Learning to Play the Game of Go

James Foulds

October 17, 2006

Abstract

The problem of creating a successful artificial intelligence game playing program for the game of Go represents an important milestone in the history of computer science, and provides an interesting domain for the development of both new and existing problem-solving methods. In particular, the problem of Go can be used as a benchmark for machine learning techniques.

Most commercial Go playing programs use rule-based expert systems, relying heavily on manually entered domain knowledge. Due to the complexity of strategy possible in the game, these programs can only play at an amateur level of skill. A more recent approach is to apply machine learning to the problem. Machine learning-based Go playing systems are currently weaker than the rule-based programs, but this is still an active area of research.

This project compares the performance of an extensive set of supervised machine learning algorithms in the context of learning from a set of features generated from the common fate graph – a graph representation of a Go playing board. The method is applied to a collection of life-and-death problems and to 9×9 games, using a variety of learning algorithms. A comparative study is performed to determine the effectiveness of each learning algorithm in this context.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The board game 'Go', also known as Wéiqí, or Baduk, is one of the oldest known games still played in its original form [32]. Ironically, despite its age and simplicity, Go is the last remaining major board game where the best artificial intelligence players perform only at the level of a weak amateur [29].

This report investigates the use of supervised machine learning techniques to learn a move evaluation function for Go, and presents a comparative study of the performance of various learning algorithms when applied to a particular approach to Go playing.

The focus of this report is to extend the work done by Graepel et al. [21], who introduce a graph representation of a Go board position, and apply supervised machine learning techniques to a set of features generated from this representation.

This report first provides an overview of the game of Go, including the history and rules of the game and a survey of existing approaches to artificial intelligence Go playing. Relevant material from [21] is summarized for convenience.

The results of a comparative study of supervised learning methods are then presented. The study evaluates the performance of a set of machine learning algorithms when learning a move evaluation function for Go, using the methods described in [21]. The performance of each algorithm is measured in terms of ability to choose moves that solve life-and-death problems, and score against an automated benchmark opponent in games of Go on a 9×9 playing board.

2 Background

In this section, the nature of the game of Go is described, including the rules of the game, the history, and the rudiments of game-playing strategy. Go puzzles (*tsumego*) are used as a benchmark for supervised learning algorithms in this project, therefore they are described briefly here. The methodology of supervised learning is then summarized, and a brief review of artificial intelligence approaches for Go is presented.

2.1 Go

Go is an ancient oriental strategy board game for two players. It can be considered to be very elegant, in the sense that its rules are extremely simple to state, but give rise to incredibly complex strategic possibilities.

The game is interesting from the point of view of artificial intelligence, as it represents a task where skilled humans are currently far superior to machines. The creation of an artificial intelligence Go-playing machine capable of beating human experts would mark an important milestone in the history of computer science, as Go is the last remaining popular strategy board game for which this has not been achieved.

2.1.1 Description of the Game

Go is classified as a deterministic, zero-sum, perfect information, two player strategy game [11]. This means that the outcome of the game is entirely determined by the moves made by the players, and a victory for one player necessarily equates to a loss for his or her opponent. Complete information about the current state of the game is available to both players at all times.

The players, labeled *Black* and *White*, take turns to place stones of their own colour on a 19×19 board (although beginners may practice with smaller 9×9 or 13×13 boards) called the *goban*. Stones are placed on the intersections of the grid lines, and cannot be moved once they are placed, only removed from the board when they are captured. Black always plays first. Players may also pass.

There is an additional rule, called *ko*, which disallows the placing of a stone that repeats a previous board position. In some variants of the rules, only the *last* board position before the current move must not be repeated, but in others no previous board position encountered in the current game may be repeated. This has been dubbed *super ko*.

Transitively adjacent sets of stones of the same colour are called *groups*. Synonyms include *strings*, *blocks* and *worms*. Each empty point adjacent to a group is called a *liberty*. After a player has placed a stone, any of her opponent's groups with no liberties are considered to be captured, and are hence removed from the board. If any of her own groups have no liberties after the removal of captured opponent groups, these are removed also.

The game ends when both players pass consecutively. Before calculating the score the players agree to remove *dead* groups from the board. These are groups that would inevitably be captured. If the status of a group is disputed, this is resolved by further play.

In the Chinese version of the rules, each player's score is calculated as the number of points of her colour on the board, plus the number of empty points enclosed by her colour. The Japanese rules count instead the number of opponent's stones captured plus the number of enclosed empty points. The two systems almost invariably produce the same outcome of a game.

An additional consideration in scoring is the concept of *komi*, which compensates the white player for the disadvantage of playing second by adding several stones to their score at the end of the game. The komi is predetermined before the game starts, and depending on the local version of the rules, can vary between around 4.5 stones and 8.5 stones.

2.1.2 The History of Go

Go was developed in China somewhere between 3000 and 4000 years ago [32]. The exact origins of the game are not known, although there are several competing legends. The most popular states that the game was invented by the Chinese emperor Yao, who reigned from 2357 to 2256 B.C. According to the legend, Yao invented the game in order to teach his dim-witted son, Shang Kiun, the concepts of discipline, concentration and balance. However, the game most likely has more humble origins.

In ancient Imperial China, Go was ranked as one of the Four Arts of the Chinese Scholar, along with calligraphy, painting, and playing the guqin, a seven-stringed instrument which is a relative of the zither.

Go gained an enormous amount of popularity when it gradually spread to Japan around 600-800 AD, which it still retains today [32]. In Japan, Go was considered to be of such importance that the government established state-sponsored academies for training and playing Go.

In the early 17th century, very soon after becoming Shogun and unifying Japan after one and a half centuries of internal conflict, Tokugawa Ieyasu instituted the Go Academy, called the "*Go In*". Honinbo Sansha, one of the most famous Go players of all time, was appointed as the head of the organization. The Academy was split into four houses, each with its own school and preferred playing style. The houses were sponsored by the Japanese government until the end of the Tokugawa Shogunate in 1868.

Go is now played all over the world, including several highly competitive international tournaments sponsored by large Asian companies such as Fujitsu, Toyota and Samsung.

An interesting recent development in the history of the game is the advent of internet Go. The Internet Go Server (IGS) [1], founded as early as 1992, was one of the earliest such servers, and is still probably the most popular. Another popular server is the Kiseido Go Server (KGS) [2]. Internet play has since become a popular phenomenon, with many more servers now operating. Thousands of players are competing around the world online at any one time. According to the website of IGS, a significant number of professional players compete on their servers regularly.

[1]www.pandanet.co.jp/English/
[2]www.gokgs.com/

Go has also become of interest to artificial intelligence researchers. Enormous difficulty has been encountered in creating a competitive artificial intelligence Go-playing program. In the last ten years, Go programs have become sophisticated enough to play respectable novice Go, but they still cannot even remotely approach a professional level of play. Some researchers, such as Russell and Norvig [29] believe that progress in computer Go will be beneficial to the artificial intelligence community in general.

2.1.3 Elementary Strategy

Important short term goals in Go include killing opponent groups, and creating stable groups of one's own. One must strike a balance between these sometimes contradictory aims.

An important concept for creating stable groups is the notion of an *eye*. An eye is an empty space surrounded by pieces of one colour that are all part of the same group. The term usually refers to single-point eyes, but can also refer to situations where the enclosed empty space contains more than one point.

A group with an eye is harder to capture because once it has been completely surrounded externally, the player must play an additional move on the eye to complete the capture. Groups with two eyes are unconditionally *alive*. Because only one stone may be played per turn, it is impossible to fill in both eyes in order to deprive the group of all of its liberties.

2.1.4 Player Rankings and Handicaps

There is a sophisticated system for rating and ranking players in Go. This is useful because it provides us with a precise measure of the strength of a player, at least relative to her peers.

The system is very accurate in describing the relative abilities of human players – a player with only a slightly higher rating can consistently beat her opponent without a handicap. While the rating system is somewhat useful in rating the ability of artificial intelligence Go-playing programs, there is a problem in that a competent human player who knows the weaknesses of an otherwise strong computerized opponent may be able to beat it even with an extremely high handicap.

The rating system is closely linked to the handicapping system of the game. The handicapping system is as follows. A stronger player may allow a weaker player to place extra stones on the board in a prescribed pattern before taking her first turn. The number of handicap stones required to make the game even is the measure of difference in strength between the two players.

There are separate scales of rating for amateur and professional players.

Weak amateurs are given a rank from 1-*kyu* upwards, where higher numbers mean less ability. Absolute beginners are generally placed somewhere between 20-*kyu* and 30-*kyu*.

There is a theoretical difference of a one stone handicap between each consecutive rating. For instance, a 5-*kyu* player would give a 7-*kyu* player two stones handicap in order to have an even game. Ratings can be determined by playing against others whose ranks are known, or are sometimes awarded by national Go organizations. Unfortunately, the exact strength of a given rating varies slightly over time and geographical location.

A player who can consistently beat a 1-*kyu* opponent with a single stone handicap is given the rank of amateur 1-*dan* (*shodan*). The *dan* scale is similar to the *kyu* scale, except that higher numbers are better. The amateur *dan* scale ranges from 1-*dan* to 7-*dan*.

Professional players are also given a *dan* rating, but the rating system is distinct from the amateur scale. The professional *dan* scale ranges from 1-*dan* to 9-*dan*.

2.1.5 Tsumego

Go puzzles are called *tsumego*, meaning literally "packaged Go". They serve a similar function to chess puzzles, and are often featured in Japanese newspapers. Tsumego problems are less contrived and artificial than their chess counterparts however, and often represent situations that can arise in actual games [8]. In this report, tsumego puzzles are used as a benchmark for the evaluation of machine learning techniques.

In a tsumego puzzle, a board position, or partial board position, is given, along with the goal of the problem. The goal is always either to try to find a sequence of moves that guarantees the survival of one's own group, or a sequence of moves that guarantees the death of another group.

The concepts of life and death are very fundamental to Go, and the purpose of tsumego puzzles is to teach players how to recognize these situations. The life and death status of groups must be determined in order to calculate the score at the end of the game. They are also very important to game playing strategy, as players need to know which groups to protect, and which are already lost.

2.2 Artificial Intelligence

Artificial intelligence (AI) is difficult to define precisely due to the differing paradigms existing within the field. Russell and Norvig [29] identify four different definitions, varying in two different dimensions. The definitions all agree that the goal of AI is to build specific types of systems, but the nature of these systems appears to be controversial.

Russell and Norvig state that each definition describes these ideal systems as *(a)* rational, or *(b)* similar to humans, and the success of the system is measured in terms of actions or thoughts. So the four possibilities are: artificial intelligence is the science of building systems that {think like humans, think rationally, behave like humans, behave rationally}.

In practice, the field of AI encompasses all four definitions. In this report, we are specifically interested in systems that act rationally. An ideal Go-playing system should make decisions that are optimal, regardless of whether humans would have selected the same responses. Such a system should be evaluated in terms of its behaviour – its ability to *win* is more important as an end result than the inner processes, mental or otherwise, that produce such behaviour. Of course, if successful, the processes used by such a system are likely to be of interest in themselves.

2.3 Supervised Learning

This report investigates the use of supervised learning as an approach to Go-playing. Supervised learning is a type of machine learning where a function is learned from a set of examples [29]. The typical supervised learning scenario involves a set of training examples that an agent is to learn from. Each example consists of a vector of *features* that represent the properties of the example, and a *class*. The class is the output value of the function for that example.

A supervised learning algorithm attempts to learn a function that maps an instance from feature space to a class value. It must generalize from the training data to predict the output of the function for any arbitrary example. If the class is of a nominal type, this process is called *classification*. In the case where the class is of a numeric type, the learning process is called *regression*.

2.4 Artificial Intelligence for Game Playing

Artificial intelligence techniques have been applied successfully to most popular strategy board games. The most famous example is Deep Blue [9], which defeated Garry Kasparov, the then-reigning World Chess Champion in a six-game match in 1997.

Deep Blue is the most recent, and almost certainly final, incarnation of IBM's chess-playing computer. It uses an optimized version of the minimax algorithm, coupled with a sophisticated evaluation function in a massively parallel computing environment in order to generate its moves. The system consists of 30 IBM RS/6000 processors and 480 custom single-chip processors.

Minimax is a game-playing algorithm where a search is done on the tree of possible sequences of moves [29], called the game tree. Moves are chosen assuming that each player plays optimally. If the entire game tree can be traversed by the algorithm, optimal moves are found by considering the outcome at the end of the game in a given branch, and propagating this information back up the tree.

In practice, however, for interesting games such as chess there are too many possible sequences of moves to investigate the entire game tree. In this case, a cut-off function is used to determine when to terminate the search. At this point, the result of the minimax search is estimated using a board evaluation function.

Minimax can be optimized by careful pruning, and sophisticated cut-off tests and evaluation functions. Alpha-beta pruning is a method for pruning the search tree without affecting the result of the minimax search. Other pruning methods are possible, but they may not guarantee that optimal moves are found. Quiescence search is a method where the evaluation function is not applied to highly unstable game positions, and the search is continued until quiescent positions are reached.

The authors of Deep Blue state that it uses a "highly non-uniform" search – i.e. it uses a quiescence search, and some branches of the game tree are explored much more deeply than others.

The minimax approach has been applied successfully to many other games, creating world champion artificial intelligence players for checkers [30], Qubic [25] (3-dimensional tic-tac-toe on a 4 x 4 x 4 board) , Go-Moku [2](connect 5, traditionally played on a Go board with Go playing stones) and Nine-Men's Morris [19]. In most of these champion game-playing programs, end-game databases are precomputed in order to terminate the search sooner.

Applying brute-force searching algorithms is not always enough, however. When the depth of search space is limited, an evaluation function must be carefully chosen. Machine learning has often been applied successfully to this problem.

In 1997, three months after Deep Blue defeated Kasparov, an Othello playing program called Logistello [7] defeated Takeshi Murakami, the world champion at Othello, with six victories to none. Logistello uses a variant of alpha-beta minimax called *ProbCut*, which uses supervised learning techniques to find heuristics for reducing the search space. It also learns both an evaluation function and an opening book.

Backgammon also required some extra inspiration beyond mere brute-force. Since backgammon includes an element of chance, the opponent's move at a given time cannot be unambiguously predicted. Hence, the subtrees for all possible outcomes must be investigated in order to determine the expected value of each move. Because of this increase in the branching factor of the game tree, deep searches are much more computationally expensive.

To counter this, much work was done to find an accurate board evaluation function [29]. A program called BKG [3] won against the world champion of the year, Luigi Villa, as early as 1979. BKG uses a complex hand-coded evaluation function and only searches one ply of the game tree. However, this was only a short exhibition match, and not a world championship match. Its author admits that the program made several mistakes in the game, but was lucky with the dice.

In the end, it was a machine learning program that solved the problem. A program called TD-Gammon [33] learnt a neural network for board evaluation using a reinforcement learning method called *temporal difference learning* (TD). By playing against itself more than a million times, it became strong enough to be consistently ranked as one of the top three players in the world. TD-Gammon plays at a level far superior to BKG.

2.5 Artificial Intelligence for Go

Much work has been done towards creating a successful artificial intelligence Go program. Many of the techniques that were used to create programs that play at a world-class level in other games have been applied to Go with limited success. In this section we describe the challenges that Go presents for an artificial intelligence program. We give an overview of some of the methods that have been applied to the problem, and describe the current state of the art.

2.5.1 The Problem with Minimax

Given the success of AI programs using the minimax algorithm in other games, it may come as a surprise that this success does not transfer to Go.

One of the main reasons for this is the branching factor of the game tree. For instance, at the beginning of the game there are $19 \times 19 = 361$ possible legal moves. While this reduces as the game progresses, for most of the game the branching factor remains far higher than for all of the games mentioned previously. For instance, in chess the average number of legal moves is around 35. This can be reduced to a branching factor of around 3 after heuristics have been applied to prune low quality moves[29].

Another reason for the poor performance of minimax is the difficulty of writing a good board evaluation function [24]. The success of the TD-Gammon program in backgammon demonstrates that it is possible to build a strong player for a complex game where the search space is very large, as long as a good enough evaluation function can be found. Hence, in theory such a function could circumvent the problem of the large branching factor in Go. Such a function for Go has proved elusive as of yet.

A further problem for a minimax solution to Go is the lack of a clear-cut end to the game. Unlike in chess, where a checkmate is a well-defined scenario that determines the end of the game, in Go both players must agree to finish the game by consecutively passing. Few games of Go are played until the board is completely filled in; they are instead ended when both players can agree on the outcome, which is usually much earlier. A minimax search is thus much more reliant on its evaluation function, since a given game is unlikely to get close enough to the end of the game to allow a complete search of the remaining portion of the game tree.

2.5.2 Other Challenges

At a high level of play, Go is very unforgiving of mistakes. Although the best Go programs generally make reasonable moves most of the time, they also make some crucial blunders that are likely to cost them the game [24].

The length of the game is also a factor. As each game can last for up to around 300 moves, a human opponent has a lot longer to learn the weaknesses of a computer program. In some cases, a human that knows how to exploit the weaknesses of an otherwise competent program can win decisively, despite giving the program a huge handicap.

These difficulties have meant that computer scientists have not been able to develop AI players that are competitive against humans at an advanced level of play.

2.5.3 Board Evaluation Functions vs Move Evaluation Functions

Until this point, we have only considered one class of evaluation function, known as a board evaluation function. Such a function is a mapping from a board position to a numeric value, typically between 1 and -1, indicating the predicted outcome of the game from that position. Board evaluation functions are useful when applying minimax search, because they allow the search to terminate early and instead estimate the result produced by the remainder of the game tree.

An alternative approach is to evaluate *moves* instead of board positions. In this case, each legal move is mapped to a numeric value representing the quality of the move. A program would then select the highest ranked move to play.

In Go, move evaluation functions are often more appropriate, because minimax is generally not used as the core of the program's decision-making process, and good moves are, after all, what we are fundamentally interested in finding. In practice, many programs use both move and board evaluation functions [24].

2.5.4 Current Go Programs – the State of the Art

As many of the current top programs are commercial or private software, it is not always easy to determine the techniques used. However, it is probably safe to assume that these programs are built using similar methods to their public contemporaries.

In any case, the winner of the Go event at the most recent annual Computer Olympiad, held in June 2006, was GNU Go[3], an open source project from the Free Software Foundation. This competition is the most prominent competition in computer Go. Other important tournaments include the Kiseido Go Server tournaments and the Computer Go Ladder.

The best Go programs existing currently, such as GNU Go, Go++[4], and The Many Faces of Go [17] rely heavily on large internal databases of hand-coded domain knowledge. They apply localized alpha-beta minimax searches, rule-based expert systems and pattern matching. Müller states in a 2002 review article [24] that "Most competitive programs have required 5-15 person-years of effort, and contain 50-100 modules dealing with different aspects of the game".

The general approach of most of the current top programs is an inversion of the traditional minimax scenario. For a practical implementation of minimax, the game tree is searched, and an evaluation function is applied when we can search no more. In Go, small localized searches are often applied as part of the evaluation function, rather than the other way around.

Most modern Go programs, such as GNU Go and The Many Faces of Go, produce a set of candidate moves using a wide variety of methods such as pattern matching and hand-coded domain information heuristics. Candidate move sets are necessary because of the sheer number of possible moves on the board – their application allows more time to closely investigate the most promising moves. These candidate moves are then evaluated and the best ranking move is selected.

2.5.5 Pattern Matching

All of the top existing Go programs have an internal database of patterns. These are localized positions of part of a Go board, combined with information per-

[3]Publicly available from http://www.gnu.org/software/gnugo/

[4]A commercial program that was the winner of the 9th KGS Computer Go Tournament in 2005, and the Computer Olympiad in 2002. While its closed-source nature means that its details are not publicly available, its website at www.goplusplus.com mentions some very high level information about the techniques the program uses.

taining to that position such as a suggested move, or sequence of moves, further preconditions for the applicability of the pattern, goals associated with the pattern and a measure of importance. Patterns are invariant under the symmetries of rotation, reflection and inversion of colour. Human players often train by studying patterns of a very similar nature, called *joseki* in Go terminology.

Patterns can be added manually by experts, or discovered by machine learning techniques [10]. The Go++ program uses a pattern database containing over 23,000 hand-made patterns, and an additional 300,000 patterns that were generated automatically from professional games. The program's author employs a 6-*dan* Japanese player full-time to add new patterns into the system. The pattern acquisition method used by Go++ has not been released into the public domain, but the program's website hints that machine learning is involved.

While pattern matching is an important component of virtually every major Go program, this method alone is not enough to make a strong player. For instance, the program "Wally" is a very simple player that is based almost entirely around pattern matching. Apart from pattern matching, its only rules are to capture whenever it can, and never play suicide moves. If there is no pattern that it knows, it plays a random move.

Wally's playing strength is estimated to be only around 30-*kyu*, which is the rating of an extreme beginner human player. It does however play a non-trivial and consistent game.

2.5.6 Localized Goal-Directed Search

Although it is not feasible to perform a total minimax search on all of the available branches of the game tree, small localized searches are helpful in several situations. The search space is cut down immensely when we are only considering points relevant to a given high-level goal on small portions of the board.

For instance, GNU Go uses minimax to detect *dragons* – sets of stones that are provably connectable despite any opposition attempt to separate them. It also uses minimax to determine the life-and-death status of groups.

Cazenave [10] uses a minimax search to evaluate candidate patterns. His program, GoGol, acquires patterns for its pattern rule set by enumerating candidate patterns based on general templates, and then evaluates them with respect to simple goals such as making an eye. It performs minimax to determine if the goals can be achieved.

2.5.7 Monte Carlo Methods

Monte Carlo methods are non-deterministic stochastic methods for simulating the behaviour of a system. The approach is to repeatedly generate randomized scenarios and evaluate their outcomes. They are often used when state information is unknown, or when other methods are infeasible due to computational complexity. They have applications in many areas such as finance, physics, and gambling (in fact, the name is a reference to the casino in Monte Carlo). In the field of game-playing AI, Monte Carlo methods have been successfully applied to several games such as poker, bridge and backgammon.

In 1993, Brügmann [6] created the first Go program based around this technique. His system, called Gobble, evaluates each possible move by repeatedly finishing the game via randomized play, and averaging the results to obtain an expected outcome. A random sequence of moves for each player is generated in advance, based on current move value estimates, and the score is calculated at the end of the game.

Moves are evaluated based on the score for any game in which they were played, no matter when in the game they were played. Simulated annealing is used to control the probability that a move is played out of order. Despite having almost no explicit domain knowledge built into the program, Gobble plays "respectable novice go"[6] at a level of about 25-*kyu*.

Bouzy and Helmstetter [4] found that simulated annealing was not significantly superior to using a fixed temperature. Monte Carlo Go players play a respectable strategic game, but perform poorly at lower level tactics [24]. Bouzy and Helmstetter tried to solve the tactics issue by starting the random games with a one-ply minimax search, using the Monte Carlo method as the evaluation function. No improvement was found with this method, however.

2.5.8 Machine Learning Approaches

The top programs currently do not make much use of machine learning techniques. However, a new approach is needed in order to make progress, and machine learning techniques, like Monte Carlo methods, are attractive alternatives to the manual creation of domain knowledge playing heuristics. This is an active area of research.

Reinforcement learning is a common approach, for example [1]. In contrast to supervised learning, where a set of pre-labeled training examples are provided to a learning agent, reinforcement learning provides an agent with feedback responses to its actions as it interacts with its environment.

In particular, the temporal difference (TD) reinforcement learning algorithm has been frequently tried in Go-playing applications (see, for example, [28], [22] and [12]). The success of TD in backgammon, a complex game that has resisted other artificial intelligence approaches, led to the hope that it will prove useful when applied to the game of Go. However, like all other approaches, it has produced only lukewarm results in Go [22].

Neural networks have been used, often in conjunction with TD learning. Dahl [14] trains a neural network to learn strong local board position shapes with supervised learning, and also trains two additional neural networks by self-play using TD learning, to estimate the safety of group structures, and value the territorial potential of unoccupied points. Shell et al. [31] use genetic algorithms to train a neural network to play the Capture Game, a version of Go with simplified objectives.

Enzenberger [16] uses a neural network, trained via self-play and TD learning, to segment a game position into smaller independent subgames to simplify position evaluation. Russell and Norvig [29] suggest that work on such segmentation may have important repercussions for the AI community in general.

A literature survey shows surprisingly little research into supervised learning in the game of Go. Graepel et al. [21] try to learn a move evaluation function using a support vector machine and a kernel perceptron. They also introduce a novel representation for board positions, which they call a "common fate graph" (CFG). The CFG is a graph-based representation that takes into account the fact that connected stones share a common fate - they live or die in the same circumstances, and hence can be treated as a single unit. Feature vectors are extracted from the CFG representation. This is the method investigated in this report.

3 A Supervised Learning Approach to Go

Graepel et al. [21] presented a new representation for a Go board position, and accompanied this with a method for extracting feature vectors from this representation, suitable for use with supervised learning techniques. This project is an extension of that work. For convenience, their work is summarized here.

3.1 Common Fate Graph Representation

The *common fate graph* (CFG) is a graph representation of a Go board position. It is founded on the observation that adjacent stones of the same colour have a common fate - if one stone is captured, then all of the friendly stones belonging to the same group must be captured also. Hence, all of these connected stones can be conveniently represented as a single node in a graph representation of a position.

Consider the simplest possible graph representation of a Go board position. In this model, a Go position is represented by 19×19 nodes, representing the 361 intersections on the board. Each node stores either the colour of a stone at

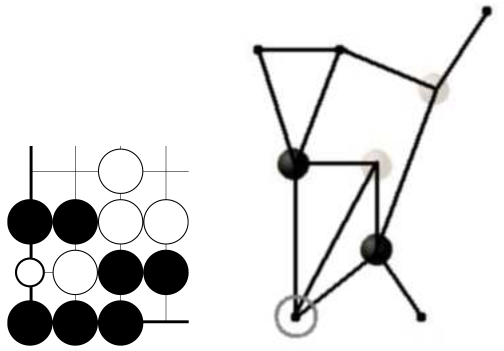

Figure 1: Example board position, and corresponding CFG. The points marked with a circle on each image correspond with each other.

that intersection, if there is such a stone, or the fact that it is empty. Each node in the graph is adjacent to its geographical neighbors above, below and to its sides. We will follow [21], and refer to this as the *naïve full graph representation.*

The CFG is built from the naïve full graph representation by repeatedly merging all non-empty nodes with adjacent nodes of the same colour, until there is no non-empty node of the graph adjacent to another node of the same colour. This is the defining feature of the representation.

Formally, let $G = (E, V)$ be the näive full graph representation of a Go position with edges E and vertices V. While $\exists p_1, p_2 \in G$ such that $\{p_1, p_2\} \in E, p_1.\mathit{colour} = p_2.\mathit{colour}$ and $p_1.\mathit{colour} \neq \mathit{empty}$, merge them using the following transformation:

$$V \mapsto V \setminus \{p_2\} \tag{1}$$

$$E \mapsto (E \setminus \{\{p_2, p_3\} \in E\}) \cup \{\{p_1, p_3\} : \{p_2, p_3\} \in E\} \tag{2}$$

See Figure 1 for an example board position and the corresponding CFG.

The CFG is a useful representation because it abstracts from the notion of 'stone' to the notion of 'group', which is the level of abstraction at which competent human players generally reason about Go positions. It helps to capture the structure of the board position, as it represents the adjacency relationships between the different groups, which indicates the level of interaction, in terms of attack and defense patterns, surrounding and counter-surrounding.

There are limitations, though. The representation is 'lossy' in the sense that information about the size and shape of the groups is not retained in the CFG representation. Every full graph position maps to exactly one CFG, but each CFG could map to many full graph positions.

Graph	Count
•	0
●	2
○	1
• •	0
• ●	0
• ○	0
● •	3
● ○	4

Figure 2: The first few subgraph features extracted from the board position in Figure 1, relative to the intersection marked with a circle.

3.2 Relative Subgraph Features

Graepel et al. extract feature vectors from their CFG representation by counting the occurrences of different types of subgraphs starting from any particular empty node in the graph. They pick a maximum subgraph length, s, and enumerate all of the different possible subgraphs of a CFG of length s or less.

Here the notion of "subgraph" is restricted to connected subgraphs with the shape of chains without branches or loops. To meet this specialized definition, there must be a path from any node to any other node in the graph, every node must be adjacent to at most two other nodes, and the graph must be acyclic. Hence, the notion of length makes sense – it is defined to be the order of the subgraph. Throughout the remainder of this document, the term "subgraph" will refer to this specific restricted class of subgraphs unless stated otherwise.

Given an *empty* intersection on the board, which corresponds to the move of placing a stone at that intersection, Graepel et al. count the number of occurrences of each possible type of subgraph of length $\leq s$ encountered, starting from the nodes adjacent to that point. The original *empty* intersection is not counted, and cannot form part of the subgraph. The feature vector is just the vector of the counts of the different subgraphs.

If a real-valued or discrete class label representing the quality of a move at that point can be appended to this feature vector, it can be used as part of a training set for any standard supervised machine learning algorithm.

For an example of a relative subgraph feature vector, see the example given in Figure 2. The features have been extracted from the CFG shown in Figure 1, relative to the empty intersection marked with a circle. This feature set corresponds to the move of placing a stone on that intersection.

3.3 Tsumego Problems

Graepel et al. apply their relative subgraph feature extraction method to tsumego problems. Tsumego problems are Go-playing puzzles, with a similar function to chess puzzles. A board position is given, and the player must determine a sequence of moves in order to surround an opponent's group, or prevent her own groups from dying. Tsumego problems are described more completely in Section 2.1.5.

Graepel et al. take a set of tsumego problems generated by Thomas Wolf's GoTools program [35], and learn a move-evaluation function using a support vector machine, and also using a kernel perceptron, two popular supervised learning techniques. They found that both algorithms had approximately a 65% success rate at giving the highest evaluation to one of the moves selected by GoTools.

As both the support vector machine and the kernel perceptron produced similar results, the authors conjectured that "...the (common fate graph) representation is so crucial that the difference between the learning algorithms becomes negligible". The results of a more comprehensive experiment conducted for this report using many more learning algorithms (described in Section 5) show that this is not the case, however.

3.4 Playing Go

Graepel et al. also use the same training paradigm to learn a move evaluation function for a simple AI Go-playing program. The training examples were acquired from a collection of 9×9 internet Go games that were collected and preprocessed by Nicol Schraudolph. The relative subgraph features were extracted for each move in the sample games. Each resulting feature vector was labeled as a "good" move. For each good move, a random legal move was selected as a "bad" move example. The program always selects the highest ranking legal move according to the predictions of the learning algorithm.

The resulting player was not able to beat GNU Go, but showed some promise and successfully managed to threaten its opponent's pieces (*atari*) and even make captures in some games. Graepel et al. suggest that their method may eventually be used as a *move generator* heuristic for selecting which moves to consider for further evaluation, as part of a broader strategy.

As mentioned before, programs such as GNU Go use a wide variety of move generator heuristics in order to select a small set of promising moves to examine thoroughly. This method allows more time to be spent evaluating moves that are likely to be played, rather than wasting time by examining every possible legal move on the board.

4 An Implementation of the Relative Subgraph Feature Extraction Algorithm

In order to perform benchmark experiments with a set of machine learning algorithms on tsumego problems and 9×9 games, it was first necessary to implement the software for creating the common fate graph representation and extracting the relative subgraph features. A Go-playing engine capable of communicating with a graphical user interface and other Go programs was also built. The design of the system and the algorithms used are presented here.

4.1 Implementation Issues

The relative subgraph feature extraction system, and the tsumego-solver and Go-playing engine were implemented using Sun Java version 1.5.0.

Graphs are represented by a class named *CFG*, which is implemented as an adjacency list [20] data structure. This allows for fast access to the sisters of a given node, while still allowing for easy modification to the structure. The design of the system is described in Appendix C.

4.2 Building Common Fate Graphs

To build the common fate graph representation of a board position, the 2-dimensional array representing the board position is first converted into the naïve full graph representation by creating a 2-dimensional array of nodes with the same dimensions as the board array. The colours are set appropriately, and every node is connected to its vertical and horizontal neighbours.

The procedure used for transforming the naïve full graph representation into a common fate graph is an iterative one. In a loop, we examine all vertices in the graph. Each non-empty vertex is compared with its neighbours, looking for nodes of the same colour. Each neighbour with the same colour is merged into the current node.

When an iteration produces no change to the model, the process can be terminated. Pseudocode for the algorithm is provided in Algorithm 1.

4.3 Extracting Relative Subgraph Features

From the common fate graph representation, the relative subgraph features are computed using a simple recursive depth first search with cycle detection.

The node coloured *empty*, which we will label p, for which the subgraph features being computed are relative to, is marked as ‘*visited*’. A counter for the remaining search depth is set to s, the maximum subgraph length. The

Algorithm 1 Transform naïve graph representation into common fate graph

```
$G \Leftarrow$ Naïve graph representation
repeat
  $somethingChanged \Leftarrow false$
  for all $v$ in $G$ do
    if $v.colour = empty$ then
      continue
    end if
    for all $e$ in edges($v$) do
      if $e.other.colour = v.colour$ then
        merge($v$, $e.other$)
        $somethingChanged = true$
      end if
    end for
  end for
until $somethingChanged = false$
```

feature vector, *counts*, is initialized as an empty array of length s. Then a depth first search is applied to all of the neighbours of p, using the procedure described in Algorithm 2.

Algorithm 2 Count subgraph features

```
Procedure countSubgraphs(depthRemaining, currentNode, subgraphCounts,
currentGraphString)
  if depthRemaining = 0 then
    return
  end if
  currentGraphString = currentGraphString + currentNode.getColour()
  subgraphCounts[subgraphIndex(currentGraphString)]++
  currentNode.setVisited(true)
  for all outEdge $\in$ currentNode.getEdges() do
    other = outEdge.getOtherNode(currentNode)
    if other.visited() = false then
      countSubgraphs(depthRemaining - 1, other, subgraphCounts,
          currentGraphString)
    end if
  end for
  currentNode.setVisited(false)
```

4.4 Enumerating Subgraphs

In order to implement the extraction of relative subgraph features from a common fate graph representation of a board position, it is necessary to uniquely identify each different possible subgraph of length $\leq s$, the chosen maximal length. Furthermore, an ordering must be imposed on the subgraph types in order to consistently define a feature vector.

In practice, this is implemented as an index into an array of integers representing the counts of each subgraph type. When a subgraph is encountered, we must somehow determine the index corresponding to the subgraph of that type so that the count of that subgraph type may be incremented.

Clearly, what is required is an enumeration of the subgraph types, i.e. a bijective mapping $f\colon G \to \mathbb{N}$, where G is the set of all subgraph types and $\mathbb{N}$ is the set of natural numbers.

Unfortunately, Graepel et al. do not describe such an enumeration. Hence, a new method for enumerating subgraphs, presented here, was developed during the course of this project.

To simplify our terminology, without loss of generality we will view each subgraph as a string over the alphabet $A = \{0, 1, 2\}$. 0 represents an *empty* coloured node in the subgraph, and 1 and 2 represent *black* and *white* coloured nodes respectively. The mapping between strings and subgraphs is defined in the obvious way.

For instance, the string "010012" represents a subgraph which contains an empty node, followed by a black node, two consecutive empty nodes, another black node and finally a white node. Clearly, the mapping between strings and subgraphs is a bijection. Because the two notions are equivalent, the terms will sometimes be used interchangeably, as long as the meaning is clear.

The order of the subgraphs in the enumeration sequence is defined to be the lexicographical order of the corresponding strings. Hence, the first few entries in the sequence are the subgraphs corresponding to: $\{$0, 1, 2, 00, 01, 02, 10,12, 21, $\ldots\}$. Note that there is no "11" entry – this is because adjacent non-*empty* coloured nodes are merged together during the creation of the common fate graph. Such subgraphs cannot occur, so they are omitted from the sequence. Also note that the empty subgraph is not included.

The ordering of the subgraphs has now been described precisely, so the enumeration function is completely defined. A little more work is needed to make this useful in practice, however.

We would like to be able to efficiently evaluate the enumeration function $f(g)$ for a given $g \in G$ without having to visit every single subgraph that exists earlier in the sequence.

A recurrence relation describing the number of subgraphs of a given length is required. Consider the relation $p\colon \mathbb{N} \to \mathbb{N}$, where $p(n)$ is defined to be the number of subgraphs (strings) of length n or less. Then we have:

Lemma 4.1. $p(n) = \begin{cases} 0 & \text{if } n = 0; \\ 3 & \text{if } n = 1; \\ 10 & \text{if } n = 2; \\ 3p(n-1) - p(n-2) - p(n-3) & \text{if } n \geq 3. \end{cases}$

Proof. It is easy to verify the cases where $n \leq 3$ by simply counting the strings. We just need to show that the lemma is true for the case where $p \geq 3$.

Let α be a string of length k, ending in a '0'. Then α has three successors of length $k+1$: $\alpha 0, \alpha 1$, and $\alpha 2$. Also, if a string β does not end in a 0, it has two successors: $\beta 0$, and $\beta 2$ if β ends in a 1, otherwise $\beta 1$.

So the number of strings of a given length is three times the number of "*zero enders*" of the previous length plus two times the non-zero enders of the previous length. Let $M_k = \{\alpha 0 : \alpha \in G, length(\alpha) = k-1\}$ be the set of zero enders of length k, and $N_k = \{\beta[1|2] : \beta \in G, length(\beta) = k-1\}$ the set of non-zero enders of length k. So we have:

$$p(n) = p(n-1) + 3|M_{n-1}| + 2|N_{n-1}| \tag{3}$$

Also notice that every zero ender $\alpha 0$ of length k has a predecessor α of length $k-1$. Conversely, every such α of length $k-1$ has a zero ender successor. So the number of zero enders of length k is precisely the number of strings of length $k-1$, which is equal to $p(k-1) - p(k-2)$. The non-zero enders form the remaining strings of length k. Then

$$\begin{aligned} |M_k| &= p(k-1) - p(k-2) && (4) \\ |N_k| &= (p(k) - p(k-1)) - |M_k| \\ \Leftrightarrow |N_k| &= (p(k) - p(k-1)) - (p(k-1) - p(k-2)) \\ \Leftrightarrow |N_k| &= p(k) - 2p(k-1) + p(k-2) && (5) \end{aligned}$$

Substituting into Equation 3, we get

$$\begin{aligned} p(n) &= p(n-1) + 3(p(n-2) - p(n-3)) \\ &\quad + 2(p(n-1) - 2p(n-2) + p(n-3)) \\ \Leftrightarrow p(n) &= p(n-1) + 3p(n-2) - 3p(n-3) \\ &\quad + 2p(n-1) - 4p(n-2) + 2p(n-3) \\ \Leftrightarrow p(n) &= 3p(n-1) - p(n-2) - p(n-3) \end{aligned}$$

□

Lemma 4.1 is very useful in computing $f(g)$. Note that instead of attempting to solve this recurrence relation, it is sufficient to compute the required values of the sequence in a bottom-up fashion and store them in a lookup table.

Let $H_g = \{i : i \in G, length(i) = length(g), i \prec g\}$ be the set of strings of the same length as g, that precede g in the enumeration sequence. Then

$$f(g) = p(length(g) - 1) + |H_g| \tag{6}$$

Consider the elements of H_g. As each such element is ordered lexicographically earlier than g, it must begin with a prefix in which every character strictly precedes the corresponding character in g. For example, assume $g =$"120". Then H_g is the set of strings of length 3 which begin with the prefixes "0", "00", "01", "10", "11". The strings beginning with "00" and "01" are already covered by strings starting with "0", so we can drop them from our list of prefixes while retaining complete coverage of H_g.

In order to count the elements of H_g, we iterate through the prefix lengths, from 1 to $length(g)$. At each prefix length, we only need to consider the case where all but the last letter of the prefix are identical to the corresponding letters of g – all other elements were counted earlier in the process. We consider the possible options for the last letter of the prefix, where the letter is strictly less than the corresponding letter in g, but can legally follow the second to last letter of the prefix, if any. There are only ever at most two such options.

For the prefix under consideration, we add to our running total the number of strings with the same length as g beginning with this prefix. The last letter of the prefix, and the length of the remaining substring completely determine this number.

If the last character of the prefix is a '0', any string of the correct length may follow – and we can use the recurrence relation to find the number of these strings. If the last character is not a '0', the possible number of strings is equal to the total number of strings whose length k is $length(g) - length(prefix)$ minus the zero-starter strings (there are $p(k-1) - p(k-2)$ of them), divided by two since in this case, the non-zero last character of the prefix cannot be followed the same symbol.

An example may help to illustrate this. As before, let g be the subgraph represented by the string "120". We want to find $f(g)$. First, from our recurrence relation lookup table we know that $p(2) = 10$. In order to apply Equation 6, we now only need to calculate $|H_g|$.

At prefix length one, we must only consider the prefix "0". The string "1" is not strictly less than the first character of g, so we discard it. There are $p(2) - p(1) = 7$ strings of length two which may be preceded by a '0'.

Now the prefixes of length two must be examined. We only need to consider legal prefixes which start with the first character of g, and whose second character is lexicographically less than the second character of g. The only such prefix is"10". There are 3 strings of length one that can follow a '0'.

So $|H_g| = 7 + 3 = 10$.

Hence, $f(g) = 10 + 10 = 20$. Manually computing the sequence of possible strings confirms that g is in fact the $21st$ element in the sequence, as expected (recall that the enumeration sequence starts from zero).

5 Tsumego Experiment

A comparative study of a large set of supervised machine learning algorithms applied to tsumego problems was performed. This study compares the performance of a set of machine learning algorithms in learning a move evaluation function for solving tsumego problems. In this section, the experimental setup is described, and the results of the study are presented.

5.1 Experimental Setup

The experimental setup is similar to that used by [21]. The dataset used is a subset of a database of around 40,000 tsumego problems and solutions generated by Thomas Wolf's program GoTools.

GoTools solves tsumego problems with a combination of domain-knowledge heuristics and alpha-beta minimax search [35]. It has been rated by top amateur players to have a strength of 4-6 amateur *dan* (strong amateur, close to professional level) [36].

For each problem in the database, GoTools provides a set of solutions, ranked in order of quality. To build each instance in the training set for supervised learning, a randomly selected problem from the database is transformed into a common fate graph representation. The highest ranked solution provided by GoTools is selected as an example of a "good" move. The lowest ranked solution (apart from the top-ranked move), if it exists, is selected as an example of a "bad move".

Subgraph features, relative to the position of the first move in these selected solutions, are generated, creating feature vectors for use in learning. Class values are appended to the vectors, describing the quality of the moves. "Good" move examples are given a class value of 1, and "bad" move examples are given a class value of 0. The learning problem is thus a classification problem, and standard supervised learning techniques can be applied.

Following [21], the maximum subgraph length parameter, s, was set to 6. This makes for feature vectors with just over 400 attributes. It should be noted that this value is large enough to allow the subgraphs to cover a good portion of the board, due to the compact nature of the common fate graph representation, and the fact that tsumego puzzles tend to only refer to a subsection of the full 19×19 grid. For the tsumego database, at least in the subset of the full dataset used in this study, the average board size for the tsumego puzzles was approximately 6×7.

Evaluation is performed slightly differently to training. In the evaluation scheme used, unlike the method for generating the training set, each problem in the tsumego database generates exactly one test case.

First, the common fate graph transformation is performed on the tsumego board positions. For each legal move in the test data (ignoring ko restrictions), the relative subgraph features are computed. Then, the classifier performs a move evaluation by predicting the class value of the relative subgraph feature vector of that move. The move with the highest predicted probability for a class value of 1 is selected as the move chosen by the algorithm. Nothing sophisticated was done to break ties – moves encountered earliest took precedence.

If the move selected by the learning algorithm is one of the moves nominated by GoTools as the start of a solution sequence, then this is counted as a success for the classifier. Otherwise, it is counted as a loss.

Due to the large size of the tsumego dataset (40,907 problems), only a subset could be used. Instances were selected by random sampling without replacement. The running times of the algorithms were the limiting factor in the size of the test/training sets. For this reason, a training/test split of 6600/4000 instances was used.

5.2 Initial Parameter Tuning

Given the large number of classifiers used in the experiment, it was not feasible to tune every possible parameter on each of the learning algorithms. In the initial experiment, parameter selection was performed only where there was just one parameter trading off computational expense against performance. In particular, the number of iterations was varied for the ensemble learners. We also varied the number of nearest neighbours for IBk, and the number of iterations for VotedPerceptron. Graphs of the varied parameters against performance are provided in Figures 3 to 8.

For all other learning algorithms, the default parameter settings in Weka were used. Some parameter tuning was later done with a separate tuning dataset to try to fine-tune the best-performing algorithms. This is summarized in Section 5.5.

The ensemble learners recorded increased performance with more iterations,

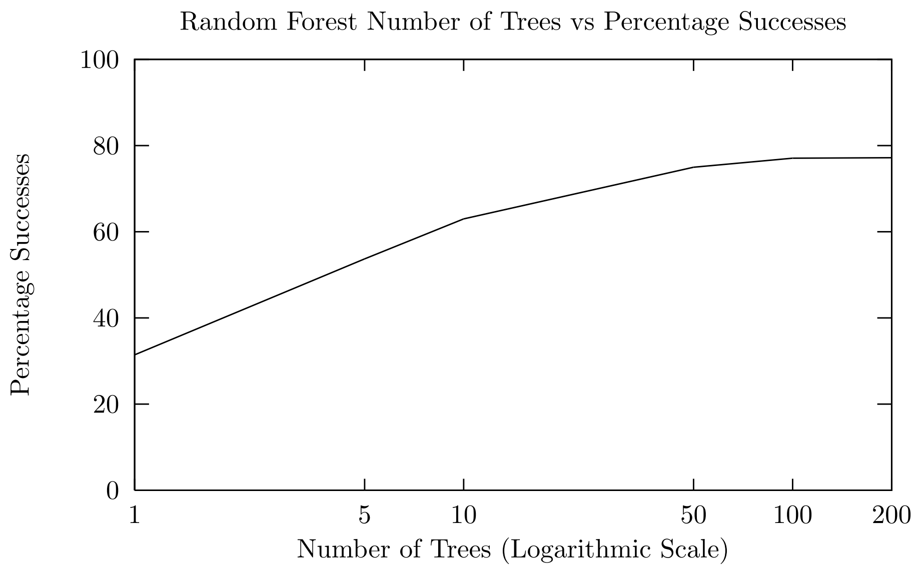

Figure 3: Parameter selection for Random Forest

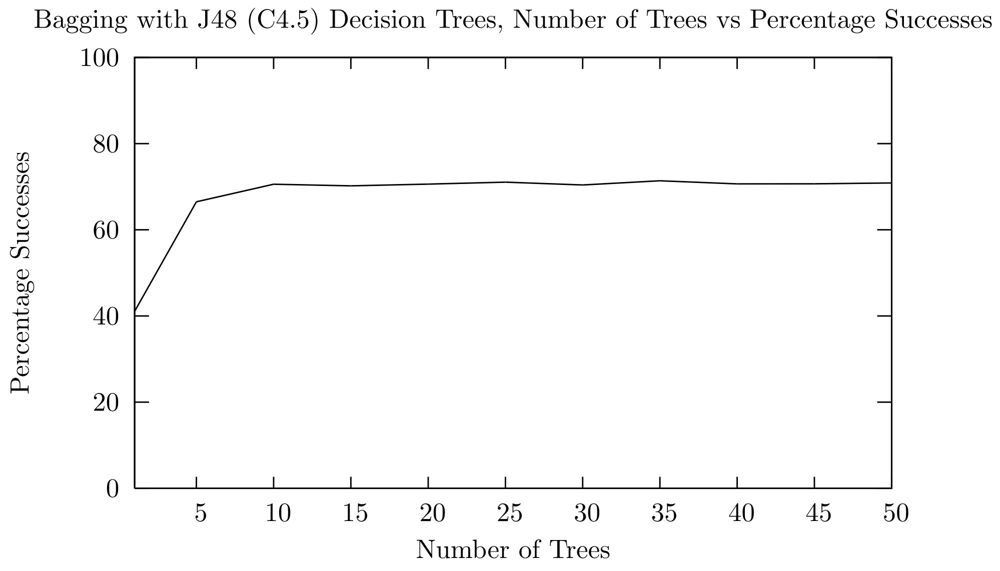

Figure 4: Parameter selection for Bagging

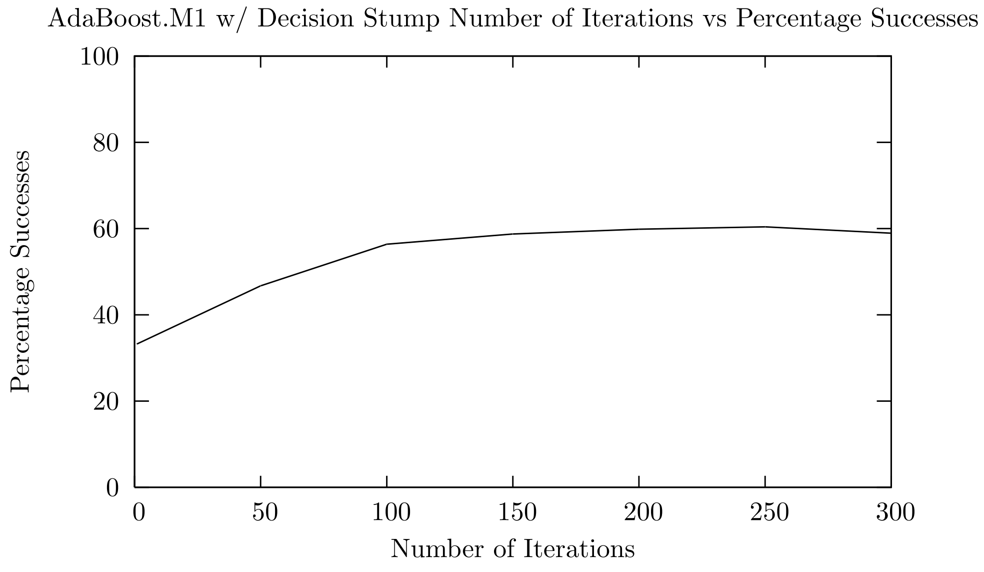

Figure 5: Parameter selection for AdaBoost.M1 with Decision Stump as the base classifier

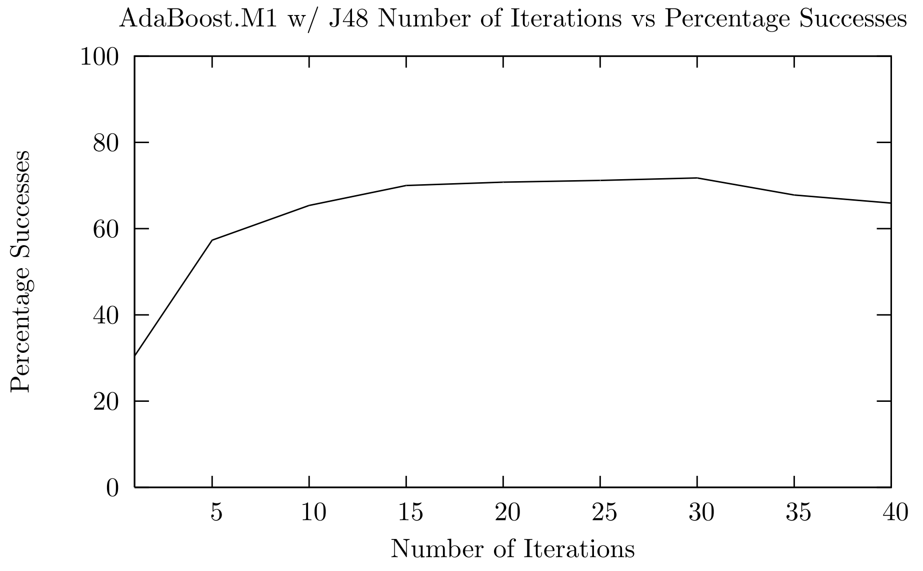

Figure 6: Parameter selection for AdaBoost.M1 with J48 (C4.5) as the base classifier

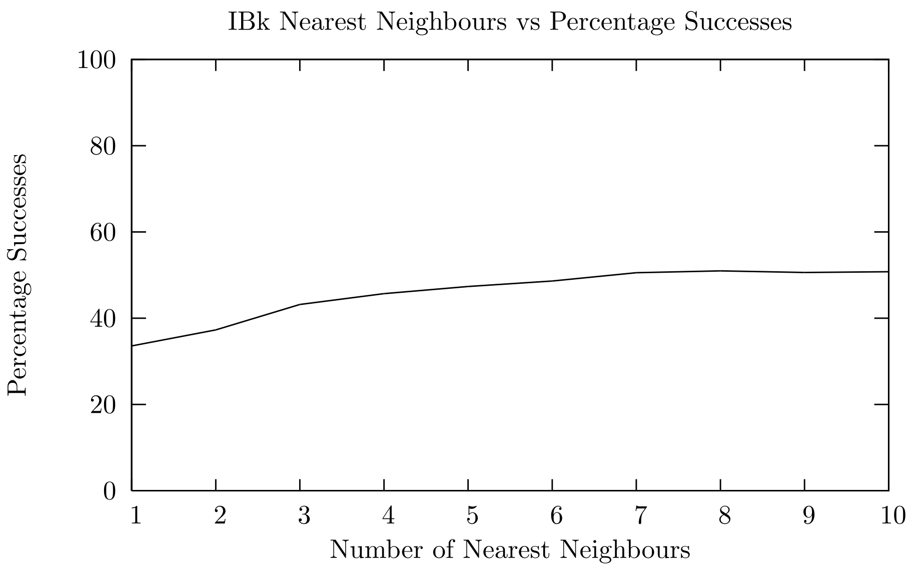

Figure 7: Parameter selection for Nearest Neighbour classifier

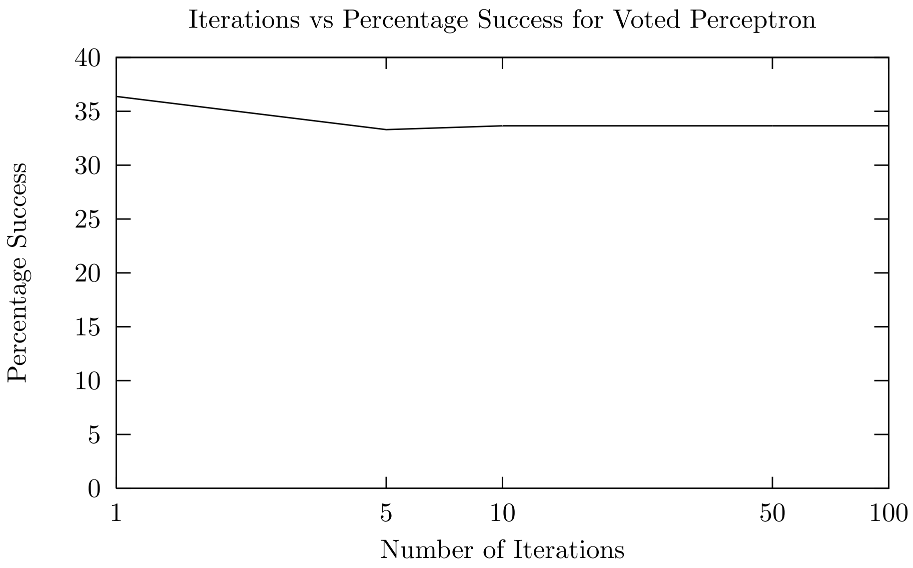

Figure 8: Parameter selection for Voted Perceptron

until the improvements gradually trailed off. At this point, greater numbers of iterations made no difference, except for the case of AdaBoost.M1 with both J48 and DecisionStump as the base learners, where eventually a slight decrease in performance was detected. The most likely explanation for this decrease is that the classifier began to overfit the training data.

RandomForest's performance peaked after around 100 trees, with a success rate of 77.08%. Using J48 as the base learner, AdaBoost.M1 peaked after 15 iterations while Bagging peaked after only 10 iterations. AdaBoost.M1 gained very little accuracy after around 100 iterations.

Interestingly, VotedPerceptron performed best with only one iteration. Performance gains leveled off after around 7 neighbours for IBk.

5.3 Experimental Results

The above experimental method was applied to a wide variety of supervised machine learning algorithms. All of the algorithms used are implemented as part of the Waikato Environment for Knowledge Analysis (WEKA) [34], an open-source data mining suite developed at the University of Waikato, New Zealand.

The results of the experiment are summarized in Table 1. For a brief explanation of each of the classifiers used, see Appendix A. More thorough descriptions of each algorithm can be found in [34].

To test for statistical significance in the differences between each of the learning algorithms' results, McNemar's test was applied with a significance level of $\alpha = 0.05$.

McNemar's Test, in the context of evaluation of machine learning algorithms, is described elegantly in [15]. At the core, McNemar's test is simply an application of the sign test. Because the sign test is distribution-free [23], this test can be used without making too many assumptions about the data.

McNemar's Test was applied to each pair of learning algorithms, A and B. The null hypothesis is that cases where A produces a success and B has a failure ('A+B-') are equally as likely as cases where B has a success and A has a failure ('A-B+'), measured according to the instances of the test data. Each of these two random variables has a binomial distribution.

To test whether there is a statistically significant difference between the two algorithms, we count the number of instances where the outcome is A+B-, and the number of instances where the outcome is A-B+, ignoring cases where both successfully predict a solution recommended by GoTools, or neither did. Then, the sign test is applied to these counts. The result of this is the probability that the observed result or a result further away from the null hypothesis would

Name of WEKA classifier	Number	Statistical Wins	Percentage successes
AdaBoostM1 w/ DecisonStump (100 iterations)	1	9	56.38
AdaBoostM1 w/ J48 (15 iterations)	2	12	70.0
Bagging w/ J48 (10 iterations)	3	12	70.58
Bagging w/ J48 (unpruned, 10 iterations)	4	12	70.15
IB1	5	3	33.58
IB5	6	6	47.35
IB10	7	8	50.78
J48	8	2	30.45
JRip	9	6	46.6
NaiveBayes	10	1	28.30
PART	11	5	39.95
RandomForest (10 trees)	12	11	62.98
RandomForest (50 trees)	13	16	74.98
RandomForest (100 trees)	14	17	77.08
RandomForest (200 trees)	15	17	77.18
SMO w/ Polynomial kernel ($C = 1, Exp = 1$)	16	9	56.75
SMO w/ RBF kernel ($C = 1, Gamma = 0.01$)	17	14	71.43
VotedPerceptron (1 iteration)	18	4	36.38
ZeroR	19	0	25.15

Table 1: Results of tsumego experiment

have occurred if the null hypothesis were true. This value is multiplied by two in order to make the test two sided.

If the observed result is very unlikely given the null hypothesis, this is evidence that the null hypothesis is false. When the probability is less than α, we conclude that the difference between the algorithms is significant.

5.4 Analysis of Results

Table 1 lists the results for the algorithms in alphabetical order. The top few algorithms are all either types of support vector machines or ensemble learners. The best-performing algorithms are the support vector machine learner SMO, using a radial basis function kernel, and RandomForest, an implementation of the Random Forest algorithm [5]. Random Forest is a meta-learning algorithm which works by bagging random trees.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	-	L	L	L	W	W	W	W	W	W	W	L	L	L	L	-	L	W	W
2	W	-	-	-	W	W	W	W	W	W	W	W	L	L	L	W	L	W	W
3	W	-	-	-	W	W	W	W	W	W	W	W	L	L	L	W	-	W	W
4	W	-	-	-	W	W	W	W	W	W	W	W	L	L	L	W	L	W	W
5	L	L	L	L	-	L	L	W	L	W	L	L	L	L	L	L	L	L	W
6	L	L	L	L	W	-	L	W	-	W	W	L	L	L	L	L	L	W	W
7	L	L	L	L	W	W	-	W	W	W	W	L	L	L	L	L	L	W	W
8	L	L	L	L	L	L	L	-	L	W	L	L	L	L	L	L	L	L	W
9	L	L	L	L	W	-	L	W	-	W	W	L	L	L	L	L	L	W	W
10	L	L	L	L	L	L	L	L	L	-	L	L	L	L	L	L	L	L	W
11	L	L	L	L	W	L	L	W	L	W	-	L	L	L	L	L	L	W	W
12	W	L	L	L	W	W	W	W	W	W	W	-	L	L	L	W	L	W	W
13	W	W	W	W	W	W	W	W	W	W	W	W	-	L	L	W	W	W	W
14	W	W	W	W	W	W	W	W	W	W	W	W	W	-	-	W	W	W	W
15	W	W	W	W	W	W	W	W	W	W	W	W	W	-	-	W	W	W	W
16	-	L	L	L	W	W	W	W	W	W	W	L	L	L	L	-	L	W	W
17	W	W	-	W	W	W	W	W	W	W	W	W	L	L	L	W	-	W	W
18	L	L	L	L	W	L	L	W	L	W	L	L	L	L	L	L	L	-	W
19	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	-

Table 2: Result of McNemar's test for all pairs of algorithms

The range in performance among the algorithms is remarkable. There is a difference of over fifty percentage points between the best and worst performing algorithms. This may be partly to the nature of the dataset – there are many attributes, all of which are correlated weakly to the class, and the same attributes may not always be important in different cases.

If this theory is correct, the algorithms that make greedy choices about which attributes are important may underperform because they are not able to use all of the available data – a "big picture" kind of approach is required. The attributes that are highly correlated with the class in some areas of instance space may not be so highly correlated in others, as a different situation on the board may require different types of move to be played.

Thus, the ensemble algorithms may perform better because they have more diversity in their approach. Many classifiers acting at once on the problem may help the meta-classifier from falling into the trap of overfitting to a narrow range of game-playing situations. Support vector machines are also able to use all of the features if necessary. More work is required to determine if this theory holds.

Graepel et al. used a support vector machine and a kernel perceptron for learning on the tsumego data. As the performance of both of these algorithms was very similar, they conjectured that the common fate graph representation was the crucial factor in the result, and the choice of learning algorithm was unimportant. The above results show that the opposite is true – the selection of algorithm here has a huge impact on the result.

Interestingly, the VotedPerceptron learner did not perform nearly as well as the kernel perceptron used in [21]. This may be because the VotedPerceptron classifier used a polynomial kernel, while [21]'s kernel perceptron used a radial basis function kernel. This is consistent with the results for the support vector machine classifier, where the RBF kernel achieved a higher accuracy than the polynomial kernel, although the difference between the classifiers using the two kernels was not nearly so marked in that case.

5.5 Further Parameter Tuning

Having gained a good estimate for the relative strength of each algorithm in the tsumego move evaluation problem, some further parameter tuning was performed in order to fine-tune the strongest algorithms. In particular, parameter tuning was applied to the support vector machine algorithms. These classifiers were among the best-performing algorithms even with their default settings, and it was considered that these in particular were likely to benefit from the tuning.

A separate dataset was constructed consisting of 3300 training instances and 2000 test instances, randomly selected without replacement from the unused portion of the GoTools tsumego database. Parameter space was scanned systematically, and the algorithms were rerun on the original train/test data using the parameters that performed best on the separate dataset.

5.5.1 Support Vector Machine using an RBF Kernel

The gamma parameter and the "C" parameter were varied for SMO (an implementation of a support vector machine learner) with a radial basis function kernel. The C parameter is the upper limit on the absolute values of the support vectors, the variation of which allows control of how closely the model fits to the data. The gamma value is the inverse of the width of the Gaussian kernel used, and also influences the fit to the training data.

Parameter space was scanned systematically, with both parameters being varied in exponentially increasing increments. Figures 9 to 11 show the results.

The parameter values that produced the best results on the separate dataset were Gamma = 1.0 and C = 1.0 or greater. The lowest sufficient value to reach peak performance, setting C to 1.0, was selected.

The support vector machine was rerun on the actual test and training data using these parameter values. The result was a success rate of 79.45%, making it statistically even with the first-placed algorithm, RandomForest.

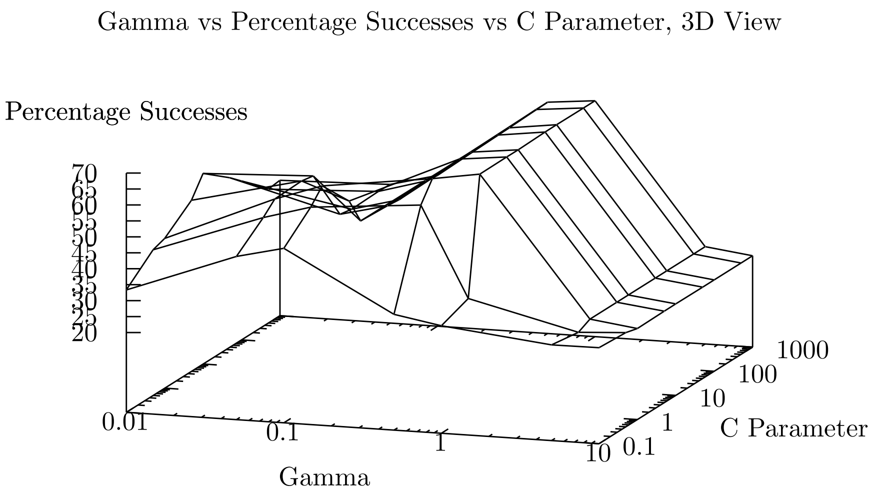

Figure 9: Parameter tuning for the support vector machine with the RBF kernel, 3D View

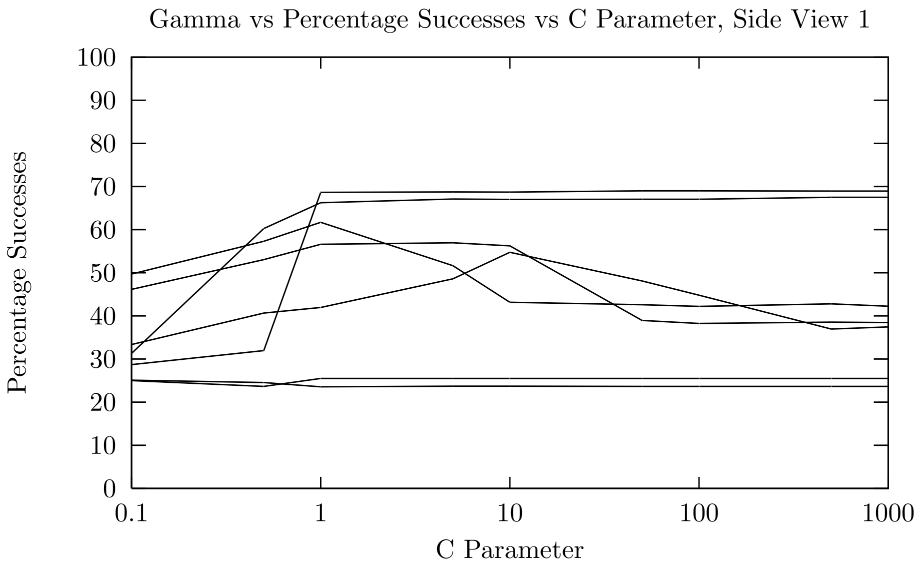

Figure 10: Parameter tuning for the support vector machine with the RBF kernel, Side View 1

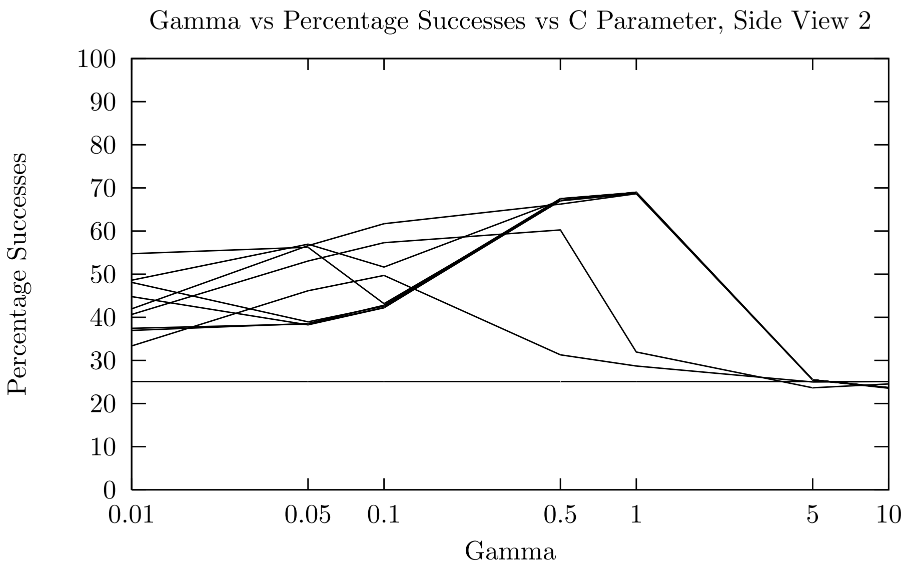

Figure 11: Parameter tuning for the support vector machine with the RBF kernel, Side View 2

5.5.2 Support Vector Machine using a Polynomial Kernel

The C parameter and the size of the exponent of the polynomial kernel were varied as the resulting classifier was evaluated on the separate dataset. The C parameter was once again varied in exponential increments, while the exponent was varied linearly. The results are shown in Figure 12.

The best parameter values on the separate dataset were Exponent = 6.0 and C = 10.0 or greater, with a result of 72.1% accuracy. Setting the polynomial exponent to 5.0 had very similar results, but interestingly there was a sharp dip in performance with the exponent value set to 7.0. The exception to this was the result where the polynomial exponent was set to 7.0 and C was set to 0.1, where the outcome was 72.1% accuracy – making it tie for first place. However, the points immediately around it had much lower percentage accuracy values, so it was decided to treat this point as an outlier.

The support vector machine using a polynomial kernel was evaluated on the actual test/train sets using an exponent of 6.0 and a C value of 10.0, the smallest value of C sufficient to reach the peak in performance on the holdout set. However, when the algorithm was run with this 'best' parameter selection, the performance on the test/train set was disastrous. The result was a success rate of only 2.45%!

This was the lowest success rate seen throughout the entire course of the

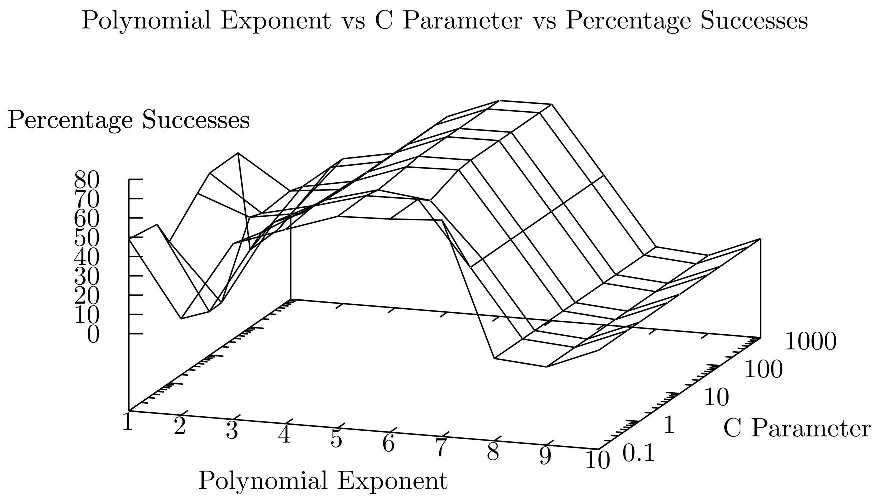

Figure 12: Parameter tuning for the support vector machine using a polynomial kernel

experiment, with a result very much less than would be acquired by random chance (around 25%). This result is difficult to explain.

Further investigation revealed that the classifier using these parameters that was trained on the separate training dataset was able to retain its performance on the actual test set, recording a success rate of 71.15%.

Given these results, it appears that there is something in the actual training set that creates a problem for the classifier. The most likely explanation for this is that there is a bug in the WEKA implementation of the support vector machine learner which is triggered by some property of this particular dataset.

6 9 × 9 Go Experiment

Just as beginners often practice on boards much smaller than the usual 19 × 19 grid, it is common for programs in their earlier stages to be developed for the simpler problem of smaller board sizes. 9 × 9 boards are most commonly used for this purpose – see, for instance, [21], [16] and [4]. Even smaller boards, such as 5 × 5 [28] and 7 × 7 [12] are sometimes used for Go programs that learn from self-play. However for pure supervised learning this is problematic as it is harder to find game record databases for these.

Playing Go on smaller boards is still an interesting problem in itself. For instance, the Internet Go Server has a highly competitive 9 × 9 Go ladder. Therefore, in this section we present experimental results for the performance of a Go machine trained on 9×9 games using supervised learning on extracted relative subgraph features.

6.1 Experimental Setup

An AI Go program was created using the relative subgraph feature algorithm. It was trained on 9×9 Go game records, and played against a benchmark weak opponent called "Wally"[5], discussed in more detail below. A variety of features, including the final score and the ability to create stable groups, were used to evaluate each of the learning algorithms in this context.

6.1.1 Training Sets

A set of 9×9 Go game records was used for training. The database of game records was compiled and archived by Nicol Schraudolph, from internet games played on the Internet Go Server. For the purposes of learning, only games where both players had a rating of *shodan* (1-*dan*) on the amateur scale were selected. A set of game records meeting the required player rating criterion was randomly selected from the database. Since the program played as the *white* player in all games, only *white* moves were used for training.

6.1.2 Program Architecture

The architecture of the Go-playing program is very simple. The chosen machine learning algorithm is trained using relative subgraph features, extracted from board positions in the same manner as in the tsumego program described earlier.

For each *white* move in the selected games, subgraph features are generated relative to the location of the move. Each of these feature vectors is considered to be an example of a "good" move, and given a class value of 1. A corresponding "bad" move example is generated for each good move by picking a random empty intersection and extracting subgraph features relative to this point. To this feature vector, a class value of 0 is appended.

Move selection is performed similarly to the case of tsumego solving. For a given board position, every possible move is evaluated by generating the subgraph features relative to the corresponding board location. The utility of each move is predicted from the subgraph features by the learning algorithm. The move with the highest predicted utility is selected for playing.

Suicide moves are not considered as candidate moves, in order to avoid violation of the ko rule. The program is also prevented from filling in its own

[5]Wally is a simple Go-playing program by Bill H. Newman. It is in the public domain, and comes packaged with its source code. There is no official website for Wally, but it is available for download from http://www.britgo.org/gopcres/gopcres1.html

single-point eyes – this was the only domain knowledge given to the program.

Filling in an eye is a disastrous move that no competent human player would ever make. Implementing this rule is necessary since the program is unable to give a rating to pass moves, and so would most likely perform this move regardless of a poor utility rating if no other option were available, such as at the end of the game when there are no other legal moves.

It may be possible to get away without this rule by setting a threshold move utility value to determine whether the player should pass, provided that explicit training examples were given labeling this type of move as "bad". It is not clear how this threshold value should be determined, however. The authors of [4] also implement the 'no eye-filling' rule as the only domain knowledge heuristic built into a Go program using a Monte Carlo approach.

6.1.3 Game Setup

For each classifier, twenty-five 9×9 games were played against a program called "Wally". Wally is often used as a benchmark weak player – see, for instance, [21] and [28]. It works almost entirely by simple pattern matching, with no other heuristics or algorithms built in except that it always captures when it is able to, and will not play suicide moves. It also detects ko situations so that it will always play legal moves. If it can find no matching patterns, it plays a random move. According to comments in its source code, Wally's author estimates its playing strength as around 30-*kyu*.

Wally was originally written in C. However, for this project a C++ port of the program called "Wallyplus" was used. This version was selected because Wallyplus implements the Go Text Protocol, a modern communication protocol used by Go programs to communicate with each other and with graphical user interfaces.

In all of the games played, the supervised learning-based program played as White, and Wally played as Black. The reason for this is that both players are almost completely deterministic, which leads to a lack of variation between games if Wally does not play first. If the other player starts, play proceeds in a deterministic fashion until a position occurs such that Wally can match no pattern in its database. In these situations, Wally is programmed to play a random move. However, in many cases in the experiment this did not happen at all, resulting in games being repeated in their entirety.

Fortunately, this is avoidable by allowing Wally to play the first move. If Wally is the Black player and hence plays first, the initial board position matches no pattern in its internal pattern library. This causes it to play a randomly chosen first move, resulting in very different games being played each time.

6.1.4 Evaluation

To evaluate the learning algorithms, for each game the number of a program's pieces left at the end of the game was recorded. The mean number of pieces on the board throughout the entire game, and the score at the end of the game were recorded as well. The median of each of these values over 25 games was then computed. The number of eyes formed by the player, and the number of unconditionally alive groups were also counted.

The median was selected as the measure of central tendency for results over all of the games due to its robustness. On the other hand, the average number of pieces on the board during each game was estimated by its arithmetic mean, in order to better take into account all of the data rather than just the middle values. If a program steadily increases the number of its stones on the board throughout the game, a median measure would determine only the number of stones in the middle of the game, which is not what we want to measure.

Scoring in Go is problematic, in particular for computer programs, since the players must agree on which groups are alive, and which are dead groups that must be removed from the board. Hence, there is no clear-cut algorithm for correctly determining the score at the end of the game. Most graphical front-ends for Go, such as GoGUI and Jago[6], ask the user to specify the life-and-death status of groups in order to determine the score.

In this experiment, the score was calculated by Wally's score estimation routine, which makes no attempt to remove dead groups. This means that in the cases where the machine learning player has surviving stones left on the board, the difference in score is often underestimated. It still gives a good relative indication of the difference in performance between the machine learning algorithms, however.

6.2 Experimental Results

It was quickly discovered that Wally was superior in playing performance to the supervised learning player. While the program was unable to defeat Wally no matter which machine learning algorithm was used, the results of the comparative study are still interesting. The results are shown in Tables 3 to 7.

In Table 5, the scores are shown as the overall score of the game, calculated as Black's individual score minus White's individual score (including a komi of 6.5 points in compensation for starting second). Hence, larger numbers indicate a greater victory for the Black player, Wally.

[6]GoGUI and Jago are open source graphical user interfaces for playing Go. They are publicly available from http://gogui.sourceforge.net/ and http://www.rene-grothmann.de/jago/ respectively.

Name of Weka Classifier	# Stones surviving
JRip	0
ZeroR	0
Voted Perceptron (1 iteration)	0
Bagging w/ J48 (10 iterations)	2
IB10	2
Naive Bayes	2
Random Forest	4
SMO (Polynomial Kernel)	9
PART	19
SMO (RBF Kernel)	20
AdaBoostM1 w/ J48 (10 iterations)	20
AdaBoostM1 w/ DecisionStump (100 iterations)	21
J48	22

Table 3: Number of the program's stones surviving at the end of the game. Median over 25 games for each classifier.

Name of Weka Classifier	Average stones on board
Voted Perceptron (1 iteration)	3.19
ZeroR	3.27
JRip	3.45
Bagging w/ J48 (10 iterations)	4.95
Naive Bayes	6.76
Random Forest (100 iterations)	7.22
IB10	7.23
SMO (Polynomial Kernel)	7.77
J48	11.49
PART	11.49
AdaBoostM1 w/ J48 (10 iterations)	11.5
SMO (RBF Kernel)	11.68
AdaBoostM1 w/ DecisionStump (100 iterations)	12.49

Table 4: Mean number of the program's stones on the board throughout the game. Median over 25 games for each classifier.

Name of Weka Classifier	Overall game score
JRip	+113.5
Voted Perceptron (1 iteration)	+112.5
ZeroR	+111.5
Bagging w/ J48 (10 iterations)	+106.5
IB10	+104.5
Naive Bayes	+103.5
Random Forest (100 iterations)	+97.5
SMO (Polynomial Kernel)	+78.5
PART	+40.5
SMO (RBF Kernel)	+39.5
AdaBoostM1 w/ DecisionStump (100 iterations)	+38.5
AdaBoostM1 w/ J48 (10 iterations)	+36.5
J48	+36.5

Table 5: Overall game score – Wally's score minus the machine learning player's score. Median over 25 games for each classifier.

Name of Weka Classifier	Number of eyes created
JRip	0
Naive Bayes	0
ZeroR	0
Bagging w/ J48 (10 iterations)	4
PART	4
J48	5
IB10	6
Random Forest (100 iterations)	1
Voted Perceptron (1 iteration)	0
SMO (Polynomial Kernel)	27
AdaBoostM1 w/ DecisionStump (100 iterations)	29
AdaBoostM1 w/ J48 (10 iterations)	42
SMO (RBF Kernel)	87

Table 6: Total eyes created, summed over all 25 games for each classifier

Name of Weka Classifier	Number of games where safe groups were created
IB10	0
J48	0
JRip	0
Naive Bayes	0
Random Forest (100 iterations)	0
Voted Perceptron (1 iteration)	0
ZeroR	0
PART	1
Bagging w/ J48 (10 iterations)	2
SMO (Polynomial Kernel)	3
AdaBoostM1 w/ DecisionStump (100 iterations)	8
SMO (RBF Kernel)	16
AdaBoostM1 w/ J48 (10 iterations)	17

Table 7: Number of games where groups with two eyes were formed

While a clear winner did not emerge, the stand-out learning algorithms were AdaBoost (a boosting algorithm) using both DecisionStump and J48 (C4.5) decision tree classifiers as the base learners, and SMO (a support vector machine-based classifier) using a radial basis function kernel. These classifiers learnt to create eyes and form stable groups.

J48 and PART successfully kept large groups alive, but this was only due to the simplicity of their opponent. They were unable to form eyes. None of the algorithms managed to capture any of Wally's stones.

Interestingly, the results did not completely match the outcome of the tsumego scenario. The support vector machine learner with the RBF kernel was once again one of the best-performing algorithms, but RandomForest was not as successful as it was at the tsumego task.

6.3 Observations of Game Play

The differences observed in playing behaviour between the algorithms were remarkable, especially considering that each was trained using exactly the same training set. Algorithms of different levels of sophistication made quite different moves, and were consistent in their styles of play across all games. The trends observed over the 25 games for each algorithm are described here.

Given that the ZeroR classifier does not make any use of the feature vector information from instances given at prediction time, it is unsurprising that it was completely unable to distinguish between the possible choices available. This algorithm always predicted the same utility value for each move.

Combining this with the fact that ties in utility value were broken by selecting the first move examined, its behaviour was always to play the first legal move encountered in the move selection process. Effectively, this meant that starting from the left edge of the board, it filled each column of the grid from bottom to top, before doing the same thing on the next column.

JRip was also unable to distinguish between moves. It also just filled in the intersections in columns, starting from the bottom-left of the board.

While J48 appears to have performed well at keeping pieces on the board, an examination of the game records found that this success is largely illusory. Its secret was to play a few pieces smatteringly around the board in the early game, and then merely plough columns of stones from the bottom left in the fashion exemplified by ZeroR. Wally was usually too distracted in capturing the scattered pieces to attack the large but extremely weak group built on the left hand side. This group was often allowed to live all the way until the end of the game. J48 was not competent at creating eyes, however, and would therefore have been immediately wiped out by a superior opponent.

IB10 behaved similarly to J48 but was less successful in distracting Wally, so its large left-hand groups were usually captured early in the game.

Bagging with J48 as the base learner always started the game with several pieces spread out around the board. It then usually played a few moves to expand on an existing group. For the remainder of the game, almost every move was placed on intersections with three black stones adjacent – a disastrous move resulting in the capture of the piece in the next move.

Random Forest and SMO, the support vector machine learner, using the polynomial kernel, played similarly to Bagging with J48. They made the same disastrous moves, but generally played more stones next to their own stones to develop their groups. It was interesting that the Random Forest algorithm, which performed at a similar level to the support vector machine in the tsumego experiment, was only mediocre in the 9×9 game playing scenario.

The boosting algorithms always opened by playing individual stones near all of the edges of the board. They then expanded on these, sometimes making eyes and even stable groups, but doing very little in the way of attempting to capture the opposition's groups. An interesting feature was the tendency to place stones close to but not adjacent to existing groups, and only connect these up later on. The effect of this was to surround more territory with any fixed number of stones. This is a fundamental technique used by competent human players [32].

The behaviour of the support vector machine learner with the RBF kernel was very different to the other algorithms. This classifier played more aggressively and also formed eyes – the foundations of stable groups – far more frequently than the other algorithms did.

In 16 out of 25 games, groups with two or more eyes were created. Forming groups with two eyes is one of the most important subgoals in Go, as such groups are unconditionally alive. They cannot be captured unless the player deliberately and suicidally fills in one of the eyes themselves.

In Go, there is no advantage to be gained by creating groups with more than two eyes – a group needs exactly two eyes to become unconditionally alive. Further eyes are redundant and are generally a waste of moves. The classifier did not stop after achieving two eyes in a group, however. In some cases it went on to build groups with as many as four eyes.

The program played aggressively by placing its stones next to its opponent's stones, right from the very first move. This is in contrast with the other learning algorithms, which usually played far away from the enemy stones whenever possible.

The algorithm was unable to capitalise on its early aggression, however. The attacking groups were usually developed by adding further stones, but the program preferred to defensively secure its position by creating eyes or retreating instead of trying to completely surround its opponent's groups. Similarly to the other learning algorithms, it did not successfully capture any of its opponent's stones in any of the games.

7 Conclusions and Future Work

In this project, supervised machine learning techniques were applied to the game of Go, in the form of tsumego problems and 9×9 games. The relative subgraph feature extraction algorithm presented by Graepel et al. [21] was implemented in order to extract feature vectors for learning.

Comprehensive experiments were performed for the scenarios of tsumego solving and 9×9 game playing. A wide range of machine learning algorithms were evaluated in each of these contexts.

It was found that the support vector machine classifier using a radial basis function kernel with the gamma and C parameters both set to 1.0, and the Random Forest classifier, were the algorithms that performed best at predicting the solutions to tsumego problems generated by the GoTools program. These classifiers both achieved success rates of close to 80% – a significant improvement on the results recorded in the earlier work by Graepel et al.

An interesting result was the broad range in performance between the different types of classifier in this scenario. The support vector machine and ensemble learners far outperformed the simpler methods. An investigation into the precise reasons for this unusual outcome may have important repercussions for the machine learning community in general.

There was not a clear winner in the 9×9 game-playing experiment. The support vector machine learner with the RBF kernel, using the same parameter settings as above, was one of the best learning algorithms at playing 9×9 games. A boosting algorithm, AdaBoost, using two different types of decision tree as its base learner, was also notable.

Simple artificial intelligence programs using these classifiers for move selection were not able to beat a weak benchmark Go-playing program, but showed some promise as they learned to consistently make eyes and create stable groups. Although non-trivial defensive play was learned, the algorithms were weaker offensively, being unable to capture any of their opponent's stones.

A weakness of the common fate graph method is the loss of important information regarding the shape and size of groups, and location of the boundaries of the board. If the common fate graph representation and the subgraph feature vectors were extended to preserve more of the information, better results might be obtained.

A possible extension is to merge provably connectable sets of groups during the common fate graph transformation. These sets of groups should be considered to be one unit (a *dragon*, in Go terminology) rather than several separate objects.

The machine would need a separate module to connect the groups whenever this became necessary, however. GNU Go detects dragons using localized minimax searches, and uses this same technique to make connections between groups when required. It performs most of its analysis on the level of abstraction of dragons rather than groups.

One potential use of the move evaluation function learned from the features extracted by the relative subgraph feature extraction algorithm is as a move generator heuristic within a larger Go-playing system [21]. Move generators suggest candidate moves for further consideration, to enable the program to spend its time investigating only the most promising moves.

Further work is required to determine if the method is effective when applied to a practical scenario in this manner, but the experimental results presented here are encouraging.

Acknowledgments

The author would like to thank his supervisor, Dr Eibe Frank, for guidance and helpful insights, especially with report writing, and Thomas Wolf for providing the tsumego datasets.

References

[1] Myriam Abramson and Harry Wechsler. A distributed reinforcement learning approach to pattern inference in go. In *Proceedings of the International Conference on Machine Learning and Applications (ICMLA 2003)*, pages 60–65, 2003.

[2] L. Victor Allis, H. Jaap van den Herik, and M. P. H. Huntjens. Go-Moku solved by new search techniques. *Computational Intelligence*, 12:7–23, 1996.

[3] Hans J. Berliner. Backgammon computer program beats world champion. *Artif. Intell.*, 14(2):205–220, 1980.

[4] B. Bouzy and B. Helmstetter. Developments on Monte Carlo Go, 2003.

[5] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[6] Brugmann. Monte Carlo Go, 1993.

[7] M. Buro. Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 134(1–2):85–99, 2002.

[8] J. A. Campbell. *Computer Game Playing: Theory and Practice*, chapter Go, Introduction, pages 136–140. Ellis Horwood Limited, 1984.

[9] Murray Campbell, Joseph Hoane, and Feng hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.

[10] Tristan Cazenave. Automatic acquisition of tactical Go rules. In H. Matsubara, editor, *Proceedings of the 3rd Game Programming Workshop*, Hakone, Japan, 1996.

[11] Tristan Cazenave. Integration of different reasoning modes in a Go playing and learning system. In E. Freuder, editor, *Proceedings of the AAAI Spring Symposium on Multimodal Reasoning*, Stanford, CA, 1998. AAAI Press.

[12] Horace Wai-Kit Chan, Irwin King, and John Lui. Performance analysis of a new updating rule for TD(lambda) learning in feedforward networks for position evaluation in go game. In *IEEE International Conference on Neural Networks*, pages 1716–1720, 1996.

[13] William W. Cohen. Fast effective rule induction. In *ICML*, pages 115–123, 1995.

[14] Fredrik Dahl. *Honte, a Go-playing program using neural nets.* Nova Science Publishers, New York, 2001.

[15] Thomas G. Dietterich. Approximate statistical test for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923, 1998.

[16] Markus Enzenberger. Evaluation in go by a neural network using soft segmentation. In *10th Advances in Computer Games conference*, pages 97–108, 2003.

[17] David Fotland. Knowledge representation in The Many Faces of Go. *Manuscript available from www.smart-games.com/manyfaces.html*, 1993.

[18] Eibe Frank and Ian H. Witten. Generating accurate rule sets without global optimization. In *ICML*, pages 144–151, 1998.

[19] Ralph Gasser. Solving Nine Men's Morris. *Computational Intelligence*, 12:24–41, 1996.

[20] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples.* John Wiley and Sons, Inc, 2002.

[21] Thore Graepel, Mike Goutrie, Marco Krüger, and Ralf Herbrich. Learning on graphs in the game of Go. In G. Dorffner, H. Bischof, and K. Hornik, editors, *Proceedings of the International Conference on Artificial Neural Networks (ICANN-01)*, pages 347–352, Vienna, Austria, 2001. Springer-Verlag.

[22] Imran Ghory May. Reinforcement learning in board games., 2004.

[23] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics*, pages 509–511. W. H. Freeman and Company, fourth edition, 2002.

[24] Martin Müller. Computer Go. *Artif. Intell.*, 134(1-2):145–179, 2002.

[25] Oren Patashnik. Qubic: 4x4x4 tic-tac-toe. *Mathematics Magazine*, 53(4):202–216, 1980.

[26] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998.

[27] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[28] R.Ekker, E.C.D. van der Werf, and L.R.B. Schomaker. Dedicated TD-learning for stronger gameplay: applications to Go. In *Proceedings of Benelearn 2004 Annual Machine Learning Conference of Belgium and The Netherlands*, pages 46–52, 2004.

[29] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[30] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer-Verlag New York, Inc., 1997.

[31] D. Shell, G. Konidaris, and N. Oren. Evolving neural networks for the capture game. In *Proceedings of the 2002 SAICSIT Postgraduate Symposium*, 2002.

[32] Arthur Smith. *The Game of Go: the National Game of Japan.* Charles E. Tuttle Company, Inc., 1956.

[33] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3):58–68, 1995.

[34] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, second edition, 2005.

[35] Thomas Wolf. The program gotools and its computer-generated tsume go database. In H. Matsubara, editor, *Game Programming Workshop in Japan '94*, pages 84–96, Computer Shogi Association, Tokyo, Japan, 1994.

[36] Thomas Wolf. Forward pruning and other heuristic search techniques in tsume go. *Information Sciences*, 122(1):59–76, 2000.

A Description of Machine Learning Algorithms

AdaBoost.M1 An ensemble learner that iteratively builds models that are weighted towards correctly classifying instances that were misclassified by the earlier models. Effectively, a set of base classifiers are built, each of which specializes in different parts of instance space. A weighted vote is conducted to create an overall prediction. To generate class probability estimates, the weighted vote is viewed as estimating the log-odds of the set of class values.

Bagging An ensemble learner that builds sets of models on different subsets of the training data. A vote between the base models is taken at prediction time. To estimate class probability distributions, the probability distribution estimates from the base models are averaged.

DecisonStump A one-level decision tree. Class probability estimates are generated based on observed frequencies of each of the class values in the leaf nodes.

IBk A k-nearest neighbours learning algorithm. The k instances in the training data that are closest to the test instance in instance space are located. The prediction is just the majority vote of the neighbours. Probability estimates for each class value are generated based on the frequency of that class value among the k neighbours.

J48 An implementation of the C4.5 decision tree learner developed by Ross Quinlan [27]. Class probability estimates are generated based on the observed frequencies of the classes in the leaves.

JRip Implements Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [13], a propositional rule learning algorithm. Class probability distributions are estimated by the frequencies of the different class values of the instances covered by the rule that covers the test instance.

NaiveBayes The simple naive Bayesian classifier that estimates the probability of each class value being present given the observed features, using Bayes' rule. The assumption is made that each attribute is conditionally independent of the others. This is often not true in practice, but good results can sometimes be achieved regardless in cases where these incorrect class probability estimates are close enough to still rank the class values in the correct order.

PART Generates a PART [18] decision list, using a separate-and-conquer method for deriving rules from partial C4.5 decision trees. Class probability estimates are generated similarly to JRip.

RandomForest A meta-learning algorithm based on bagging ensembles of random trees [5]. Class probabilities are estimated by averaging the estimates generated by the base classifiers.

SMO An implementation of the sequential minimal optimization algorithm [26] for training a support vector machine classifier. A logistic function is fitted to the output of this classifier to obtain probability estimates.

VotedPerceptron Implements the voted perceptron algorithm for voting each of the weight vectors encountered in the creation of a kernel perceptron. This implementation uses a polynomial kernel. For the experiments in this report, the exponent of the kernel was set to 1. Class probability estimates are generated similarly to SMO.

ZeroR A rudimentary algorithm that merely predicts the majority class encountered in the training data. Class probabilities are estimated based on the frequency that each class value occurs.

B Dictionary of Go Terms

alive A group is alive if it cannot currently be captured.

atari Term used to warn an opponent that a group of stones is about to be captured.

dan Unit of rank for strong amateurs and professional players. There are separate *dan* scales for amateurs and professionals. Higher numbers are better.

dead A group is dead if it can be captured against any defense.

dragon Set of groups that are considered to be one unit for strategic purposes. The groups in a dragon can be connected to each other against any defense.

eye Enclosed set of intersections within a group. Eyes are fundamental concepts in Go strategy, as they are required to build groups that are unconditionally alive.

fuseki Sequence of opening moves, similar in function to a chess opening.

goban The playing board for Go. Usually consists of a grid of 19×19 intersections, although smaller board sizes such as 9×9 are sometimes used.

group Set of stones that are strictly connected. Some authors use the term to mean a set of several such strictly connected objects that can be viewed as one unit for the purposes of strategic consideration (described here under the term *dragon*).

joseki Playing patterns, consisting of sequences of moves usually played in the corner of the board that generate an outcome seen as fair to both players.

ko A rule of the game that prevents moves that repeat the immediately preceding board position. The main purpose of the rule is to prevent infinitely repeating gameplay, but in practice the strategic implications of ko are quite significant. The situation arises frequently in actual play.

komi Extra points given to the White player at the beginning of the game to compensate for the disadvantage of playing second.

kyu Unit of rank for weak to moderate amateur players. Beginners start with a rank of around 30-*kyu*, and progress to 1-*kyu*, after which they are measured on the *dan* scale.

liberty The total number of empty spaces adjacent to a group. Groups with no liberties are considered to be captured and are immediately removed from the board.

tsumego A type of Go puzzle, similar in function to a chess puzzle. A board position is given, and the goal is either to protect a group from death or to kill an opponent's group.

stone Playing piece.

super ko Variant of the rules where the ko rule is extended to disallow the repetition of *any* previously encountered board position, rather than just the immediately preceding one. This rule is used in the New Zealand rule set.

C Design of the System

The software for the tsumego learner and game-playing engine was implemented in Sun Java version 1.5.0. The classes that were developed for this system are briefly described here.

AIplayer The game-playing engine, which handles move generation. It loads a serialized machine-learning model from a file, and queries this model on each possible legal move during gameplay.

It also handles communication to a graphical user interface and to Go-playing opponents. This is achieved using the Go Text Protocol (GTP), a modern text-based computer-Go protocol. Most of the GTP functionality is inherited from the GtpDummy class packaged with the GoGUI graphical user interface by Markus Enzenberger.

ArrayBoard An array-based implementation of a board representation. Stones are placed on the board via the playMove() method. When moves are played, captures are detected and the board is updated accordingly. Moves that violate the ko rule are identified at this stage, and stored in a list for quick access. This class extends the *Board* abstract class.

Board An abstract class providing a framework for representation of a Go board position. Abstract methods are specified for playing moves, accessing board information and basic information such as the number of stones of each colour on the board. Concrete methods are provided for making random moves and printing the board, in order to help with debugging.

CFG A graph data type for representing a common fate graph. This is implemented using an adjacency list data structure, for flexibility and fast access to the sisters of any given node.

The constructor accepts a *Board* object, and immediately converts this into the näive full graph representation. It then applies the common fate graph transformation to itself. The CFG transformation algorithm and relative subgraph feature extraction procedure are implemented in this class.

Edge A helper class for *CFG*, representing an edge in a directed graph. Implements the *Comparable* interface to allow instances of *CFG* to maintain sorted lists of edges, for efficiency.

Main The main class, which sets up the playing engine and logging facilities.

McNemarsTest A class to perform McNemar's test to statistically compare the results of different classifiers on the tsumego data. It reads a list of the outcomes of each tsumego problem (where each problem outcome is either a success or failure) for each classifier. It then performs the statistical test on each pair of classifiers.

The results of the test are output to a CSV file, readable in a spreadsheet program such as Microsoft Excel. Another CSV file is created, recording whether there was a win, loss, or draw between each pair of classifiers given a fixed significance level α.

ModelBuilder An executable main class for building machine learning models on game records, and then serializing them.

ParseTG Parser for tsumego problems created by Thomas Wolf's GoTools program. As an implementation of the WEKA Filter class, it expects the problems to have been converted to *arff* format by *TsumegoArffMaker*.

ParseGameRecord Parser for game records stored in the *olf* format, such as the 9×9 game records from Nicol Schraudolph's database. This is implemented as a WEKA Filter, and hence expects the game records to have been packaged within an *arff* file. The output of the filter is a set of relative subgraph feature vectors pertaining to the moves played in the game record.

PlayerRankFilter Inherits from the Filter class from the WEKA framework. It processes game records to find the games where both players were above a given playing strength rating.

SimpleSGF The Smart Go Format (SGF) is a sophisticated tree-based file format for representing games of Go. Its main strength is the ability to support variations in the game-tree for "what-if" scenarios. The SimpleSGF class parses this format, but relies on the assumption that there are no such variations present in the game record.

TsumegoArffMaker Reads data files containing tsumego problems generated by GoTools, and packages them in the arff format. Each tsumego problem in a given data file is individually identified. The name of each problem and the board layout information are extracted and stored together in a single instance of the output dataset.

TsumegoLearner The program for learning to solve tsumego problems and evaluating the results. The percentage success rates of each classifier on the test tsumego data are calculated, and the outcomes for each individual tsumego problem are written to a file for later use when performing McNemar's test. This class also handles the selection of the test, train and holdout datasets, using the method of random selection without replacement.

Vertex A helper class for *CFG*, representing a vertex in a graph. Stores the colour of the node, coordinates for one of the intersections within the graph, and a *TreeSet* of *Edge* objects connecting it to adjacent vertices. A method for counting the number of liberties of the node is provided. Instances can also be marked as *visited* when graph searches are performed.