NTNU

Norwegian University of
Science and Technology

# Reuse of Past Games for Move Generation in Computer Go

**Tor Gunnar Høst Houeland**

Master of Science in Computer Science
Submission date: July 2008
Supervisor: Agnar Aamodt, IDI
Co-supervisor: Helge Langseth, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

# Problem Description

Creating a good computer player for the board game Go represents a significant challenge in the fields of artificial intelligence and abstract game theory. The programs that currently exist are significantly weaker than professional human players, and new and currently unexplored artificial intelligence methods and game playing approaches may be required to improve this situation.

This master thesis project will study how to use case-based reasoning (CBR) methods to improve a computer Go playing program. This includes obtaining an overview of the existing techniques for creating Go-playing programs, and also an examination of how CBR methods have previously been used to play Go and other games. The initial task will be to design and partially implement a computer Go playing program based on existing methods. This system should then be expanded with CBR-based methods in an attempt to improve the playing skill of the system by reusing information from game records of other previously played Go games.

Assignment given: 10. September 2007
Supervisor: Agnar Aamodt, IDI

# Abstract

Go is an ancient two player board game that has been played for several thousand years. Despite its simple rules, the game requires players to form long-term strategic plans and also possess strong tactical skills to handle the complex fights that often occur during a game.

From an artificial intelligence point of view, Go is notable as a game that has been highly resistant to all traditional game playing approaches. In contrast to other board games such as chess and checkers, top human Go players are still significantly better than any computer Go playing programs. It is believed that the strategic depth of Go will require the use of new and more powerful artificial intelligence methods than the ones successfully used to create computer players for such other games.

There have been some promising new developments using new Monte Carlo-based techniques to play computer Go in recent years, and programs based on this approach are currently the strongest computer Go players in the world. However, even these programs still play at an amateur level, and they cannot compete with professional or strong amateur human players.

In this thesis we explore the idea of reusing experience from previous games to identify strategically important moves for a Go board position. This is based on finding a previous game position that is highly similar to the one in the current game. The moves that were played in this previous game are then adapted to generate new moves for the current game situation.

A new computer Go playing system using Monte Carlo-based Go methods was designed as a part of this thesis work, and a prototype implementation of this system was also developed. We extended this initial prototype using case based reasoning (CBR) methods to quickly identify the most strategically valuable areas of the board at the early stages of the game, based on finding similar positions in a collection of professionally played games.

The last part of the thesis is an evaluation of the developed system and the results observed using our implementation. These results show that our CBR-based approach is a significant improvement over the initial prototype, and in the opening game it allows the program to quickly locate the most strategically interesting areas of the board.

However, by itself our approach does not find strong tactical moves within these identified areas, and thus it is most valuable when used to provide strategic guidelines for other methods that can find tactical plays.

# Preface

This master's thesis was carried out within the Knowledge-Based Systems (KBS) group in the Division of Intelligent Systems (DIS) at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor Agnar Aamodt for his guidance, helpful suggestions and valuable feedback during this thesis work.

Trondheim, July 16th, 2008

Tor Gunnar Høst Houeland

# Contents

# Chapter 1

# Introduction

This chapter introduces the background, motivation and objectives for the thesis. The board game Go is introduced, and an explanation is given of why creating a computer Go player is an interesting and difficult challenge.

## 1.1 Background and motivation

Go is a very old board game that has been studied by humans for several millenia. It is based on very simple rules, but requires surprisingly complex strategies to play well. The challenge of producing a good computer Go player has been attempted numerous times[1, 2, 3], but the developed programs have so far not reached the level of strong human players.

Because of this, Go has attracted some attention from the AI community. Well-explored techniques highly focused on search have produced programs that beat the best human players for other superficially similar board games such as chess[4], checkers[5] and Othello[6], but these approaches do not seem to work for Go.

(There are also other games such as Arimaa[7] for which it is also very difficult to create good computer players, but these other games have typically been created specifically to be easy for human players compared to computer programs. Go on the other hand has been played since long before computers existed.)

Illustrating this significant challenge of Go, an IEEE Intelligent Systems letter from the editor in 2006 was entitled “Computers play Chess; Humans play Go”[8], which urged AI to focus more on the parts of human intelligence we do not yet understand instead of focusing on solving well-defined problems. This comparison to Go is justified, since one of the big problems for computer Go is to even discover which information is important to consider at various stages of the game.

There are specialized Go problem-solving programs[9] that can solve even very difficult Go problems, but only when these problems are seen in isolation or separated from the rest of the board by thick walls of stones. This never happens in real games between humans, since building these overly thick walls are a waste of valuable moves that could be placed elsewhere.

If a thin wall is breached in a real game, a human player will often have anticipated this possibility beforehand. The human player will then adapt and turn this into a gain for another area of the game, which can dramatically reduce the impact of such an invasion. Human players rarely fully settle for one specific interpretation of the board position until the game is nearly over.

In contrast, many traditional approaches to automatically playing games rely on having distinct and well-defined states and sub-states, that don't overlap or have unclear boundaries. Some other AI approaches such as neural networks have produced even worse initial results[10], even though these approaches are often better for handling similar uncertain problem boundaries. There have also been other attempts where neural networks have been applied in a much more narrow scope and combined with other search-based methods. For these smaller sub-problems the neural networks have produced acceptable results[11].

Somewhat surprisingly, many different approaches to creating Go playing programs tend to result in programs they can play at approximately the same level, around that of an intermediate amateur human player. There are some differences and some approaches have cleared achieved better results, but these differences are rather small compared to the wide span of different player strengths seen in humans players.

All current computer programs are easily beaten by serious amateur players who have devoted significant time to studying the game, and the programs are nowhere close to beating professional players. This shows that there is still room for significantly better artificial intelligence approaches for playing Go.

## 1.2 Objectives

The overall objective of this master's thesis is to examine how computer programs can learn to play the board game Go, with an emphasis on methods that achieve this by reusing past experience. Contemporary methods for constructing Go-playing systems will be examined, and a new system will be proposed for playing the game with a focus on reusing previous experience as part of the playing process. The following have been the main goals for the thesis:

- Examine Go theory and previous approaches for creating computer Go players.
- Examine how experience can be used to improve game-playing performance based on previous games.
- Develop a new computer program prototype that can play Go.
- Examine how this program can use experience to improve its performance, with a focus on using case-based reasoning methods.

Examining previous Go and computer Go theory and developing the new computer program prototype will be large parts of the thesis work. Creating a working Go playing program is a significant challenge in itself, and before attempting to extend and improve such a program the fundamental theory underlying its construction must be well known.

## 1.3 Go

Go is an ancient strategic board game that originated in China and has been played for thousands of years. The rules and methods of playing have remained largely the same, but it is still a highly challenging game today and the complex strategies that result from the game's few and simple rules are not yet fully understood.

Go was traditionally played primarily in eastern Asia, and has only relatively recently spread to the western world. It spread early on from China to Japan and some time afterwards to Korea, and all the best players in the world today are still generally from these three countries. The theories about the game have also been heavily influenced by eastern philosophy, and this is still apparent today.

In the last 50 years it has become more popular to take a more analytical approach to the game and clearly specifying internally consistent sets of recommendations and guidelines. However, professional-level books concerning Go are still more focused on such aspects of the game as elegance, balance and fairness between the players, and only a few of these advanced books have been translated to English.

## 1.4 Computer Go

There have been many attempts to create programs for playing Go, but these have not yet been as successful as for other games such as chess or Othello. The main reason for this is that the successful approaches for other games are heavily based on searching through many possibilities, while the number of possibilities a Go player faces is of a different order of magnitude. This means that a complete search of the possible move sequences is clearly not feasible, and there is a need for good heuristics to guide such a search to limit the number of positions that have to be explored[1].

This ties in with the other big problem for computer Go, which is that it is very difficult to evaluate Go board positions. In chess and Othello, you can often get a reasonable first approximation of a board position by counting the number of game pieces each player has of each kind, where e.g. in chess having a queen is usually much better than not having one. In contrast, such considerations are rarely as useful in Go. In an even game the two players will normally have almost exactly the same number of stones on the board, and these small differences in number of stones are normally not at all useful for determining who is winning. This is because it is the empty intersections remaining that will determine the winner, and the stones played so far only indirectly influence who will control these intersections in the end.

Because of this it is often beneficial to play moves that strengthen the influence in an area or indirectly capture the opponent's stones, while a purely game piece material-driven search would instead aim to directly capture as much as possible. Human players normally learn that this directly aggressive approach is strategically unsound after a handful of games, and even weak amateur players know how to achieve their real goals indirectly while also expanding their territory.

## 1.5 CBR and Go

Case-based reasoning (CBR) is a problem solving and machine learning approach[12] that relies on explicitly stored previous experience to generate solutions to new problems. Since human players often use what they have learned from previous games and tactical situations when thinking about possible moves, a CBR approach may be useful for programs learning to play Go.

The computer program developed as part of the thesis work will employ CBR-based methods to reuse experience from previous game records. This will be done by comparing the overall strategic objectives and local tactical problems in the current situation to the previous games. Then the moves performed in a previous game will be adapted to discover new moves for the current board position, which are also likely to be good moves if the board positions in the current and previously stored game are substantially similar.

## 1.6 Research method

Part of the thesis will consist of exploring the existing literature in the field of computer Go and analytical analysis of Go games. This will be followed by a further, narrowed focus on the most promising methodological approach for creating a Go playing program.

The results of this theoretical study will be used to design a new computer Go playing system, and include developing a new program as an initial prototype implementation of the proposed system. The development of this prototype implementation will be a large part of the thesis work. This implementation will largely be based on methods used in existing Go programs, and will allow us to get an in-depth understanding of the state of the art for current computer Go approaches. When implemented, the program will also be used as a basis for experimenting with possible further improvements of our basic system.

The main focus for such enhancements will be on allowing the program to learn from previous games, based on game records containing the move sequences played in these previous games. The prototype will be expanded to attempt to include methods for reusing these previous games, both to examine whether they can be easily integrated in practice and to evaluate the effect they have on the playing skill level of the program.

The impact of these suggested improvements will be qualitatively evaluated by comparing different versions of the program with and without these enhancements.

## 1.7 Evaluation

The last part of the thesis will be an overall evaluation of the thesis work, and a discussion of to what extent these initial objectives have been accomplished.

This part will also include an evaluation of the proposed computer Go playing system and the prototype implementation, and an assessment the significance of the results achieved by creating and experimentally modifying the prototype.

# Chapter 2

# Go

The objective of Go is to score more points than the opponent, by enclosing areas of the board with your stones. The opposing player will naturally try to prevent this, but it is also possible to capture the opponent's stones by fully surrounding them, and this is worth points as well. Because of this capturing rule it is not beneficial to attempt overly aggressive moves that don't survive, and much of the challenge of the game is in finding the proper balance between offense and defense.

## 2.1 Capturing

Each group of stones has a number of *liberties*, which are empty intersections adjacent to the stones, as shown in figure 2.1. The single stone on the left has 4 liberties, marked with *a*, while the 4-stone group in the bottom right has 6 liberties, marked with *b*.

If the opponent fills in all the liberties of a group, the group is captured and removed from the board. In figure 2.2, white captures the black stone by filling in the last liberty when playing the marked stone. The black stone is removed from the board and counts as one point scored for white. The now empty intersection will also be worth one additional point for white if he still controls it at the end of the game.

Sometimes one of the players can capture the opponent's stones whenever

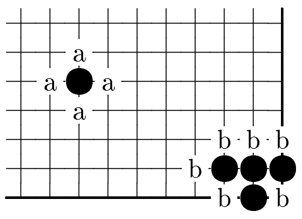


Figure 2.1: An illustration of group liberties in Go. The single black stone has 4 liberties marked *a*, while the black group in the lower right corner has 6 liberties marked *b*.

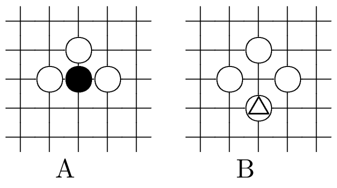


Figure 2.2: An example of the capturing rule. White captures the black stone by playing the marked white stone.

he wants, without any risk of first having his own stones surrounded instead. In this case, the situation is typically left as it is until the end of the game, since the captured player will only lose more stones if he plays in the area, while the capturing player is in no rush and can focus on first scoring points in other parts of the board.

## 2.2 Seki

However, in close games there are often situations where both players try to capture each other first. The player whose group has the most liberties typically wins these *capturing races*, but this is not always the case. Sometimes this even leads to situations where neither player is able to capture the other. This is known as *mutual life* or *seki* and is illustrated in figure 2.3. In this example, neither player wants to play at *a* or *b*. If black plays at *a*, white can then respond at *b* to capture the marked black stones, and vice versa.

When a seki occurs, sometimes the surrounding stones will be captured later in the game, which will then allow one player to safely capture the other. Otherwise the situation simply remains a seki until the end of the game, in which case both groups are counted as being alive.

## 2.3 Ko

In some situations, the game could go into a loop where the players capture each other back and forth continually. An example of this is shown in figure 2.4. To avoid this, there is also an additional *ko rule* which states that a player may not repeat the previous game board state. This means that they cannot

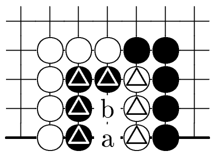


Figure 2.3: An example of seki (mutual life). The marked black and white stones cannot be captured because a move in this area by either player would let the other player capture first.

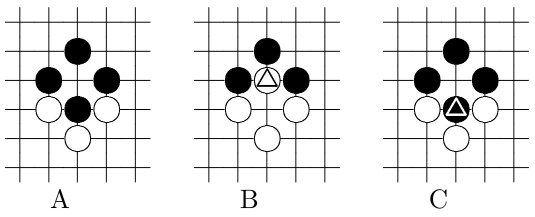


Figure 2.4: An example of a ko. Because of the ko rule, black cannot immediately recapture the stone as shown on the right, but must first play somewhere else to avoid repeating the previous position.

immediately recapture a single stone without first playing at least one other move on the board somewhere else. In figure 2.4, the black move in step C cannot be played immediately after white captures in B. Black must first play another move somewhere else before he can play to capture this white stone. Such capturing situations that involve the ko rule are simply called *kos*.

## 2.4 Scoring

For the example game position shown in figure 2.5, black has surrounded the area at the top, while white has surrounded the area at the bottom. By playing in the middle of the lower line, white can capture the black group in the lower left corner, gaining 2 points for capturing the stones and another 2 points for now surrounding the area they occupied. Board B shows the resulting position after this capture, and the surrounded area for each player is shown in board C.

There are a number of slightly different rule sets for playing Go. These rule sets also include small variations in how the final board positions are scored. However, there will normally never be more than a 0 or 1 point difference in the final score based on these different scoring rules. This is small compared to the number of points the players normally get during play. Between evenly matched top professional players, even these minor differences can sometimes be important, but by comparison the skill difference between professional Go players and computer Go programs corresponds to about 100 points. Because of this the slight differences between these different scoring rules will not be

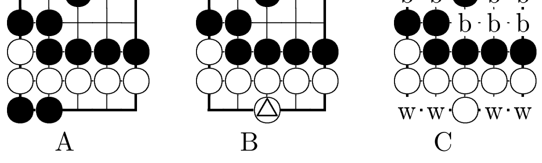


Figure 2.5: Scoring for an example 5x5 board. At the end of the game white plays the marked stone to capture two black stones, and then both players pass. The surrounded intersections marked with *b* are points for black, and the ones marked with *w* are points for white.

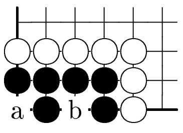


Figure 2.6: A black group with two eyes marked *a* and *b*. This group is alive and cannot be captured by white.

further explored in this thesis.

Using a common scoring rule known as *area scoring* for the example in figure 2.5, black has 7 points from the marked surrounded intersections and 7 more points from the black stones remaining on the board, for a total of 14 points. White has 4 marked intersections, 7 stones and 2 captured stones for a total of 13 points.

This would mean that black wins. However, black always plays the first move, and this gives the black player a significant advantage. To make the game fair, white receives a number of bonus points known as *komi* as compensation. How many komi points white receives depends on the rule set used, but values of 6.5 or 7.5 points are currently the most common for a 19x19 board. (The half point is used to break ties.)

The use of komi is relatively recent and has only become common in the last century. Previously the game without komi was also known to give black an advantage, but playing white was simply considered a handicap for the stronger player. This allowed for tracking the progress and relative strength of players based on the ratio of black to white wins, while today komi is used to instead make all games relatively fair for both players.

(The true value komi should have for an even game at a $19 \times 19$ board is not known, and the value used in tournament play has been slowly increased. A value of 5.5 was used for many years, but in the last decade most competitions have increased komi to values around 7, which appears to be more reasonable.)

## 2.5 Groups

Because of the way the capturing rule works, it is possible (and common) to have groups of connected stones on the board that can no longer be captured by the opponent. The most common example of this is a group that has two *eyes*, e.g. as shown in figure 2.6.

To capture the black stones, the white player has to play at both *a* and *b*. However, both of these moves would have to be played at the same time, otherwise the white stone played would be a suicide, which is not allowed. Since white cannot play two stones at once, there is no way to kill the black group (unless black intentionally plays *a* or *b* himself, which would be very bad and self-destructive moves).

This concept of unkillable groups can be further divided into two kinds. The first classification is for groups that can never be killed even if the opponent always passes and never responds to your moves, which will be referred to as *pass-alive*[13]. This concept is mostly used in academia and detailed game-theoretic analysis of board positions, since no real life opponent would ever behave in this way.

The other, and more common, concept of life is a group that will end up as being alive at the end of the game if you answer all of the opponent's moves correctly, no matter which moves the opponent plays. For these kinds of groups, not answering an opponent's threat or answering incorrectly could still make the group die, but with correct play they will end up living at the end of the game.

Such groups are simply said to be *alive*. However, sometimes these groups may require winning a ko fight for the group to be alive, which will not always be possible depending on the situation on the rest of the board. Such alive groups that depend on a ko fight are said to be *conditionally alive* , and groups that do not depend on a ko fight are called *unconditionally alive*[1]. Being unconditionally alive is a significantly better result in actual play, since winning ko fights can often be very difficult and always involves some sacrifice on another part of the board.

## 2.6 Game stages

A large part of the high-level and strategically oriented Go theory is related to what objectives are most important during various parts of the game, and these are typically classified into three different stages. Games normally contain significant overlap between these stages instead of consisting of distinct parts, but discussing the various stages is nevertheless useful for describing what kind of trade-offs the players should consider and what they should focus on to obtain the best result at that point in the game.

The moves considered important earlier in the game are typically classified this way because they are the biggest and most valuable plays, and should thus normally always be played if possible. If instead an opponent prematurely plays a move that should have been postponed until a later stage, the player can gain an advantage by ignoring the move and taking the initiative to play first in another area of the board. This will typically involve sacrificing some points in the undefended area, but lets the player gain a larger number of points in return by gaining a larger influence over the new area.

### 2.6.1 The opening moves

The game of Go has been extensively studied for hundreds of years, and the very early stages of the game have been examined in great detail. All games start with an empty board, so there are fewer possible board positions early in the game than later. The theory regarding these first opening moves is known as *fuseki*, which concerns what kind of board positions tend to work well together. In particular, many different initial move sequences have been examined for the corners of the board, which are known as *joseki* sequences[14].

These sequences are considered fair and balanced for both players, and either player would normally end up at a loss if they deviate from these established moves. For this reason professional players typically know dozens or hundreds of possible joseki variations, and can also recognize when special considerations

[1]Unfortunately the term unconditional life is ambiguous and has also been used to mean pass-alive. In this report the terms will be used as defined in this section, where responses to opponent moves may be required for groups that are considered to be alive.

based on the rest of the board mean that the joseki sequence should *not* be followed.

### 2.6.2 Middle game

Afterwards follows the middle game, which is characterized by fighting for territory. In the opening, the players have laid out an initial claim on areas of the board, but at some point either an area or the border between two areas becomes contested. The cost of losing an entire such area of the board will be too big to ignore, so both players continue to play moves around this area until it is settled.

There have also been some attempts to classify middle game joseki sequences, but because of the wide variety of possible sequences at this point it has not been studied as extensively. Players often have their own preferences for these situations related to their playing styles, instead of relying on any general consensus on what the best responses are, as is possible for the corner joseki sequences.

In this middle part of the game, the players no longer know the full extent of their moves and possible responses, but are guided by principles such as placing stones in *good shapes*, which means stones that are efficient and flexible. These stones perform their specific task, such as attack, defense or even both at the same time, using a minimal number of stones. At the same time such well-placed stones will also have the flexibility to be sacrificed if necessary, but still indirectly provide some benefit for the neighbouring areas even if they will later be surrounded and captured.

Apart from playing in positions that give good shape, it is also important that the stones played work well together. When playing a new move, it is often very important to adapt the exact position played so it interacts well with your own previously placed stones, rather than directly answering the opponent's last move. It is considered especially important not to make a previous move worthless, or to make a previous move seem oddly placed. Even if it is not immediately obvious from the local tactical situation, making stones work well together in this way tends to be advantageous later in the game.

Among strong players, these considerations are planned many moves in advance, and guidelines for such play are organized in concepts such as following the *direction of play* and preserving the *flow of the stones* when deciding on moves. These and other similar concepts are meant to appeal to human intuition and have so far not been systematized in a way that clearly explains when they are applicable and how they should be used in a way that allows them to be performed automatically by a computer player.

### 2.6.3 Endgame

After the fights on the board have been settled and there are no longer any groups whose life and death status are unknown, the last stage of the game begins. At this point there will no longer be any attacks on the board that realistically attempt to kill groups, but instead the focus will be on playing the remaining moves in the best way and particularly in the most beneficial order.

If neither player makes any large mistakes there will no longer be any major sudden shifts in the expected score result, but in close games between evenly matched players even these comparatively smaller differences can often determine who ends up winning.

The situations encountered in the endgame can to a much larger degree be isolated from the rest of the board and studied by themselves than the other parts of the game. Some specific endgame situations[15] have been studied in great detail, and the exact number of points they are worth has been calculated with optimal play, even for complicated situations that do not arise during normal play.

This theory is currently rarely used in actual games, and it is mainly of academic interest and used for especially thorough analysis of previously completed games.

## 2.7 As a mathematical game

As a mathematical game, Go is classified as a two-player adversarial game with perfect information. It is quite similar to some other board games like chess, checkers and Othello, but there are also some important differences. The game tree complexity for all of these games, containing all possible moves and responses until the end of the game, grows exponentially with the length of the game.

Go games are both longer and have more possibilities at each step than the other games mentioned, which means that the game tree is of a different order of magnitude[1, 16].

Despite the enormous game trees that occur in Go, human players can play the game quite well. One of the main reasons for this is that in Go, pieces are never moved around on the board, and once placed they are only rarely removed by being surrounded and captured. This allows a visual image of the board position to remain valid for much of the game, which leverages the advanced pattern recognition abilities humans possess to quickly understand the situation on the board.

# Chapter 3

# Go tactics

In addition to an understanding of the strategic objectives that should be pursued during the game, players need to be able to carry out their plans using tactically strong moves.

Go tactics deal with local areas of the board, where the players are fighting each other based on groups that are close together. The most valuable fights typically involve the life or death status of these groups, but good tactical plays can also force the opponent to spend extra moves in an area or just provide a few extra points to a skilled player.

The most basic tactical plays are those that aim to conclusively capture the opponent's small groups or single stones, when the player's own stones in the area are already strong. These are relatively simple to understand and are typically one of the first things new players learn after understanding the rules of the game.

## 3.1 Ladders

A ladder is a situation where one player continuously reduces an opponent's group to only having one liberty, which threatens to capture the group on the next move if he doesn't respond. However, in a successful ladder, the opponent is only able to increase the number of liberties to two, and is immediately reduced to one liberty again on the next move. This continues while the fight moves across the board, until it either runs into the edge of the board or some other stones on the board.

An example of this is shown in figure 3.1. The marked stone on the left starts the ladder, and even though the white stone tries to escape it is trapped in a ladder formation by black. Eventually black captures the entire white group at the edge of the board with ⓬.

If it runs into the edge or the attacking player's stone, the escaping player loses all his stones, and it is said that the ladder *works*. Amongst good players, ladders like this are never played out entirely, but simply assumed to kill the escaping stones if they cannot reach another friendly stone.

If the escaping player wants to save the stone, he must first play an assisting stone elsewhere on the board, known as a *ladder breaker*, that will prevent the ladder for working. Similarly, if the attacking player wants to capture a stone

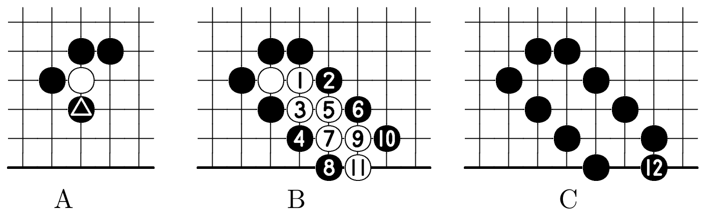


Figure 3.1: An example of capturing stones using a ladder. The white stone tries to escape, but black finally captures the entire white group at the edge of the board.

in a ladder that doesn't currently work, he must first play an assisting stone elsewhere that can assist future ladder formation.

## 3.2 Nets

Another basic technique for capturing stones are *nets*. They require a slightly stronger initial position for the attacking player, but allow for capturing the stone locally, without being affected by the rest of the board like a ladder is. Using a net the stone is not attacked directly, but instead locked out from having any future route of escape, which is why it is referred to as being trapped in a net. The stone is only actually captured and taken off the board if the escaping player tries to save it.

An example of this is shown in figure 3.2. The marked black stone traps the white stone in a net, and the white stone is considered dead and will eventually be captured. If white tries to escape, black shuts him in as shown on the right, and captures the white group with ❹.

## 3.3 Snap-back

Another basic but slightly less common way to capture stones is with a *snap-back*. They are normally used to gain some extra points, whereas ladders and nets often capture strategically important single stones but without directly gaining many points through captures.

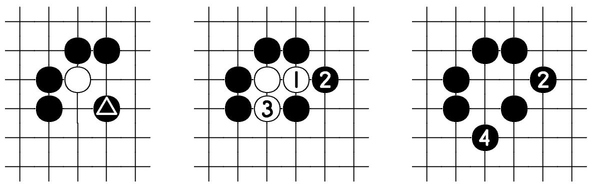


Figure 3.2: An example of using a net to capture stones. The marked black stone traps the white stone in a net formation. The white stone cannot escape, and will be shut in and finally captured if the white player attempts to save it.

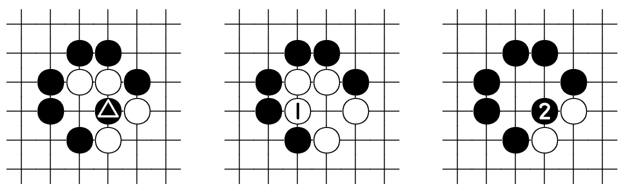


Figure 3.3: A capture using snap-back. When white captures the marked stone, black responds by capturing the entire white group.

Snap-backs work by immediately recapturing a group after it has captured one of your stones, because the opponent's group reduced itself to only one liberty when capturing. This often comes as a bit of a surprise to new players, who do not think that capturing a stone could be bad. However, the group was actually already killed by the initial stone, since the following snap-back normally cannot be avoided.

An example of this is shown in figure 3.3. After white plays ① to capture the marked black stone, black responds by playing ❷ at the same spot as the captured stone, which now captures the white group.

## 3.4 Eyes

The most important criterion for determining if a group will live or dies, is whether it has (or can always form) two eyes. If a group has two separate *eyespaces* where such eyes can be created, it cannot be killed unless the opponent can somehow create his own group inside the eyespaces, which is rare. Because of this, groups with only one eyespace are the most interesting for the purpose of this project, and groups with two eyespaces can simply be considered to be alive.

Reducing the eyespace of a group to only containing one possible eye is thus an effective way to kill the group. If a group is reduced to only one eye in this way and the group is disconnected from other groups belonging to the same player, it will eventually be killed regardless of the surrounding areas (the exceptions being if it can kill a neighbouring group first, or achieve mutual life in seki).

Usually a larger eyespace is better than a smaller one, since they are worth more points and a large eyespace is more likely to allow for two separate eyes. However, there are some notable exceptions, and even amateur players typically know the most basic eyespace forms and their corresponding life-and-death status. These basic eyespace forms are shown in figures 3.4 through 3.8.

Some of the shown groups are dead, some are alive, but the most interesting groups are those that are *unsettled*. The life or death of these groups depends on which player plays the next move in the area. For the unsettled groups the points marked with $x$ are the *vital points* of these groups. White can play on the vital point to kill the groups, while black playing there saves the group[1]. These vital points are usually at the center of the eyespaces, and human players develop an

[1]For the larger eyespaces there may also be other black moves that save the group, but the vital point is normally preferred since it also removes any remaining ko threats.

intuition for quickly finding these vital points even when the surrounding stones are not as fully settled as in these illustrative figures.

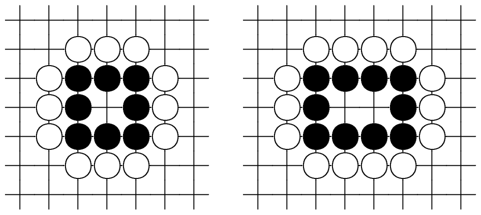


Figure 3.4: Groups with a small eyespace containing 1 or 2 spaces are dead.

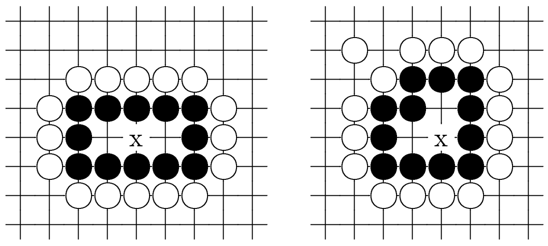


Figure 3.5: Groups with 3 spaces are unsettled and depend on who plays next. If white plays $x$ the groups die, while black paying $x$ saves the groups.

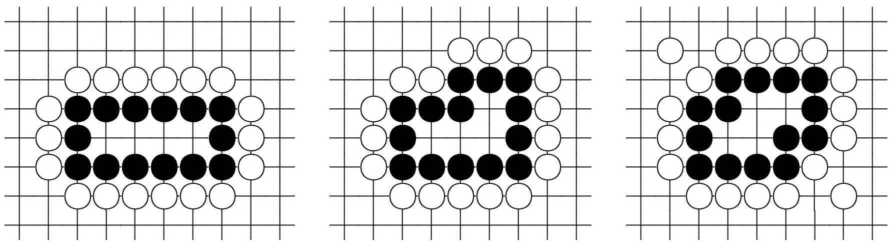


Figure 3.6: These three versions of 4 point eyespaces are unconditionally alive. As long as black answers any white move inside the eyespaces, these groups cannot be killed by white.

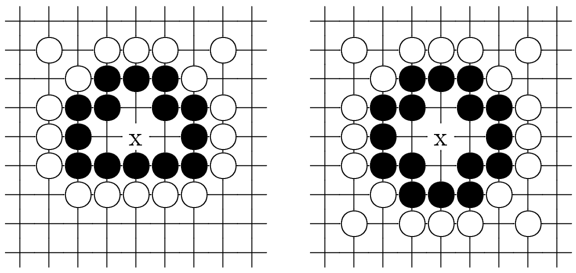


Figure 3.7: 4 or 5 points in a cross are unsettled and depend on who plays next. The vital points for these groups are marked with *x*.

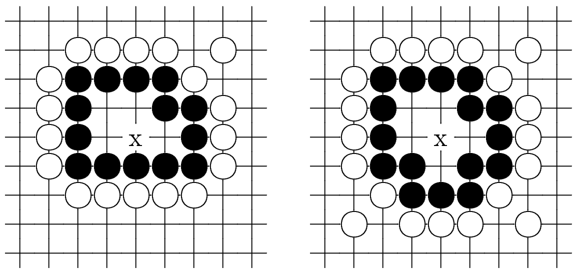


Figure 3.8: The bulky five and flower six are unsettled and depend on who plays next. The vital points for these groups are marked with *x*.

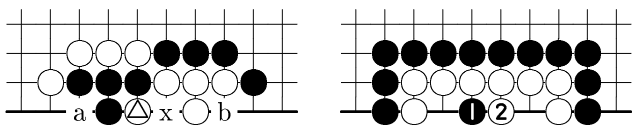


Figure 3.9: An example of a ko fight. The ko is shown on the left, and the player that wins the ko will be able to kill an additional 4 stones. On the right there is an unconditionally alive white group that contains a possible ko threat for black.

## 3.5 Ko tactics

Kos are situations where both players can capture a stone back and forth, but because this could lead to an endless loop, the ko rule forces the players to play elsewhere in between. Ko captures by themselves are only worth one point, but important kos can often be worth much more indirectly, typically by deciding whether an entire group lives or dies. An example of this is shown in figure 3.9, where the black group will die unless it can capture the marked white stone.

If white gets to keep the white stone and also play at *a*, the black group will die, while if black can capture the marked stone by playing at *x* and then also play at *b*, the white group will be captured instead.

Assuming that white captured a black stone at *a* with the marked white stone, the ko fight has started and black cannot immediately recapture by playing at *x*. First, he has to play somewhere else on the board, someplace where white will want to respond instead of capturing the group by playing *a*. Because of this, the move has to pose a significant threat if it is left unanswered, which is why such moves are called *ko threats*.

An example of a good ko threat is to play inside a straight 4-point eyespace of an otherwise surrounded group, as shown on the right hand side of the figure. After white plays the marked stone in the ko fight, black answers by playing ❶ as a ko threat. The group is alive and cannot be captured as long as white answers correctly, and both players know this. However, white has to answer this ko threat immediately, otherwise black can play another stone inside the eyespace afterwards, and this would kill the group. To avoid losing the group, white answers at ②, but now black can play at *x* in the ko fight and then white is the one who has to find a new ko threat or lose the ko fight.

## 3.6 Double-purpose moves

Another very effective type of move is a *double-purpose move*, a move that is useful in two different ways at once. For example, this can be attacking two groups at once, attacking a group while at the same time strengthening your own group, or setting up a ladder-breaker that also threatens to capture a group.

## 3.7 Go playing skill

For a human Go player, it is important to both have a decent understanding of the high-level strategic objectives and the skill and experience to find good moves in a local tactical situation. It is natural to assume that a computer player would also benefit from having this level of knowledge and understanding. This means that our computer Go playing program preferably should have some way to determine strategically important moves, and also a method to find good tactical moves for local situations.

# Chapter 4

# Game theory and analysis

A lot of research has gone into games and how they can be played, and this also includes the game of Go. In this chapter we will examine some theories regarding the analysis of Go positions, and also comparisons between Go and other similar board games.

These theoretical considerations can be useful to determine how various parts of the game should be addressed by our Go playing program. They can also provide guidelines for how difficult it will be to find methods for reasoning about these various parts of the game, and estimates for how well these approaches can be expected to perform.

## 4.1 Game characteristics

The number of possible board positions and the number of possible games of Go are numbers beyond normal human understanding, and also significantly larger than for most other similar board games. A comparison with other popular games is shown in figure 4.1[1, 16].

The number of game positions is an estimate of the number of legal states the game can be in, while the number of game sequences (the number of leaf nodes in the game tree) is an estimate of the total number of ways the game can be played. These measures are related, but they are not directly proportional because the branching factor (the number of legal moves at each position) also greatly affects the size of the game tree.

| Game | Game positions | Game sequences |
|---|---|---|
| Tic-tac-toe | $10^3$ | $10^5$ |
| Connect Four | $10^{15}$ | $10^{20}$ |
| Checkers | $10^{20}$ | $10^{30}$ |
| Othello | $10^{30}$ | $10^{60}$ |
| Chess | $10^{50}$ | $10^{120}$ |
| Shogi | $10^{70}$ | $10^{230}$ |
| Arimaa | $10^{43}$ | $10^{300}$ |
| Go | $10^{170}$ | $10^{360}$ |

Table 4.1: Complexity of some common games.

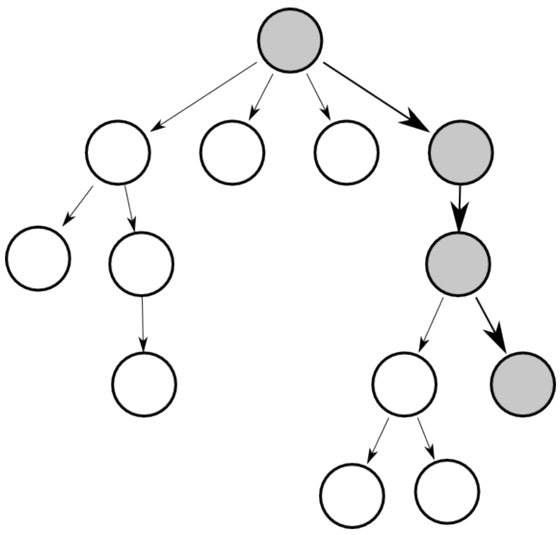


Figure 4.1: An example of a game tree.

Shogi is a Japanese variant of chess where previously captured pieces can be placed anywhere on the board instead of making a normal move. This increases the number of possible positions and games, but still allows for many of the same types of strategies and tactics as normal chess. Creating computer shogi players is not expected to require any fundamentally different artificial intelligence methods from those used to play chess, but instead mostly continuations of existing methods using increased computational power.

Arimaa[7] is a recent game that was created to be easy to learn for human players but difficult to solve using AI methods. It has a very large number of possible moves at each position, and a correspondingly large number of possible game sequences. Human Arimaa players are currently much better than computers, even though there have been some attempts to produce good computer players. However, the game is only about 6 years old and both humans and computers are still getting significantly better and discovering new strategies.

The other games listed in the table are more common and assumed to be well known. Tic-tac-toe, Connect Four and checkers have already been solved, which allows a computer player to play them optimally. Computer Othello players are vastly better than any human players, and computer chess players are better than even top human professionals, but not flawless.

There is an obvious correlation between the game tree complexity and computer player achievements, but this is not caused by the game tree alone. In Go the reason for the comparatively good human results are to a large degree attributed to the human ability to form complex abstractions and our very strong pattern recognition skills. For some other games that have very large game trees, computer players are still vastly better than humans because in these games there are no such discernable patterns that aid humans in very effectively dealing with only extremely narrow parts of the game tree the way they can for Go.

Figure 4.1 shows an illustrative example of a small game tree. The marked nodes show one possible path through the game tree, where each node represents one board position and each arrow represents a move. The example shown has a maximum of 4 children for the root node and consists of 5 node levels. A full game tree for Go would have an average of about 100 children for every node, and about 200 levels of such nodes. Exhaustively searching through even 10 of these levels is impossible using modern computer systems, and the number of nodes grows exponentially at every level.

### 4.1.1 Computational complexity analysis

Finding the optimal move in Go is known to be a very hard problem. Some results from the field of computational complexity theory have analysed parts of Go to determine just how difficult they really are. These consider a generalized version of Go played on an $N \times N$ board, and find bounds for exactly how much *more* difficult the tasks become as $N$ grows.

#### Complexity classes

Problems in complexity class P[17, 18] are commonly regarded as problems that are tractable, or efficiently solvable on a computer. Formally, the complexity class P consists of those decision problems that can theoretically be solved in polynomial time by a deterministic Turing machine.

Using big O notation, these are problems where a solution to a problem of size $N$ can be found in time $O(p(N))$ where $p(N)$ is a polynomial function of $N$, such as $p(N) = N^3$. This means that the solution to any such problem can be solved by one algorithm that will always be able to find the solution in less than a constant multiple of $N^k$ steps, where the constant multiplier is mostly used to allow for different "base step sizes" and is rarely considered important for the analysis.

A well-known example of this is sorting, where general comparison-based sort is known to be $O(N \times logN)$, and many algorithms are known that fall within this category. (For comparison-based sorting this is also known to be a tight bound, i.e. the minimal bound is $\Omega(N \times logN)$. This notation means that it is impossible to find the solution in less than a constant multiple of $N \times logN$ steps.)

Another important complexity class is NP, which are decision problems that can be solved in polynomial time by a non-deterministic Turing machine. This is an imaginary machine that can have multiple possible actions for any given state, and the machine can always "magically" pick the most beneficial one. Another more intuitive explanation is that the solutions to problems in NP can be *verified* quickly, but finding these solutions might be very difficult.

All problems in P can also be solved by these non-deterministic machines, which means that all problems in P are also in NP. However, it is believed that there are many problems in NP that are not in P, and are thus considered more difficult. (It is not actually known if this is true, and whether P = NP remains an important open question[19]. An affirmative answer would have large ramifications for many parts of computer science, for instance the most common forms of contemporary computer cryptography would be rendered ineffective.)

This leads to some additional definitions of how hard a problem is, where e.g. an NP-hard problem is at least as hard as any problem in NP. This means that if you have an efficient method of solving the NP-hard problem, any problem in NP could be converted to input for this problem in polynomial time, which would essentially mean that the same solution could be used to solve all problems in NP as well.

If a problem is both in NP and NP-hard, it is said to be an NP-complete problem. Such problems exist for other complexity classes as well, and can be considered the most difficult problems within each class. If these problems are solved, they would allow all other problems in the same complexity class to be solved as well.

PSPACE consists of problems that can be solved using an amount of memory that is polynomial in the size of the input, regardless of the number of steps this takes. These problems are even harder, and PSPACE is at least as big as NP (although it is not known if NP is a true subset or whether they're actually identical sets of problems).

EXPTIME is the set of problems that can be solved by a deterministic Turing machine in exponential time, i.e. $O(2^{p(N)})$, where $p(N)$ is a polynomial function of $N$. Since the number of separate states in a system is exponential in the amount of memory, e.g. $2^N$ for $N$ bits of memory, these problems are at least as hard as those in PSPACE, since they could simply examine all possible states (again, whether this means strictly harder is not known, and even NP and EXPTIME could actually be the same).

### Complexity of Go

Determining the result of ladders in Go is known to be in PSPACE[20], and it is also at least as hard as any other problems in PSPACE, making it PSPACE-complete. Go endgames have been shown to be PSPACE-hard[21], at least as hard as any problems in PSPACE, although they may possibly be even harder. The game of Go itself is known to be EXPTIME-complete[22] (using Japanese ko rules).

However, these results in themselves do not mean that it must be prohibitively difficult to create a Go player, since many other games are also EXPTIME-complete when properly generalized, such as chess and even versions of tic-tac-toe known as m,n,k-games.

What it does indicate is that creating an algorithm that will always produce the correct result is very unlikely, even for smaller and often relatively simple parts of the game such as ladders and the last plays in the endgame. This is an important result, since it means that our program cannot be expected to be able to handle all situations correctly. Instead our program should focus on finding solutions that work reasonably well most of the time, especially for those situations that are most likely to occur in real games.

Another result is that playing a game on a $19 \times 19$-sized board can be expected to be much more difficult than on a $9 \times 9$-sized board, which matches the experience of computer Go programmers. This suggests that the same type of algorithms that work on a small board do not necessarily translate to a larger board, which also corresponds to how humans play.

Human players approach a $19 \times 19$ board game with a larger emphasis on the opening and early middle game, while a $9 \times 9$ board game is understood to be a more tactical situation, where the skills normally used in the late middle game and endgame are used almost exclusively.

## 4.2 Board position analysis

Many different parts of the game have been separately analysed in detail, with methods that give provably correct results although only for their specific subset of game situations. Some of the methods used could be useful for creating a general computer player, and some concepts and insights developed in these specialized analyses may be useful in a broader context.

### 4.2.1 Combinatorial game theory

Combinatorial game theory (CGT)[23] deals with two-player games consisting of various positions where the players can perform moves, and the current position and possible moves are known to both players. The goal is to find the optimum move, which is usually difficult unless the games are rather simple.

A well-known and important example is *Nim*[24], a game consisting of stacks of various sizes, and the players alternately pick a stack and take as many objects as they wish from that one stack. The same moves are available to both players, and the only difference between them is which player is the next to move in each position. These Nim games can be solved, and in fact the Sprague-Grundy theorem[25] states that every such two-player game where the same moves are available to both players is equivalent to a *nimber*, which describes a specific Nim position.

The situation becomes vastly more complex in *partisan games*, where some moves in a given position are available to one player and not the other. Most well-known board games are partisan games, e.g. in Go only the first player can play black stones, while the second player plays the white stones.

Combinatorial game theory is not directly applicable to Go games, but most endgame Go positions can be transformed to *chilled Go*[15] positions, and these chilled positions can be solved using CGT. The correct solution to chilled Go positions will normally also apply to the original Go position, except in some cases involving ko and seki. Chilled Go is played according to normal Go rules, except that each play costs 1 extra point as a special tax. This form of chilled Go is not particularly interesting to play in itself, but is only considered because it allows for the mathematical analysis of otherwise cumbersome positions.

This theory can sometimes find optimal solutions to the very last part of the game, and even sometimes point out moves by professional players that caused them to lose the game. Using the developed theory, combinatorial game theorists can create Go endgame positions that are quite difficult but could have reasonably resulted from normal play and do not appear overly constructed. However, in these positions the theory can be used to defeat even professional Go players regardless of which side the professional chooses to play. (Typically the player's scores in these positions are very close, and the positions are constructed in such a manner that there are many possibilities for suboptimal plays that will lose the game.)

Interesting as this may be, it is not directly useful for creating a general computer player, since it can only be solved for a very limited number of possible situations and not positions that occur earlier in the game. It could be useful for creating an automated solver for endgames, but this is unlikely to result in more than a couple of points difference compared to normal non-optimal play and as such is not nearly enough to allow computer players to approach human level playing skill.

### Game temperature and move urgency

One important contribution from combinatorial game theory is the concept of game temperature. Playing a “hot” move is much more urgent than a “cold“ move, and this classification is based not only on the expected gain, but on what possibilities the opponent would have if you do not play them. This corresponds to concepts that are already known in traditional Go theory but more loosely defined. The most important such Go concepts are known as *sente* and *gote* plays.

A sente play is one that the opponent has to answer rather directly, which means that the player will retain the initiative and get to choose which area to play in once more. Being the first to play sente moves in all interesting areas of the board would be a very significant advantage, and a better understanding of sente and gote plays is often essential to reduce a weaker player’s initial advantage in handicap games.

The opposite term is gote, which simply means a move that does not have to be answered. Playing gote moves is not always bad, since gote moves are often required to make sure that a group lives, or to kill one of the opponent’s groups. However, weaker players typically play smaller gote moves too soon, and a gote move worth only a couple of points should almost always be postponed until the endgame. Some more complicated concepts such as reverse sente also exist, which is a gote move that prevents an opponent from getting a sente move in the area. This is usually much better than a normal gote move, and worth playing earlier in the game.

In CGT these concepts are analysed more rigorously and at a very exact level, where the different areas are considered as separate *surreal numbers* which can be analysed individually and later combined. The temperature then corresponds to the size of the most important such play, and naturally the players should attempt to play moves that are worth as close to this amount as possible. The temperature is said to noticeably drop when moving from the opening to the middle game, and from the middle game to the endgame, and it is often very advantageous to get the last move before such a large temperature drop. This is natural since the last move was worth a large number of points, and after a decrease in temperature most future moves will be worth much less, so such a play will give the player one more big move than the opponent.

## 4.3 Group life

An algorithm known as Benson’s algorithm[13] can be used to determine which groups are pass-alive, i.e. groups that cannot be captured even if the opponent is allowed to play an infinite number of moves in a row. It is a static algorithm that finds such groups without any form of search or experimental play, but the fact that it only finds groups that are pass-alive drastically reduces its usefulness during normal play.

A group that is pass-alive is obviously also alive in the traditional sense, but a computer player needs to also know the status of groups that are not played to this level of completion. Typically only groups with small eyespaces will be pass-alive, and they still often do not become pass-alive until the latter part of the endgame.

A proficient Go player has to determine whether the group may be likely to die much earlier, so that the dying group can be connected to other groups for life or sacrificed in a more meaningful manner that provides some benefit. There are no known very good methods for doing this analysis in a computer program, but counting and evaluating the eyespaces is a reasonable approach that sometimes works sufficiently well.

## 4.4 Score estimation

Estimating what the score will be at the end of the game from a given board position can be very difficult in Go. Amateur human players typically only have a very rough idea of what the score will be, sometimes only on the level of whether the game is hopeless and at a point where they should resign or not. Instead of playing based on the expected outcome from each possible move, various areas of the board are thought of as being important or big, and this is used to guide the player in finding an appropriate move. The specific move within this selected area is then picked based on tactical skill and experience.

Computer Go players are instead often based on at least some form of search, where the different possibilities have to be evaluated in a manner that can be expressed in a computer program. Because of this, estimating the score for each resulting position is a common approach – a good estimate would in return yield good moves. However, it is very difficult to calculate the expected score from a position, because of the many tactical considerations that appear in an area before the game is over.

Computer score estimates are often reasonably good in the opening game, where the stones can be considered somewhat independent without drastically reducing the accuracy of the score estimate. Using different methods, computers can also often give good estimates during the very last parts of the endgame, when the board has been partitioned into clearly separated areas and stones are only affected by very close neighbours.

However, none of these methods actually have a real understanding of how the stones affect each other, which makes them poor at estimating the score during the middle game. To estimate the score during the middle game, a human player analyzes the different areas of the board tactically and is able to also estimate how the various stones will affect each other much later in the game.

Contemporary computer players do not have this level of understanding, and often give worse estimates for tactical position than even weaker amateur players they could beat in an even game.

# Chapter 5

# Computer Go

Since very strong computer players have been developed for many other board games, there has also been significant interest in developing computer programs that play Go. Many such programs have been developed, and the playing strength achieved by state-of-the-art programs gradually increases.

However, at this time even the best programs are still playing at an amateur level, and are easily defeated by professional players. When developing Go programs many difficulties have been encountered that are not as significant in other board games. These mostly involve a larger emphasis on strategic considerations and the fact that Go board positions allow for a vast number of possible continuations.

The earliest programs typically contained some rules and considered each stone to spread some influence to the nearby area. GNU Go is a program that primarily utilizes a more advanced and expanded form of this approach with many generalized rules patterns. It is currently one of the best computer programs available, and is often used as a basis for contrasting and evaluating other programs.

More recently an approach based on Monte Carlo simulations has been introduced, which is based on using a very large number of computations and evaluations and finding good moves based on analyzing these evaluations statistically. This approach has become popular and achieved good results within the computer Go community, particularly for smaller board sizes where this approach has even led to significantly better results than any other currently known methods.

## 5.1 Rule-based systems

The oldest kind of computer Go-playing programs[1] contained various rules, where the program has been explicitly told to perform in certain ways based on the board position. An example of this could be “If there is a bulky five shape on the board, play on the vital point. Otherwise check the other rules.” This rule is rather simplistic - while playing at the vital point is often a very good move, it might be possible to connect the group to another eye in response to this play, or even some other plays that are worth more, such as a game-deciding ko fight.

Another example of a more advanced rule is to always play in the position that will provide the biggest increase to the player's influence over the board. This sort of rule is unusual for traditional rule-based systems, but quite common for even the early Go programs since the other rules typically only covered a limited number of situations, while the program should be able to play even in positions that weren't covered.

Another very difficult problem is one that is common to many such rule-based systems, which is ensuring that the rules remain consistent and actually perform as intended. In a Go-playing program there would typically be hundreds of rules, and keeping these rule sets functional and properly ordered can be a significant challenge. To somewhat reduce this problem, the rules can be generalized and sorted into groups and subsystems, which is fairly effective if the groups of rules can be separated in such a way that the different groups do not affect each other and can be analyzed separately.

Stone patterns and generalized stone patterns where some of the intersections are ignored are often used to implement such rules[1]. The benefits of this approach are that the patterns can often be relatively easily checked against the current board position, and that they are mostly easy to read for human experts when designing the system.

However, such programs are normally only as good as they are explicitly told to be. Other mechanisms are needed to allow these systems to learn from experience, such as reordering or weighting rules according to how well they've performed in played games. The trade-off for this possible increase in playing performance is that these reorderings or rule weights may no longer be as intuitive or even understandable to the human experts.

### 5.1.1 Joseki sequences

An important use for these pre-established rules is in the early opening game, where even human professional players know a large number of possible opening move sequences in the corners called joseki sequences. The moves in these joseki sequences are considered the best moves in these situations, and in general any deviation from them will produce a worse result. Human professionals know many such sequences, but they understand them at a deeper level and can adapt the moves when the specific board situation demands it, which happens on a relatively frequent basis. Another very important skill professionals possess is knowing which of the sequences to follow in each situation, since there are typically several that start from the same initial stone position.

For a computer player, getting this deeper understanding of the moves does not seem to be feasible at this time, since a solid understanding of Go is required to be able to study these sequences. Because of this, extended study of such sequences is only recommended for stronger amateur human players, which are players that devote a significant amount of time to the game and can beat the best contemporary Go programs today. Weaker human amateurs typically know a couple of sequences, but not the meaning behind each play or how to react if the opponent plays something else.

A computer player can easily replicate the plays from the general joseki sequences, and at least against weaker amateur players this may be enough to achieve a decent opening strategy. Because of this, some very useful joseki sequences are likely to be beneficial as part of a program's Go knowledge. However, the value of adding a larger set of joseki sequences is likely to be limited by the lack of ability to apply them properly in the way stronger human players can.

### 5.1.2 GNU Go

One of the best knowledge-based computer Go programs is GNU Go[2], an open source project that is developed by volunteers all over the world. It uses large amounts of expert knowledge encoded into the program for various purposes and at several different levels of abstraction.

The program begins by separating the stones into *worms*, which are also often called *strings* and consist of contiguous 4-connected groups of stones (which will always share the same life and death status). Afterwards these worms may be combined into *dragons*, which are not expressly connected but will still be considered as a whole by the program. This is useful because live groups can be made more efficiently by not having all the stones directly connected, since fully connecting them would naturally require more moves. The program assumes that these dragons are connected and will also live or die together, just like the stones in a worm.

Afterwards, GNU Go uses several special-purpose move generators to obtain a list of possible next moves, which include moves to play specifically in the opening game (based on a fuseki database), to capture/defend a worm, kill/defend a dragon, or break into the opponent's territory. It can also perform pattern-based moves that try to enable a future attack or defense, or make good shape if specific recognized patterns of stones are on the board. It also includes special endgame moves that are only considered after there are no more other interesting moves to be made.

Then these possible moves are evaluated, which is primarily based on the difference they make to the players' expected territories, but also including some estimates of strategic importance and good shape. This separate step is performed to discover when a move that was suggested by one generator can also be beneficial for another purpose, and as mentioned earlier such double-purpose moves are worth significantly more than moves that only do one thing.

Another benefit is that this separate evaluation allows for some kinds of tuning, such as how to value influence compared to territory, or playing safer moves when ahead and more risky moves when behind.

## 5.2 Neural networks

There have also been many attempts[1, 11, 26, 10] to use neural networks for playing Go, with the hope that playing skill and understanding would emerge naturally. Most of these experiments have ended in failure, but when used in a larger framework to analyse specific subsets of the game, these methods have been shown to be viable and helpful.

Neural networks have been successfully trained to evaluate parts of Go, such as shape and the potential for territory in an area. Some success has also been achieved in segmenting the board into different parts that can then be analyzed. Since these areas are often not clearly defined, strictly separated or independent, it is possible that a neural network approach might be better for this problem than many other techniques.

The resulting plays from neural network programs also often seem more natural and human-like than other computer playing programs, which can be considered an advantage for some purposes.

Unfortunately the use of neural networks for playing Go does not seem to be as extensively examined as other approaches, perhaps because the most basic intuitive method of using the entire board as input produces discouragingly bad results.

Other forms of subsymbolic, evolutionary or biologically inspired approaches have not received significant attention for implementing Go programs. This is probably because neural networks can more easily be adapted to use board positions as input, and are generally more popular for creating game-playing programs.

## 5.3 Monte Carlo Go

Monte Carlo[27] methods are a class of algorithms for solving tough computational problems. They are based on probabilistic processes and have bounded execution times by only performing a certain number of computations, but because only some possibilities are tried they do not necessarily produce an optimal solution.

Monte Carlo methods are typically used when the dependencies between multiple variables are known, but the problem is too difficult or costly to solve analytically or using other methods. An example of this is primality testing, that is determining whether a certain number is prime. It is possible to determine this with 100% certainty in polynomial time, but using probabilistic methods is the most popular way because they can detect composite numbers with a very high probability much faster than any known deterministic algorithm.

In Computer Go, Monte Carlo methods are used to find a good result, but without guaranteeing that the opponent has no clever moves which would lead to a poor result for the computer player. Such Monte Carlo Go methods[28, 3] have become very popular recently, mostly because of the very good results achieved using an algorithm known as UCT to guide the probabilistic search.

These methods are based on performing the same very simple procedures thousands or millions of times, which aligns well with the brute computing power of modern machines. The aggregate of the results from these many experiments are then used to eventually decide where to play.

These procedures are naturally too simplistic to directly capture the strategical and tactical depth in Go. Instead the approach relies on analysing very many games to detect important moves and tactics indirectly as part of the normal experimentation, without the computer having a true understanding of their meaning and purpose.

For Go, the success of these approaches strongly correspond to how much of the game tree they explore, and because of this the best results have been obtained for the smaller $9 \times 9$ board size. For the exponentially larger game tree associated with full $19 \times 19$ boards, this partial exploration has not been as successful in discerning important plays, since the number of possibilities is of a completely different order of magnitude.

### 5.3.1 UCT algorithm

The Upper Confidence Bounds applied to Trees (UCT)[29] algorithm provides a way to choose between exploiting previous good results and exploring other possibilities that provide uncertain results, as in the classic multi-armed bandit problem. UCT also support selecting between such nodes when they are part of a tree, and for computer Go it is used to determine which node to further examine in the planned future game tree.

The choice between examining known good nodes and uncertain nodes that could potentially be better is represented by using a bias $c_{t,s} = \sqrt{\frac{2 \ln t}{s}}$, where $t$ is the total number of node evaluations performed, and $s$ is the number of times the specific node has been evaluated. The node with the highest sum of expected reward and bias is selected as the next node to examine. Because of the $\ln t$ term, even bad nodes are guaranteed to be revisited eventually, and given unlimited time the algorithm will always converge to the best result.

The original paper also includes a stronger convergence proof that relies on using biases multiplied by some problem-specific constants $C_p$. Assuming that all these problem-specific constants are known, it guarantees that the failure probability at the root node converges to zero at a polynomial rate for all evaluations after the first $N_0$ evaluations. However, this result is mostly of theoretical interest since finding the $C_p$ constants is not feasible for Go game trees and $N_0$ might be very large and never even reachable by the computer player.

When used in practice for computer Go[30], the algorithm simply performs as many evaluations as possible under the game's time restrictions, without any guarantees about failure rates. Existing implementations typically perform these node evaluations by playing the game from the node position to the end of the game using randomized moves, and determine a score of 0 or 1 based on whether the computer player lost or won at the end of this hypothetical continuation.

An illustration of how the UCT algorithm works is shown in figure 5.1. Every simulation begins at the root node, and the UCT values for exploring each of its child nodes are calculated. Then the algorithm moves to the child nodes with the highest calculated UCT value, and the process is repeated for this node's children.

In this way the algorithm moves through the game tree until it reaches a leaf node. At that point the game will be quickly simulated using a much simpler random play method until end position is reached, and the score for this end position is calculated. The value of the leaf node will be updated according to whether or not the computer player won the game using this hypothetical continuation of the game.

### 5.3.2 Efficient game simulation

The most important aspect of these approaches is that they rely on a very large number of game simulations, and because of this the system should be able to perform the individual simulations as quickly as possible. The strength of the approach depends on the number of simulations, and the number of simulations possible in a game playing situation naturally depends on the time it takes to perform each individual simulation.

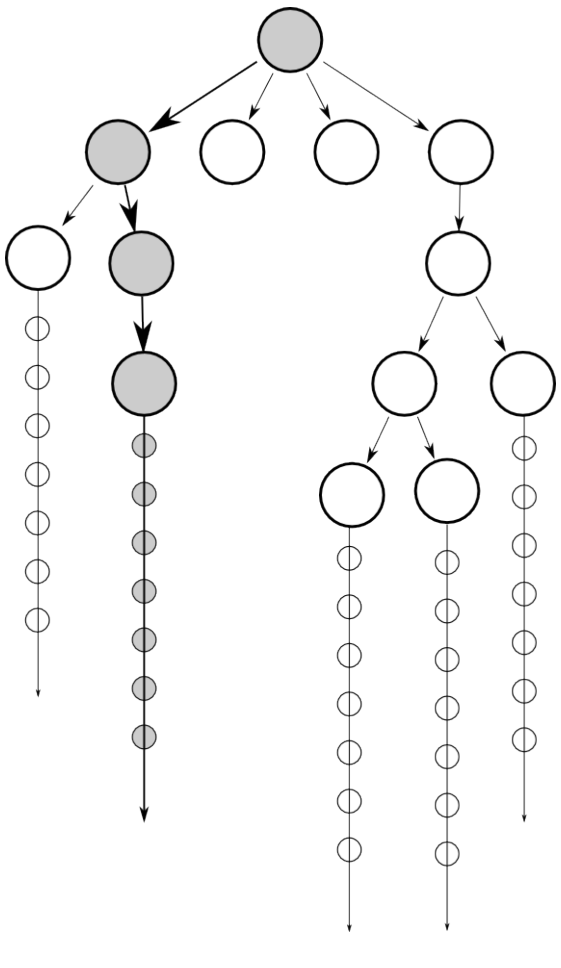


Figure 5.1: An illustrative example of a small UCT node tree explored using Monte Carlo methods. Every node in the graph has been further explored using a random playing algorithm (shown using rows of smaller circles) until a final end position has been reached. This includes the initial evaluations of the internal nodes before they were expanded, but for clarity it is only shown for the current leaf nodes in this figure. The actual UCT node trees used to play Go are naturally much larger.

This leads to a trade-off between the quality of the simulations and the time it takes to ensure this quality. Having each simulation consist of purely random play is one possibility, and often the starting point for most methods[3]. The next step is to add some understanding to the moves such that the games bear a closer resemblance to how real Go games are played. In practice some extra knowledge about Go has been shown to improve results, but the amount should be surprisingly limited. Two reasons for this are the aforementioned reliance on quick simulations, and the fact that often the best move will not be considered when using these more advanced methods.

This is natural, since otherwise the methods themselves could just be used to play the game directly. Random moves on the other hand will always consider all possible legal moves, and with a large enough number of simulations the beginning of much better sequences may be encountered even in this random manner. The problem is that the number of simulations required to get the same confidence in the results grows exponentially in the number of steps in the sequence. This means that there is another trade-off between the number of moves explored at each step, and the maximum length of sequences that can be covered.

The way these trade-offs should be handled has not been fully determined, but as mentioned the use of some limited form of Go knowledge when playing simulations has produced the best results.

## 5.4 Playing strength

The three outlined approaches, heavily knowledge-based, neural network and Monte Carlo simulations, are all quite different. Monte Carlo simulations have a greater emphasis on building and searching the game tree extensively, although all methods consider different possible future move sequences to some degree.

Surprisingly, the methodologies tend to produce program of somewhat similar strengths, which is around that of a mediocre amateur human player. All the approaches also have some characteristic quirks and weaknesses, and even weaker human players can typically beat the programs after playing a number of games against them and learning to take advantage of these weakness.

Knowledge-based programs were the first attempts at computer Go players, and neural network approaches appeared as a bit of a contrast to the more traditional approach. The development of strong Monte Carlo programs is relatively recent, and unlike the other approaches, Monte Carlo programs have also been proven to be rather strong at the smaller $9 \times 9$ board, at a level where they can often beat quite strong amateur players and even provide a challenge for professional players.

# Chapter 6

# CBR and games

Case-based reasoning (CBR) is a powerful problem solving approach based on reusing previous experience. Since the main focus for improving the Go player developed in this thesis will be through reusing previous experience, previous case-based approaches to game playing may provide valuable insights.

This chapter will examine some existing frameworks for developing and using case-based methods, and also examine previous attempts to use CBR as part of game AI.

## 6.1 jCOLIBRI

jCOLIBRI[31] is a domain independent Java framework for CBR development. It allows the usage of generic problem-solving methods that can be applied to many different tasks and domains through common ontology-based knowledge representations. Because of this jCOLIBRI can be useful for comparing and contrasting knowledge-intensive CBR approaches.

In our project, we will examine and experiment with different ways to generate and reuse experience. Using existing implementations for existing problem-solving methods would not be suitable for this kind of work

Generalizing the developed method to handle other problems would be an interesting further development which could benefit from using jCOLIBRI, but the current project is narrowly focused on developing a method that only has to work for playing Go. However, any such future development would be unlikely to benefit from reusing the specific prototype implementation of the program developed in this project.

## 6.2 TIELT

TIELT[32] is a software tool for integrating AI systems with gaming simulators. Compared to jCOLIBRI, TIELT provides an abstraction of the games played instead of focusing on assisting the internal reasoning processes in the AI system. Much of the work done on TIELT concerns real-time strategy games and military combat, which typically deal with imperfect information and randomness. Our AI will be focused entirely on Go, and is not intended to work in other games or scenarios. Because of this we will not develop a TIELT interface for our Go playing program.

## 6.3 Planning tactical operations

Other projects have also attempted to use case-based reasoning as a part of game-playing agents, sometimes using TIELT to provide the game mechanics. These approaches are normally based on having various low-level operations that can be performed in real-time tactical combat systems, and reuse plans for ordering sequences of such operations.

The most prominent examples of this are programs that play game types based on Civilization[33] and Warcraft[34], two popular computer games that have been used to evaluate AI approaches since they are non-trivial and it is not easy to beat human players.

In these games the significant improvement achieved by using CBR has often been that such plans of operations are retrieved and reused to directly react to the opponent's behavior, while many other methodologies have only worked (but often extremely well) against static opponents that do not adapt or change strategies during play.

## 6.4 Creating game stories

Another use of case-based reasoning in the game domain has been as part of plot generation[35]. A problem with many computer games is that the most interesting parts of the games contain specific plots that were created explicitly by the human game designers. This means that although the game can be very entertaining, these plot elements will always remain the same, which means that replaying the game a second time will typically not be as enjoyable.

An approach to address this is to use case-based reasoning to generate such plots automatically from a case base of possible plot elements. Although such automatically produced plots can often be of a lower quality and a bit bland, the prospect of doing this automatically is nevertheless appealing.

## 6.5 Using CBR for Go games

For this project, a large number of professional game records are available for reuse, but the game records only contain the move sequences and no information about their purpose or how the moves are related to each other, neither tactically nor strategically. This information is considered to simply be unavailable, since automatically reconstructing these relations would require a deep understanding that in itself would be sufficient to play the game better than any current computer programs. An alternative approach would be to have human experts annotate the games, but doing so for a significant number of games is not feasible as part of this project.

Because this information is not available, the tactical plan-based reuse employed in real-time games is not directly applicable, and some other method will have to be used. A similar approach to the one used for game story creation could be applied to Go to make the computer program behave in different ways each game. This could be useful, especially to let human players enjoy playing against the computer without having it do exactly the same blunders every time, but it does not directly make the computer player stronger.

Because of this the approach used in this project will be slightly different. It will focus on being able to quickly reuse game positions, where the reuse process and the game positions do not rely on deep analysis or use of Go-specific theory. Instead the process will be used to quickly find moves that may be good, and rely on the other parts in the system to actually verify whether they produce good results in the given game position.

This approach allows all the available game records to be reused directly without a human expert to analyze and transform them. It should also hopefully work well with the Monte Carlo UCT approach to playing Go, which is also based on quickly performing a large number of possibly inaccurate evaluations.

# Chapter 7

# Using cases

Human players can be said to use experience in at least two different ways. One is by generalizing these experiences into abstract concepts that are then remembered and used to understanding future situations. When these concepts are reused, they may also be modified based on how applicable they were, or if something unexpected happened.

Another way experience is used is in the case of unexpected events. These events are typically remembered explicitly, and humans may be reminded of the particular previous incident if something similar is about to happen. Normally only the most important and characteristic parts of such incidents are remembered, which indicates that some method of abstraction is applied even in such cases.

When trying to reuse experience for computers, similar approaches have been used. The results of these attempts show that generalizing and abstracting seem to be very difficult concepts to implement in a sufficiently advanced manner. Because of this, there have been many attempts to let the computer somehow abstract the information on its own, with the human programmer only designing and knowing the general approach and not exactly how the abstractions are formed or represented.

Another approach is to exploit the vast storage available in computers, and not generalize anything until it is necessary. This typically means to wait until something should be analyzed, and then only looking at the experiences that are most relevant for the specific problem at hand. It is hoped that this type of delayed learning will allow for more accurate analysis, or just that simpler methods may be sufficient to solve a specific complex task if a similar solution already exists.

This latter approach will be employed in this master's thesis, and the experience to be reused will be complete game records of previous professional games. When the computer program is playing, these will be consulted to see if any of the professional players' moves can be adapted to the current situation.

## 7.1 Previous games

The program will attempt to reuse game records of previous games that only contain the order in which the moves were played on the board, with no sort of deeper semantic understanding or explanations. This means that the game record itself cannot be used to directly determine exactly what is important on the board at any given time. We will not attempt to infer exactly what the original purpose behind each move was, but instead use the professional player's moves as guidelines to find which parts and areas of the board are the most interesting to evaluate.

This restriction is natural, since there is currently no form of syntax or terminology that accurately describe the concepts these professional players apply when deciding on moves. As has been noted before, much of the advanced Go theory is formulated through proverbs and inexact statements that do not always apply, and they should instead be processed and understood as vague guidelines.

Amateur players are typically not able to understand the reasoning behind the moves found in professional games, because the amateur player does not know how to locate and evaluate the most important aspects of the situations that occur in such games. By having a better sense of which moves and tactics will become interesting in a situation, a strong player will often be able to look ahead a dozen moves further than an amateur, which corresponds to evaluating a completely different order of magnitude of possible game sequences.

Instead amateur players are suggested to simply look quickly through such advanced games, to get a "feel" for how moves are placed in succession instead of focusing too much on analyzing exactly how the move affects the local tactical situation.

This has been recreated with some success in computer programs[36, 37], by using statistical approaches based on tens of thousands of professional and strong amateur player games to build up fairly accurate move predictions. Some of the best programs can even predict the exact move a professional player will play about 1/6th of the time, which is rather impressive compared to programs that attempt to actually play.

When these techniques are used to actually generate moves to play in a game, this is typically much less successful. Normally many of the moves will be very good, but then a few of them will be disastrous and negate most of the advantage of all the professional-level moves preceding it, due to the computer program simply having no idea how the stones actually work together.

However, in Monte Carlo-based approaches similar concepts have been used to produce more plausible game sequences when combined with some local tactical patterns. Although each game is still fairly poorly played, combining thousands or millions of such games as is typically done has been shown to provide better results than random play[3].

## 7.2 Reusing strategic plans

In our approach, the previous games will instead be used to attempt to locate the most important parts to play on the board. As noted above, this is optimistic since even fairly strong amateur human players do not really understand the move sequences in professional games. However, it may provide better results than approaches based on random play.

There are two main differences between this approach and the other Monte Carlo methods outlined above. One is that we will attempt to use the previous game experience only at the game tree exploration level, and still use random play to determine the result of each game. The other is that the games will be stored explicitly and only a handful of the most similar board positions will be used to analyze each position.

This is in contrast to the statistical approaches, which use all the hundreds of thousands of games to build up probability tables that are then always applied in the same way without any means to differentiate between original games that matched the current game better. In our approach, we will attempt to find moves specifically for the current game, instead of finding moves that may be good in general.

### 7.2.1 Matching similar board positions

The first important challenge is determining which of the previous games are similar enough to the current game that the moves may be adapted for reuse. Since the goal is to find the general strategic direction the game should move in, this comparison will be based more on the strength and influence players have in areas than on the local tactical situation.

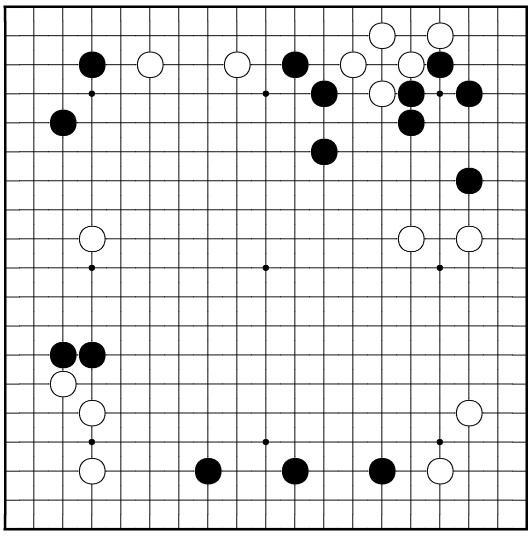
To accomplish this, an influence map is created for each board position, which contains the estimated areas of the board each player is in control of. This is similar to the approaches normally used for score estimation, where each stone spreads a certain amount of influence to the surrounding area.

In our approach, this influence is spread as a flow that cannot move through stones, whereas the most common approach is to simply base the influence on the distance from a stone without considering the surrounding area. When manually inspecting the influence maps generated, this flow-based influence spreading seemed to provide more accurate results. An example influence map generated by this algorithm is shown in figure 7.1.

The similarity between different board position is then calculated by counting how many intersections on the board has the same player controlling them in the generated influence maps. This provides fairly good results, at least for the first opening moves.

### 7.2.2 Evaluating and adapting the previous plan

Another problem is determining how to generate a new move based on the current board position and the professional player's move in the matched game. We take a simplistic approach, where the areas of the board that are in nearly the same position as the professional player's move are given priority over other moves. This is done by exploring these moves first when evaluating possible moves, and providing a small bonus to the estimated score for these moves.


Original board position

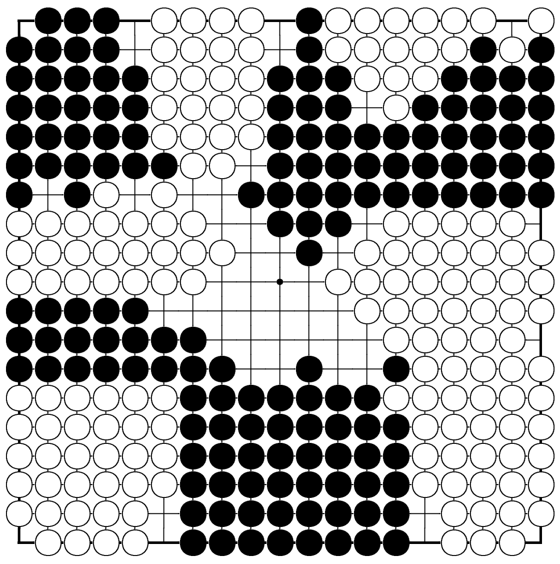

Influence map generated for the shown position

Figure 7.1: Influence map for a sample board position. The original board position is shown at the top, and the corresponding influence map is shown below.

## 7.3 Finding vital tactical moves

Another interesting use for previous game records is in reusing local tactical moves. If a local tactical situation is the same as in the previous game, the moves from the previous game may be reused in the current situation. If it is a perfect match the moves can simply be reused directly, while otherwise they may be used as guidelines for which types of moves are most promising.

### 7.3.1 Matching fighting situations

The benefit of only matching local tactical situations is that they are much smaller, and thus more likely to exactly match something that has been played before. Unfortunately these local situations are never completely independent of the rest of the board, most directly through the number of ko threats available, which often decides the outcome of a fight.

In our approach, only the common eyespace shapes outlined in chapter 3 will be used for tactical matching. Reusing the professional game case base would be preferable, but this would require a method for discovering when fights occur in these games. This can be very difficult since professional-level players rarely enter a fight unless they are likely to win, and instead indirectly threaten to create such local situations that would give them an advantage unless the opponent responds. Because of this, even strong amateur players may have trouble determining when a group of stones is actually threatened, which means that it is unlikely to be easy for a computer program.

### 7.3.2 Comparing status of affected groups

An important aspect for the validity of the local tactical matching is the life or death status of the connected groups. Determining this status can often be very difficult, so our approach will ignore this aspect and always consider the eye-stealing moves interesting.

This means that the tactical moves will always be considered as possible next moves during move generation, and explored further than they normally would, but they will only actually be played if they lead to an increased chance of winning. The problem of actually determining how and when to reuse the moves is also addressed by only considering it for situations where the move can be directly copied. This limits the possible benefit from this approach since it will not always be applicable, but can be implemented rather easily in the cases where it works.

## 7.4 Overall use of previous experience

The most significant reuse of previous experience in the proposed Go playing system will be through biasing the moves in the opening game towards where professional players have played in similar situations. The system will not attempt to understand the underlying intention behind playing in these areas, but it is believed that playing better will nevertheless be an advantage compared to only using the UCT algorithm and random play.

Another proposed use of previous experience is through specific tactical situation cases created by a human expert where the most interesting positions have been marked. These interesting positions can then be focused on for the further game tree exploration, which should allow them to be explored sooner and deeper than during normal search. Since for this use the tactical moves will be reused almost directly with no adaptation, this approach is somewhat similar to the use of generalized patterns in other computer Go approaches.

# Chapter 8

# System Construction

In this chapter we present the overall design for our Go playing system, and give some details about how the various components interact to generate moves from a given positions.

The initial prototype implementation is also presented, with a description of how some of the most important parts of the system work.

## 8.1 Main system components

Our proposed Go playing system consists of four main components:

- Game representation
- Score estimation
- UCT game tree exploration
- Case-based reuse of previous games

All of these components will be included in various degrees as part of our prototype implementation in order to get a working computer Go playing program. The first three main parts are based on how other contemporary Go playing programs are constructed. The last part will be an example implementation of the main idea presented in this thesis, which is to use explicitly stored previous game records to influence game play.

A diagram of our system is shown in figure 8.1. **GoGame** stores all information about the games, including most importantly the current board position. The **CaseBase** component is used to retrieve the stored game that is most similar to the board position for the current game, based on the minimum difference in estimated board influence. The **UCT** component performs the UCT game tree exploration, based on random play score estimations performed by the **ScoreEstimator**. This score estimate also includes a bonus for moves that are close to the professional player move in the game record retrieved from the **CaseBase**.

The system works by first retrieving the most similar previous game from the case base. This is used to find the location of the professional player's move for a position that is similar to the current board situation, which means that

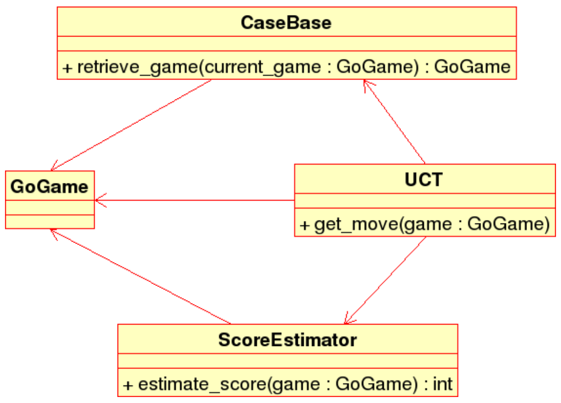

CaseBase
+ retrieve_game(current_game : GoGame) : GoGame
GoGame
UCT
+ get_move(game : GoGame)
ScoreEstimator
+ estimate_score(game : GoGame) : int

Figure 8.1: Diagram for the proposed Go playing system.

the same area of the board is likely to be interesting for the current game as well.

Afterwards the possible legal moves in the current situation are analyzed by the **UCT** game tree component. When reaching a leaf node during UCT evaluations, the system will play the rest of the game randomly using the **ScoreEstimator** component, and given an extra bonus if it is close to the move retrieved from the professional game. After performing a large number of such UCT game tree explorations followed by random play, the built up UCT tree will be used to find the move with the highest winning rate, and this move is returned by the system as the next move to play.

## 8.2 Game representation

In our program, a game is represented as a sequence of full board positions. Support has been added for reading game records in the Smart Game Format (SGF)[38], which is commonly used to store Go games. The SGF file format itself only contains the sequence of moves the players performed, and the exact results have to be calculated by the computer program reading them. Our program reads a sequence of moves from an SGF file, and returns a sequence of the resulting board positions, with the Go rules applied to remove stones that have been captured.

### 8.2.1 Board positions

Two different approaches for storing the individual board positions were implemented and compared. The first was a very simple approach, where each board simply contained an array of $N \times N$ intersections to store the information (either empty, black stone or white stone).

The code for evaluating possible moves then had to implement all the game logic, such as identifying groups and determining whether they would get captured. Additionally, to implement the ko rule, Zobrist[39] hashes of all the previous board sequences in a game were stored to make sure a board position was not repeated. Zobrist hashing is a fast, reasonably collision resistant hashing method commonly used for chess and Go positions. Another benefit is that the hashes can be very quickly incrementally updated.

The other approach also included all the intersection data and Zobrist hashes, but in addition it explicitly stored information about what groups were on the board and where their liberties were. This allowed the algorithms for evaluating moves and estimating score to be much simpler and quicker to execute. However, the trade-off was that some additional computation (but simpler than that normally performed during evaluation) has to be done every time a new move is played on the board.

## 8.3 UCT

Due to the recent success of UCT-based Go playing programs[1, 40], the underlying game playing approach for this thesis is also based on UCT. The thesis includes a new UCT implementation created specifically for this thesis, to be able to easily modify parts of the algorithm for experimentation. Fully implementing the algorithm also allowed for a better understanding of how the algorithm works and which parts of it are suitable for such modifications.

### Node scores

When using UCT one of the most important decisions is which formula to use for scoring individual nodes in the tree, since this directly affects the order in which the nodes are explored. For Go playing programs using UCT, each node evaluation normally results in a value of 0 or 1, depending on whether the game was won or not.

The most common scoring algorithm for Go programs[30] is

$$UCTvalue = \frac{wins}{plays} + \sqrt{\frac{ln(parent.plays)}{5 \times plays}} \tag{8.1}$$

where *plays* is the number of times the node has been evaluated, *wins* is the number of times this has resulted in winning, *parent.plays* is the number of times the parent node in the game tree has been played and $ln()$ is the natural logarithm function.

This algorithm is also used in this thesis, although some modifications have been experimentally tested, such as also including the margin of victory in the individual evaluations instead of just whether it was a win or not. The scoring algorithm does not know that the win rate cannot be below 0 or above 1, and therefore will explore other options in an attempt to find win rates above 1 even if it has found a move that seems to never lose.

Some experiments using different expressions that avoid this problem have also been performed, but visual inspection of the resulting trees built using these alternative score equations have not resulted in any significant differences. Since the estimated win rate using random play tends to be around 0.3 - 0.8 after many simulations have been performed, this is not particularly surprising. Although these alternatives were equally viable and did not decrease performance, the more common UCT scoring algorithm for Go listed above will be used to allow for easier comparison with other work.

However, as was previously mentioned, a small bonus was also added on top of the UCT score when evaluating early opening moves that are placed close to a professional player's move in the most similar previous game. This encourages the program to explore these moves more deeply. When the program decides on a move this bonus is also included in the final node evaluation, which biases the program towards playing these moves unless they are clearly inferior to other moves the program has tested.

## 8.4 Result estimation

The UCT algorithm works very well in practice, and will explore the tree according to how good the results are for different branches in the game tree. However, this exploration is based on being able to estimate the score at any given node, and this is a difficult problem.

### 8.4.1 Score estimation

One very simple approach is to use normal score estimation techniques at each leaf node, such as our approach based on influence maps. The problem with this is that these score estimators typically have biases and there are many tactical situations they do not correctly classify. In particular, our score estimator performs no specific tactical analysis, which means that it rarely determines the correct status for tactical life and death problems. If the move generation is directly guided by such estimators the computer player will also have these problems, resulting in a poor playing performance.

Another aspect of score estimators is that they are normally deterministic and will always produce the same result for a given board position. For use with UCT tree exploration algorithm, it is normally better to use a leaf node evaluation function that will return a stochastic result based on the remaining uncertainty in the position, since the algorithm handles this very well and by repeated application it can produce a score distribution which is more expressive than one single score result.

### 8.4.2 Random play

Another very popular approach, and the basis for the recent success and popularity of UCT-based algorithms, is to use a Monte Carlo-inspired approach and estimating the score by randomly playing the game to the end and evaluating this end result[28]. The individual results will be close to random, but if they are biased towards winning for good initial positions, the UCT exploration approach will be able to relatively quickly identify the most successful moves.

When sufficiently many games (at least tens of thousands) are played, this approach also works in practice. Random play does not always work as well in highly tactical situations, but as long as enough trials are performed it will normally find many good game sequences simply by chance.

Two random play approaches were implemented for this thesis work. One is based on playing the game one stone at a time until the end, using mostly random play but influenced by some guidelines and patterns to achieve somewhat natural-looking play. Using some guidelines and pattern in this way has been successfully implemented by other UCT-based program authors[3], and our version also performed better than a purely random version without these features enabled.

The other approach is based on using an even quicker but less accurate method to play the rest of the game. All the remaining empty intersections on the board are filled in randomly with black and white stones, without considering any game rules. Then all the smaller groups on the board consisting of 3 stones or less are removed, and the resulting position is then used for the evaluation after removing dead groups that do not have two eyes. This method is significantly faster and actually also produces ending board positions that look somewhat similar to real game results.

### 8.4.3 Group status evaluation

Evaluating the status of a group can be very difficult, particularly in the case of nearby groups that both threaten each other and where neither of them will be alive without killing the other group.

In our approach the only place where this group status evaluation is absolutely needed is for the final part of the random fill algorithm for board score estimation. Since this approach is based on performing a comparatively larger number of trials with less accuracy, it also means that the group status evaluation does not necessarily have to be accurate, as long as it is fast.

With this in mind, a simple group status evaluation approach was created, which is simply based on the number and size of the eyespaces bordering each group. A group that encloses two eyespaces (or one eyespace that is larger than 5 spaces) is assumed to be alive, and a group that is only connected to one eyespace and the opponent's groups is considered dead.

If eyespaces are shared between groups of both players and neither of the groups are independently alive or dead according to this procedure, they are assumed to both be alive in seki.

This approach produces relatively good results for the kind of positions that result from the random fill procedure, even though it is not nearly general enough to work correctly for all endgame positions, and it does not work at all for evaluating group status in the middle of the game.

### 8.4.4 Local tactical play

Similarly, the functionality for determining decent tactical play in an area also has to be relatively quick, which means that it cannot be too advanced and certainly not always correct (which in itself is known to be too difficult due to the possible complexities that can theoretically occur, as previously mentioned.)

To accommodate this, the tactical play evaluation consist of a number of separate tactical evaluators that are then used to determine a move.

The first of these is based on capturing, where the program will capture or defend groups if necessary, where bigger groups are given priority.

Another evaluator uses the local playing patterns employed for tactical play in MoGo[3] (a strong Monte Carlo-based Go program), which was shown to produce significantly better results than random play in their experiments.

Thirdly the program will attempt to play in relatively free parts of the board around the 3rd or 4th line, which based on our personal playing experience are

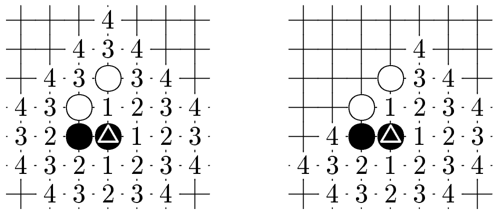


Figure 8.2: Distance from each intersection to the marked black stone. A common metric-based approach is shown on the left, while our flow-based approach is shown on the right. The benefit of our algorithm is that the influence is limited by nearby stones, which corresponds better to how influence is evaluated by human players.

often valuable positions.

Otherwise, if there are no moves that fulfill any of these requirements, the program will simply choose and play a random legal move.

## 8.5 CBR approach

The main idea behind our CBR-based approach is the reuse of previous games as explained in section 7.2. An important part of this approach is our influence map algorithm, since this algorithm forms the basis for assessing the similarity between various board positions.

An illustrative example of how this flow-based algorithm works in practice is shown in figure 8.2. On the left a more common distance-based method is shown[1], while our flow-based approach is shown on the right.

In each case the numbers represent the distance each intersection is away from the marked black stone. The influence a stone exerts on an intersection is inverserly proportional to the distance to this intersection. For our algorithm, intersections that are more than 5 steps away are not counted as being influenced by a stone.

(Our final algorithm is similar to the one shown in the example, but it actually sends out multiple such streams of flows simultaneously in each direction. This further enhances the ability of nearby stones to block or reduce the amount of influence that can be exerted around them, but is based on the same concept as shown in the illustration.)

Before games are added to our case base, they are processed using this algorithm to spread influence from every stone on the board. This is used to generate the influence map based on the sum of influence exerted on each intersection by the black and white stones. If the sum of black influence for an intersection is signficantly higher than the sum of white influence, this intersection will be marked as belonging to black in the influence map. A similar process is used for white influence, and the intersection is marked as empty if the sums of white and black influence are nearly equal (and in particular when both sums are 0 because there are no stones nearby).

[1]In this case using Manhattan distance, which emphasis the 4-connectedness requirement for stone groups. Euclidean distance is also often used as an alternative distance metric.

Each case in our case base contains the corresponding influence map as an index which is used when finding the most similar board position in the case base. When the current board position should be used to retrieve one of the previous games from the case base, an influence map is first generated for the current board position as well. This influence map is then compared to the influence maps for all the previous board positions stored in the case game.

Each individual influence map comparison is performed by counting the number of intersections that has the same classification in both influence map. This means that for each intersection there are only two possibilities, either they are exact matches or they are not. No additional penalties are applied for intersections that belong to black in one influence map and white in the other.

During retrieval, the case base will return the game position with the influence map that is most similar to the current board position's influence map when scored in this way.

## 8.6 Implementation

The code created as part of the thesis work consists of prototype implementations of the main system components outlined above. Some of the planned features were first tested in Python to make sure they were feasible, while the final system was programmed in C++ because of the demand for very quick execution of the most frequently repeated code in the system. Using the UCT algorithm, the resulting skill of the computer player is directly affected by how many operations the program can perform and how deep the searches can explore the game tree.

A class diagram illustrating the implementation at a high level is shown in figure 8.3. The code contains many other utility functions and low level details that are not included for brevity, but they implement the same functionality as shown in the abstracted class diagram.

The **GoBoard** class contains an array describing the board position itself, and whether each intersection has a black stone, a white stone or is empty. Additionally it keeps track of the current player to move, the current score based on komi and captures and it maintains the Zobrist hash code for the current board position.

The **GoBoard** class also contains some of the most important methods in the entire system. First among these is the $play()$ function, which adds a stone to the specific position for the current player, and updates the board position according to the Go rules. Another important method is $evaluate\_move()$, which performs a tactical evaluation of a proposed move and determines how big or important the move is, as described in section 8.4.4. The final important method in the **GoBoard** class is $estimate\_score()$, which quickly plays the game until the end using random moves and returns the resulting score at the end of this hypothetical continuation.

The **UCT** and **UCT_node** classes respectively implement the game tree exploration algorithm and the representation of nodes in the game tree. The $perform\_simulation()$ method will explore the game tree until it reaches a leaf node and then call $estimate\_score()$ to obtain an evaluation of the board position at the leaf node. The $get\_best\_move()$ method will return the move with the highest chance of success (biased by proximity to the professional player's

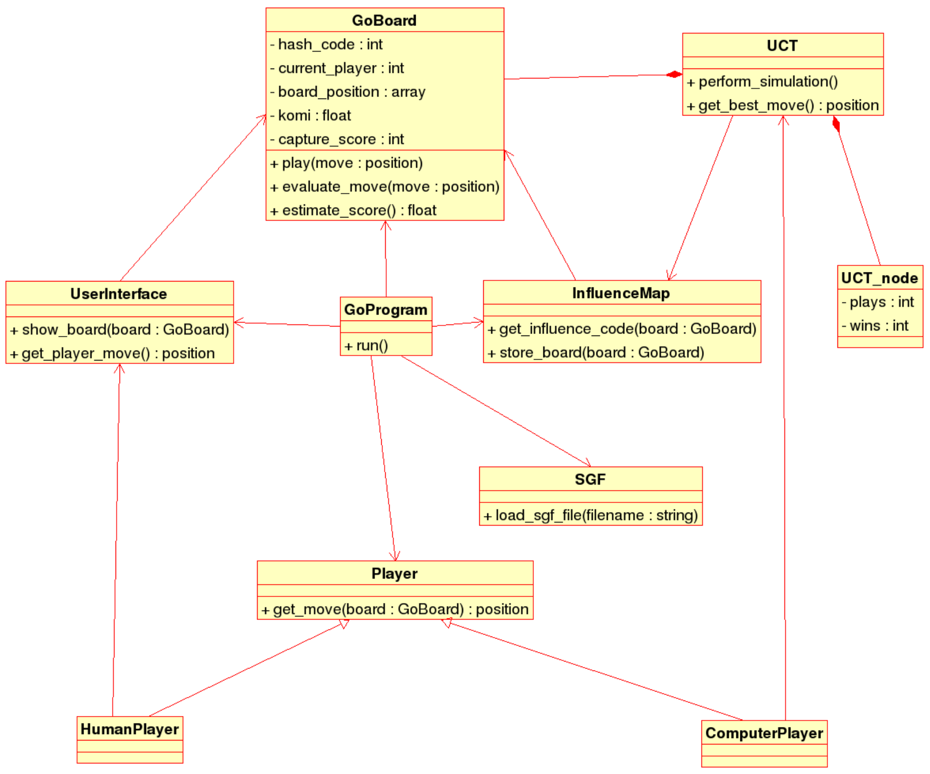

GoBoard
- hash_code : int
- current_player : int
- board_position : array
- komi : float
- capture_score : int
+ play(move : position)
+ evaluate_move(move : position)
+ estimate_score() : float
UCT
+ perform_simulation()
+ get_best_move() : position
UserInterface
+ show_board(board : GoBoard)
+ get_player_move() : position
GoProgram
+ run()
InfluenceMap
+ get_influence_code(board : GoBoard)
+ store_board(board : GoBoard)
UCT_node
- plays : int
- wins : int
SGF
+ load_sgf_file(filename : string)
Player
+ get_move(board : GoBoard) : position
HumanPlayer
ComputerPlayer

Figure 8.3: Class diagram illustrating the major parts of the prototype implementation.

move) and is called by the **ComputerPlayer** class after all the simulations that will be performed for the current move have been completed.

The **InfluenceMap** class contains the case base of professional game records and the functionality for retrieving these based on the influence map of the current board position.

The **Player** class is an abstract base class that allow computer and human players to be used interchangeably by the control system, which allows for changing who controls the black and white player, and the possibility of letting the computer play against itself.

The **UserInterface** class provides a simple textual user interface that displays the current board position and allows the player to input moves by specifying the desired coordinates, which is used by the **HumanPlayer** class to let the human operator control one of the players.

The **GoProgram** class is the main entry point for our program and is used to set up game conditions and initiate instances of the other classes. It also uses the **SGF** class to load the files containing professional game records and add them to **InfluenceMap**.

### 8.6.1 Previous game representation

Before board positions are stored in the **InfluenceMap** class, the influence map method described in 7.2.1 is used to determine an influence code representing the board position. Each stored game position consists of such an influence code that is used during retrieval to find the most similar case, and the position played by the professional player which is reused to provide a bias during move generation.

To retrieve a previous game from the case base, the influence code describing the current position is compared to the influence codes for all the previously stored game board positions, and the position corresponding to the most similar influence code is returned. The way these influence codes are compared is by examining each intersection and determining whether it is equal to the current board or different. The previous game position that has the same classification for the highest number of board intersections is used as the closest match.

Figure 8.4 shows an example of a previous game record as stored by the **InfluenceMap** class. The **code** element contains the influence map encoded as a sequence of bits using three bits per intersection. Exactly one of these three bits will be set, representing whether the intersection is empty, controlled by black or controlled by white.

By using this bit-based representation, the similarity comparisons can be performed very quickly. When two influence codes are being compared, first a bitwise AND operations is performed, and then the matching score can be found by counting the number of bits that are set in the result.

The **position** element contains the position that was played by the professional player in this situation, while **player** contains which player's turn it was at this point. (This is encoded as 1 for black and 2 for white.) During case retrieval, only cases where it's the same player's turn will be considered. Finally the **filename** element contains the name of the SGF file this board position was taken from. This is only used for informational and debugging purposes.

| | |
|---|---|
| **code** | 010001010010010010010010100001010010010010001100100100001100010 1 |
| | 0001001001001001010000101001001001000110010010010010010010010001 |
| | 0010010100100100010010010010100100100100100100100100100100010001 |
| | 1001001000100100100101001001001001001001001001001001001001001001 |
| | 0001001001001010010010010010010010010010010010010010010010000101 |
| | 0010100100100100100100100100100100100100100100100100100001001010 |
| | 0010101000011000010100100100100100011001001000010010100100100100 |
| | 1001001001001001001001001001000110000100101001001001001001001001 |
| | 0010010010010010010010010001001001001010010010010010010010010010 |
| | 0100100100100100010010010010010010011001001001001000010100100100 |
| | 1001000100100100100100110010010010010010010001001001001001010000 |
| | 1001100001001100100100100100100100100010010010010010010100100100 |
| | 1001001001000100100100100100100100100100100100101001001001001001 |
| | 0010001001001001001001001001001001001001010010010010010010010001 |
| | 0010010010010010010010010010010100100100100100100100100010010010 |
| | 0100100100100100100100011001001001001001001001000010100100100100 |
| | 10001010010010001100100100100100100100100001010010010010001 |
| **position** | R18 |
| **player** | 1 |
| **filename** | games/sgf/Honinbo.Shusaku-Ota.Yuzo-1853-05-15.sgf |

Figure 8.4: Schematic for an example board position case from a professional game. The **code** element is used to index these cases for retrieval purposes, while the **position** and **player** are reused to generate to new moves using our approach.

# Chapter 9

# Results

In this chapter the results from the developed prototype are presented. It includes a demonstration of how the program is used to generate new moves, and the results from improving the system by reusing previous game records.

After the results from some experiments that led to further improvements of our initial implementation, these improvements and of our final prototype implementation are also presented with some additional explanations and comments.

We evaluate the games played by the prototype implementation according to our amateur understanding of Go, and our impressions of what types of moves strong and weak human players normally tend to play.

## 9.1 Implemented system

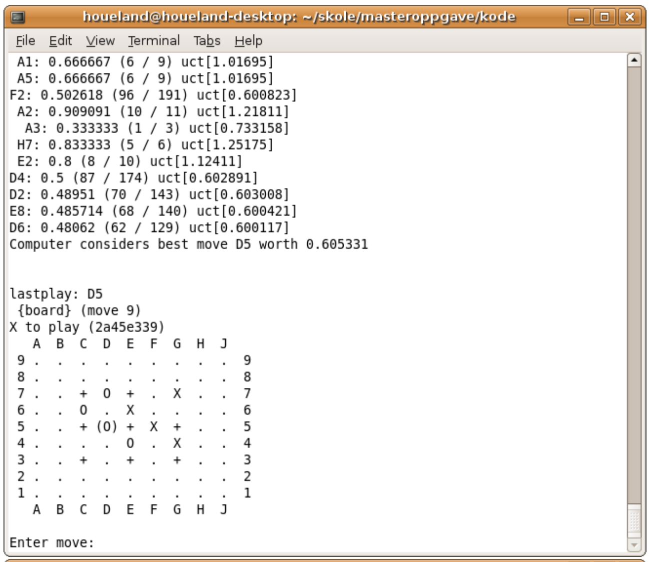
The prototype implementation provides a stable and functional computer Go player that can play reasonably well on a $9 \times 9$ board and provide a challenge to weaker amateur players. An example screen shot from a $9 \times 9$ game played against the implemented prototype is shown in figure 9.1.

However, for the larger $19 \times 19$ board, the implementation does not find good moves quickly enough to be a worthy opponent. In the very beginning of the opening game the implementation can find good moves relatively quickly by relying on the CBR-based reuse of previous professional games, but this is only feasible for approximately the first 20 moves. To play a real game, at least about 150 moves have to be performed, and often significantly more to finish the endgame.

### 9.1.1 Reuse of tactical situations

The planned case base also included descriptions of local tactical situations created by human experts, which could be reused directly if they were matched. However, this is not included in the implemented system.

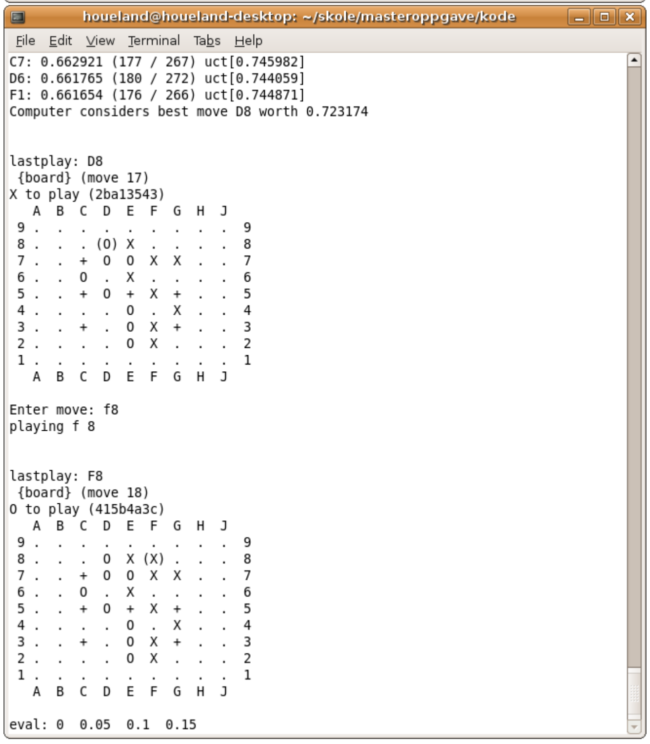
The reason for using tactical cases in our system was to locate vital points to kill groups with single eyespaces. However, our implementation of the basic UCT algorithm turned out to already be capable of finding these simple vital points and many others simply through exploring the game tree.


houeland@houeland-desktop: ~/skole/masteroppgave/kode
File Edit View Terminal Tabs Help

```
 A1: 0.666667 (6 / 9) uct[1.01695]
 A5: 0.666667 (6 / 9) uct[1.01695]
F2: 0.502618 (96 / 191) uct[0.600823]
 A2: 0.909091 (10 / 11) uct[1.21811]
  A3: 0.333333 (1 / 3) uct[0.733158]
 H7: 0.833333 (5 / 6) uct[1.25175]
 E2: 0.8 (8 / 10) uct[1.12411]
D4: 0.5 (87 / 174) uct[0.602891]
D2: 0.48951 (70 / 143) uct[0.603008]
E8: 0.485714 (68 / 140) uct[0.600421]
D6: 0.48062 (62 / 129) uct[0.600117]
Computer considers best move D5 worth 0.605331


lastplay: D5
 {board} (move 9)
X to play (2a45e339)
   A  B  C  D  E  F  G  H  J
 9 .  .  .  .  .  .  .  .  .  9
 8 .  .  .  .  .  .  .  .  .  8
 7 .  .  +  O  +  .  X  .  .  7
 6 .  .  O  .  X  .  .  .  .  6
 5 .  .  + (O) +  X  +  .  .  5
 4 .  .  .  .  O  .  X  .  .  4
 3 .  .  +  .  +  .  +  .  .  3
 2 .  .  .  .  .  .  .  .  .  2
 1 .  .  .  .  .  .  .  .  .  1
   A  B  C  D  E  F  G  H  J

Enter move:
```


houeland@houeland-desktop: ~/skole/masteroppgave/kode
File Edit View Terminal Tabs Help

```
C7: 0.662921 (177 / 267) uct[0.745982]
D6: 0.661765 (180 / 272) uct[0.744059]
F1: 0.661654 (176 / 266) uct[0.744871]
Computer considers best move D8 worth 0.723174


lastplay: D8
 {board} (move 17)
X to play (2ba13543)
   A  B  C  D  E  F  G  H  J
 9 .  .  .  .  .  .  .  .  .  9
 8 .  .  . (O) X  .  .  .  .  8
 7 .  .  +  O  O  X  X  .  .  7
 6 .  .  O  .  X  .  .  .  .  6
 5 .  .  +  O  +  X  +  .  .  5
 4 .  .  .  .  O  .  X  .  .  4
 3 .  .  +  .  O  X  +  .  .  3
 2 .  .  .  .  O  X  .  .  .  2
 1 .  .  .  .  .  .  .  .  .  1
   A  B  C  D  E  F  G  H  J

Enter move: f8
playing f 8


lastplay: F8
 {board} (move 18)
O to play (415b4a3c)
   A  B  C  D  E  F  G  H  J
 9 .  .  .  .  .  .  .  .  .  9
 8 .  .  .  O  X (X) .  .  .  8
 7 .  .  +  O  O  X  X  .  .  7
 6 .  .  O  .  X  .  .  .  .  6
 5 .  .  +  O  +  X  +  .  .  5
 4 .  .  .  .  O  .  X  .  .  4
 3 .  .  +  .  O  X  +  .  .  3
 2 .  .  .  .  O  X  .  .  .  2
 1 .  .  .  .  .  .  .  .  .  1
   A  B  C  D  E  F  G  H  J

eval: 0  0.05  0.1  0.15
```

Figure 9.1: Screenshots showing the prototype implementation playing a $9 \times 9$ board game. The first screenshot shows the position in the game after move 9, and the second screenshot shows moves 17 and 18 from later in the same game. Our implementation plays as white in these screenshots, marked as O, while a human player plays the black stones, marked with X.

Implementing the reuse of tactical cases might allow such moves to be found more quickly, which could possibly be beneficial for the large $19 \times 19$ board where our system is relatively slow. However, the isolated situations where this would be beneficial seem unlikely to result in a noticeable increase in general playing speed, and were thus left out of the prototype implementation.

## 9.2 Board representations

The two implemented board representations could both fully represent the current board position, but differed substantially in how computationally expensive it was to perform various computations using them. The first board representation that only contained the individual stone positions could usually be updated very quickly, but occasionally required expensive computations to find larger groups of stones and determining captures.

The second board representation contained a list of all the stone groups currently on the board and their liberties, which allowed for simpler algorithms when using the board state to determine tactical moves and also when performing captures. This representation is better suited for performing complex calculations on the board positions, but adds a small amount of overhead after every play to update the internal data structures.

In practice, the differences between these two board representations seemed to cancel each other out and did not result in any significant differences in playing speed. As expected, the first board representation was faster for purely random play, where a large number of moves were generated but with fewer computations at each step. When using the improved random player which incorporated some tactical considerations and simple playing patterns there were no substantial differences in total execution time between the two board representations. (An example of this is that one experiment using the first board representation and 10,000 simulations to generate the first move took 2.4 minutes, while using the second board representation this took 2.5 minutes.)

The second board representation was kept as the main board representation for further experiments, since it encapsulated some of the complex consequences of the Go rules without incurring a significant performance penalty. This allowed for simpler implementations of the other classes, which meant that it was easier to perform additional experiments without introducing programming errors.

## 9.3 UCT implementation

The UCT implementation developed as part of the system performs well and efficiently focuses on searching the most promising parts of the game tree. It is a basic implementation of UCT and does not include some of the optimizations that more advanced Monte Carlo programs include, which means that it can require more simulations than these programs to find good moves. However, our implementation is sufficiently efficient to perform the planned modifications and experiments and plays reasonably well on a small board.

An example of the UCT node tree after performing 10,000 simulations at the beginning of a new game on a $9 \times 9$ board is shown on the next page, as displayed by our prototype implementation. Each line corresponds to one node in the game tree, consisting of the move considered, the win ratio for the considered move and the UCT value associated with further exploring this node.

```
D5: 0.5 (261 / 522) uct[0.559404]
 E5: 0.8 (12 / 15) uct[1.08885]
  A2: 0.5 (2 / 4) uct[0.867971]
   A1: 0.666667 (2 / 3) uct[0.970673]
  A1: 0.333333 (1 / 3) uct[0.758229]
  A3: 0 (0 / 1) uct[0.735942]
 C7: 0.733333 (22 / 30) uct[0.937583]
  B8: 1 (2 / 2) uct[1.5832]
   A1: 0 (0 / 1) uct[0.37233]
  A8: 0.5 (2 / 4) uct[0.912383]
  A1: 0.333333 (1 / 3) uct[0.809512]
 F6: 0.730769 (19 / 26) uct[0.950168]
 D9: 0.666667 (10 / 15) uct[0.955519]
 F4: 0.666667 (10 / 15) uct[0.955519]
F6: 0.497967 (245 / 492) uct[0.559156]
 G5: 0.73913 (17 / 23) uct[0.971294]
  B5: 1 (1 / 1) uct[1.7919]
  A6: 0.6 (3 / 5) uct[0.954147]
  A1: 0.333333 (1 / 3) uct[0.790535]
 E5: 0.7 (14 / 20) uct[0.948967]
  B4: 1 (1 / 1) uct[1.77405]
  A5: 0.5 (2 / 4) uct[0.887023]
  B3: 0.5 (2 / 4) uct[0.887023]
 C7: 0.666667 (4 / 6) uct[1.12122]
 F5: 0.666667 (10 / 15) uct[0.95415]
 G7: 0.666667 (10 / 15) uct[0.95415]
E4: 0.493298 (184 / 373) uct[0.563572]
 F7: 0.75 (12 / 16) uct[1.02207]
  A9: 1 (1 / 1) uct[1.74466]
 C5: 0.733333 (11 / 15) uct[1.01432]
 E3: 0.714286 (10 / 14) uct[1.00514]
G3: 0.485294 (165 / 340) uct[0.5589]
E7: 0.48503 (162 / 334) uct[0.559294]
E8: 0.48503 (162 / 334) uct[0.559294]
F4: 0.48503 (162 / 334) uct[0.559294]
```

The first line `D5: 0.5 (261 / 522) uct[0.559404]` means that moving at D5 next is considered to be the most valuable move, with a win rate of 50%. It has been explored 522 times, and this has resulted in 261 wins for the computer player. The UCT value for exploring this node is also shown, which in this particular case is around 0.56.

The next line (`E5: 0.8 (12 / 15) uct[1.08885]`) is indented, which indicates that it is a child node of the line above. In this case it represents the opponent playing E5 after the computer first played D5. This has been explored 15 times, which has resulted in an 80% win rate for the opponent, and it is thus considered the most likely next move if the computer begins by playing D5.

The other lines are similar and combined they show the most interesting subset of the game tree after it has been explored by UCT for 10,000 simulations.

The most valuable next moves are considered to be, in order of preference, D5, F6, E4, G3, E7, E8 and F4. Most of these suggested moves are reasonable, but our amateur evaluation of them suggest that E8 is weaker than the others and that it should not be used as an opening move.

The computer considers moves at E5, C7, F6, D9 and F4 to be the strongest responses to a play at D5. Significantly fewer plays of these child nodes have been performed, which means that there is a much higher uncertainty regarding how strong these moves actually are. D9 is actually a very poor response and has only resulted in a high percentage of wins by chance, but in our opinion the other responses suggested by the computer are reasonable. At the third level in the game tree the uncertainty surrounding these moves are even greater, and many of these considered responses-to-responses are actually rather bad moves.

By performing a larger number of simulations, the computer program will examine a larger part of the game tree and find more accurate estimates for the considered moves. A smaller subset of a UCT game tree the implementation produced after performing 100,000 simulations is shown below. Using this increased number of simulations, the program is better able to evaluate moves properly and it produces a more reasonable list of strong moves. Correspondingly the program will also play stronger moves using this increased number of simulations (and play at around a medium amateur level), but for our implementation this takes around 20 minutes per move even on a 9 × 9 board.

```
E5: 0.55662 (9649 / 17335) uct[0.568145]
 F4: 0.501109 (904 / 1804) uct[0.534004]
  F5: 0.666667 (16 / 24) uct[0.916629]
  E3: 0.591549 (42 / 71) uct[0.736878]
 C3: 0.491262 (759 / 1545) uct[0.526808]
  E8: 0.612903 (38 / 62) uct[0.766807]
  F8: 0.612903 (38 / 62) uct[0.766807]
 D5: 0.475034 (352 / 741) uct[0.52636]
 E6: 0.471139 (302 / 641) uct[0.526324]
E4: 0.534617 (16602 / 31054) uct[0.543228]
 E7: 0.518549 (2502 / 4825) uct[0.539255]
 E3: 0.501403 (4288 / 8552) uct[0.516956]
E6: 0.532843 (10651 / 19989) uct[0.543576]
```

The evaluations for the first and second levels of the game tree are reasonable when using 100,000 simulations, but at the third level the program should not have considered E8 and F8 as good moves. However, this is still a significant improvement from using 10,000 simulations, where at the third level moves such as A1, A2 and A3 were considered, which would be very bad moves at that point in the game.

## 9.4 Score estimator

The score estimator developed based on the influence map algorithm provides reasonably accurate scores for many relatively difficult game positions. However, it does not include any knowledge about tactical situations or life and death evaluations, which means that it does not provide reasonable estimates if there are any big dead groups on the board.

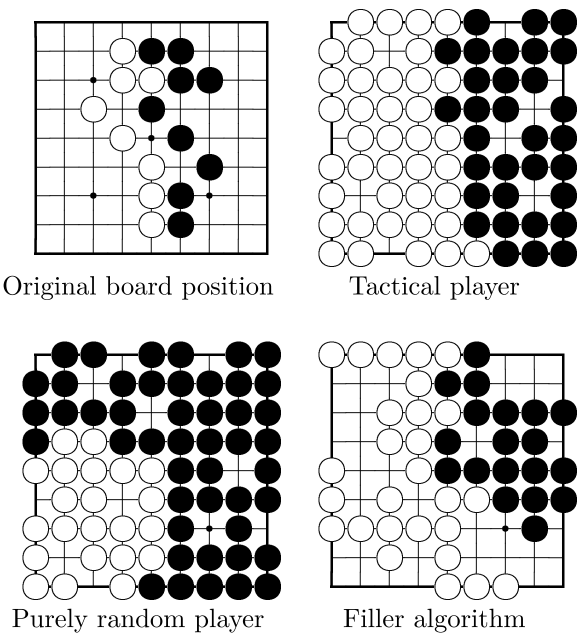


Figure 9.2: Random play continuations using the three implemented Monte-Carlo based methods.

Attempting to use the score estimator to provide evaluations for the UCT algorithm at the leaf nodes produced substantially worse results than using random play. It was also many times faster, but altogether it cannot be recommended as a viable alternative, particularly because it does not improve much by increasing the number of simulations performed. This was expected, since the UCT algorithm is designed to handle large numbers of stochastic estimates, and does not perform as well when using deterministic leaf node evaluations such as our score estimator.

## 9.5 Random play continuation

To evaluate the positions in the leaf nodes of the UCT game tree, three different Monte Carlo-based methods were implemented as part of our prototype implementation. Figure 9.2 shows an example of the kind of resulting positions these different methods generated.

Our first method was based on purely random play, where every legal move was considered equally likely at every step. This is the most basic, unbiased Monte Carlo-based approach, one that will never overlook any possible moves since all moves are considered equally important. Because of this it also has a large variance and it does not always lead to natural-looking final game positions.

As described in 8.4.4, a second method that extends the first purely random playing method was also implemented. This second method was based on the approaches that have successfully been used in other Monte Carlo-based UCT programs. It tries to capture and defend nearly surrounded groups when possible, and also contains some patterns for creating natural-looking move continuations. This bias means that it will not consider every possible resulting move sequence, but in return it will almost always provide more natural-looking positions at the end of the game.

The third method was based on the observation that our Monte Carlo-based methods were rather slow on the larger $19 \times 19$ board. This third method randomly fills in all the remaining empty intersections of the board, and then performs some basic life and death evaluations to make the end position look more like the end of a real Go game. The third method was significantly faster and provided good results in the early opening game for $19 \times 19$ boards, but unfortunately it performed poorly in highly tactical situations. This is not surprising though, since it does not contain the same tactical considerations that were explicitly added for the second method. This third method also has a large variance and can sometimes produce end positions that bear little resemblance to the original position. However, this variance is handled well by the UCT algorithm as long as a larger number of simulations performed. On a $19 \times 19$ board this method is an order of magnitude faster than the previous two methods, which allows many such extra simulations to be performed.

For our program the second method was used exclusively for $9 \times 9$ boards because of the increased emphasis on tactical play for this board size. For the full $19 \times 19$ board the third method which produces faster estimates is used for the first 40 moves. This allows the program to perform early opening moves much more quickly, while the corresponding loss in tactical accuracy is less important for this stage. After 40 moves the program will switch to the slower second method also for the large board games, because the faster third method does not work well in the middle game where tactical situations become increasingly important.

## 9.6 Reusing previous games

Our reuse of previous games is based on a collection of 1339 professional games in Smart Game Format (SGF) format. Some of these contained errors or unrecognized commands, while 1323 of them contained at least 40 moves that could be read by our SGF implementation. Examples of these errors were such things as the same player playing twice in a row or on top of another stone. Such illegal moves could have been used to indicate that the player resigned (which is how some human players indicate resignation), but our implementation simply considers this an error. Our implementation also discarded games that contained comments, since these were sometimes also used to indicate important information such as a player resigning. Since most of the games could be read successfully and form a sufficiently large case base on their own, our implementation simply ignores the games that contain such errors or unhandled elements.

Only the first 40 moves of each game were used, since the opening game is our main focus for strategically reusing previous games. The results from testing our implementation indicate that this restricted focus was a wise choice, because our approach is not particularly useful for finding good tactical moves. Such tactical evaluations often become very important in the middle game, and our CBR-based approach is thus not very useful for this part of the game.

These games cover a large collection of good opening sequences, but after about 20 moves the new games typically do not match any of the stored games any more, because at this point there are simply too many possible minor variations between the current game position and the professional game record. To address this problem our implementation only uses these game records to assist in move generation for the first 20 moves.

Without using influence maps for the similarity comparison, only around a maximum of 8 moves could be placed before the game would no longer be expected to match any of the stored games. (If a non-standard opening has been used, it probably has never been played before at all. Assuming that about 40 of the approximately 350 legal moves in the opening are viable alternatives, there are on the order of 6 trillion possible opening sequences consisting of 8 moves.)

## 9.7 CBR-assisted opening play

Our approach of finding similar professional games to aid the program during the opening game made a significant improvement to the very first moves played during each game. The main advantage of our CBR-based approach is that it allows the program to quickly focus on the most strategically interesting areas of the board, even using relatively few simulations.

Figure 9.3 shows a screenshot of our implementation running on a 19 × 19 board, using our approach for reuse of previous games. By using the professional game records as guidance to bias the UCT exploration, our program is able to generate moves in reasonable areas of the board, even though only 100 simulations per move are performed in this example.

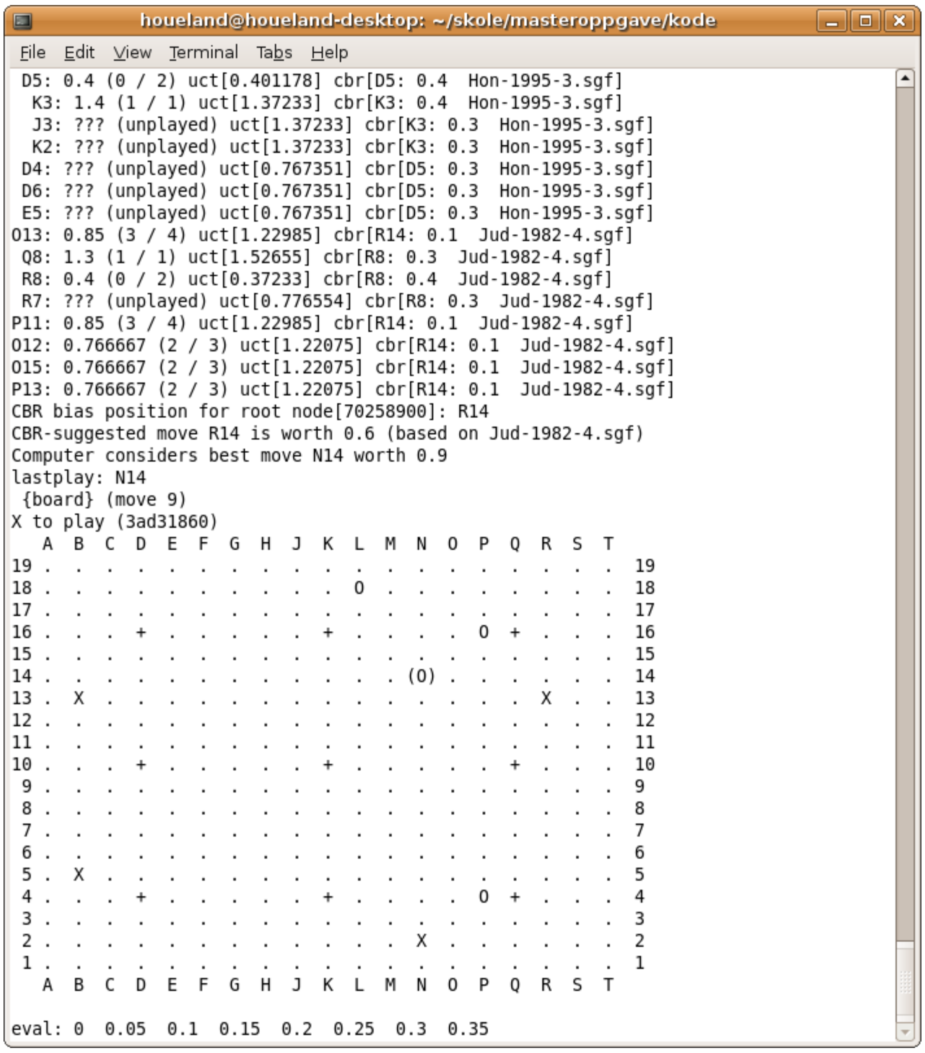
With our CBR-based approach enabled, each UCT node also has an additional section containing the bonus generated from the most similar professional game. An example of a line representing one node in this expanded UCT tree is shown below.

```
P11: 0.85 (3 / 4) uct[1.22985] cbr[R14: 0.1  Jud-1982-4.sgf]
```

This format used for nodes in the expanded UCT tree is almost the same as for our basic UCT implementation, but it contains an additional `cbr` element that is used to guide the implementation towards certain moves based on previous games.

In this example the previous game record that was most similar to the current board position was found in `Jud-1982-4.sgf`, which is a record of a game played between Cho Chikun and Otake Hideo in 1982 as part of the annual competition for the Judan title. The professional player's next move from the matched board position was R14, while the UCT node shown is for a move at P11.

The move at P11 is considered to be somewhat close to R14, and thus gets a 0.1 point bonus using our CBR approach. This means that the resulting node score is 0.85, after having won 75% of the simulations and adding the CBR-based bonus. (This is done using simple addition, where 0.75 + 0.1 gives the resulting score of 0.85. This bonus is also added to the UCT value, where in this case the original unbiased UCT value would have been 1.12985.)



```
 D5: 0.4 (0 / 2) uct[0.401178] cbr[D5: 0.4  Hon-1995-3.sgf]
  K3: 1.4 (1 / 1) uct[1.37233] cbr[K3: 0.4  Hon-1995-3.sgf]
  J3: ??? (unplayed) uct[1.37233] cbr[K3: 0.3  Hon-1995-3.sgf]
  K2: ??? (unplayed) uct[1.37233] cbr[K3: 0.3  Hon-1995-3.sgf]
 D4: ??? (unplayed) uct[0.767351] cbr[D5: 0.3  Hon-1995-3.sgf]
 D6: ??? (unplayed) uct[0.767351] cbr[D5: 0.3  Hon-1995-3.sgf]
 E5: ??? (unplayed) uct[0.767351] cbr[D5: 0.3  Hon-1995-3.sgf]
O13: 0.85 (3 / 4) uct[1.22985] cbr[R14: 0.1  Jud-1982-4.sgf]
 Q8: 1.3 (1 / 1) uct[1.52655] cbr[R8: 0.3  Jud-1982-4.sgf]
 R8: 0.4 (0 / 2) uct[0.37233] cbr[R8: 0.4  Jud-1982-4.sgf]
 R7: ??? (unplayed) uct[0.776554] cbr[R8: 0.3  Jud-1982-4.sgf]
P11: 0.85 (3 / 4) uct[1.22985] cbr[R14: 0.1  Jud-1982-4.sgf]
O12: 0.766667 (2 / 3) uct[1.22075] cbr[R14: 0.1  Jud-1982-4.sgf]
O15: 0.766667 (2 / 3) uct[1.22075] cbr[R14: 0.1  Jud-1982-4.sgf]
P13: 0.766667 (2 / 3) uct[1.22075] cbr[R14: 0.1  Jud-1982-4.sgf]
CBR bias position for root node[70258900]: R14
CBR-suggested move R14 is worth 0.6 (based on Jud-1982-4.sgf)
Computer considers best move N14 worth 0.9
lastplay: N14
 {board} (move 9)
X to play (3ad31860)
   A  B  C  D  E  F  G  H  J  K  L  M  N  O  P  Q  R  S  T
19 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   19
18 .  .  .  .  .  .  .  .  .  .  O  .  .  .  .  .  .  .  .   18
17 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   17
16 .  .  .  +  .  .  .  .  .  +  .  .  .  .  O  +  .  .  .   16
15 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   15
14 .  .  .  .  .  .  .  .  .  .  .  . (O) .  .  .  .  .  .   14
13 .  X  .  .  .  .  .  .  .  .  .  .  .  .  .  .  X  .  .   13
12 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   12
11 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   11
10 .  .  .  +  .  .  .  .  .  +  .  .  .  .  .  +  .  .  .   10
 9 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   9
 8 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   8
 7 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   7
 6 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   6
 5 .  X  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   5
 4 .  .  .  +  .  .  .  .  .  +  .  .  .  .  O  +  .  .  .   4
 3 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   3
 2 .  .  .  .  .  .  .  .  .  .  .  .  X  .  .  .  .  .  .   2
 1 .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   1
   A  B  C  D  E  F  G  H  J  K  L  M  N  O  P  Q  R  S  T

eval: 0  0.05  0.1  0.15  0.2  0.25  0.3  0.35
```

Figure 9.3: Screenshot showing the prototype implementation playing on a $19 \times 19$ board using our CBR-based approach. In this example only 100 simulations are performed for each move, but the implementation is still able to generate moves in strategically valuable areas of the board.

This extra bonus based on the most similar game ranges from up to 0.4 for an exact match, to no bonus added at all for moves that are not in the general vicinity of the professional player's move.

A comparison of the opening game played with and without the use of this CBR approach is shown in figure 9.4 and figure 9.5. In both cases the program is playing against itself with 100 simulations per move. The game played using CBR is a vast improvement over the one played using the unassisted approach, which is unable to generate any good moves at all with only 100 simulations per move.

The opening moves in the CBR-assisted game in figure 9.5 are generally placed in strategically important areas. With only 100 simulations the program is mostly unable to find good tactical positions for the moves within these areas, but overall the opening game looks significantly better than without using the CBR method.

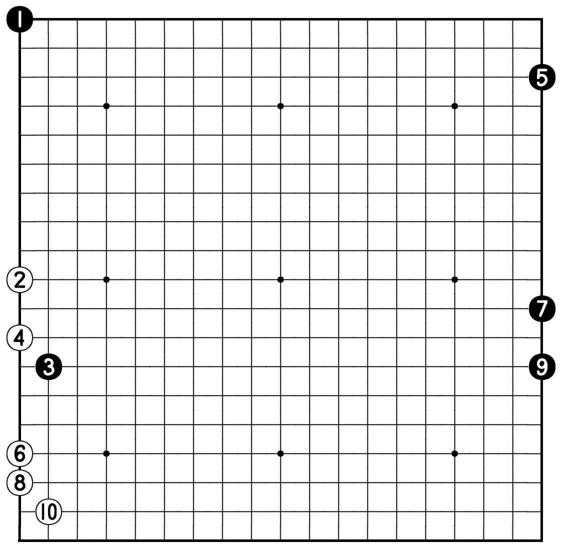


Figure 9.4: The first opening moves in a game played using 100 simulations per move, based only on the UCT algorithm and random play.

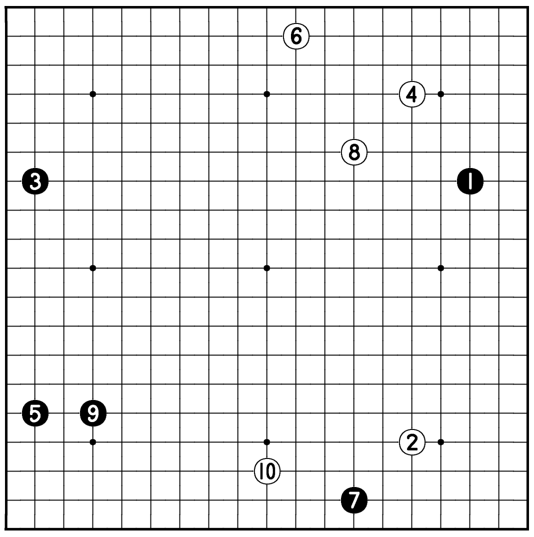


Figure 9.5: The first opening moves in a game played using 100 simulations for each move, biased by our CBR-based approach for reusing previous games.

Figure 9.6 shows an opening game using the unassisted UCT algorithm with 1,000 simulations per move instead, and figure 9.7 similarly shows an opening game using our CBR approach and 1,000 simulations per move.

In this case the moves generated by our CBR-based approach are also significantly better than the unaided approach, and in fact the moves generated by the CBR approach using 100 simulations is comparable to the unassisted UCT exploration with 1,000 simulations per move.

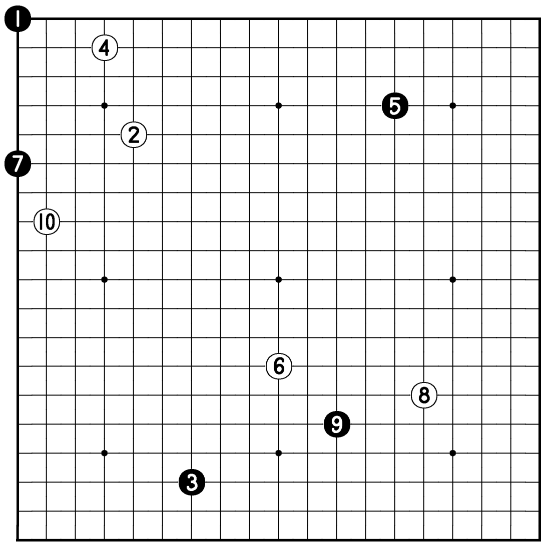


Figure 9.6: A game played using 1,000 simulations for each move, using the basic UCT algorithm.

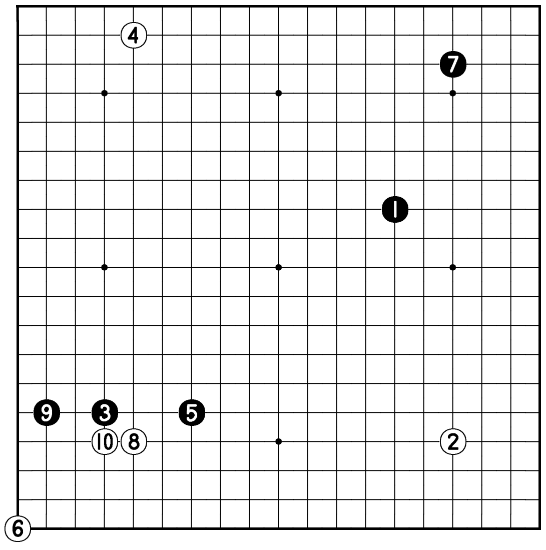


Figure 9.7: A game played using 1,000 simulations for each move, and our CBR-based approach. The 6th move at A1 is actually a very bad move, but by chance the computer happened to win all 13 times simulations that started with A1 at that point in the game.

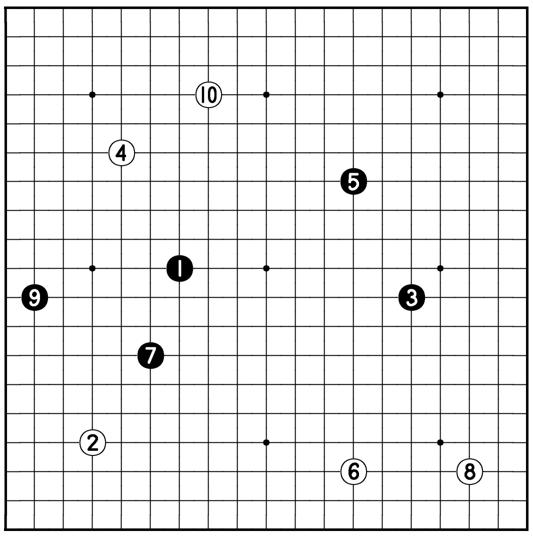


Figure 9.8: A game played between a computer player using the basic UCT algorithm with 10,000 simulations for each move, and our CBR-directed approach using 1,000 simulations per move. The CBR-assisted approach plays as white, and has placed greater emphasis on the corner areas of the board.

Figure 9.8 shows a game where the black player uses the unassisted UCT algorithm with 10,000 simulations per move while the white player uses the CBR-assisted approach and 1,000 simulations per move. Even though it is based on fewer simulations, the CBR approach is better able to focus on the corners of the game, which are normally considered to be more valuable and are also played first by human players.

### 9.7.1 Limitations of our method

One problem with our CBR approach is that it will only find strategically interesting areas of the board to play in. It is not able to find strong tactical moves within these areas by itself, and will have to rely on other methods for achieving this. Our basic prototype implementation of the Monte Carlo-based tactical random player is too weak to properly address this issue for the full $19 \times 19$ board. Similarly, our approach does not recognize situations where there are tactically important moves that need to be performed in the opening, which can happen if the opponent is playing overly aggressive moves.

An example of this is shown in figure 9.9, where our program plays black and the white player prematurely attacks the black corner. The program will not recognize this as an important tactical situation, but will instead continue to play other strategically interesting moves elsewhere on the board.

In itself this does not necessarily lead to a bad result for black, since the moves on the other parts of the board are worth many points. However, it would typically be better to respond to these white moves using local tactical moves, rather than simply sacrificing the black stone.

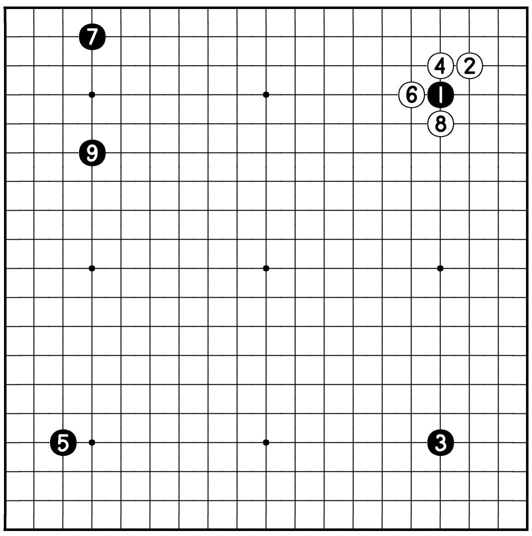


Figure 9.9: An example of an aggressive white player against our CBR approach playing black. In this case the generated black moves are not bad moves, but it would be preferable to instead respond tactically to the white moves.

Another weakness in our approach is that it will only find good moves in situations that are similar to the professional games in the case base. In the example game in figure 9.8, the black stone at ❺ places a very weak influence on the upper right corner. This means that the white player could easily invade this area, which would be worth a large number of points.

Because our CBR approach is only based on professional games it will not be able to find this possibility. No professional player would play at ❺ or anywhere similar, and because of this there are no games in our case base that suggest that a good white response to such moves would be to invade the corner.

Our approach to reusing previous games is useful for quickly determining which areas of the board are the most strategically interesting, but it will only be able to do so if the opponent also plays good, strategic opening moves.

# Chapter 10

# Discussion

This chapter contains an evaluation of our master's thesis and a discussion regarding the initial thesis objectives and to what extent they have been fulfilled. It also includes an overall evaluation of the results from our prototype implementation, and a discussion of the impact of reusing previous games to improve the playing ability of our program.

## 10.1 Go theory

Many interesting aspects of Go theory have been examined in some detail. This includes theory regarding how to play that is applicable for humans, and also theories at an abstract mathematical level where the game aspects can be formally analyzed and the game positions can be processed through automatic methods.

The practically oriented theory and techniques employed by very strong human amateur and professional players have not been explored as deeply, since such material is difficult to apply in games even for strong amateur players. This advanced theory is typically presented as guidelines and ideas, which does not have clear boundaries that indicate where they can be applied or exactly how they should influence moves.

This makes it very difficult to translate these strategies into computer code that can be used as part of move generation, so only amateur-level theory that has already been examined and structured in greater detail has been used during the development of this (and most other) Go playing programs.

## 10.2 Computer Go approaches

Several different approaches to creating computer Go players have been examined and the most popular recent approach, Monte Carlo-based UCT game tree exploration, has been reused for this project.

However, other approaches have also been shown to produce programs of nearly the same playing strength, and are likely to also contain useful methods and techniques for developing Go playing programs.

It is not clear how these different approaches can be combined in an efficient manner, so our project only attempts to add reuse of experience to the chosen

UCT approach. Combining our CBR-based move biasing with methods from other computer Go playing approaches could be an interesting direction for future work.

## 10.3 Experimental Go implementation

The prototype implementation of our Go playing computer system works and the implementation contains no known major bugs. It can be used to play games both with and without reusing previous game records to guide the move generation.

The playing strength of this prototype implementation is not nearly as strong as the best Monte-Carlo based Go programs. This is understandable since these other programs have been developed and improved by teams of developers over several years. However, this difference in playing strength means that the effects of reusing previous game records in our system does not necessarily translate to these stronger programs that already have other methods to handle the early opening game.

On a $9 \times 9$ game board, our system plays acceptably well, but still at a weak amateur level. For the full $19 \times 19$ board, our system can play reasonably well and often very good moves during the opening game by using our CBR approach. In the middle game it does not find good moves in a timely manner, and the program can only play the rest of the game at a novice level.

### 10.3.1 Estimated Go program playing strength

A common system for describing Go player skill is the use of kyu grades, where lower numbers are better and an absolute beginner start at around 25-30 kyu. Differences in kyu grades approximately correspond to fair handicaps, and a game between 13 kyu and 9 kyu players would be reasonably fair if the 13 kyu player starts the game with 4 handicap stones (and a komi of 0.5, since komi points are not used in handicap games).

For strong amateur players an additional set of dan grades exists, where 1 dan is stronger than a 1 kyu, and in this case a higher dan grade is better. The best amateur players are 6 to 7 dan, and at these level they are comparable to some professional players. (Professional players also use dan grades, but even a professional 1 dan is at around a 6 to 7 amateur dan level and thus much stronger than an amateur 1 dan.)

Our estimate of the prototype implementation's playing skill would be that it is around a 10 kyu player for $9 \times 9$ boards, where it uses only the UCT algorithm and our version of a random player with some additional tactical considerations as described earlier.

The first opening moves on a $19 \times 19$ board generated by our program are around the level of a 6 to 8 kyu player, but this is only for strategic moves and our implementation is not able to properly respond to early tactical fights against very aggressive opponents.

For the opening game our program reuses previous game records to guide the game tree exploration, and it also uses our faster third method for random play which can only produce reasonable results for the opening game. The reuse of previous game records is the most significant reason for this result, and using only this method combined with the tactical random player it may instead be considered to play at around the level of a 7 to 11 kyu player.

During the middle game and endgame on a $19 \times 19$ board, our implementation only uses the UCT algorithm, which causes it to play at around a 15 to 20 kyu level. Without our method for reusing previous games, this approach would also be used for the opening game which would result in a similar novice playing level (15 to 20 kyu) for the opening game.

GNU Go plays at around a 3 to 7 kyu level, while the best Monte Carlo-based programs are currently playing at about a 1 to 4 kyu level on $19 \times 19$ boards, and an impressive 5 dan amateur level on smaller $9 \times 9$ boards. Programs created using other approaches such as neural networks generally play at about a 5 to 10 kyu level for $19 \times 19$ boards.

(At the beginning of this thesis work, GNU Go and the Monte Carlo approaches were more similar in playing strength. However, during the work on this thesis, GNU Go has not changed significantly and has not improved more than one kyu grade, while the Monte Carlo-based programs have gained approximately 2 to 4 kyu grades, and if this continues they might reach a clear amateur dan level even for large boards in the next year.)

## 10.4 Use of experience to improve playing level

Even using the influence-based similarity measure which works rather well for finding similar games, our approach to reusing previous games is only useful for the beginning phases of the opening game. This is because later in the game there are too many other surrounding stones that influence the position, and our method does not properly adapt the moves to these nearby stones.

Another problem is that even though the opening moves the computer program plays using this approach are good, the computer program is only mimicking professional players and does not actually know how to use the moves later in the game.

Because of this our program normally does not take full advantage of the opening moves for the rest of the game, even if the opening moves are very good. Experiments using the entire previous game record to guide the rest of the game have not been successful, simply because there are too many ways for the opponent to respond and he is unlikely to pick exactly the same one as the original opponent.

The use of case-based methods for finding good tactical moves was not included in our implementation, because our program was able to find relatively simple tactical moves such as vital points in eyespaces by only using our implementation of Monte Carlo-based UCT game tree exploration.

It is possible that such tactical reuse could be applied for other situations where unaided UCT tends to perform poorly, but it is not clear that this local CBR-based reuse will be more effective than the purely pattern-based approaches successfully employed in other programs.

This is because the approaches used in e.g. GNU Go are already very good, and the strong Monte Carlo programs have shown that many tactical situations can also be solved without large amounts of expert knowledge. One important exception is long sequences of relatively simple moves, such as predicting how a long ladder formation moves across the board.

However, reusing tactical cases does not seem to be the most useful approach for these exceptional tactical situations. Reuse of previous game records appears to the most useful for high-level strategic considerations, which are most common in the opening game.

# Chapter 11

# Conclusion

Our prototype implementation of the Go playing program works, and in the opening game the approach based on reusing previous games usually results in better moves than the straight-forward application of the underlying UCT algorithm.

However, it is unlikely that this will result in any drastic improvements if the same approach is added directly to contemporary computer Go playing programs. One reason for this is that these programs are already rather strong, and often already contain some specialized approaches for the opening game.

Another important reason is that even when good moves are found using our approach, these moves usually depend on the computer program being able to follow them up correctly later, and current computer playing programs are not able to do this in a satisfactory manner. In particular, our implementation of a tactical player is too weak on a full $19 \times 19$ board, which means that our combined prototype Go playing program is unable to play at a competitive level.

Our approach is likely to be more useful for a computer program that contains a deeper knowledge of the purpose and intentions behind each move it performs, but today this is still closer to how a human plays than a computer. No automated computer system implementations with this level of understanding currently exist, nor are they likely to be feasible to develop in the near future.

Today, the most promising computer Go approaches seem to be a continued development of the statistical- and heavily computational-based Monte Carlo methods.

# Bibliography

[1] B. Bouzy and T. Cazenave, "Computer Go: An AI oriented survey," *Artificial Intelligence*, vol. 132, no. 1, pp. 39–103, 2001.

[2] "GNU Go." http://www.gnu.org/software/gnugo/. Retrieved on July 16th, 2008.

[3] S. Gelly, *A Contribution to Reinforcement Learning; Application to Computer-Go*. PhD thesis, Université Paris-Sud, 2007.

[4] C. Donninger and U. Lorenz, "The chess monster Hydra," in *FPL* (J. Becker, M. Platzner, and S. Vernalde, eds.), vol. 3203 of *Lecture Notes in Computer Science*, pp. 927–932, Springer, 2004.

[5] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen, "Checkers Is Solved," *Science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[6] M. Buro, "An evaluation function for Othello based on statistics," tech. rep., NEC Research Institute, 1997.

[7] O. Syed and A. Syed, "Arimaa - a new game designed to be difficult for computers," *International Computer Games Association Journal*, 2003.

[8] J. Hendler, "Computers Play Chess; Humans Play Go," *IEEE Intelligent Systems*, vol. 21, no. 4, pp. 2–3, 2006.

[9] A. Kishimoto and M. M. 0003, "Search versus knowledge for solving life and death problems in Go," in *AAAI* (M. M. Veloso and S. Kambhampati, eds.), pp. 1374–1379, AAAI Press / The MIT Press, 2005.

[10] P. Donnelly, P. Corr, and D. Crookes, "Evolving Go playing strategy in neural networks." AISB Workshop on Evolutionary Computing, 1994.

[11] M. Enzenberger, "The integration of a priori knowledge into a Go playing neural network." http://www.cgl.ucsf.edu/go/Programs/neurogo-html/NeuroGo.html, 1996. Retrieved on July 16th, 2008. This web site is a mirror, the author's original publication site is no longer available.

[12] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches," *AI Communications*, vol. 7, pp. 39–59, March 1994.

[13] D. B. Benson, “Life in the game of Go,” *Information Sciences*, vol. 10, no. 1, pp. 17–29, 1976.

[14] “Wikipedia article on Go opening theory.” http://en.wikipedia.org/wiki/Go_opening_theory. Retrieved on July 16th, 2008.

[15] D. W. Elwyn Berlekamp, *Mathematical Go: Chilling Gets the Last Point.* A K Peters Ltd, Natick, MA, 1994.

[16] L. V. Allis, *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, University of Limburg, 1994.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[18] “Wikipedia article on computational complexity theory.” http://en.wikipedia.org/wiki/Computational_complexity_theory. Retrieved on July 16th, 2008.

[19] S. Cook, “The P versus NP problem.” Official problem description as part of the Clay Mathematics Institute Millenium Prize Problems, 2000.

[20] M. Crasmaru and J. Tromp, “Ladders are PSPACE-complete,” in *Computers and Games*, pp. 241–249, 2000.

[21] D. Wolfe, “Go Endgames Are PSPACE-Hard,” in *More Games of No Chance*, MSRI Publications, pp. 125–136, Cambridge University Press, 2002.

[22] J. M. Robson, “The Complexity of Go,” in *IFIP Congress*, pp. 413–417, 1983.

[23] R. K. G. Elwyn R. Berlekamp, John Horton Conway, *Winning Ways for Your Mathematical Plays vol 1-2.* Academic Press, London, 1982.

[24] C. L. Bouton, “Nim, a game with a complete mathematical theory,” *The Annals of Mathematics*, vol. 3, pp. 35–39, 1901.

[25] “Wikipedia article on the Sprague-Grundy theorem.” http://en.wikipedia.org/wiki/Sprague-Grundy_theorem. Retrieved on July 16th, 2008.

[26] A. Lubberts and R. Miikkulainen, “Co-Evolving a Go-Playing Neural Network,” in *Coevolution: Turning Adaptive Algorithms upon Themselves* (R. K. Belew and H. Juillè, eds.), pp. 14–19, 7 2001.

[27] “Wikipedia article on Monte Carlo methods.” http://en.wikipedia.org/wiki/Monte_Carlo_method. Retrieved on July 16th, 2008.

[28] B. Brügmann, “Monte Carlo Go.” http://www.ideanest.com/vegos/MonteCarloGo.pdf, 1993. Retrieved on July 16th, 2008. This web site is a mirror, the author’s original publication site is no longer available.

[29] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *ECML* (J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, eds.), vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293, Springer, 2006.

[30] S. Gelly and Y. Wang, "Exploration exploitation in Go: UCT for Monte-Carlo Go," *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*, 2006.

[31] J. J. Bello-Tomás, P. A. González-Calero, and B. Díaz-Agudo, "JColibri: An Object-Oriented Framework for Building CBR Systems," in *ECCBR*, pp. 32–46, 2004.

[32] M. Molineaux and D. W. Aha, "TIELT: A Testbed for Gaming Environments," in *Proceedings of the Sixteenth National Conference on Artificial Intelligence* (M. M. Veloso and S. Kambhampati, eds.), pp. 1690–1691, AAAI Press, 2005.

[33] R. Sánchez-Pelegrín, M. A. Gómez-Martín, and B. Díaz-Agudo, "A CBR Module for a Strategy Videogame," in *ICCBR Workshops* (S. Brüninghaus, ed.), pp. 217–226, 2005.

[34] M. J. V. Ponsen, S. Lee-Urban, H. Munoz-Avila, D. W. Aha, and M. Molineaux, "Stratagus: An open-source game engine for research in real-time strategy games," in *Papers from the IJCAI Workshop on Reasoning Representation and Learning in Computer Games* (D. W. Aha, H. Munoz-Avila, and M. van Lent, eds.), 2005.

[35] B. Díaz-Agudo, P. Gervás, and F. Peinado, "A case based reasoning approach to story plot generation," in *ECCBR* (P. Funk and P. A. González-Calero, eds.), vol. 3155 of *Lecture Notes in Computer Science*, pp. 142–156, Springer, 2004.

[36] D. Stern, R. Herbrich, and T. Graepel, "Bayesian pattern ranking for move prediction in the game of Go," in *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pp. 873–880, ACM, 2006.

[37] F. A. de Groot, "Moyo Go Studio." http://www.moyogo.com/. Retrieved on July 16th, 2008.

[38] "SGF file format." http://www.red-bean.com/sgf/. Retrieved on July 16th, 2008.

[39] A. L. Zobrist, "A new hashing method with applications for game playing," Tech. Rep. 88, University of Wisconsin, Computer Science Department, 1970.

[40] "Go results from the 12th Computer Olympiad." http://www.grappa.univ-lille3.fr/icga/tournament.php?id=167, 2007. Retrieved on July 16th, 2008.