to evaluate and rank moves (actions). A candidate move is evaluated with the decision tree features by descending each decision tree in the forest, with the descent path determined by the query results w.r.t. the candidate move. When a leaf node is reached, the weight stored at the

[5]The freely available tool was optimized by Detlef Schmicker, a contributor on the Oakfoam engine. These optimizations, available in the Oakfoam codebase, enabled the use of larger training data sets — with the optimizations, more training data can be handled with the same amount of memory, and the upper limit on memory (present in the original tool) has effectively been removed; the limit is now the available memory of the computer used.

leaf node is used as the evaluation of the candidate move by the relevant decision tree. A final evaluation of the candidate move can then be formed by combining this weight with the feature level weights of the other features in the feature instance (such as other decision tree, tactical or pattern features) by taking their product, as described in Section 2.6.

At this point, the feature instance and its action evaluation can be used either in testing or engine usage.

4.5 Testing

In order to measure the move prediction performance of a feature instance, it is used to evaluate all the positions in the testing data set. By evaluating all the legal moves in a position, an ordering of the moves can be formed. This move ordering can then be used to test the accuracy of the feature instance, by measuring the rank of the correct move, according to the testing data. For each position, the rank of the correct move in the ordering, and the observation probability according to the GBTM, are measured. In this way, each position in the testing data is used to evaluate the move prediction accuracy of the feature instance.

4.6 Engine Usage

To use a feature instance (including decision tree features) in an MCTS engine, they are used to evaluate moves, as described in Section 4.4, and incorporated into the selection or simulation policies of MCTS, as described in Section 2.4. In this work, they are integrated into Oakfoam, an open source MCTS-based computer Go engine. Oakfoam has a number of relevant attributes:

- MCTS engine with many adjustable parameters (all were left as default unless specified otherwise).
- Tactical, pattern and decision tree feature instances can be constructed and used (after implementing the pipeline described in this chapter).[6]
- In the MCTS selection policy, progressive widening is used to incorporate domain knowledge. The selection policy also includes other

[6]During this work, the use of features in Oakfoam was improved in a number of ways: the implementation of decision tree features (and all the associated components), larger pattern features, and a streamlined implementation of the pipeline described in this chapter.

enhancements (such as RAVE and LGRF) over the basic MCTS described in Chapter 2. Refer to Appendix A and the source code for more details on the exact selection policy used.

- The simulation policy for the playouts uses a number of hand-crafted playout heuristics (based on similar policies in successful engines such as MOGO [23] and PACHI [20]).
- The opening book was disabled for this work, in order to not limit testing of moves near the beginning of a game.
- Parallelization was disabled for this work, to minimize any effects introduced by multi-core or cluster parallelization.

In this work, features are integrated into OAKFOAM with progressive widening in the selection policy. In this way, at each node during the MCTS selection phase, only a subset of the children moves are considered, where the considered subset is determined by evaluating moves with features.

4.7 Conclusion

This chapter described major implementation considerations of a pipeline for constructing and using feature instances with decision tree features for computer Go. This pipeline can be summarized as follows:

- Divide the data into training and testing data sets.
- Grow the decision forest of τ trees, using the query selection policy and parameter ρ (the number of games used).
- If applicable, harvest patterns with parameter ξ (the number of games used).
- Train weights for the forest leaf nodes (and any other features), using MM and parameter ϕ (the number of games used).
- Use the decision forest (and any other features) to evaluate moves.
- Use the feature instance evaluations to test the accuracy of the generated features, or use the evaluations and progressive widening to mitigate the branching factor in MCTS.

The next chapter will evaluate the performance of decision tree features, and the impact of the feature instance settings, using the implementation described in this chapter.

Chapter 5

Experiments and Results

This chapter evaluates the performance of decision tree features for computer Go, and compares them to state-of-the-art tactical and pattern features, using the system described in Chapter 4. The goals of this chapter are as follows:

- Reproduce a state-of-the-art feature instance using tactical and pattern features.
- Evaluate decision tree features by:
  - measuring the impact of the relevant settings on move prediction;
  - identifying the settings with good performance in terms of move prediction and playing strength; and
  - evaluating the feasibility of decision tree features, by comparing results to the state of the art.
- Evaluate a combination of state-of-the-art and decision tree features, to determine whether adding decision tree features will improve the state of the art.

The remainder of this chapter is structured as follows: Section 5.1 introduces the testing methodology used in this chapter. Section 5.2 describes the training and testing data, Section 5.3 presents the list of tactical features used, and Section 5.4 presents examples of decision tree features. Sections 5.5–5.10 outline and evaluate the move prediction performance of various feature instances. Finally, Section 5.11 evaluates a select few feature instances in terms of playing strength.

5.1 Testing Methodology

This section introduces the testing methodology used in the testing of this work, as presented in this chapter.

In this chapter, the term *test* is used to refer to the evaluation of one or more feature instances in terms of move prediction or playing strength. Each feature instance is generated by following the pipeline described in Chapter 4, and selecting either testing or engine usage in the final step, for a move prediction or playing strength test respectively.

Decision tree features were conceived as a way to improve upon the pattern features of state-of-the-art feature instances. Additionally, while decision tree features are history-agnostic (independent of previous moves and their order), tactical features are not. As such, the majority of feature instances with decision tree features will also contain tactical features. Section 5.10 will briefly explore feature instances with just decision tree features by modifying tactical features to be history-agnostic.

There are a number of relevant settings that potentially impact the performance of feature instances. These settings are summarized as follows (only a subset of them are relevant for most feature instances):

- The number of games used to harvest patterns for pattern features (ξ).
- The query system(s) that describes the decision tree/forest.
- The quality criterion used for query selection when growing the tree/forest.
- The number of trees in the decision forest (τ).
- The number of games used to grow the trees in the forest (ρ).
- The number of games used to train the weights for all the features (ϕ).

The relationship between the effects of the above settings is unclear, and this work does not make an attempt to construct a theoretical model of their interaction. Furthermore, while the time it takes to evaluate a feature instance can vary, it is consistently long enough to prohibit a comprehensive search of the multi-dimensional space which corresponds to the interaction of all the settings. However, we do have some expectations regarding the effects of the settings. As such, the settings will be tested systematically, in such a way that their effects can hopefully be isolated to some extent, and our expectations can be validated.

In this work, move prediction is measured by generating move orderings for the positions in the testing data set and recording the rank of the correct

move for each position, according to the testing data. Move prediction tests are described in more detail in Section 5.5.

Testing a feature instance in terms of playing strength is achieved by integrating the feature instance into an MCTS engine, and measuring the performance of the modified engine by playing a series of games against a reference opponent. In this work, progressive widening is used when integrating feature instances into the MCTS engine, Oakfoam. Playing strength tests are further described in Section 5.11.

A playing strength test is the most direct way of measuring the performance of a feature instance for computer Go. However, in comparison with a move prediction test, a playing strength test takes much longer to give an accurate evaluation. Furthermore, early investigations indicated that move prediction performance is a good indicator of playing strength. As such, move prediction tests will predominantly be used to evaluate feature instances, and the results will be verified by testing the playing strength of a limited number of feature instances.

The machine used to perform the move prediction tests in this chapter has a four-core Intel Core i5-3570K (3.4 GHz) and 16 GB of RAM. The playing strength tests were performed with a single core per game, on a cluster of machines, each with an eight-core Intel Xeon E5440 (2.83 GHz) and 16 GB of RAM.

As described in Section 4.3, the MM algorithm is used to train the feature level weights. In order to execute the algorithm in a reasonable time, all the GBTM competitions are loaded into memory so that they can be iterated over; ϕ is therefore limited by the available memory of the testing machine. However, to further complicate the issue, increasing τ increases the memory required for each GBTM competition (because the number of individuals in each team increases) and therefore potentially further limits the feasible values for ϕ. Even though the testing machine has 16 GB of RAM, ϕ was still limited by the available memory, especially when testing larger forests. This interaction between ϕ, τ, and the available memory plays a role in a number of the tests and the choice of setting values for feature instances.

5.2 Training and Testing Data

The data set used in the training and testing in this chapter is a collection of high-level 19x19 games. The games are from an online archive of games played on the KGS[1] Go server from 2001 to 2009 [35]. All the games that have handicap stones were filtered out, to ensure the initial game state for

[1]KGS is a recursive backronym for KGS Go Server.

all games in the data set is an empty board.[2] After filtering, the data set is divided into training and testing data sets before the construction of each feature instance, as described in Section 4.1. In this work, the testing data set always consisted of 1000 games.

5.3 Tactical Features

Table 5.1 lists the tactical features, and their various levels, used in this work. The table also includes example weights for two feature instances: one using tactical features with $\phi = 16000$, and another using tactical and decision tree features with $SG_\emptyset$, WWLS, $\tau = 16$, $\rho = 800$ and $\phi = 16000$.

There are some noticeable differences in the γ values (weights) between the two feature instances for some of the tactical features. In these differences, the weights for the feature instance with decision tree features are much closer to 1, possibly indicating that these tactical features have been incorporated into the decision trees to some extent.

Furthermore, for both feature instances, the last four tactical features (those that make use of the distance to recent moves) have a number of levels with reasonably high γ values. This seems to indicate that the distances to recent moves are important features, and is further explored in Section 5.10.

5.4 Example Decision Tree Features

This section presents two example feature levels from a forest of decision tree features. The forest is part of a feature instance from Section 5.9.2 with tactical and decision tree features, with setting values $SG_\emptyset + IG_\emptyset$, WWLS, $\tau = 16$, $\rho = 3200$ and $\phi = 16000$. The two selected feature levels were chosen from the highest and lowest trained weights in the decision forest, and both are from trees with the $SG_\emptyset$ query system.

Each feature level corresponds to a descent path down one of the trees in the forest, and only these descent paths are presented in this section (not the rest of the relevant decision trees, or the outcomes not found in these paths).

Figure 5.1 presents the first example feature level. This feature level has a weight of 117.8, which is the highest trained weight in the decision forest. This feature level represents a move on the 4-4 point in an empty corner, with an opposing stone in the far corner of the board. This corresponds with our intuition: a move near an empty corner is a good opening move. Furthermore, the move is most likely the second move of the game and in the

[2]The *komi* of the filtered games is not restricted [1].

Feature	Level	γ^T	γ^{DT}	Description
Pass	1	2.75	2.03	Pass after a normal move
	2	264.67	193.54	Pass after another pass
Capture	1	1.81	0.29	Capture a chain
	2	1.71	2.59	Capture a chain in a ladder
	3	17.19	0.86	Capture, preventing an extension
	4	34.12	1.05	Re-capture the previous move
	5	86.65	6.44	Capture a chain adjacent to a chain in atari
	6	36.66	4.49	Capture a chain as above, of 10 or more stones
Extension	1	13.90	1.05	Extend a chain in atari
	2	1.56	0.30	Extend a chain in a ladder
Self-atari	1	0.30	0.76	Self-atari of 5 or fewer stones
	2	0.12	0.10	Self-atari of more than 5 stones
Atari	1	3.82	0.84	Atari a chain
	2	1.63	0.36	Atari a chain and there is a ko
	3	4.21	1.08	Atari a chain in a ladder
Distance to border	1	0.44	1.07	
	2	1.07	1.04	
	3	1.63	1.07	
	4	1.22	1.02	
Circular distance (δ°)	2	13.21	10.41	
to previous move	3	6.67	6.62	
	4	3.80	3.51	
	...	...	...	
	10	1.32	1.57	
Circular distance (δ°)	2	1.67	1.55	
to the move preceding	3	1.53	1.74	
the previous move	4	1.02	1.21	
	...	...	...	
	10	0.95	1.15	
CFG distance	1	3.50	2.69	
to previous move	2	3.90	2.97	
	3	4.02	3.09	
	4	1.97	2.01	
	...	...	...	
	10	0.80	0.98	
CFG distance	1	6.37	2.81	
to the move preceding	2	4.55	2.29	
the previous move	3	3.54	1.84	
	4	2.69	1.58	
	...	...	...	
	10	1.31	1.08	

Table 5.1: List of tactical features, with example weights from two feature instances: γ^T for tactical features with $\phi = 16000$ and γ^{DT} for tactical and decision tree features with $\text{SG}_\emptyset$, WWLS, $\tau = 16$, $\rho = 800$ and $\phi = 16000$. The weights for level zero of each feature (where none of the other levels apply) are fixed at 1. When multiple feature levels are applicable, the highest level is selected.

opposite corner to the first move (considering the data set used), reducing the opponent's options and matching typical high-level play.

Figure 5.2 presents the second example feature level. This feature level has a weight of 0.0194, which is one of the lowest trained weights in the decision forest. This feature level represents a move on the third line, with three or more opposing stones in closer proximity than any supporting stones.[3] This also corresponds with our intuition: a move in an area of the board with many opposing stones in close proximity is usually a bad move.

5.5 Move Prediction Outline

This section describes the evaluation of the move prediction of various feature instances. The move prediction of a feature instance is evaluated by performing the following process:

- A feature instance with the desired settings is generated, and a testing data set is formed by randomly selecting 1000 games from the data set, as described in Chapter 4.
- For each position of each game in the testing data set:
  - All the legal moves are evaluated using the feature instance, resulting in weights for the moves.
  - The move weights are used to construct a move ordering, and the rank of the move actually made in the testing data is determined.
  - The move weights are also used to construct a GBTM competition and the observation probability of the correct move winning is calculated.
- The move ranks are then used to construct a cumulative distribution M over the testing data, where $M(x)$ is the probability that the correct move has rank x or less.
- The observation probabilities are used to determine the mean log-evidence (MLE) of the testing data: the log of the likelihood of the testing data observed, divided by the number of observations.

The above process allows feature instances to be compared in terms of the measured M distribution and/or the MLE. The ideal situation is when

[3]This description of the second example feature level omits some details for simplicity.

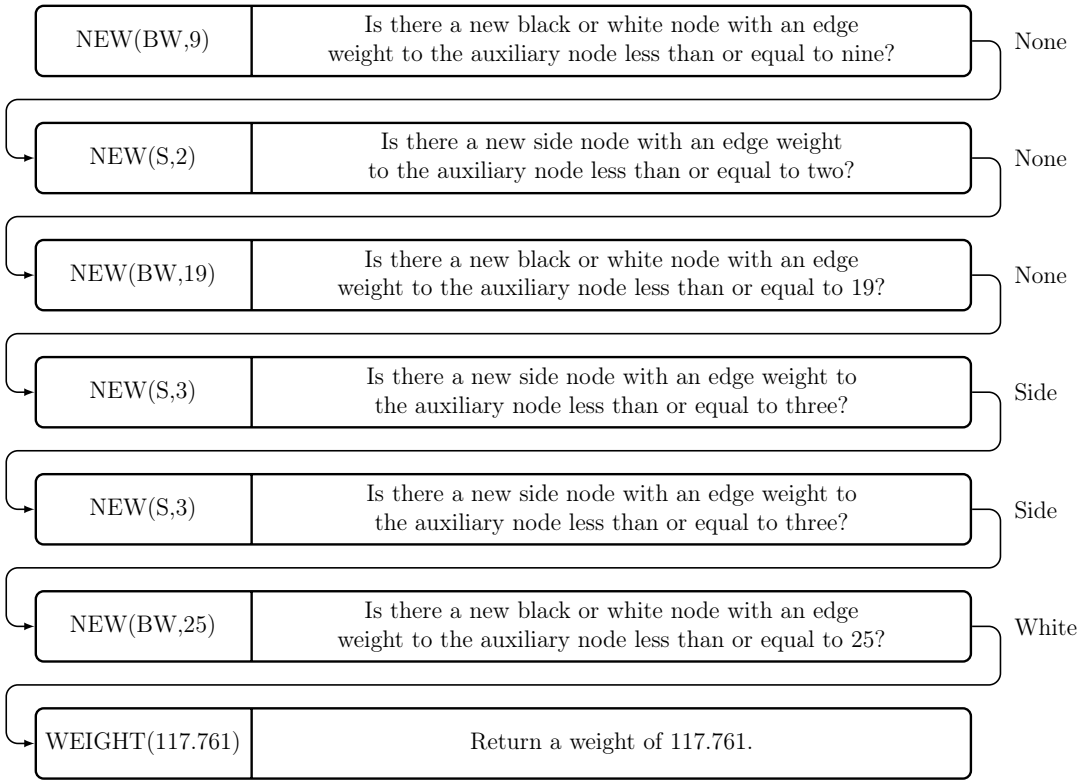


Figure 5.1: First example descent path from an $SG_{\emptyset}$ decision tree. The tree is from a feature instance with $SG_{\emptyset}$ + $IG_{\emptyset}$, WWLS, $\tau = 16$, $\rho = 3200$ and $\phi = 16000$.

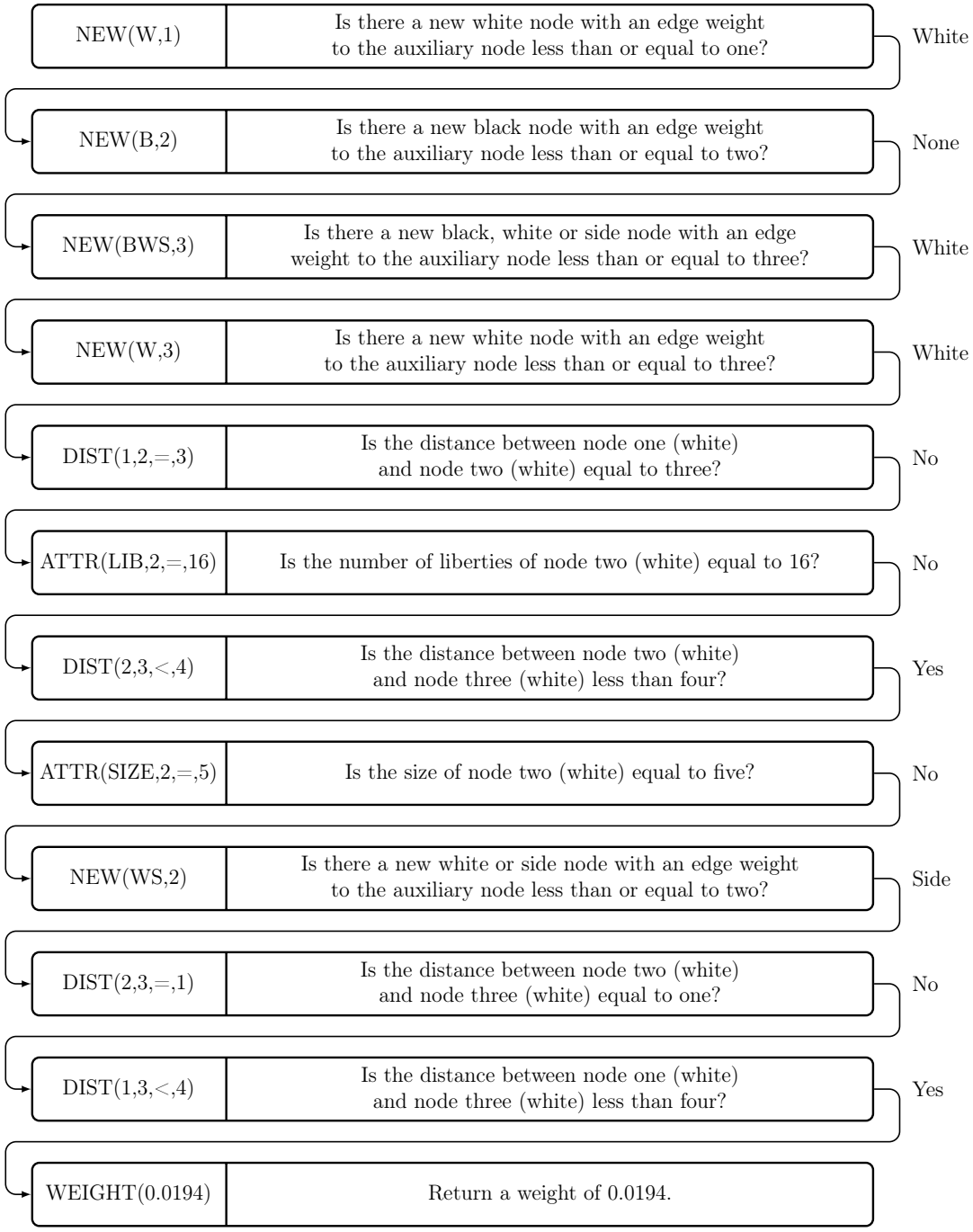


Figure 5.2: Second example descent path from an $SG_{\emptyset}$ decision tree. The tree is from a feature instance with $SG_{\emptyset}$ + $IG_{\emptyset}$, WWLS, $\tau = 16$, $\rho = 3200$ and $\phi = 16000$.

$M(1) = 1$ and MLE $= 0$; for each game position, this would correspond to the correct move always having the largest move weight and the chance of the correct move winning in a GBTM competition being 100%.

As progressive widening will be used to integrate some of these feature instances into an MCTS engine, it indicates that comparing the M distributions will be the most appropriate.[4] However, it is unclear how to concisely compare two different cumulative distributions. Comparing the area under the curve (AUC) is a possible method; however, due to the use of progressive widening, only the first few x values of $M(x)$ will be relevant in practice. Furthermore, preliminary tests (shown in Section 5.6.2) indicated that for feature instances with the same type of features, the shape of their cumulative distribution curves are very similar, and curves with higher $M(1)$ values stay consistently above those with lower $M(1)$ values; $M(1)$ therefore already seems to be a good metric for comparison. As such, tests in the following sections will primarily compare feature instances based on their values for $M(1)$.

In order to objectively compare the $M(1)$ values of different feature instances, some indication of the variance of these values is required. We first computed the maximum possible width of a 95% confidence interval in terms of the testing data. This was done by using a normal approximation to the binomial distribution corresponding to an $M(1)$ value. Assuming that there are at least 100 moves (typical of 19x19 games) in each of the 1000 games in the testing data set, and $M(1) = 0.5$ (resulting in the maximum interval width), the maximum possible width of a 95% confidence interval for an $M(1)$ value w.r.t. the testing data set is:

$$2z\sqrt{\frac{p(1-p)}{n}} = 2 \cdot 1.96\sqrt{\frac{0.5 \cdot 0.5}{100000}} = 0.00620. \tag{5.1}$$

Next, the variance of $M(1)$ in terms of the entire process (including feature instance construction and testing) was empirically measured, with setting values that could be run fairly rapidly. 50 feature instances with settings $\text{SG}_\emptyset$, WWLS, $\tau = 8$, $\rho = 500$ and $\phi = 4000$ were constructed and tested; the selection of training and testing data sets for each feature instance was independent. The standard deviation of the $M(1)$ values of these feature instances was determined to be $\sigma = 0.00173$, which corresponds to a 95% confidence interval width of 0.00678 using a normal approximation. Due to the relatively narrow confidence interval of this test, and the small theoretical maximum confidence interval width w.r.t. the testing data set, confidence

[4] If the feature instances were used to select moves in the simulation policy in proportion to their weights, MLE might be a more meaningful metric for comparison.

intervals are not shown in the move prediction results in this chapter. Furthermore, for assessing significance, it is assumed that the 95% confidence interval width of all the move prediction results in this chapter is 0.00678 (0.678%).

The length of time required for generating and evaluating the move prediction of a feature instance can vary drastically depending on the features used and their settings. Due to the fact that the decision tree feature implementation was not optimized, generating and evaluating the move prediction of a feature instance with decision tree features sometimes took up to 56 hours on the testing machine. As such, some tests limited the settings to reduce the time needed (in addition to the limitation imposed by ϕ and weight training).

Note that, in order to highlight the variation in results, the vertical axes of the figures in this chapter show only portions of the possible range, and some of the horizontal axes use a logarithmic scale; all relevant axes are labelled to this effect.

In these tests, harvested patterns, decision forests, and whole feature instances are reused if the settings are the same. As such, except where explicitly stated, feature instances with the same setting values share the components with the setting values in common.

First, Section 5.6 measures the move prediction performance of feature instances with tactical and pattern features, reproducing a state-of-the-art feature instance. Then, Sections 5.7–5.9.1 investigate the impact of the various settings on feature instances with tactical and decision tree features. Using the findings from these tests, Section 5.9.2 evaluates the best-performing feature instances, with various combinations of tactical, pattern and decision tree features, in terms of move prediction. Finally, Section 5.10 introduces the history-agnostic modification for tactical features, and discusses the results of move prediction tests on feature instances using this modification.

5.6 Tactical and Pattern Features

In this section, feature instances with tactical and pattern features are compared. First, Section 5.6.1 evaluates feature instances with just tactical features and varying ϕ. Then, Section 5.6.2 measures the utility of $M(1)$ as an indicator for move prediction. Finally, Section 5.6.3 evaluates feature instances with tactical and pattern features, varying ξ and ϕ in an effort to reproduce a state-of-the-art feature instance.

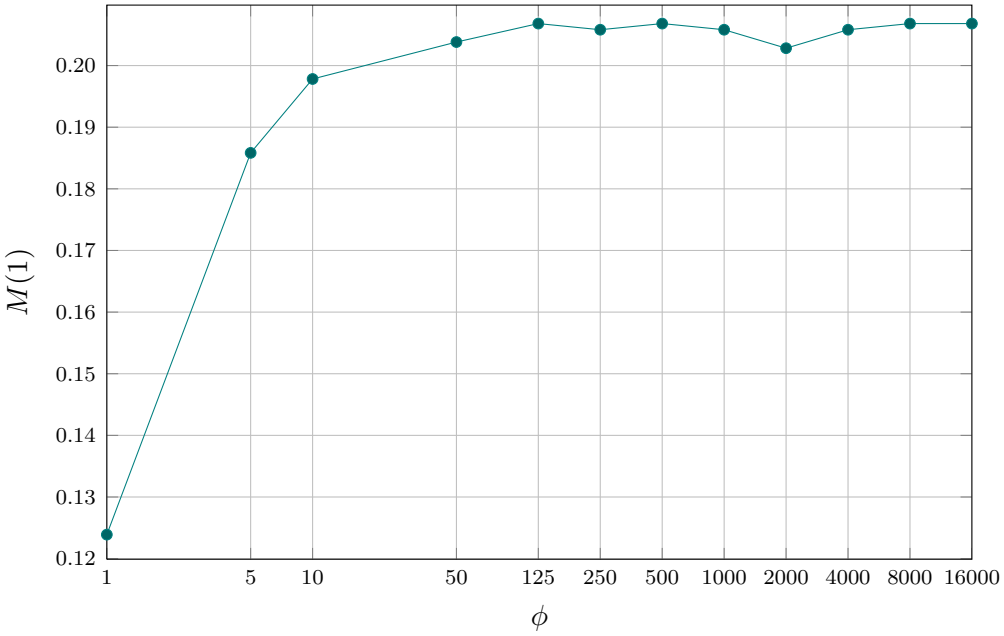


Figure 5.3: The effect of the number of games used for weight training (ϕ) on the move prediction of feature instances with tactical features.

5.6.1 Tactical Features

The move prediction of tactical features (without pattern or decision tree features) is evaluated in this test. In these feature instances, the only relevant setting is ϕ, the number of games used for weight training. Feature instances evaluated in this test were generated with a variety of ϕ values. Due to the lack of pattern and decision tree features, there are very few weights that need to be trained; as such, we conjecture that a relatively low ϕ will still result in feature level weights close to optimal.

Figure 5.3 presents the move prediction results of tactical features, with different values for ϕ. These results show that $\phi = 50$ is enough data for weight training when only tactical features are used. As such, in the following tests (where $\phi \geq 4000$), it will be assumed that the tactical feature portion of a feature instance always has a sufficiently large ϕ, and only the pattern and/or decision tree features potentially require a larger ϕ for training optimal weights.

5.6.2 Utility of the $M(1)$ Value

As stated in Section 5.5, $M(1)$ is used to compare different feature instances. The purpose of this test is to confirm this decision by determining if the $M(1)$

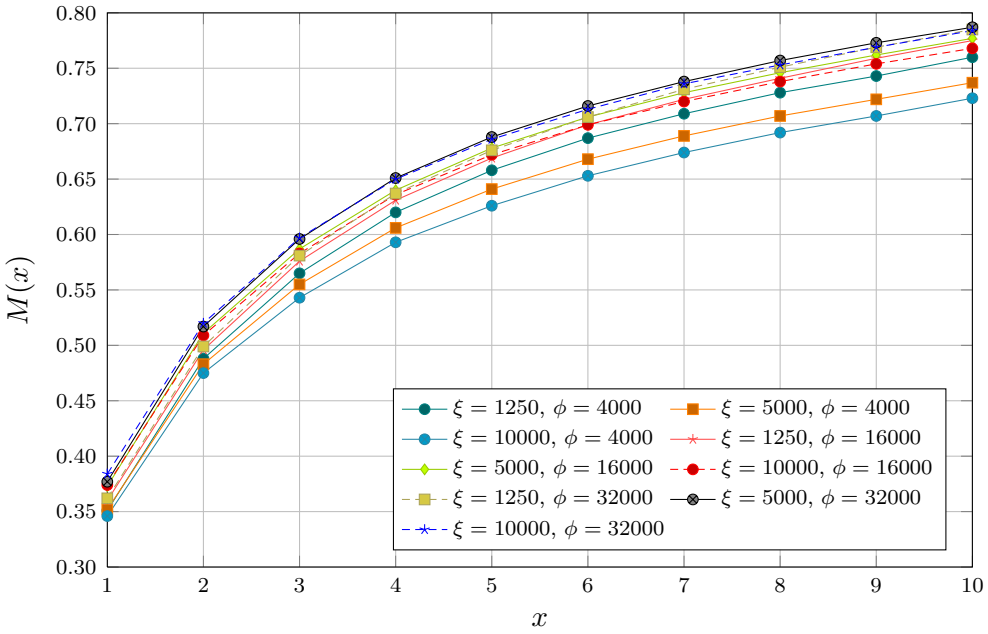


Figure 5.4: Move prediction performance of feature instances with tactical and pattern features, for different numbers of games used for harvesting patterns (ξ) and weight training (ϕ).

value of a feature instance is a good indicator of its overall move prediction, for tactical and pattern features. For this test, a number of feature instances using both tactical and pattern features, and a number of ξ and ϕ setting values, were evaluated.

Figure 5.4 shows the move prediction of four feature instances with varying ξ and ϕ. These results indicate that the $M(1)$ value is a good metric for comparing different M distributions: if feature instances A and B have cumulative distributions M_A and M_B respectively and $M_A(1) > M_B(1)$, then for meaningful values of x, $M_A(x)$ is consistently greater than $M_B(x)$. This matches the argument presented in Section 5.5. As such, the other move prediction tests in this chapter will compare only the $M(1)$ value of the various feature instances.

5.6.3 Impact of ξ and ϕ

The move prediction of feature instances with tactical and pattern features is measured in this test, in order to reproduce a state-of-the-art feature instance. The two settings relevant for these feature instances are ξ and ϕ. The impact of these settings will be measured, and the strongest feature instance used

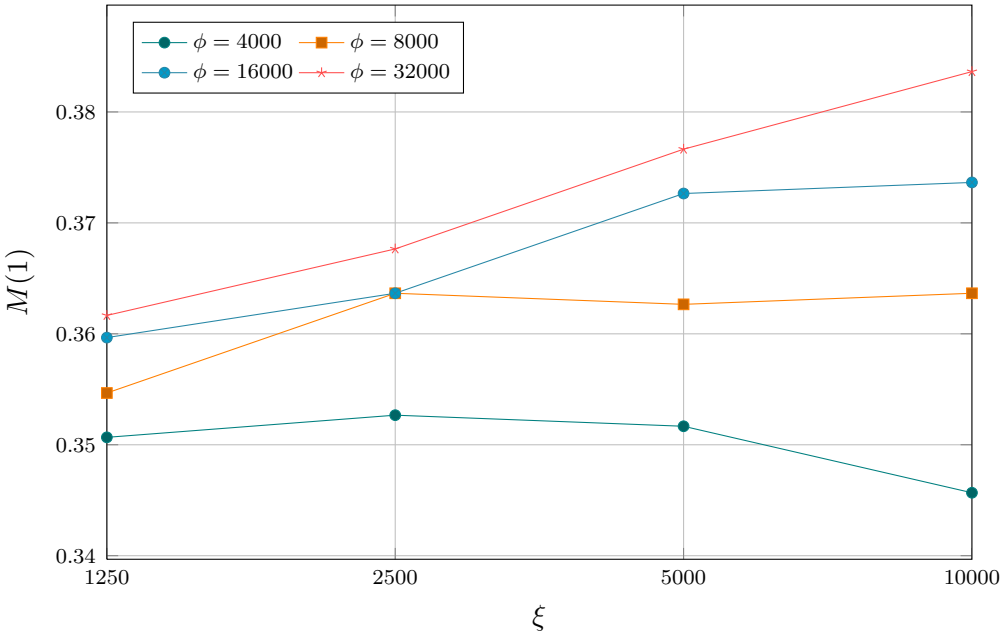


Figure 5.5: The effect of varying the number of games used for harvesting patterns (ξ) for various tactical and pattern feature instances with select values for the number of games used for weight training (ϕ).

as a proxy for the state of the art if it shows comparable performance. We expect a larger ξ and/or ϕ to result in improved move prediction performance.

Figures 5.5 and 5.6 present the $M(1)$ values of various feature instances with a number of values for ξ and ϕ. Figure 5.5 shows that, given a sufficiently large ϕ, an increase in ξ results in an improvement in move prediction. The figure also shows that if ϕ is not large enough, then a larger ξ can be detrimental. The latter effect is presumably due to the high ratio of feature levels to training data for weights in these situations, resulting in noisy feature level weight estimates.

Figure 5.6 illustrates that a larger ϕ generally results in better move prediction, as expected. The graph also indicates that the effect is more pronounced for larger values of ξ, possibly because smaller values of ξ in this test have ϕ values that are approaching optimal.

The feature instance with setting values $\xi = 10000$ and $\phi = 32000$ has move prediction of $M(1) = 38.4\%$. Table 2.1 in Section 2.4.1 listed the state-of-the-art move prediction performance of feature instances with tactical and pattern features, using various algorithms for training weights. In this work the GBTM and MM are used for feature level weight training, and the $M(1)$ value of the feature instance from this work (with $\xi = 10000$ and $\phi = 32000$)

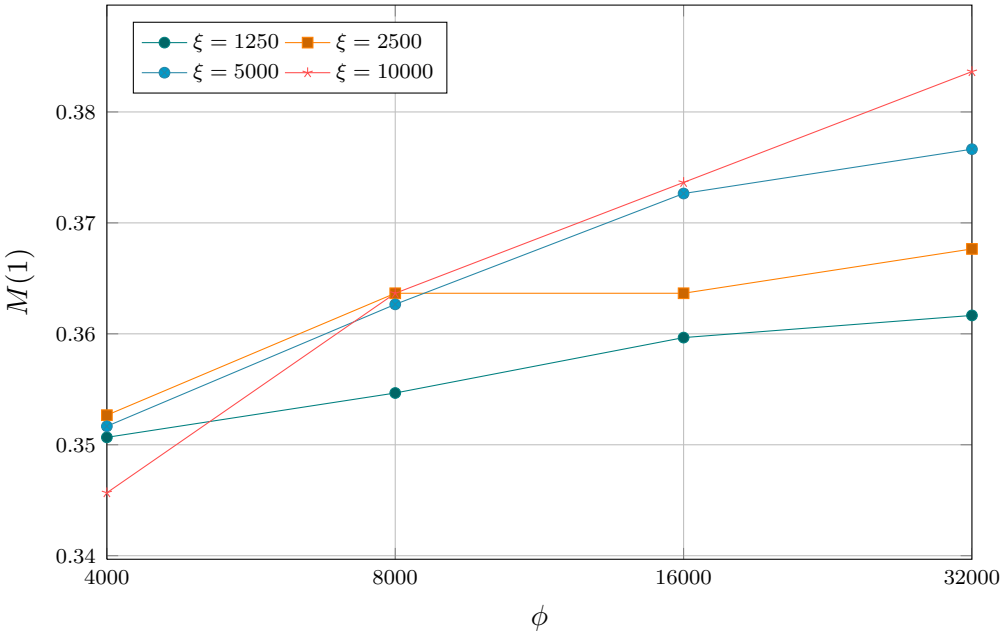


Figure 5.6: The effect of varying the number of games used for weight training (ϕ) for various tactical and pattern feature instances with select values for the number of games used for harvesting patterns (ξ).

compares favourably to the state of the art using GBTM and MM ($M(1) = 37.9\%$). As such, this feature instance will be used to represent the state of the art in the remainder of this chapter.

5.7 Query Systems and Quality Criteria

This section investigates the impact of the choice of query system and quality criterion on the move prediction performance of various feature instances with tactical and decision tree features. This is done in an attempt to investigate the impact of the various query systems and quality criteria, and select a number of promising query systems and the most promising quality criterion for further testing. Additionally, the impact of ϕ on some of these results is considered.

We expect that some query systems will perform better than others, but we expect them all to be comparable. Furthermore, we expect the separate class of quality criteria, especially those with the weighted modification, to perform better than the the split class of criteria.

First, Section 5.7.1 compares the move prediction performance for different quality criteria. Section 5.7.2 then compares the relative performance using different query systems. Section 5.7.3 evaluates the move prediction performance of different query systems, separated into game stages. Finally, Section 5.7.4 considers the impact of ϕ on the move prediction performance of various feature instances.

5.7.1 Quality Criteria

In this work, different quality criteria (divided into two classes: split and separate) were proposed in Section 3.4.2. In order to reduce further testing, the impact of the various quality criteria on move prediction performance was investigated, with the aim of selecting only the best criterion for subsequent tests. Each combination of query system and quality criterion was tested, and all feature instances in this test used $\tau = 8$, $\rho = 500$ and $\phi = 4000$. We expect the separate class of quality criteria, especially those with the weighted modification, to perform better.

The move prediction performance of various quality criteria for feature instances with $\mathrm{SG}_{\emptyset}$ is listed in Table 5.2, sorted by descending $M(1)$ value. These results show that number of leaves in the decision forest is highly dependent on the quality criterion used, and that the ordering in terms of $M(1)$ and MLE is very similar, reaffirming that $M(1)$ is a good metric for comparing move prediction. Similar observations were made for the other query systems.

Table 5.4 presents the $M(1)$ values of all the evaluations measuring the impact of the quality criteria, highlighting the highest $M(1)$ value for each query system.

Table 5.3 summarizes the results from Table 5.4 by listing the best-performing quality criteria for each query system. From these results it is clear that WWLS and WSS are the best performing quality criteria overall. WWLS will be chosen for further testing as it appears to be slightly better than WSS in most of these results.

Section 3.4.2 makes a number of conjectures regarding the relative performance of various quality criteria. The results indicate that, in general, the separate class of criteria have better performance than the split class, as conjectured. The results also verify that the weighted variants of the separate criteria tend to perform better than the non-weighted variants. As conjectured, the results show that the pairs of DS/LS and EDS/ELS have very similar results. Additionally, the results show that in contrast to our conjecture, WWLS performed better than WSS.

Quality Criterion	Leaves	$M(1)$	MLE
Weighted Win-Loss-Separate	12549	31.0%	-2.88
Weighted Symmetric-Separate	9617	31.0%	-2.90
Symmetric-Separate	11738	29.7%	-2.94
Entropy Win-Split	11998	29.5%	-3.00
Win-Loss-Separate	14139	28.9%	-3.03
Naive Descent-Split	17154	28.6%	-3.04
Descent-Split	16233	28.6%	-3.05
Loss-Split	16343	28.1%	-3.06
Win-Split	10355	27.9%	-3.09
Winrate-Entropy	11855	27.8%	-3.07
Entropy Loss-Split	30605	27.5%	-3.14
Weighted Winrate-Entropy	12244	27.1%	-3.12
Winrate-Split	14563	27.0%	-3.19
Entropy Descent-Split	30658	26.6%	-3.16

Table 5.2: Comparison of quality criteria for tactical and decision tree features with the $\text{SG}_\emptyset$ query system. All feature instances use $\tau = 8$, $\rho = 500$ and $\phi = 4000$. The leaves column for a feature instance shows the total number of leaves in the relevant decision forest.

Rank	$\text{SG}_\emptyset$	SG_C	$\text{IG}_\emptyset$	IG_C	IG_E	IG_{CE}
1	WWLS	WSS	WWLS	WWLS	WRS	WRS
2	WSS	WWLS	WSS	WSS	SS	SS
3	SS	NDS	WWE	SS	WWLS	EWS

Table 5.3: Summary of results from Table 5.4 showing the top three quality criteria in terms of $M(1)$, for all query systems. All feature instances use $\tau = 8$, $\rho = 500$ and $\phi = 4000$.

Quality Criterion	$SG_\emptyset$	SG_C	$IG_\emptyset$	IG_C	IG_E	IG_{CE}
Naive Descent-Split	28.6%	29.0%	22.1%	22.5%	20.9%	21.3%
Descent-Split	28.6%	27.8%	21.9%	22.1%	20.8%	21.0%
Win-Split	27.9%	28.1%	22.2%	22.0%	21.0%	21.4%
Loss-Split	28.1%	27.8%	22.0%	22.0%	21.0%	21.1%
Entropy Descent-Split	26.6%	25.5%	25.0%	24.0%	20.7%	20.6%
Entropy Win-Split	29.5%	28.6%	25.9%	23.6%	21.1%	21.5%
Entropy Loss-Split	27.5%	25.5%	25.4%	24.0%	20.6%	20.5%
Winrate-Split	27.0%	25.5%	22.2%	21.7%	**21.6%**	**21.8%**
Win-Loss-Separate	28.9%	27.2%	24.5%	23.9%	21.2%	21.3%
Weighted Win-Loss-Separate	**31.0%**	30.3%	**27.6%**	**25.8%**	21.4%	21.4%
Symmetric-Separate	29.7%	28.0%	24.9%	24.5%	21.5%	21.7%
Weighted Symmetric-Separate	**31.0%**	**30.7%**	27.1%	25.5%	21.3%	21.5%
Winrate-Entropy	27.8%	27.3%	24.7%	23.3%	21.4%	21.3%
Weighted Winrate-Entropy	27.1%	26.0%	26.0%	23.8%	21.2%	21.3%

Table 5.4: Comparison of the $M(1)$ values for tactical and decision tree feature instances with various quality criteria. All feature instances use $\tau = 8$, $\rho = 500$ and $\phi = 4000$.

Furthermore, an unexpected observation from these results is that the best result between IG_E and IG_{CE} is only better than one result from any of the other query systems. As such, the next test will attempt to determine if IG_E and IG_{CE} will be discarded for further testing.

In Section 3.4.2, we conjectured that the separate class of criteria would result in trained weights that are far from 1, in comparison to the split class of criteria (due to the winrates of the descent statistics tending to 0 or 1 for the separate class). To evaluate this conjecture, the variance of the natural logarithm of the weights (VLW) for different quality criteria was measured, and the results are shown in Table 5.5.

The results show that for $SG_\emptyset$ and SG_C there is no discernible difference between the VLW of the two classes of criteria. For $IG_\emptyset$ and IG_C there is a marked difference between the VLW of the separate and split classes of criteria, with the VLW of the split class being much lower. For IG_E and IG_{CE}, the VLW of all the criteria is also much lower than for other query systems; this is possibly related to the poor performance of these two query systems.

In Section 3.4.2, we also conjectured that the non-weighted separate criteria would tend to have very unbalanced decision trees. To evaluate this conjecture, the variance of the length of the descent path for different quality criteria was measured, and the results are shown in Table 5.6.

The results show that the variance of WLS is much higher than WWLS. Although the variances of both SS and WSS are not very large (in comparison to other criteria), SS tends to have a higher variance than WSS. The variances of both WE and WWE are high (in comparison to the other criteria), and there is also not a large difference between them. As such, we conclude that WLS, WE and WWE tend to construct very unbalanced decision trees. Furthermore, WS and EWS for $IG_\emptyset$ and IG_C seem to exhibit similar behaviour, possibly due to a low winrate of the descent statistics, and the potential difficulty in finding candidate queries that satisfy the suitability conditions.

In conclusion, a number of characteristics of feature instances with various quality criteria were measured, confirming most of the conjectures from Section 3.4.2. Furthermore, WWLS was shown to be the best-performing quality criterion, and is therefore used in further testing.

5.7.2 Query Systems

In order to identify the feasibility of the different query systems, this test measured the move prediction performance of feature instances with different

Quality Criterion	$SG_\emptyset$	SG_C	$IG_\emptyset$	IG_C	IG_E	IG_{CE}
Naive Descent-Split	0.413	0.399	0.164	0.134	0.009	0.078
Descent-Split	0.429	0.425	0.123	0.082	0.004	0.004
Win-Split	0.368	0.354	0.126	0.119	0.015	0.008
Loss-Split	0.412	0.408	0.111	0.093	0.005	0.071
Entropy Descent-Split	0.162	0.162	0.186	0.217	0.004	0.005
Entropy Win-Split	0.431	0.391	0.277	0.195	0.029	0.011
Entropy Loss-Split	0.197	0.194	0.237	0.191	0.007	0.005
Winrate-Split	0.307	0.291	0.133	0.114	0.051	0.054
Win-Loss-Separate	0.463	0.422	0.349	0.309	0.090	0.137
Weighted Win-Loss-Separate	0.432	0.393	0.293	0.257	0.061	0.062
Symmetric-Separate	0.462	0.421	0.302	0.225	0.066	0.087
Weighted Symmetric-Separate	0.407	0.378	0.270	0.209	0.032	0.036
Winrate-Entropy	0.438	0.408	0.326	0.266	0.146	0.158
Weighted Winrate-Entropy	0.437	0.396	0.330	0.280	0.109	0.142

Table 5.5: Variance of the natural logarithm of the decision tree weights for tactical and decision tree feature instances with various quality criteria. All feature instances use $\tau = 8$, $\rho = 500$ and $\phi = 4000$.

Quality Criterion	$SG_\emptyset$	SG_C	$IG_\emptyset$	IG_C	IG_E	IG_{CE}
Naive Descent-Split	1.01	1.47	5.05	5.33	7.91	20.19
Descent-Split	1.25	1.06	8.16	6.71	9.23	9.75
Win-Split	5.69	5.78	444.83	843.07	56.06	74.41
Loss-Split	1.18	0.94	6.65	6.50	9.85	11.77
Entropy Descent-Split	1.49	0.84	2.04	55.93	9.25	9.04
Entropy Win-Split	6.80	4.93	431.21	548.91	33.37	81.70
Entropy Loss-Split	1.60	0.78	10.85	5.71	10.16	8.86
Winrate-Split	9.41	8.71	16.63	17.06	31.25	20.48
Win-Loss-Separate	140.89	195.45	821.22	1083.32	518.36	230.47
Weighted Win-Loss-Separate	11.19	4.75	103.29	25.65	68.15	56.07
Symmetric-Separate	12.06	7.61	23.49	19.67	132.27	39.50
Weighted Symmetric-Separate	6.49	4.96	13.47	13.58	46.48	52.22
Winrate-Entropy	149.87	233.35	1052.30	1485.64	687.27	754.38
Weighted Winrate-Entropy	345.55	374.30	941.24	1452.90	640.26	787.53

Table 5.6: Variance of the length of the descent paths for the decision forest in feature instances with various quality criteria. All feature instances use $\tau = 8$, $\rho = 500$ and $\phi = 4000$.

query systems. All feature instances have setting values WWLS, $\tau = 8$, $\rho = 500$ and $\phi = 4000$ (the same as the previous test). Due to the poor performance of IG_E and IG_{CE}, additional feature instances with these query systems, but with other quality criteria (WSS and WRS) and values for ρ ($\rho = 2000$ or $\rho = 4000$) were tested. The move prediction performance of just tactical features with $\phi = 4000$ are also included for comparison.

The results of this test, shown in Table 5.7, indicate that SG has better move prediction than IG, and that the query systems with no modifications seem to be stronger than others in the same query system class. Furthermore, the four best-performing query systems ($\text{SG}_\emptyset$, SG_C, $\text{IG}_\emptyset$ and IG_C) all have significantly better move prediction performance than just tactical features.

The results also show that, even with larger trees than in the previous test (indicated by the number of leaves in the decision forest), tactical and decision tree feature instances with IG_E and IG_{CE} are not able to perform much better than just tactical features. A possible explanation is that the empty modification discards important information from the board-move pair. As such, only $\text{SG}_\emptyset$, SG_C, $\text{IG}_\emptyset$ and IG_C were deemed feasible for use in further testing. The combination of various query systems will be tested in Section 5.9.1.

5.7.3 Query Systems per Game Stages

In Section 3.3 we conjectured that the various query system classes would perform better at different aspects of Go — we conjectured that the SG class of query systems would perform better at the beginning of the game, while IG would perform better at life and death scenarios, which tend to occur in the later part of the game. In order to validate this, game stages were formed by separating the positions from the testing data set into independent groups, depending on their move number in their respective game. In this work, the width of each game stage was fixed at 30 moves, i.e. the first stage was composed of moves 1 to 30, the second stage is composed of moves 31 to 60, so on. The length of a Go game varies considerably, with very few games in the data set with more than 300 moves. As such, the testing data set is limited to the first 300 moves of each game. Feature instances have setting values WWLS, $\tau = 8$, $\rho = 500$ and $\phi = 4000$ (the same as the previous two tests).

Figure 5.7 compares the $M(1)$ values of the various feature instances, separated by game stage. The results show that, as expected, SG performs better than IG at the beginning of the game. However, in the later part of the game SG and IG have similar behaviour — the move prediction results tend to steadily improve from move 120 at approximately the same rate, for

Query System	QC	ρ	Leaves	$M(1)$
$SG_{\emptyset}$	WWLS	500	12549	31.0%
SG_C	WWLS	500	9288	30.3%
$IG_{\emptyset}$	WWLS	500	12262	27.6%
IG_C	WWLS	500	7352	25.8%
IG_E	WWLS	500	3977	21.4%
IG_{CE}	WWLS	500	3341	21.4%
IG_E	WWLS	2000	8104	21.7%
IG_E	WSS	2000	5924	21.6%
IG_E	WRS	2000	6836	21.3%
IG_{CE}	WWLS	2000	4356	21.4%
IG_{CE}	WSS	2000	5195	21.9%
IG_{CE}	WRS	2000	6674	21.7%
IG_E	WWLS	4000	12608	21.5%
IG_E	WSS	4000	9706	21.4%
IG_E	WRS	4000	12462	21.1%
IG_{CE}	WWLS	4000	14268	21.2%
IG_{CE}	WSS	4000	14823	21.6%
IG_{CE}	WRS	4000	16775	21.4%
Only tactics	-	-	-	20.7%

Table 5.7: Comparison of the $M(1)$ values of tactical and decision tree feature instances with the different query systems, with additional feature instances with IG_E and IG_{CE}. All feature instances use $\tau = 8$ and $\phi = 4000$.

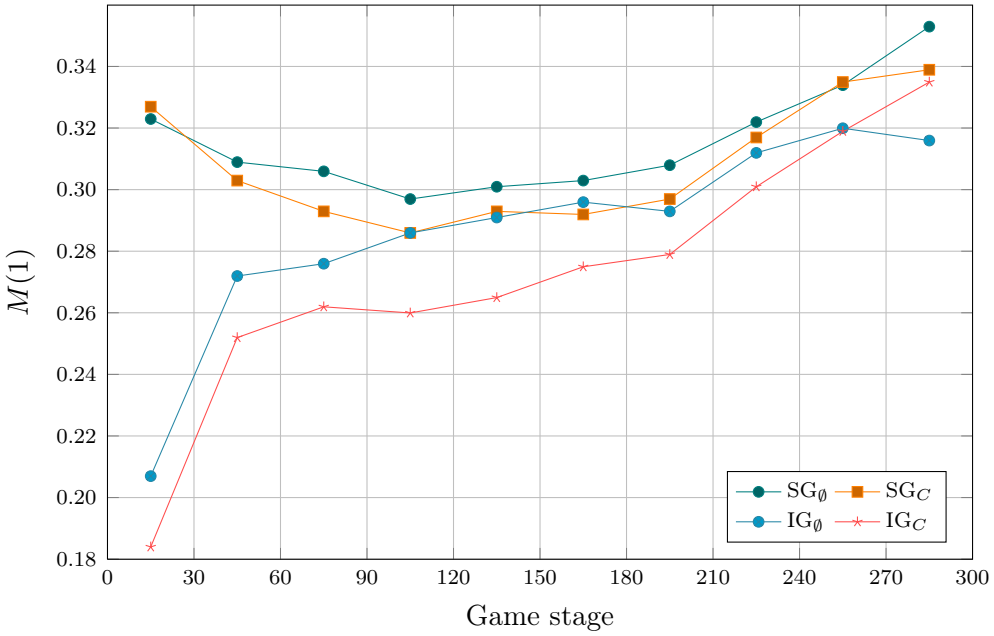


Figure 5.7: Move prediction performance of tactical and decision tree feature instances with different query systems, separated by game stages. A game stage consists of a period of 30 moves, measured from the start of the game. The results for a game stages are plotted in the center of the relevant interval. All feature instances use WWLS, $\tau = 8$, $\rho = 500$ and $\phi = 4000$.

all query systems. An intuitive explanation of this is that diversity of the positions in the later part of a Go game is lower than during the earlier part.

5.7.4 Impact of ϕ

To confirm that $\phi = 4000$ used in the tests of Sections 5.7.1–5.7.3 was sufficient, this test measured the impact of ϕ (the number of games used for weight training) on the feature instances from the previous tests, that compared quality criteria and query systems. All feature instances used WWLS, $\tau = 8$ and $\rho = 500$.

Figure 5.8 shows the $M(1)$ values of the various feature instances, in terms of ϕ. These results show that an increase in ϕ will typically result in a better $M(1)$ value, and that $\phi = 4000$ was a meaningful value for the previous tests, as there is considerably more improvement up to this value in comparison with beyond this value. For future tests that require a large ϕ value (such as measuring the impact of ρ), a value of $\phi = 16000$ will be

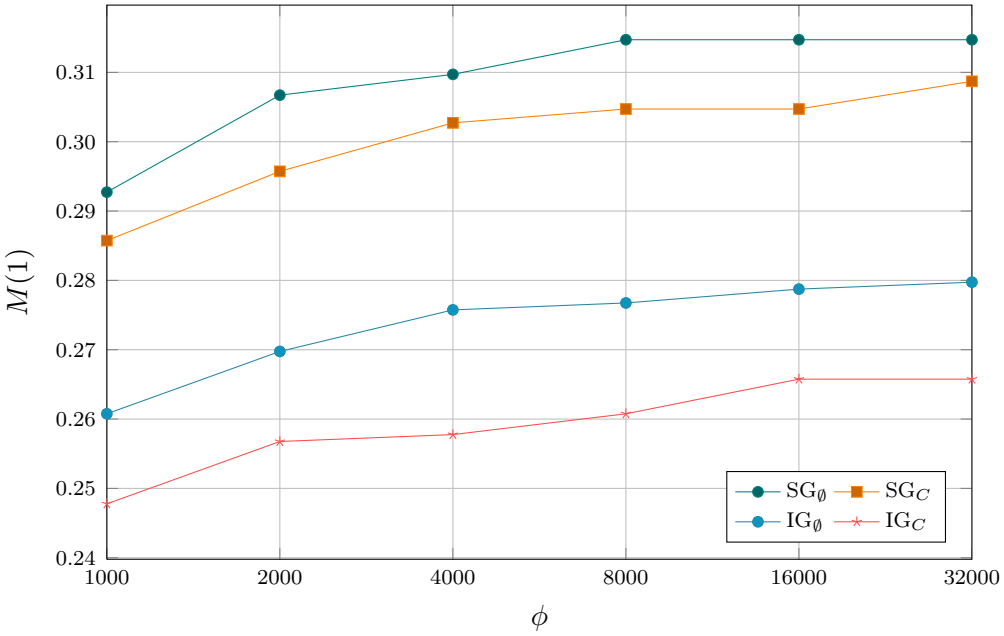


Figure 5.8: The effect of varying the number of games used for weight training (ϕ) for tactical and decision tree feature instances with each query system. All feature instances use WWLS, $\tau = 8$ and $\rho = 500$.

used, as the results indicate such a value will offer good performance, while still allowing for larger values of other settings, such as τ.

At this point, a number of feature instances with decision tree feature have been evaluated. Results have indicated that WWLS is a good quality criterion, and that IG_E and IG_{CE} have very poor results. As such, further testing will use $\text{SG}_\emptyset$, SG_C, $\text{IG}_\emptyset$ and IG_C with WWLS; the next section will investigate the impact of τ, ρ, and ϕ on move prediction performance.

5.8 Decision Forest Parameters

This section considers the impact of the τ, ρ and ϕ feature instance settings on the move prediction performance of feature instances with tactical and decision tree features. For all of these settings, we expect a larger value to result in better move prediction performance, but with diminishing returns in each case.

First, Section 5.8.1 investigates the impact of τ (the number of trees in the decision forest) on move prediction performance. Next, Section 5.8.2 investigates the impact of ρ (the number of games used to grow the trees).

Finally, Section 5.8.3 measures the impact of ϕ (the number of games used for weight training) on the move prediction performance of various feature instances.

5.8.1 Impact of τ

Theoretically, a single large decision tree ($\tau = 1$) could give optimal results, but preliminary testing confirmed the intuition that a forest of trees ($\tau > 1$) has superior move prediction performance. As such, this test measured the impact of τ on the move prediction of feature instances with tactical and decision tree features. However, simply testing different values of τ, with all other settings fixed, will result in both the number of trees in the forest, and the overall number of leaves being altered.[5] As such, ρ will also be adjusted to keep the number of leaves in the forest (and therefore the number of feature level weights) approximately the same. To do this, the total number of tree descents in tree growth is kept approximately constant by keeping the value of $\rho\tau$ fixed.

In this test, both $\rho\tau = 4000$ and $\rho\tau = 8000$ were considered; the query systems were tested independently, and all feature instances used WWLS and $\phi = 4000$. This value of ϕ was chosen to reduce the time taken for this test, while still leaving it large enough for adequate weight training. Values of τ from 1 to 64 were used in this test. Feature instances with $\tau = 128$ were attempted, but the weight training process could not handle this setting, due to the increase in size of the GBTM competitions and therefore the increased memory requirement.

$M(1)$ in terms of τ is shown in Figure 5.9. These results indicate that a larger τ will usually result in better move prediction. While the results do not indicate a significant plateau, a setting of $\tau = 16$ was chosen for further testing as (for most of the query systems and values of $\rho\tau$) there is only very little improvement for larger values of τ, and larger values constrain other settings due to computational considerations.

5.8.2 Impact of ρ

This test evaluated the impact of ρ on move prediction. We expect a larger ρ to lead to larger trees, and thus improved move prediction performance due to the increased length (and therefore specificity) of the descent paths (corresponding to feature levels). However, when ρ is increased, the number

[5]The ρ games are independently considered for each of the τ trees, as described in Section 4.2; therefore, the number of leaves in the decision forest is closely related to $\rho\tau$.

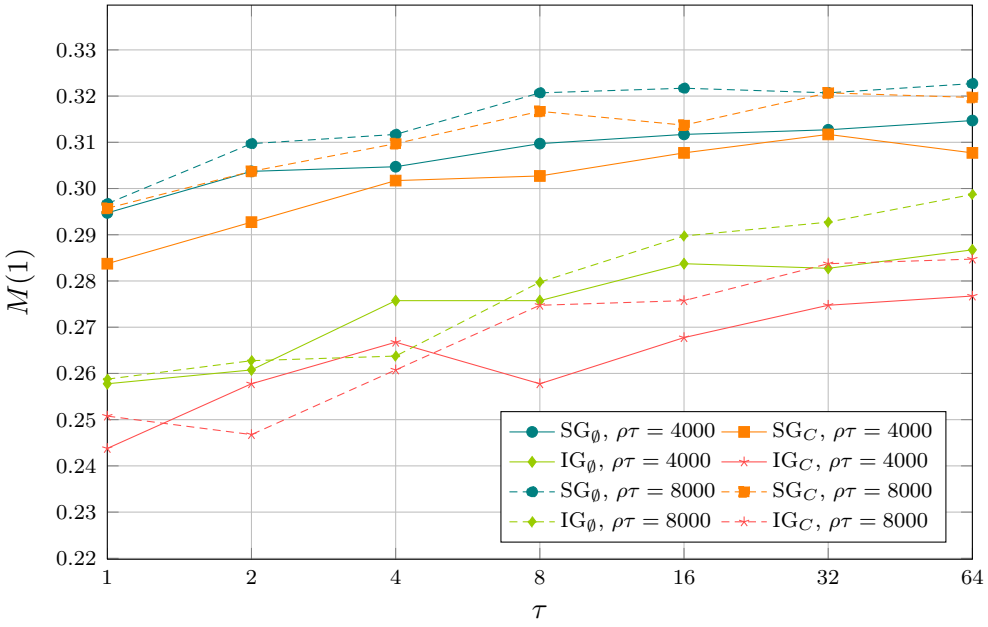


Figure 5.9: Evaluation of the impact of varying the number of trees in the decision forest (τ) of tactical and decision tree feature instances, with a fixed $\rho\tau$. All feature instances use WWLS and $\phi = 4000$.

of leaves will tend to increase, and therefore the number of feature level weights that need to be trained will also increase. As such, this test used a large value for ϕ, in an attempt to reduce the dependency of the results on the weight training. For this test, all feature instances used WWLS, $\tau = 16$ and $\phi = 16000$, for each query system.

Figure 5.10 presents the $M(1)$ results of this test. The results clearly indicate that increasing ρ will usually result in improved move prediction. Unfortunately, a number of the feature instances with $\rho = 3200$ took over 50 hours to evaluate. As such, later tests will typically not make use of such a large value for ρ, to reduce the evaluation time.

5.8.3 Impact of ϕ

While the impact of ϕ was considered in an earlier test in Section 5.7.4, subsequent tests have used significantly larger trees, which should require more weight training data. This test investigated whether the impact of ϕ on decision forests with more leaves is the same as the earlier test showed. We conjecture that the effect of increasing ϕ has significant diminishing returns, and will reach a plateau when it is large enough for the size of the trees.

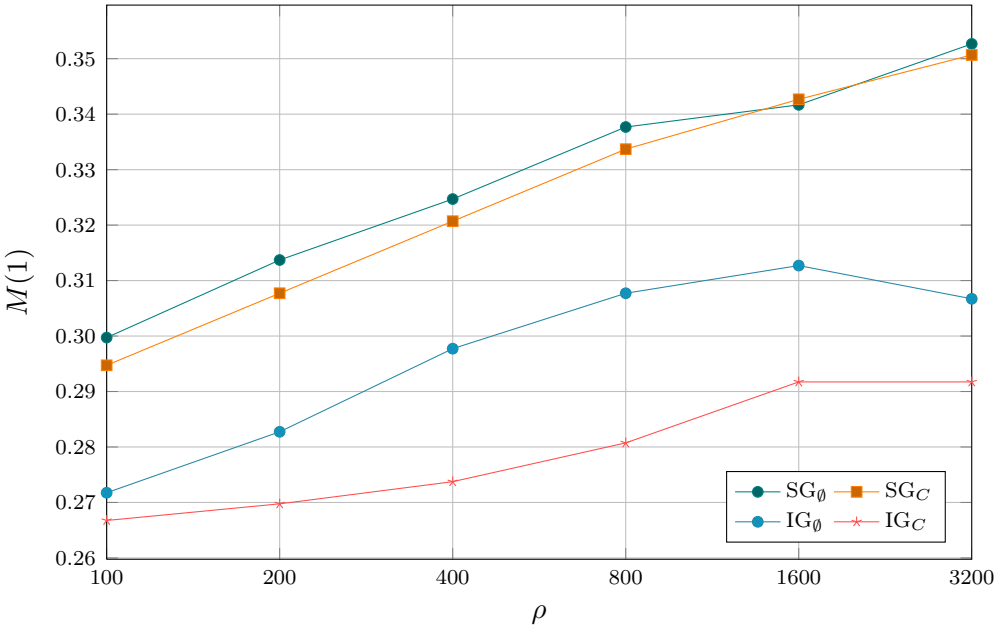


Figure 5.10: The effect of varying the number of games used for growing decision trees (ρ) for tactical and decision tree feature instances with each query system. All feature instances use WWLS, $\tau = 16$ and $\phi = 16000$.

As such, this test was performed on trees with $\rho = 800$ so that a plateau could hopefully be observed within the restrictions of the weight training. Observing a plateau in the results might permit the combined use of results from this test and the earlier test in Section 5.7.4, measuring the impact of ϕ (and indicating a plateau), to extrapolate when the plateau would be reached for an arbitrary-size forest of decision tree features. All feature instances in this test use WWLS, $\tau = 16$ and $\rho = 800$ for each query system and value of ϕ.

Figure 5.11 shows the impact of ϕ on the feature instances of this test. These results indicate that, while there are diminishing returns, increasing ϕ has a significant positive effect on relatively large trees — within the range of ϕ in this test, no plateau could be found; a larger ϕ than our system can handle is required for optimal weights for such large trees/forests. Note that $\phi = 32000$ could not be used due to computational resource limitations.

The results presented in this section have shown that feature instances with larger values of τ, ρ, and/or ϕ have improved move prediction performance. However, increasing these setting values also impacts the time and memory required to construct and test feature instances. As such, some non-optimal values are chosen in testing to reduce the computational re-

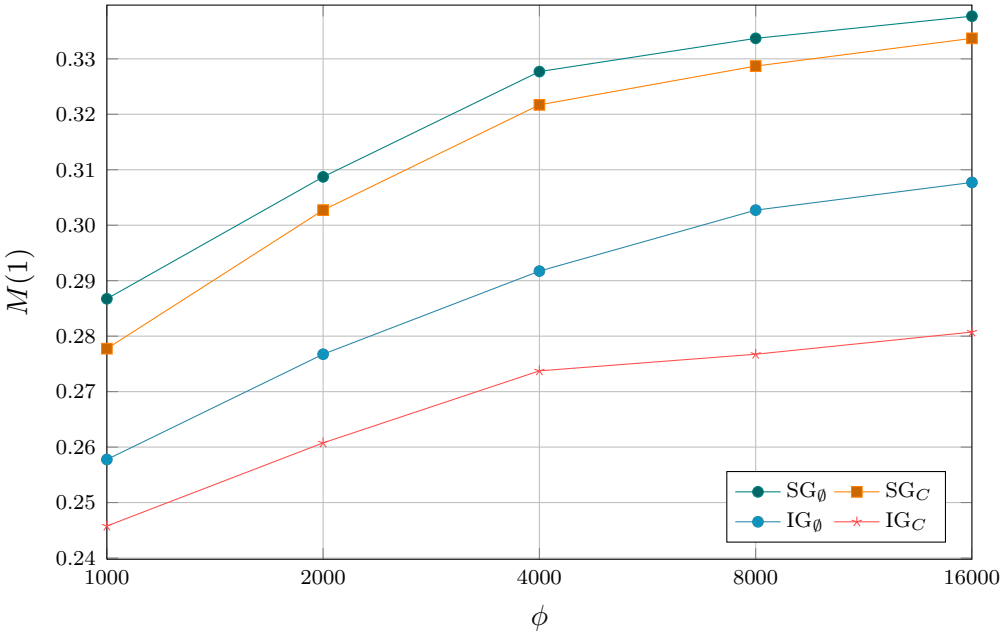


Figure 5.11: The effect of varying the number of games used for weight training (ϕ) for tactical and decision tree feature instances with each query system. All feature instances use WWLS, $\tau = 16$ and $\rho = 800$.

quirements. The next section will evaluate feature instances with multiple query systems, and use all the move prediction testing results to construct the best-performing feature instance using decision tree features.

5.9 Comparison with State of the Art

This section investigates feature instances with multiple query systems and the combination of tactical, pattern and decision tree features. We expect multiple query systems to have move prediction performance slightly above the average of the separate query systems, and we expect feature instances with the combination of tactical, pattern and decision tree features to be stronger than other feature instances without all three feature types.

First, Section 5.9.1 evaluates feature instances with multiple query systems. Then Section 5.9.2 investigates the move prediction performance of the best feature instances with various combinations of tactical, pattern and decision tree features.

	$SG_\emptyset$	SG_C	$IG_\emptyset$	IG_C
$SG_\emptyset$	33.4%	33.2%	**33.9%**	33.5%
SG_C	-	32.9%	33.8%	32.6%
$IG_\emptyset$	-	-	30.3%	29.5%
IG_C	-	-	-	27.7%

Table 5.8: Comparison of $M(1)$ values for tactical and decision tree feature instances with the combination of up to two different query systems. All features instances use WWLS, $\tau = 16$, $\rho = 800$ and $\phi = 8000$.

5.9.1 Combinations of Query Systems

This test measured the move prediction performance of tactical and decision tree feature instances, with a combination of different query systems in the same forest. All feature instances in this test used WWLS, $\rho = 800$ and $\phi = 8000$. The total size of the decision forest was fixed at 16 trees ($\tau = 16$), and each combination of two query systems was tested with 8 trees of each type. We conjecture that the combination of two query systems will be slightly stronger than their average strength, as we expect the combination will be more robust.

Table 5.8 presents the results of combining different query systems. The results indicate that the combination of multiple query systems can have some benefit, confirming the conjecture — all the feature instances with a combination of two query systems are at least slightly better than the average of the two separate query systems in terms of move prediction. Although the difference between the strongest single query system ($SG_\emptyset$) and the strongest pair ($SG_\emptyset$ + $IG_\emptyset$) is not significant, the results also indicate that the combination of two query systems is usually not significantly weaker than the strongest of the separate query systems, i.e. a single query system is not stronger than any combination containing that query system.

5.9.2 Best Feature Instances

This test first constructs a feature instance with the best setting values for decision tree features found thus far; then this feature instance is combined with pattern features; and finally, these feature instances are compared to the state-of-the-art feature instance with tactical and pattern features.[6]

[6]We will assume that comparing $M(1)$ values is appropriate, considering these feature instances have varying feature types (confirmed in Figure 5.12), and the 95% confidence interval width of 0.00678 (0.678%) empirically found in Section 5.5 also applies to the results of this test.

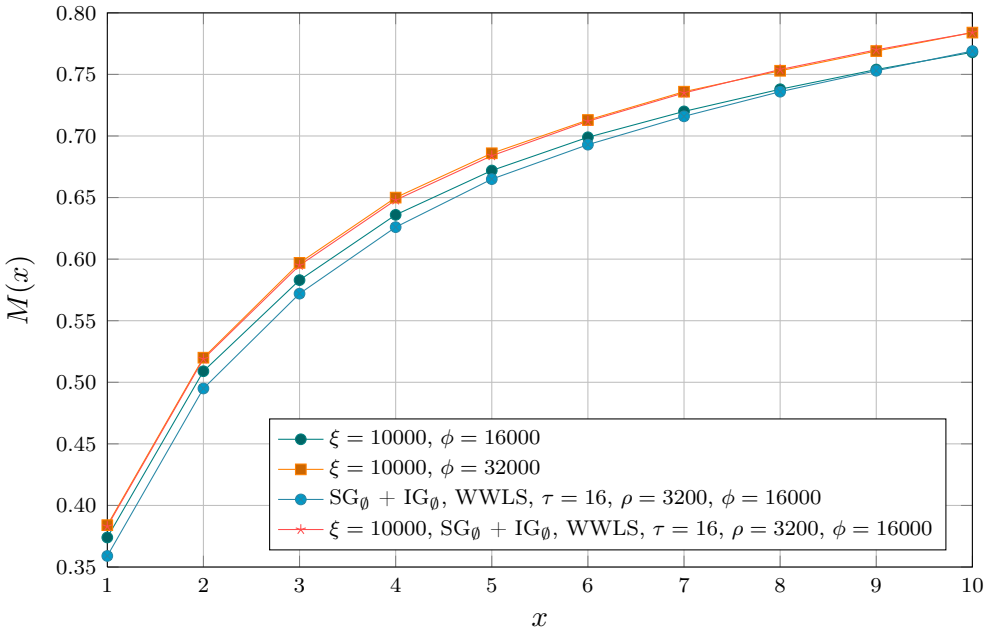


Figure 5.12: Comparison of move prediction performance of various feature instances with tactical, pattern and/or decision tree features. Feature instance setting values are indicated in the legend.

We conjecture that the move prediction performance of the first feature instance (with the best setting values for tactical and decision tree features) will be comparable to the state of the art, while the second feature instance (with tactical, pattern and decision tree features) will improve upon the state-of-the-art results.

Figure 5.12 shows the move prediction performance of the various feature instances used in this test. Setting values are indicated in the graph legend, and it is assumed that feature instances with no mention of setting values for pattern or decision tree features do not contain the relevant features. Discussion of these results follows.

The feature instance with tactical and decision tree features was constructed with settings $\text{SG}_\emptyset + \text{IG}_\emptyset$, WWLS, $\tau = 16$, $\rho = 3200$ and $\phi = 16000$. These setting values were chosen as the best values found in the previous tests (with the exception of τ, which was chosen to allow a larger ρ). The $M(1)$ value of the feature instance was measured to be 35.9%.

While this result is not quite as good as the state of the art, it is still comparable. As such, we can say that tactical and decision tree features are a feasible alternative to state-of-the-art tactical and pattern features.

The feature instance with tactical, pattern and decision tree features was constructed with settings $\xi = 10000$, $\text{SG}_\emptyset + \text{IG}_\emptyset$, WWLS, $\tau = 16$, $\rho = 3200$ and $\phi = 16000$. The $M(1)$ value of this feature instance was measured to be 38.3%.

This result shows that, while this feature instance is able to achieve equivalent performance to state of the art, it is not able to surpass it. However, the value of ϕ used for this feature instance was limited; if we compare this result to a feature instance with tactical and pattern features and $\phi = 16000$ (which obtained $M(1) = 37.4\%$ in Section 5.6.3), then the inclusion of decision tree features offers a small but significant improvement.

5.10 History-Agnostic Features

The board-move pair theoretically contains enough information to compute an accurate evaluation for move prediction.[7] However, in Go moves are often local, i.e. moves are often close in proximity to recent moves. While this shouldn't make a difference (the best move should be independent of the game history), tactical features do benefit greatly in practice from the inclusion of one or more features that take the distance to recent moves into account. This section briefly explores a modified set of tactical features that are history-agnostic, for a more fair comparison of feature instances with just decision tree features (which are only able to evaluate the board-move pair).

In this test, the abbreviations T, P and DT are used for tactical, pattern and decision tree features respectively. All pattern features used $\xi = 10000$, and all decision tree features used $\text{SG}_\emptyset + \text{IG}_\emptyset$, WWLS, $\tau = 16$ and $\rho = 3200$. ϕ is either 16000 or 32000, depending on the limitations of weight training due to limited memory.

The history-agnostic tactical features (T_{HA}) introduced in this test are identical to T (see Table 5.1), except that the last four features that measure the distance to recent moves are excluded. Of the remaining features, only two feature levels might be considered as history-dependent: atari level 2 and capture level 4. However, atari level 2 could theoretically be determined without the game history.[8] Furthermore, it is fairly tricky to remove just the single level of capture level 4 without extensive further computation, and it lacks the broad applicability of the excluded features that measure the distance to the previous two moves. As such, the atari and capture tactical features were left as-is. Besides these potential exceptions, this set of features (T_{HA}) is not dependent on the history of the game.

[7] This assumes that ko information is part of the board state.

[8] It can be determined that a potential move on the board is illegal due to the ko rule.

T	T_{HA}	P	DT	ϕ	$M(1)$
X	-	X	-	32000	38.4%
X	-	X	X	16000	38.3%
X	-	-	X	16000	35.9%
-	X	X	-	32000	26.1%
-	X	-	X	16000	25.0%
-	-	-	X	16000	24.7%
-	-	X	-	32000	24.6%
X	-	-	-	16000	20.7%
-	X	-	-	16000	9.7%

Table 5.9: $M(1)$ values of various feature instances with history-agnostic tactical features. Feature instances use some appropriate subset of the following setting values: $\xi = 10000$, $SG_\emptyset$ + $IG_\emptyset$, WWLS, $\tau = 16$, $\rho = 3200$ and $\phi = 16000$ or $\phi = 32000$ (for P, T_{HA} + P and T + P).

Table 5.9 and Figure 5.13 present the results of this test, sorted by descending $M(1)$ value. These results show that feature instances with T_{HA} do not perform as well as feature instances with T. Furthermore, the results show that, T_{HA}+DT and just DT feature instances are comparable with T_{HA}+P and just P feature instances respectively. Additionally, the results indicate that the inclusion of any type of tactical features has a greater impact on feature instances with pattern features, compared to feature instances with decision tree features. We conjecture that this is because decision tree features are more efficiently able to encode tactic-like concepts, but further testing, with a larger variety of feature instances, and more in-depth analysis is required to confirm this.

5.11 Playing Strength

This section compares a few select feature instances in terms of playing strength, when integrated into an MCTS engine, Oakfoam.

Playing strength was compared using 10000 playouts per move with Oakfoam, and features were integrated into Oakfoam with progressive widening, as described in Section 4.6. A fixed number of playouts per move was used because the aim is to determine feasibility, and the decision tree feature implementation was not optimized. For each feature instance, a series of 100 games were played on 19x19 against GnuGo (version 3.8, level 10) with 7.5 komi and alternating colors starting on consecutive games [36]. The 95% confidence interval for a series of 100 games is fairly large, making compari-

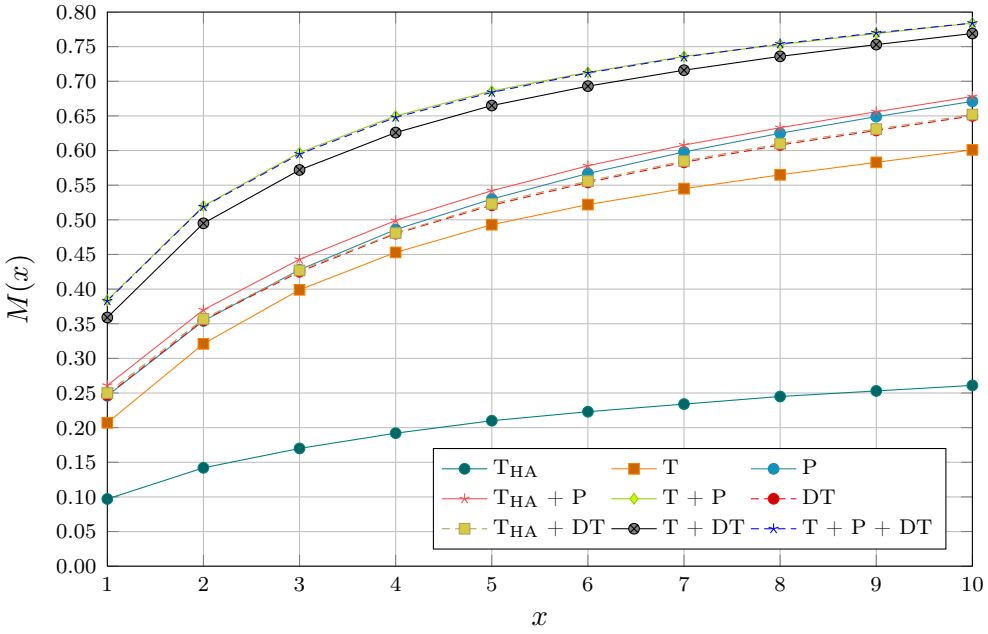


Figure 5.13: Move prediction of various feature instances with history-agnostic tactical features. Feature instances use some appropriate subset of the following setting values: $\xi = 10000$, $\mathrm{SG}_\emptyset + \mathrm{IG}_\emptyset$, WWLS, $\tau = 16$, $\rho = 3200$ and $\phi = 16000$ or $\phi = 32000$ (for P, T_{HA} + P and T + P).

son between results difficult; however, due to the non-optimized decision tree feature implementation, only a limited number of games could be played in the time available.

Results of the strength comparison are shown in Table 5.10. The results show that feature instances with decision tree features instead of pattern features are a feasible alternative (especially with the same number of games for ϕ): not as good as the state of the art, but not significantly worse. Furthermore, the combination of tactical, pattern and decision tree features performed marginally better than the state of the art, but not significantly so. These results correspond well with the move prediction results, confirming that when used for progressive widening, move prediction is a good indicator of playing strength.

5.12 Conclusion

This chapter evaluated a large variety of feature instances in terms of move prediction and playing strength. In Section 5.6, a state-of-the-art feature instance with tactical and pattern features, and setting values $\xi = 10000$ and $\phi = 32000$, was constructed and evaluated in terms of move prediction. This state-of-the-art feature instance has an $M(1)$ value of 38.4%.

In Sections 5.7–5.9.1, feature instances with tactical and decision tree features were evaluated, and the impact of the various relevant feature instance settings was measured. Section 5.7.1 showed that the WWLS quality criterion has the best move prediction performance and was therefore used for subsequent tests. Section 5.7.2 illustrated that the two query systems with the empty modification (IG_E and IG_{CE}) have poor performance in comparison with the other query systems, and as such, only $\text{SG}_\emptyset$, SG_C, $\text{IG}_\emptyset$ and IG_C were used in further tests. Sections 5.7.4–5.8.3 evaluated the impact of the other relevant settings (τ, ρ and ϕ) of feature instances with tactical and decision tree features on move prediction performance. The results in these sections showed that increasing any one of the parameters resulted in an improvement in move prediction. However, increasing these settings also resulted in the evaluation of these feature instances requiring more computational resources. As such, many tests were performed on feature instances with smaller-than-optimal setting values to reduce the required computational resources. Section 5.9.1 showed that the combination of multiple query systems can result in improved move prediction performance, and the combination of $\text{SG}_\emptyset$ and $\text{IG}_\emptyset$ showed the best performance.

Section 5.9.2 constructed the best feature instance with tactical and decision tree features, and setting values $\text{SG}_\emptyset$ + $\text{IG}_\emptyset$, WWLS, $\tau = 8$, $\rho = 3200$

Tactical	Pattern	Decision trees	ϕ	$M(1)$	Winrate
X	-	-	16000	20.7%	8% ± 5.3%
X	$\xi = 10000$	-	16000	37.4%	48% ± 9.8%
X	$\xi = 10000$	-	32000	38.4%	53% ± 9.8%
X	-	$SG_\emptyset + IG_\emptyset$, WWLS, $\tau = 16$, $\rho = 3200$	16000	35.9%	48% ± 9.8%
X	$\xi = 10000$	$SG_\emptyset + IG_\emptyset$, WWLS, $\tau = 16$, $\rho = 3200$	16000	38.3%	58% ± 9.7%

Table 5.10: Comparison of playing strength with a few select feature instances. All feature instances were evaluated by incorporating them into OAKFOAM with progressive widening. For each feature instance, 100 games on a 19x19 board were played against GNUGO, with a fixed 10000 playouts per move for OAKFOAM. The 95% confidence interval is indicated for each winrate.

and $\phi = 16000$. This feature instance had an $M(1)$ value of 35.9%, which is inferior, but comparable, to the state of the art. A feature instance that combined tactical, pattern and decision tree features was also constructed and evaluated in Section 5.9.2, but this feature instance did not improve upon the state of the art.

In Section 5.10, a history-agnostic variation of tactical features was introduced, and various feature instances using these features were evaluated. The results showed that decision tree features have comparable move prediction performance compared to pattern features without tactical features, and feature instances with history-agnostic tactical and decision tree features have similar performance to feature instances with history-agnostic tactical and pattern features.

Finally, in Section 5.11, the playing strength of a few select feature instances was measured. These results showed that the best tactical and decision tree feature instance has inferior, but comparable, playing strength performance to state of the art.

In conclusion, the results of this chapter have demonstrated that feature instances with tactical and decision tree features are a feasible alternative to the state of the art, in terms of move prediction and playing strength (with a fixed number of playout per move). However, the decision tree feature implementation was not optimized and it remains to be seen if optimization will make decision tree features practically usable in MCTS engines where there is more typically a fixed amount of time available.

Chapter 6

Conclusion

In computer Go, moves are selected for play from a given position, similar to other domains where actions are selected from given states. The use of features for encoding domain knowledge, constructing move orderings for prediction, and incorporating the features into the move selection process, has been shown to be powerful. These features include tactical and pattern features; tactical features encode hand-crafted heuristics, such as playing a move that captures an opposing chain of stones, while pattern features encode the board position surrounding a candidate move. However, it is not clear how current approaches that use features can easily be transferred to other domains — tactical features require expert knowledge to encode domain knowledge, and pattern features are Go-specific. As such, this work aimed to propose a new more general approach to extracting and using domain knowledge.

The proposed decision tree feature approach uses decision trees for extracting domain knowledge in an automated manner. These features evaluate a state-action pair by descending a decision tree, with queries recursively partitioning the input space of state-action pairs and returning a weight stored at the resultant leaf node. This approach required the design and implementation of a number of components, including a query system (the state-action pair representation and query language used by the decision tree queries) and a quality criterion for the query selection policy (used to select queries when growing the decision trees).

This work applied decision tree features to Go, in order to evaluate board-move pairs for move prediction and playing strength comparisons. While we expect decision tree features to be more easily transferable to other domains (compared to tactical and pattern features), there is a lack of comparable results in other domains. In this application, the queries of the decision tree features evaluate a candidate move by examining the surrounding board

position. This work proposed and investigated two classes of query systems, with six variants in total, as well as twelve quality criteria. Decision tree features can be combined with other features, such as tactical features, to form a feature instance. These feature instances have a number of relevant settings: the query system(s) used, the quality criterion used, the number of trees in the decision forest (τ), the number of games used for growing the trees (ρ), and the number of games used for training weights (ϕ). The hypothesis was that feature instances with tactical and decision tree features (with select setting values) would be able to extract and use domain knowledge with comparable performance to the current state-of-the-art feature instances in computer Go, as measured according to move prediction and the playing strength of a computer Go engine.

In this work, feature instances that consist of tactical and decision tree features were extensively tested. Tests investigated the impact of the various feature instance settings and showed that they can greatly affect the move prediction performance of such feature instances. Testing showed that the strongest single query system is $\text{SG}_\emptyset$ (stone graph with no compression), but that a combination of $\text{SG}_\emptyset$ and $\text{IG}_\emptyset$ (intersection graph with no compression) is possibly marginally stronger. Testing also showed that the Weighted Win-Loss-Separate (WWLS) quality criteria tends to be the best quality criteria across the various query systems, in terms of move prediction. Increasing the other settings (τ, ρ and ϕ) tended to improve move prediction performance; however, the size of the decision forest was limited by the available time, and the amount of data used for weight training was limited by the available memory. Due to these computational limitations, many tests used smaller-than-optimal values. The results suggest that optimization of the implementation to allow larger setting values will result in improved results.

The results showed that, when tactical, pattern and decision tree features are used separately, decision tree features have better performance than tactical features, but slightly inferior performance than Go-specific pattern features. The best-performing tactical and decision tree feature instance was constructed with settings $\text{SG}_\emptyset$ + $\text{IG}_\emptyset$, WWLS, $\tau = 16$, $\rho = 3200$ and $\phi = 16000$. This feature instance was shown to have comparable performance, in terms of move prediction ($M(1) = 35.9\%$) and playing strength, to a state-of-the-art tactical and pattern feature instance ($M(1) = 38.4\%$). As such, it was shown that feature instances with decision tree features are a feasible alternative to the current state-of-the-art tactical and pattern features for computer Go, when used for move prediction or playing with a fixed number of playouts.

Furthermore, due to the relative ease of applying decision tree features to other domains, it is likely that decision tree features will be a feasible method

of incorporating domain knowledge in an automated method elsewhere — if the results with computer Go are applicable, then decision tree features will have better performance than hand-crafted features (such as tactical features) while requiring less expert knowledge, and inferior, yet comparable, performance than automated domain-specific features (such as pattern features) while being more easily applied to a new domain.

6.1 Recommendations

This section makes some recommendations for applying decision tree features to domains other than computer Go.

In order to apply decision tree features to another domain, an appropriate query system must be designed and implemented.[1] The testing in this work indicates that a variety of query systems (and their combinations) can be beneficial. In terms of the query selection policy, this work indicates that WWLS might be a good choice for other domains, and it should be relatively easy to explore the relative performance of the other proposed quality criteria.

In any application of decision tree features, the various settings will need to be tuned to find the optimal values. The testing in this work suggests that for most of the settings (such as τ, ρ and ϕ in this work), values corresponding to providing more training data will typically have better performance. However, it is likely that for any domain, these settings will be limited by the computational resources available. As such, an iterative greedy process that slowly ascends the multi-dimensional manifold describing the various settings should be a relatively simple method to find the optimal, or close to optimal, setting values that can be used, given limited computational resources. At each step of this process, multiple feature instances that each increase a single setting value can be constructed and evaluated, with the best-performing feature instance used in the next step. This process depends on the fact that, according to testing in this work, the performance in terms of each of the setting values is monotonically increasing, with diminishing returns.

6.2 Future Work

Based on this work and the results found, this section proposes a number of potential avenues for further work.

In order to be practically useful for computer Go (when the time per move is of concern), the implementation in this work can be profiled and

[1] Refer to Section 3.5 for details on applying decision tree features to other domains.

optimized. It is possible that such work will result in a system that will be able to complement a state-of-the-art system and together result in improved performance. Even if these efforts are not fruitful, it is possible that analysis of some of the more important decision tree feature levels will result in a number of implementable heuristics to augment current approaches.

While the the amount of training data in this work was limited by the available computational resources, alternative techniques for modelling and training weights for features might be able to use a much larger collection of training data. Future work could explore such possibilities.

This work showed that feature instances with decision tree features are a feasible alternative to state-of-the-art computer Go feature instances that are relatively well-established. It remains to be seen if other domains without such well-established techniques for including domain knowledge can benefit from the application of decision tree features. Such future work requires the design and implementation of a domain-specific query system, and the evaluation of performance in terms of action prediction.

Future work can also investigate the possibility of using multi-core and/or cluster parallelization for decision tree features. Possible parallelization applications include: growing a decision forest (with multiple processing nodes per tree), training weights, and other trivial applications such as action evaluation (by descending each tree in the decision forest in parallel).

Furthermore, in addition to the future work proposed above, there are a number of other minor avenues for future work, such as: adding a history-dependant element to decision tree features (possibly in a similar fashion to auxiliary nodes), considering query languages that are able to query the state-action pair in a more involved manner, limiting training and testing to separate game stages, and investigating other quality criteria for query selection.

Appendix A

Reproducibility

All source code used in this work is available in the codebase of OAKFOAM, an MCTS-based computer Go engine [37]. OAKFOAM is released under the open source BSD license. OAKFOAM version 0.2.0 was used for this work and is tagged in the code repository. Default parameters were used unless specified otherwise. The following links are relevant:

- Website: `http://oakfoam.com`
- Codebase: `http://bitbucket.org/francoisvn/oakfoam`

The MM tool of Rémi Coulom was used to train feature weights. This tool is available at: `http://remi.coulom.free.fr/Amsterdam2007/`. This tool was optimized by Detlef Schmicker, a contributor to the OAKFOAM engine, to allow larger data sets for training. These optimizations are also included in the OAKFOAM codebase, and are automatically applied when the `scripts/features/mm-fetch.sh` script is executed.

The following example commands illustrate the process required to generate a feature instance and measure its more prediction performance. Note that `$` merely signifies a user prompt.

```
$ sudo apt-get install g++ libboost-all-dev mercurial
$ hg clone http://bitbucket.org/francoisvn/oakfoam
$ cd oakfoam
$ ./configure
$ make
$ cd scripts/move-prediction/
$ cp example.test params.test
$ vi params.test
$ ./run.sh params.test
```

Bibliography

[1] K. Baker, *The Way to Go.* American Go Foundation, 1986.

[2] J. Conway, *On Numbers and Games.* Academic Press Inc, 1976.

[3] E. Berlekamp, J. Conway, and R. Guy, *Winning Ways for your Mathematical Plays.* A K Peters, 1982.

[4] E. Berlekamp and D. Wolfe, *Mathematical Go: Chilling Gets the Last Point.* A K Peters, 1994.

[5] B. Bouzy and T. Cazenave, "Computer Go: An AI oriented survey," *Artificial Intelligence*, vol. 132, pp. 39–103, Oct. 2001.

[6] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–49, 2012.

[7] A. Rimmel, O. Teytaud, C.-S. Lee, S.-J. Yen, M.-H. Wang, and S.-R. Tsai, "Current Frontiers in Computer Go," *IEEE Symposium on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 229–238, 2010.

[8] N. J. Nilsson, *Principles of Artificial Intelligence.* Tioga Publishing Company, 1980.

[9] R. A. Hearn, *Games, Puzzles, and Computation.* PhD thesis, Massachusetts Institute of Technology, 2006.

[10] G. M. J.-B. Chaslot, M. H. M. Winands, H. van den Herik, J. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte-Carlo tree search," *New Mathematics and Natural Computation*, vol. 4, no. 3, p. 343, 2008.

[11] R. Coulom, "Computing Elo Ratings of Move Patterns in the Game of Go," *ICGA Journal*, vol. 30, 2007.

[12] M. Müller, "Computer Go," *Artificial Intelligence*, vol. 134, pp. 145–179, Jan. 2002.

[13] S. Gelly and D. Silver, "Combining Online and Offline Knowledge in UCT," in *24th International Conference on Machine Learning*, pp. 273–280, ACM Press, 2007.

[14] G. M. J.-B. Chaslot, L. Chatriot, C. Fiter, S. Gelly, J. Perez, A. Rimmel, and O. Teytaud, "Combining expert, offline, transient and online knowledge in Monte-Carlo exploration," *IEEE Transactions on Computational Intelligence and AI in Games*, 2008.

[15] F. Van Niekerk and S. Kroon, "Decision Trees for Computer Go Features," in *Workshop on Computer Games at the International Joint Conference on Artificial Intelligence*, 2013.

[16] J. House, "Groups, liberties, and such." Computer-Go Mailing List Archive, `http://go.computer.free.fr/go-computer/msg08075.html`, 2005.

[17] N. Wedd, "Human-Computer Go Challenges." `http://www.computer-go.info/h-c/index.html`, 2013.

[18] R. Segal, "On the Scalability of Parallel UCT," in *Computers and Games*, pp. 36–47, Springer, 2011.

[19] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," in *17th European Conference on Machine Learning*, pp. 282–293, Springer, 2006.

[20] P. Baudiš and J.-L. Gailly, "Pachi: State of Art Open Source Go Program," *Advances in Computer Games*, vol. 7168, no. Lecture Notes in Computer Science, pp. 24–38, 2011.

[21] D. Stern, R. Herbrich, and T. Graepel, "Bayesian Pattern Ranking for Move Prediction in the Game of Go," *23rd International Conference on Machine Learning*, pp. 873–880, 2006.

[22] M. Wistuba and L. Schmidt-Thieme, "Move Prediction in Go - Modelling Feature Interactions Using Latent Factors," in *36th Annual German Conference on Artificial Intelligence*, 2013.

[23] Y. Wang and S. Gelly, "Modifications of UCT and sequence-like simulations for Monte-Carlo Go," *IEEE Symposium on Computational Intelligence and Games*, pp. 175–182, 2007.

[24] S.-C. Huang, R. Coulom, and S. Lin, "Monte-Carlo Simulation Balancing in Practice," in *International Conference on Computers and Games*, pp. 81–92, Springer, 2011.

[25] T. Wolf, "Basic Seki in Go," technical report, Department of Mathematics, Brock University, 2012.

[26] L. Ralaivola, L. Wu, and P. Baldi, "SVM and Pattern-Enriched Common Fate Graphs for the Game of Go," in *European Symposium on Artificial Neural Networks*, pp. 485–490, 2005.

[27] R. A. Bradley and M. E. Terry, "Rank analysis of incomplete block designs: I. The method of paired comparisons," *Biometrika*, vol. 39, no. 3, pp. 324–345, 1952.

[28] D. R. Hunter, "MM algorithms for generalized Bradley-Terry models," *Annals of Statistics*, vol. 32, no. 1, pp. 384–406, 2004.

[29] M. Wistuba, L. Schaefers, and M. Platzner, "Comparison of Bayesian move prediction systems for Computer Go," in *IEEE Conference on Computational Intelligence and Games*, pp. 91–99, Sept. 2012.

[30] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson, third ed., 2010.

[31] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is NP-complete," *Information Processing Letters*, vol. 5, no. 1, pp. 15 – 17, 1976.

[32] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[33] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[34] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[35] "Game records in SGF format." `http://www.u-go.net/gamerecords/`.

[36] "GNU Go." `http://www.gnu.org/software/gnugo/`.

[37] "Oakfoam." `http://oakfoam.com`.