Enhancing a Tactical Framework for Go by Using Monte Carlo Statistics

Niclas Pålsson

January 15, 2008

Examensarbete för 20 p, Institutionen för Datavetenskap,
Lunds Tekniska Högskola

Master's thesis for a diploma in computer science, 20 credit points,
Department of Computer Science, Lund Institute of Technology

Abstract

Go is an ancient board game based on surrounding territory by placing stones on a grid. It has proven very difficult to create computer programs that play the game well.

Monte Carlo statistics has in recent years been a popular subject of research for creating computer programs playing Go. It relies very little on domain specific knowledge, and therefore presents an alternative to knowledge based Go, which is difficult to improve. Furthermore it doesn't suffer from the drawbacks associated with tree search in games with high branching factors such as Go. However, programs relying solely on Monte Carlo methods are still inferior to good knowledge based programs such as GNU Go. The object of study for this thesis is whether it's possible to create synergy by integrating a Monte Carlo module into the tactical framework of GNU Go, for Go played on a 9x9 board.

Four different strategies are tested for integrating the module, and for each one the degree of Monte Carlo influence is parameterised and varied to empirically find the optimal level. Two simple Monte Carlo models are tested: the Abramson's Expected Outcome heuristic, and a simplified version of Brügmann's model from Gobble, the first Go program to use Monte Carlo methods.

The study shows that it's possible to improve the playing strength of GNU Go 3.6 so that it wins a statistically significant greater proportion of games played against the unmodified program, and the best integration strategy is identified for doing so.

Sammanfattning

Go är ett väldigt gammalt brädspel baserat på att omringa territorie genom att placera stenar på ett rutnät. Det har visat sig vara mycket svårt att skapa datorprogram som spelar spelet bra.

Monte Carlo-statistiska metoder har under de senaste åren varit ett populärt ämne för att skapa Go-spelande datorprogram. De baseras väldigt lite på domänspecifik kunskap, och utgör därför ett alternativ till kunskapsbaserat Go, vilket är svårt att förbättra. Vidare dras de inte med samma svårigheter som vanligtvis är anknutna med trädsökning i spel med hög greningsfaktor såsom Go. Program som förlitar sig helt på Monte Carlo-metoder är dock fortfarande underlägsna bra kunskapsbaserade program såsom GNU Go. Ämnet som studeras i detta examensarbete är huruvida det är möjligt att förbättra GNU Go genom att integrera en Monte Carlo-modul i dess taktiska ramverk, för Go spelat på 9x9-bräde.

Fyra olika strategier för att integrera modulen testas, och för var och en utgörs Monte Carlo-inflytandet av en parameter som varieras för att empiriskt hitta optimal nivå. Två enkla Monte Carlo-modeller testas: Abramson's Expected Outcome-heuristiken, och en förenklad variant av Brügmanns modell från Gobble, det första Go-programmet att använda Monte Carlo-metoder.

Studien visar att det är möjligt att förbättra spelstyrkan för GNU Go 3.6 så att den vinner en statistiskt säkerställd större delmängd av matcher spelade mot det omodifierade programmet, och den bästa integrationsstrategin för ändamålet identifieras.

Contents

Acknowledgements

I would like to thank Eric Astor for supervising and giving me suggestions on this project, and Gunnar Farnebäck for helping me find a project related to GNU Go that was suitable for a master's thesis. Also I want to thank Mikael Pålsson and Joachim Ståhl for their feedback and suggestions.

1 Background

The game of Go has been described as *"The holy grail of computer programming"*, and *"The last refuge of human intelligence"*. It's an ancient game of strategy which originated in China some 4000 years ago, for which all attempts to create a strong computer program have been unsuccessful.

There is a component to Go that has proven incredibly difficult to solve using a computer based approach, which is probably best described as a combination of intuition and pattern recognition. It's often possible to "just know" if a move is good. Rather than relying too much on reading ahead, humans are using a shortcut, an immediate path to direct knowledge giving a quick, clear and full apprehension of a complex group of data.

The Go domain requires a different approach than other games, and much research remains to be done. This makes it an exciting field for testing new ideas, such as Monte Carlo statistics which is the subject of this thesis.

2 Introduction

It's very difficult to develop a Go program that plays well. Today's best programs are ranked around the 10 kyu level, which is the equivalent of a weak amateur player. It's possible for a complete beginner to reach this level already after studying and playing the game for a few months. This is a stark contrast to many other board games, such as chess, where the best computer programs play at an exceptional level, stronger than the human brain.

Go is typically played on a 19x19 board. This has the effect that there are on average around 250 valid plays to consider for each move, depending on how far the game has progressed. This can be compared with chess, which has an average of 35. Since the time required for searching a game tree to a certain depth grows exponentially with this factor, it has the effect that brute force searching for a good move is of limited use in Go.

Today's Go software relies heavily on pattern matching. The programs use large databases of patterns, which are manually tweaked by strong Go players. This practise is difficult, as is improving knowledge-based Go in general. See [BC01].

An interesting new development in the field is to instead play Go using Monte Carlo methods. Being a statistical approach, it has the advantage that no game specific knowledge is required, and also it doesn't involve brute force searching of the game tree. The approach has had successes, but its main weakness is in tactics, as reported in [BH03]. This suggests a Monte Carlo Go program will benefit from working in tandem with a module producing domain specific tactical information.

This thesis will research whether it's possible to improve an existing leading Go program by adding Monte Carlo methods to it. For this purpose GNU Go 3.6 will be used, because it's open source, and also one of the world's leading Go programs. It recently placed 1st in the 2006 Computer Olympiad.

3 Description of the Problem

A Monte Carlo module will be designed, implemented and integrated into the GNU Go framework. After having done this, the objective is to answer the following questions:

- Is it possible to improve GNU Go by making use of Monte Carlo statistics?
- If it's possible, how should it be integrated in order to maximise the increase in playing strength?

To clarify the latter question, the idea is to investigate how to best make the Monte Carlo module work together with GNU Go's existing move generator. Four different approaches will be tested:

1. Let MC take care of the n first moves
2. Let MC choose between the n highest valued moves
3. Let MC add a bonus to the moves
4. Let MC decide if no urgent move can be found

To determine if a particular approach is successful, automated games will be played against the unmodified GNU Go. A sufficient number of games will be played to determine the result using a 95% confidence interval for the binomial distribution. There will be a practical limit on how many automated games can be played, so if a statistically significant result cannot be obtained, it will mean that the difference in strength is too small to notice. However, it should be possible to adjust GNU Go so that it doesn't search to the very deepest level. This should significantly reduce testing time whilst keeping the effect on the results to a minimum.

Focus will not be put on designing the Monte Carlo model to use and whether changing it has an effect on playing strength. It will certainly have, but then it'll be difficult to determine whether any differences in playing strength are resulting from that, or the way the integration is made.

Two different Monte Carlo models will be used: Abramson's Expected Outcome (1990) and Brügmann (1993). The latter is the model that was used for the first ever Go program to use Monte Carlo statistics, Gobble. The reason for using two different models is simply to give a greater confidence to the results.

3.1 Expectations

I would expect Monte Carlo to be able to improve the playing strength of GNU Go. I would also expect it to be most successful in case 3, since that integration is closer linked with GNU Go. In integrations 1,2 and 4, if Monte Carlo gets to decide the move, it decides on its own which I think is a weakness. Particularly

in case 1, I think it's more efficient to use GNU Go's joseki[1] library, since the same patterns tend to recur in the opening.

In cases where GNU Go is able to accurately evaluate a local situation on the board, such as finding a move that kills[2] a group of stones, that information should take precedence over Monte Carlo. Sometimes adding a bonus to all moves, as in case 3, may obscure this information.

[1] Japanese for "Set stones". The term refers to an established sequence of moves which results in a fair outcome for both Black and White.

[2] A dead group is a cluster of stones that have no way of escaping their eventual capture.

4 The Game of Go

Go is a board game that originated in China more than 4,000 years ago. In ancient times Go was considered a martial art and was part of the training of warriors in Japan, China, and Korea. Along with calligraphy, music, and painting, Go was also one of the components of classical education for both men and women. Today, it's played by millions of people, mainly in the Far East. In Japan, Korea, China, and Taiwan, it's far more popular than chess is in the West, and players compete for large cash prizes in professional leauges.

The Go board is initially empty, and players take turns placing black and white stones on it, outlining their territory. While it's possible to surround and capture the opponent's stones, the primary objective of the game is to surround as much territory as possible. Go is typically played on a grid of 19 by 19 lines, although other sizes are possible. Professional games are always played on 19x19, and also most other games between humans. Computer Go is often played one 9x9 however, because it simplifies the game.

There are many online resources for learning more about Go, such as [WWW3].

4.1 Rules

Ignoring rules dealing with a small number of rare special cases, the game has only a handful of basic rules:

1. The game is played by two players, Black and White.
2. The board is a grid of horisontal and vertical lines.
3. The lines of the board have intersections wherever they cross or touch each other. Each intersection is called a point. That includes the four corners, and the edges of the board.
4. Black uses black stones. White uses white stones.
5. Players take alternate turns.
6. The game starts with all board points empty.
7. *Rule of Capture:* If a play surrounds the opponent's stone or stones completely, the player captures them and removes them from the board. (Figure 4.1)
8. *Rule of Suicide:* A player is not allowed to make a play that removes the last liberty of any of his own stones without doing the same to opposing stones. (Figure 4.2)
9. *Rule of No Repetition:* One may not play a move which repeats a previous board position. (Figure 4.3)
10. Pass is a valid move.

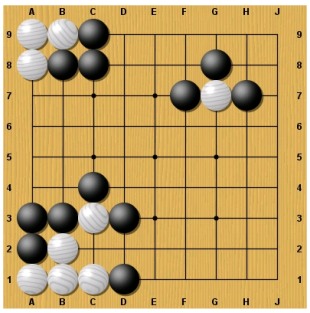


Figure 4.1: Black can capture by playing A7, C2 and G6.

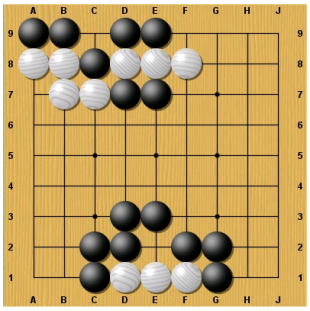


Figure 4.2: White E2 is suicide but White C9 is not.

11. *Rule Determining the Winner:* The purpose of the game is to occupy or surround more points than the opponent.

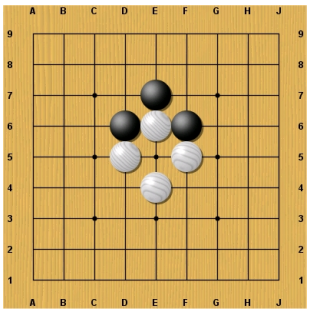


Figure 4.3: If Black plays E5 to capture E6, White is not immediately allowed to play E6 and capture back, since this would allow the game to move between the two positions forever.

4.2 Computer Go Difficulties

As touched upon in section 2, Computer Go is difficult. See [BC01] where the authors conclude that "the problems related to Computer Go require new AI problem solving methods". It's hard to improve today's programs by adding more domain specific information such as patterns and opening databases since this has already been explored. It's also ineffective to use traditional global game tree search to determine the value of a move. One of the reasons behind this is that the game has a large branching factor, but it's is also very hard to perform static evaluation of a position. To illustrate this, consider the following comparison with chess. In chess, static evaluation would comprise elements such as:

- Material (pieces on the board)
- How much of the board the pieces can threaten
- Whether the pieces threaten the center four squares

The items above are all easily calculated by performing static evaluation. The only problem lies in weighting of the different factors. Let's instead look at two concepts in Go that are very central to the game, but abstract and difficult for a program to evaluate:

- Territory
  This is the ultimate goal in Go. The problem is though that it doesn't become clear until the endgame who has more territory. During the mid and early game often only prospective territory exist. (Figure 4.4).

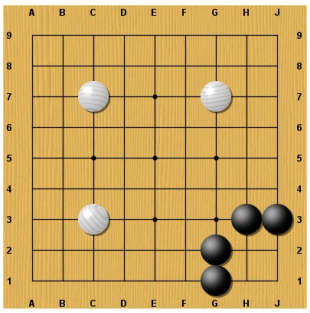


Figure 4.4: White has outlined prospective territory at the top and left, and is leading. Black plays very inefficient.

- Thickness
  The term thickness refers to a strong formation of stones, typically exerting outward influence. This concept is as important as territory. A player with thickness can often use it to an advantage and later end up with territory. (Figure 4.5)

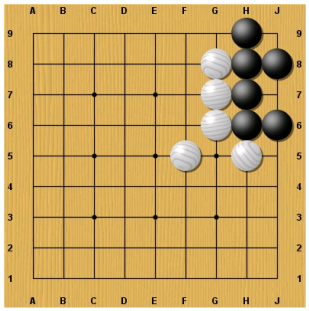


Figure 4.5: Black has 2 points of solid territory, but White's thickness is influencing the whole board. White is in the lead.

The game contains more concepts like the ones just mentioned. In both figures 4.4 and 4.5 it's trivial for a human to see which player has the advantage, but it's difficult to analyse using an algorithm. And even if an algorithm could handle these clear cut cases, it would struggle with a real game situation.

5 Monte Carlo Statistics

Section 4.2 attemps an explanation on why Go is difficult to solve using traditional methods. Monte Carlo statistics presents an interesting alternative that is neither dependent on domain specific information, nor global game tree search.

In the general case, Monte Carlo simulation is a method of analysis based on artificially recreating a chance process, running it many times, and directly observing the results. In Go, this means playing a large number of random games starting from a given position. It's then possible to draw conclusions based on observations of the games played. Two different models for doing this are presented in sections 5.2 and 5.3. Each model returns a heuristic value E for how good the position is.

With N = number of simulations, B = branch factor and L = moves per game, the time complexity for returning a move using the model described in 5.2 is

$$O(NBL)$$

This can be compared to the traditional exponential tree search which is of complexity

$$O(B^L)$$

Thus, for games with large B, such as Go[3], Monte Carlo evaluation is of much lower complexity than the tree search approach.

N however, must be sufficiently large to produce a reasonably accurate approximation of E. How large depends on the standard deviation σ of the outcome of each random game [BC06]. For Go on a 9x9 board, σ is about 35 points, which requires 4000 games to return E with an error < 1 point[4] at 95% confidence. For larger boards, the standard deviation increases, and therefore also the number of random games required.

It's worth noting that N can be increased infinitely to improve accuracy, but with diminishing returns for the increasing computation time cost. E is only a heuristic, not an absolute truth, so there is little use estimating it too accurately. [BC06] suggests 10000 to be a good value for N using a 9x9 board, producing statistical confidence and a method that works in reasonable time.

The Monte Carlo approach is often used in games with incomplete information, such as Poker [BDSS02]. It's natural in that type of game to replace hidden information by simulated randomised information, but perhaps less so in a complete information game like Go. In the case of Go however, although being a complete information game, due to its complexity much of the information can not easily be extracted. In those cases, when high quality game information cannot be obtained accurately, it appears plausible that it is instead good to use randomised information from a Monte Carlo module.

[3] Average branch factor is approximately 250.

[4] 1 point is the smallest possible quantity in Go.

5.1 Research to Date

Monte Carlo Go has been one of the main focus areas of the computer Go research community during the last few years. The very first Monte Carlo Go program, Gobble, was created in 1993 [B93]. It played on a 9x9 board, and used a simplified heuristic for returning a move. Since then, computer power has increased, and today programs can play comfortably on 9x9. In the 2006 Computer Olympiad, the 19x19 Go tournament was won by GNU Go, which is based on traditional methods. The 9x9 tournament was won by CrazyStone [C06], which is based on Monte Carlo methods. However, later tests with a larger number of games showed that GNU Go is actually stronger. [C06] suggests that CrazyStone shows a better understanding of the global board position than GNU Go, but loses on tactics such as reading long sequences ahead, and special tactical moves.

Increasing computer power may enable Monte Carlo Go to be more effective on 19x19 in the future. There have also been developments showing promising results when combining Monte Carlo evaluation with tactical search [CH05].

5.2 Abramson's Expected Outcome

Abramson (1990) [A90] proposed the expected-outcome model. It suggests that the proper evaluation of a game-tree node is the expected value of the game's outcome given random play from there on. By demonstrating that the model outperformed a standard Othello evaluator on a 6x6 board, the author showed that the expected outcome is a powerful heuristic. He concluded that the expected-outcome model of two-player games is "precise, accurate, easily estimable, efficiently calculable, and domain-independent".

5.2.1 Modifications for Go

In Go, it's possible to quantify victory, e.g. Black wins with 12.5 points. It's therefore possible to extend the model not to handle only win/loss, but to look at the winning margin in each game. This means instead of knowing Black wins $X\%$ of the games, we can say that Black wins or loses on average with n points.

This enhancement was used by Bouzy and Helmstetter [BH03] and will also be used in this experiment.

5.3 Brügmann's Model

Brügmann's model [B93], which was used in the first ever Monte Carlo Go program, Gobble, uses the all-moves-as-first heuristic. After a random game has finished with a score, instead of updating the mean value of the first move of the random game (as in Abramson's model), it updates the means of all those moves in the game that has the same colour as the first move and haven't previously been played on.

Where Abramson's model performs a Monte Carlo simulation for each possible move at ply 1 and then makes a greedy choice, Brügmann's model performs

one simulation from the current position only. This theoretically eliminates B from the time complexity, making it

$$O(NL)$$

The compromise in this model is that the heuristic is not exactly true for Go. It's only correct if a move has the same value regardless of at what stage of the game it's played, which is not the case as illustrated in figure 5.1. For the benefit of being able to increase N, this method is worth considering though.

Gobble uses additional logic such as simulated annealing, and also the concept of temperature, which controls the level of randomisation when playing a game. It was shown by Bouzy & Helmstetter [BH03] that simulated annealing wasn't effective, and that it was perfectly possible to play games with complete randomisation. Therefore these ideas will be left out and only the all-moves-as-first heuristic implemented.

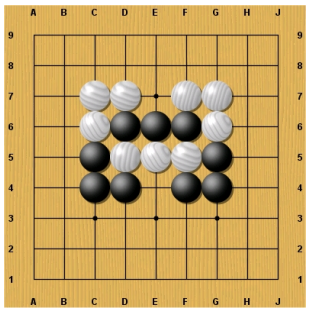


Figure 5.1: White E7 no longer has the same meaning if Black gets to capture at E4 first.

6 GNU Go

For games such as Chess and Checkers, with smaller branch factors than Go, the usual approach is to generate all possible moves, and then use a global N-ply search to calculate the best move. For Go, this is not feasible. GNU Go doesn't actually have a global search at all, but instead relies on different move generators each dedicated to find moves that are good for a particular reason. The move generators may use local search methods, for example to find the solution to a life & death (Figure 6.1) or a connection problem (Figure 6.2).

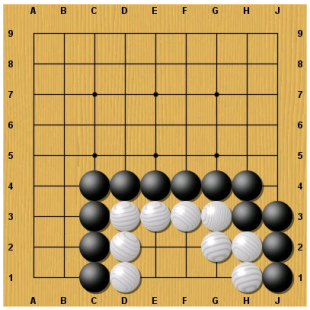


Figure 6.1: Black kills at F1. The solution is 21 ply down.

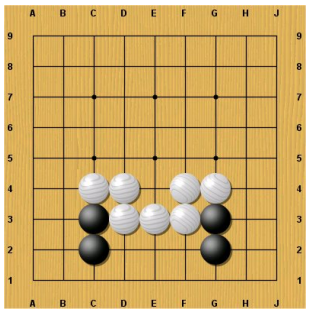


Figure 6.2: Black connects the two groups at D1 or F1, verified at 15 ply.

These situations illustrate the issue with using the traditional N-ply search. The solutions to these fairly trivial problems, at 21 and 15 ply respectively, would never be found. A Go program may solve them either by pattern recognition, or by using a goal oriented search such as λ-search [T00].

6.1 Engine Overview

There are three main steps involved in the process of finding the best move to play. GNU Go starts by trying to understand the current board position as good

as possible. Using the information found in this first phase, and using additional move generators, a list of candidate moves is generated. Finally, each of the candidate moves is valued according to its territorial value (including captures or life & effects), and possible strategical effects (such as strengthening a weak group).

6.1.1 Gathering Information

The first step is to gather as much information about the game as possible. The board is scanned for groups of directly connected stones, counting their size and number of liberties[5]. Also, which groups are connected to each other and forming larger clusters is analysed. Then it's identified which clusters are weak, and on those a more detailed life & death analysis is performed. Lastly, the balance of white/black territory is estimated.

6.1.2 Move Generators

This step involves calling different move generators, each proposing moves that may be good because of a particular reason. For instance, a move may be interesting if it captures or threatens to capture a group of enemy stones, follows a certain set pattern, or breaks into enemy territory.

Note that scoring doesn't take place here. Only the move reasons are produced. Keeping scoring in a separate module allows for more accurate results, since it's not always true that scores from separate move generators for the same move can be added together. For example, if a move is found that leads to the capture of a group of stones, it's not relevant if the move also is a threat to capture the same group of stones.

6.1.3 Move Valuation

Here, moves that can be expressed directly in terms of territory are assigned their values. Such a move may be the capture of a group, for which the value is typically two times the number of stones in the group[6]. For moves where the value is not obvious, such as combination attacks and strategical effects, a set of heuristics is used.

6.2 Architecture

GNU Go consists of two parts: the GNU Go engine and a program (user interface) which uses this engine. These are linked together into one binary. Within the engine, there are four main areas:

[5] Number of empty adjacent intersections. A group with a low number typically risks capture.

[6] Each captured stone is worth 1 point, and also the empty territory left by a captured stone is worth 1 point.

- Move Generation
  Most important is the move generation library which given a position generates a move. This part of the engine can also be used to manipulate a Go position, add or remove stones, do tactical and strategic reading and to query the engine for legal moves.

- Board Handling
  This provides efficient handling of a Go board with rule checks for moves, with incremental handling of connected strings of stones and with methods to efficiently hash go positions.

- Game Handling
  The game handling code helps the application programmer keep track of the moves in a game. Games can be saved to SGF files and then later be read back again.

- SGF Handling
  This adds support for saving or loading a game tree from a Smart Game Format (SGF) file.

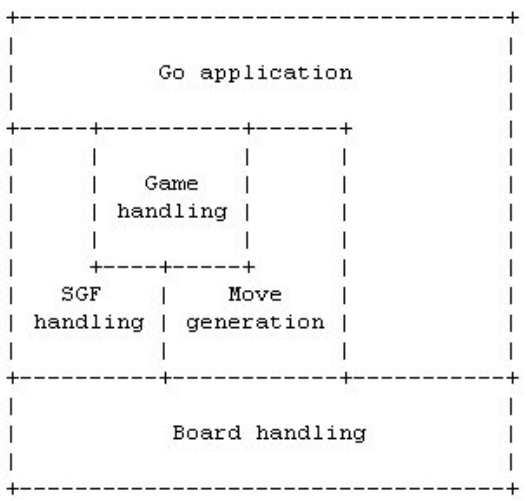


Figure 6.3: The structure of a program using the GNU Go engine

6.3 Board Code

Speed is crucial for Monte Carlo Go, to be able to simulate enough games per move to return statistically safe heuristics. GNU Go uses a board module highly optimised for traditional Go programming, but not for Monte Carlo calculations. Typically, for this purpose it's enough to use a more lightweight version since

all that is needed is to play random games to the end. There is no need to retain data structures allowing for more complex queries, or to perform undo operations. Also, since random games can be played independent of each other, Monte Carlo is very suitable for multi processor/core systems. There is however no support for that in GNU Go.

The result is that the implementation is not as fast as the Monte Carlo Go programs that are currently available[7]. The original aim of this investigation however, was not to produce a fast program, so compensation can simply be made by allowing more time per move when testing. Currently it simulates 5000 random games/s, using a 9x9 board[8].

An attempt was made to replace the board code with a simpler version, but achieved no improvement in speed, so it was dropped. The existing code should be fast enough though, and also it has the benefit of having been well tested.

[7] Crazy Stone simulates 17000 9x9 games/s on an AMD Athlon 3400+.
[8] on an Intel Core 2 Duo 2.67 GHz

7 Integrating Monte Carlo Statistics

A Monte Carlo module was designed and implemented in C, and then added to and compiled with the GNU Go engine library. It was possible to do this relatively independent of the rest of the framework. The module uses only three other functions from the engine:

`trymove()` Perform a move.

`popgo()` Retract a move.

`is_legal()` Check if a move is legal according to the Go ruleset.

The following functions outline the design of the Monte Carlo module:

`mc_getm_abram()` Retrieve Monte Carlo values for a set of moves, calculated using the Abramson model. This is public and called by the engine.

`mc_getm_brueg()` Retrieve Monte Carlo values for a set of moves, calculated using the Brügmann model. This is public and called by the engine.

`play_random_game()` Play a random game from the current position to the end, and calculate its score.

`play_random_games()` Play n random games from the current position to the end, and calculate their average score.

`simple_score()` Calculate the score for a game that has finished. This assumes that all territory consists of single intersections surrounded by either stones or the border. This is always the case after a random game has been played, since if a multi-intersection territory exists, it must contain playable moves and thus the end of the game is not yet reached.

`is_eye()` Check if an intersection is an eye. Eye handling is discussed in section 7.2.

`is_playable()` Check if an intersection is playable. This returns true if the intersection is a legal move according to the Go ruleset, and also not an eye.

`get_random_move()` Return a move selected at random among all playable moves on the board.

7.1 Monte Carlo Move Valuation

For a general description of how the different Monte Carlo models work, see sections 5.2 & 5.3. This section aims to provide formalised information about the algorithms used and how they are implemented.

The same game scoring function, `simple_score()`, is used for both the Abramson and Brügmann models. Let d be the winning margin of a game, and s the score returned by `simple_score()`. Then

$$s = \begin{cases} -d, & \textit{black victory}; \\ d, & \textit{white victory}; \end{cases}$$

Let $v_a(m)'$ be the not normalised Abramson valuation for a move m and n the number of random games played. Furthermore let s_i be the score of a game with random play commencing *after* m. Then

$$v_a(m)' = |\frac{\sum_{i=1}^{n} s_i}{n}|.$$

Finally, let $v_a(m)$ be the normalised version of $v_a(m)'$. Then

$$v_a(m) = \begin{cases} v_a(m)' - Min(v_a'), & if\ v_a(m)'\ is\ calculated\ for\ all\ m; \\ v_a(m)' - v_a(PASS)', & otherwise; \end{cases}.$$

The Brügmann model is a bit different. Let m be the move for which a value is sought, and p the number of that move.

First, let a game G be a set of moves. Then let the function $g(x)$ return the x:th move and let $c(m)$ return the colour of a move. Let R be the set of random games played. Let S be the subset of R that fulfils the criteria that for all G in S, there exists an x so that $x \geq p$ and $g(x) = m$, and for the smallest such x, $c(g(x)) = c(g(p))$. In other words, S contains all games in which the move we're seeking to value was played by the player to move at some point of the randomisation, and not previously played on[9] by an opponent.

Let $v_b(m)'$ be the not normalised Brügmann valuation for m. Also let s_i be the score of a game in S and n the number of games in S. Then

$$v_b(m)' = |\frac{\sum_{i=1}^{n} s_i}{n}|.$$

Finally, let $v_b(m)$ be the normalised version of $v_b(m)'$. Then[10]

$$v_b(m) = v_b(m)' - Min(v_b').$$

The normalised Monte Carlo scores are the ones used in the integration with the GNU Go valuation module. Normalisation is needed because Monte Carlo values are representing the average outcome of the game as a whole, so if one side has a lead, all moves will receive higher values.

[9] A stone can be played in the same place as a previous stone, if the previous stone gets captured and thereby leaves an open space.

[10] Unlike Abramson, Brügmann values all moves on the board in one go. Therefore we always have access to $Min(v_b')$.

7.2 Eye Handling

The term "eye" refers to an empty intersection on the board surrounded by stones of one colour. The concept is important because it's integral in determining whether a group of stones can be captured or not. Consider the example in figure 7.1. The white group to the left has only one eye at C6, but the group to the right has two eyes at G4 and G6. The difference is fundamental. A black play at C6 captures the first group, but the group with two eyes cannot be touched, since a black play at G4 or G6 is suicide and therefore illegal, and Black cannot play in both places at once. An attempt at a definition of eye is as follows:

Definition 1 *An* eye *is an empty intersection that fulfils one of the following criteria (counting diagonals):*

1. *At least 7 surrounding intersections are occupied by stones of the same colour or are empty but unplayable by the opponent.*
2. *Located on the board edge, and 5 surrounding intersections are occupied by stones of the same colour or are empty but unplayable by the opponent.*
3. *Located in a corner, and 3 surrounding intersections are occupied by stones of the same colour or are empty but unplayable by the opponent.*

Definition 1 is not an exhaustive list of what might be considered an eye. There are special situations that may arise, but those fall outside the scope of this thesis.

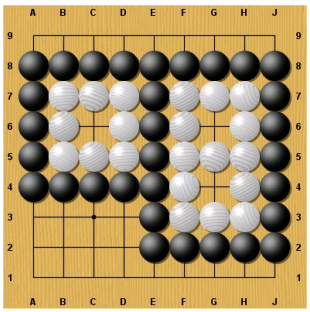


Figure 7.1: One eyed groups can be captured, but not those with two eyes.

Eyes however, pose a problem for Monte Carlo Go. The problem lies in playing a random game to the end. G4 and G6 are actually legal moves for White, but leads to the group being reduced to one eye which makes it possible for Black to capture it. Although legal, these moves are effectively suicidal, and therefore unreasonable.

If this type of move is allowed, it leads to random games not terminating. Each side will fill in their own eyes, until the opposite side can capture a group,

which will leave a new empty space to play on, and so on. It's therefore necessary to introduce further restrictions on random move generation, beyond just choosing a move that is legal according to Go rules. A simple solution is to disallow moves filling in own eyes, which is what Brügmann [B93] did, and this is also what was chosen for this research.

This additional restriction does introduce domain dependent knowledge in the Monte Carlo model, which goes somewhat against the argument that domain independence is an advantage of the Monte Carlo approach. It's necessary here though, and kept to a minumum. From a Go perspective, it's safe to say that an eye filling move would always be an unreasonable play anyway, so it may as well have been in the ruleset. This leads to the following definition:

Definition 2 *A move is referred to as* playable *if it's legal according to Go rules, and is not on an intersection which is an eye.*

The actual eye definition used by the Monte Carlo module is Definition 3, which is a simplification of Definition 1. This is because the speed of the `is_eye()` function is critical to the overall speed of the module, and the second definition leads to a faster implementation. In theory, it may incorrectly classify some intersections as eyes while they are not, but this doesn't have much impact on the statistical results since both Black and White will make the same mistakes in equal proportion. The only purpose the eye definition needs to serve is allowing games to terminate.

Definition 3 *An eye, as recognised by the Monte Carlo module, is defined by an intersection that fulfils the following criteria:*

1. *No diagonal intersections are occupied by enemy stones.*
2. *North, south, west, and east neighbours are of the same colour, or do not exist (border).*

It's easy to imagine why Definition 3 leads to faster code. All that is required is one or sometimes two comparisons per neighbour, and the intersection can be dismissed if a comparision fails, without examining the remaining neighbours.

7.3 Performance Before Integration

The examples in this section illustrate how GNU Go and Monte Carlo perform individually in different game situations. Monte Carlo on its own is not a very good player at all. It tends to be remarkably good at finding moves close to the best move however, considering that it doesn't actually have any[11] domain specific knowledge.

Figure 7.2 represents a game that is quite close to being finished. There are two important moves. White can kill Black at C8 and A2, which can be seen by reading forward a few moves. GNU Go is generally very good at situations

[11] This is not strictly true. See section 7.2.

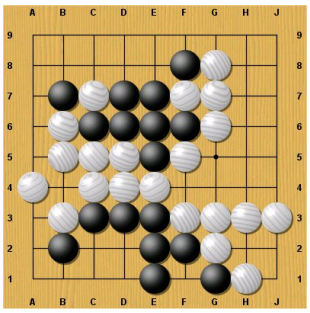


Figure 7.2: White to play in this life & death problem. White can kill either the upper group by playing C8, or the lower with A2 or C1.

where it's possible to search for a solution, but Monte Carlo struggles a bit. GNU Go ranks the moves for this example in the order C8, A2, E8, F9, which has to be considered a very accurate result. Monte Carlo finds the moves E8, C8, D8, C2, B8, which is not quite so good, but in the right direction. It prioritises the moves close to C8, the correct move, but can't quite make out which is indeed the best one. It then finds C2, which should be A2, but again the intention is right and C2 is not all that bad (it leads to ko[12]), just not as good as A2.

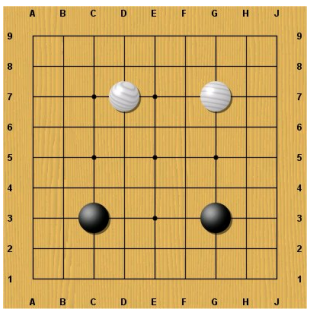


Figure 7.3: Black is next in this opening problem. There are several good moves, but C5 looks most important.

Figure 7.3 represents a game in the opening stage. Unlike the previous example given, it's not a case of searching for a particular move that kills a big cluster of stones, but instead of outlining potential territory. It's not as obvious anymore what the best move is, but in the author's opinion C5 is best. GNU Go ranks the moves for this example in the order E3, D5, E2, C5, E4. While

[12]Ko is a conditional form of life. It's better to live (or kill) unconditionally. Please refer to Go litterature for more information.

not bad, it's not that good. Especially E2 is too passive. Monte Carlo returns the moves D6, E7, E5, F7, E6. Again, these are not right, but it's worth noting that they are more aggressive. Especially the contact plays at D6, E7 and F7 could generate problems for a weaker opponent.

7.4 Integrated Move Valuation

Logic was added to the GNU Go engine to retrieve values from the Monte Carlo module and use them in the move valuation when certain conditions are met. This was integrated by adding a new separate step to the move valuation process, which after modification reads as follows:

1. Search the board for move candidates and return them together with the reasons why they were chosen.
2. Assign values to the moves according to the move reasons.
3. Retrieve Monte Carlo values for the moves, and update the existing move values according to these.
4. Check if any of the moves can be replaced with another, better move.
5. Perform a final check to see if the best move is a blunder.

Integrating the Monte Carlo valuation with the existing valuation in this way means that it's not possible to obtain a decision purely made by Monte Carlo, since the move may still be replaced or revaluated due to a blunder being detected. Thus, to let Monte Carlo choose between the n best moves would instead mean to let Monte Carlo choose between the n higest valued moves as of the first valuation, but still apply move replacements and blunder checking afterwards.

The alternative would have been to place Monte Carlo as the last step in the valuation, but this feels unnatural. If we assume that move replacements and blunder checking improve the move valuation they should be done last, otherwise their contributions could be overridden. If Monte Carlo would pick a blunder move, it would go undetected.

Four different integrations were implemented, corresponding to those outlined in section 3. A parameter exists for each scenario, representing how strong the Monte Carlo influence is.

- Let MC take care of the n first moves
  Monte Carlo valuation is used if the move number is less than or equal to n, otherwise the GNU Go valuation is used. ($n > 0$)
- Let MC choose between the n highest valued moves
  GNU Go valuation is used to select the n best moves, and thereafter Monte Carlo valuation is used to pick the best one among those. ($n > 0$)

- Let MC add a bonus to the moves
  Considering only the moves that were assigned a value by GNU Go[13], Monte Carlo valuation is scaled to the same proportions, then both valuations are added using a weighting w.

  $$V = V_{MC}\frac{MAX_{GG}}{MAX_{MC}}w + V_{GG}(1-w), \qquad (0 \leq w \leq 1)$$

  Here, V_{MC} is the score relative to the lowest score on the board when Monte Carlo scores are calculated for all moves. When only a subset of the possible moves are valued, V_{MC} is instead the Monte Carlo score relative to the pass move.

- Let MC decide if no urgent move can be found
  GNU Go valuation is used to select the moves that are within a range r of the best move, expressed as a percentage. Monte Carlo valuation is then used to select the best move from them. $(0 \leq r \leq 100)$

7.5 Automated Testing via Go Text Protocol

Go Text Protocol, GTP, is a text based protocol for communication with or between computer Go programs [WWW2]. It's supported by a number of Go programs, including GNU Go. By providing a standardised interface, it enables interaction between different software supporting the standard. For example, a GTP compatible GUI works with any GTP compatible Go engine, and it's also possible to connect two engines and let them play games automatically. The following are examples of common GTP commands:

`play white B2`	Place a white stone on intersection B2.
`genmove black`	Play a black stone where the engine chooses, and output that intersection.
`final_score`	Calculate the final score and output the result.

In the case of evaluating how Monte Carlo integration has had an impact on playing strength, it's natural to use the unmodified version of GNU Go as a benchmark. Also, enough games must be played to produce a statistically significant outcome. Because different types of integration were to be compared to find the best, each one had to be tried with a sufficient number of games. To play the games by hand would be unreasonable. An automated approach is far more appropriate, in order to play more games and achieve higher statistical accuracy.

[13] GNU Go assigns values to some moves only, so the values of other moves are unknown rather than explicitly 0, and thus they can't be used.

Using a scripting langugage, Perl in this case, it's possible to create a script that plays two programs against each other using GTP commands, as demonstrated by the following code:

```
...
open2($B_read, $B_write, "$B_cmdline");
open2($W_read, $W_write, "$W_cmdline");
...
while ($nbrpasses < 2) {
        $move = getblackresponse("genmove black");
        print $W_write "play black $move\n";
        if ($nbrpasses < 2) {
                $move = getwhiteresponse("genmove white");
                print $B_write "play white $move\n";
        }
        ...
}
$score = getblackresponse("final_score");
...
```

First, the script launches a subprocess for each of the two Go engines, and connects to the input and output streams of both. Then it's using the GTP commands `genmove` and `play` to play the game.

To be able to play a large quantity of games however, it's not good enough to manually run this script over and over again. This was solved with some additional scripting logic. First, the outcome of each game needs to be stored in a database. This is a comma separated file, with each line corresponding to one variation of Monte Carlo integration:

```
#command lines, black wins, white wins, avg score
gnugo --mode gtp --mc_first_n_moves 5 | gnugo --mode gtp,1,1,7
gnugo --mode gtp | gnugo --mode gtp --mc_first_n_moves 5,2,0,-10
```

Second, to run the first script a large number of times, another script was used. This reads from a file containing the batch instructions, comma separated:

```
#parameter, number of times to run
--mc_first_n_moves 3,10
--mc_first_n_moves 5,10
--mc_highest_n_moves 5,20
```

With the test framework in place, it's possible to specify what Monte Carlo integrations to play and the number of times to test each one, then run the entire suite. Since results are appended to a database, data can be added to from time to time.

7.6 Statistical Significance

In order to avoid statistical errors, for every game another was played with the colours switched. This is because a game may not be perfectly even for Black and White. In Go, Black always moves first, which is an advantage. To counter this advantage, White is awarded extra points, called komi. Komi is typically 5.5, 6.5 or 7.5 points, depending on which ruleset is used. The half point serves to avoid ties. This komi value is calculated based on outcomes of games played between humans. However, assuming that it would be fair in a game between two GNU Go players is not safe.

To determine whether one Go program is better than another with statistical significance[14], a sufficient number of games must be played. How many depends on the difference in playing strength between the two sides. If the difference is small, more samples are needed.

Let X be the stochastic variable that represents the number of games won by a player out of a total of n games, with p probability for the player to win. Then X is binomially distributed

$$X \sim B(n, p).$$

If $np(1-p) > 10$, we can use normal approximation to instead obtain the normal distribution

$$N(\mu, \sigma^2).$$

Thus

$$X \sim N(np, np(1-p)).$$

Ultimately, we want to be able to spot when one player is stronger than the other, so first we need the distribution for games between two equal players. It's wrong to simply assume $p = 0.5$, since games between Black and White may be uneven despite komi being used. However, as mentioned in the beginning of this section, we let each side play Black and White in equal proportion, and that takes care of the problem.

Let X_B be the stochastic variable that represents the number of games won by the player using black stones, and let X_W be the number of games won by the player using white. Let p_b be the probability for Black to win. Then

$$X_B \sim B(\frac{n}{2}, p_b)$$

and

$$X_W \sim B(\frac{n}{2}, 1 - p_b).$$

After normal approximation we get

$$X_B \sim N(\frac{np_b}{2}, \frac{np_b(1-p_b)}{2})$$

[14] with 95% confidence

and

$$X_W \sim N(\frac{n(1-p_b)}{2}, \frac{n(1-p_b)p_b}{2}).$$

Let X_{BW} represent the number of games won by a player, after playing Black and White in equal proportion. Then

$$X_{BW} = X_B + X_W.$$

Since this is a sum of two normally distributed variables it has the distribution

$$N(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2).$$

Thus

$$X_{BW} \sim N(\frac{n}{2}, np_b(1-p_b)). \tag{1}$$

Assuming no advantage is associated with playing either side, $p_b = 0.5$, and

$$X_{BW} \sim N(\frac{n}{2}, 0.25n). \tag{2}$$

Comparing the variances in (1) and (2), we observe that

$$np_b(1-p_b)) \leq 0.25n, (0 \leq p_b \leq 1). \tag{3}$$

It's shown by (1) that when playing an equal number of black and white games, the expected number of wins for a player playing an equal opponent is always $n/2$, an intuitive result. This is irrespective of p_b, i.e. unrelated to any advantages associated with playing a particular colour. The variance however, does depend on p_b. As shown by (3) the maximum is when $p_b = 0.5$. A lower variance leads to a narrower confidence interval, so for the purposes of determining confidence interval boundaries, we simply use the maximum variance, $0.25n$.

Let Z be the $N(0,1)$ normalised version of X_{BW}. Then

$$Z = \frac{X_{BW} - \frac{n}{2}}{\sqrt{np_b(1-p_b)}}.$$

To define a 95% confidence interval for Z, find z so that

$$P(-z \leq Z \leq z) = 0.95.$$

Use the inverse probability function of the normal distribution to find z

$$z = \phi(0.975) = 1.96.$$

Using $p_b = 0.5$ leads to

$$P(-1.96\sqrt{0.25n} + 0.5n \leq X_{BW} \leq 1.96\sqrt{0.25n} + 0.5n) = 0.95$$

and the interval

$$(-1.96\sqrt{0.25n} + 0.5n; 1.96\sqrt{0.25n} + 0.5n).$$

n	Lower Bound	Upper Bound	Strength Difference
50	18	32	77.78%
100	40	60	50.00%
1000	469	531	13.22%
10000	4902	5098	4.00%
100000	49690	50310	1.25%
1000000	499020	500980	0.39%

Table 7.1: The number of games n that are necessary to detect various degrees of playing strength difference.

We can now use this interval to determine with 95% confidence whether one Go program plays better than another. If the number of wins for a program falls outside the upper end of the interval, it's better than its opponent, and vice versa for the lower end.

At least 40 games must be played to satisfy the condition under which it's allowed to perform normal approximation of the binomial distribution, that $np(1-p) > 10$. Furthermore, p, although close, may not be quite 0.5 due to minor advantages for Black or White, so that would require a slightly higher n. The initial sample size was therefore set to 50 games, and then progressively increased until a statistically significant result could be obtained.

8 Experiments and Results

The results presented in this section are for Go on a 9x9 board, komi 6.5, using GNU Go 3.6 on maximum playing strength, unless stated otherwise. Although 19x19 is the official size for games between humans, 9x9 is often used for computer games, and is therefore relevant. To perform this study on 19x19 would take considerably longer[15]. There are more moves per game so games therefore take longer, and this effect is compounded because it also direcly affects the time required for a random game played by the Monte Carlo module. A greater number of random games are also required since the standard deviation of the final score is higher for a 19x19 game. Finally, effects of Monte Carlo may potentially be more subtle in a 19x19 game, which would lead to a greater number of samples required to determine differences in playing strength.

In the Monte Carlo simulation, a setting of 10000 random games is used, as is generally accepted and suggested by [BC06]. The same number is used for both the Abramson and Brügmann models to produce a fair comparison. However, it's worth noting that this makes the Brügmann model much faster, since it only plays 10000 games once to evaluate all moves rather than 10000 games for each move being evaluated (see section 5.3).

In the following sections, four different Monte Carlo integrations are tested. For each, the results are presented in a table, and there is also a graph of the relative playing strength (RPS), of the two engines. RPS is defined using the relationship

$$RPS = \frac{\textit{Games won by MC}}{\textit{Games won by GNU Go}} - 1.$$

RPS represents how many more games are won by the Monte Carlo enhanced GNU Go compared to the unmodified version, expressed as a percentage. For example, an RPS of 0% would mean the players are of equal strength. A negative value means Monte Carlo is weaker, and vice versa.

In the graphs, each bar illustrates the confidence interval for the true value of RPS for a particular type of Monte Carlo integration. With 95% confidence, it is contained within the interval. Especially, a bar that does not cross the X-axis means that one side has been determined stronger or weaker, and same for two disjunct intervals[16]. If more samples are collected, it leads to a narrower interval and a therefore a shorter bar. For reasoning around the statistical significance of the results, see section 7.6.

[15] It would take a serious number crunching machine or cluster. The 9x9 study has taken weeks to simulate on the author's desktop PC.

[16] Actually, the intervals don't quite have to be disjunct. It's enough if the confidence interval for the difference $\mu_1 - \mu_2$ doesn't contain 0.

8.1 MC Plays the First Moves

To let Monte Carlo freely play the first few moves on its own proves an unsuccessful strategy. As discussed in 7.3, Monte Carlo is normally most useful when it can not be calculated accurately what the move values are, and this is certainly the case in the opening. However, Monte Carlo is simply too weak to select moves by itself, and significantly underperforms the opening library of GNU Go. Monte Carlo is statistically weaker in every case except when Abramson plays the first move, where the outcome is unknown since the difference is too small. I would guess the behaviour would be even more disastrous on a 19x19 board, since there are many more choices, and opening patterns are firmly entrenched.

Moreover, even if this strategy had proven fruitful, it would probably be possible to tweak the opening strategy of the existing software, rather than having a complete Monte Carlo module just for a few moves. For instance, Monte Carlo likes to open in the center, which is an unusual but still perfectly valid strategy. It would be easy to change this behaviour by simply adding opening patterns to GNU Go.

In conclusion, this is not to say that Monte Carlo is useless in the opening, on the contrary, but it needs to work together with other methods.

The results are presented in table 8.1. See figure C.1 in the appendix for an illustration.

Model	n	Won MC	Won GG	RPS	RPS Low	RPS High
Abramson	1	704	726	-3.03%	-12.60%	7.56%
Abramson	3	39	61	-36.07%	-58.76%	-4.69%
Abramson	5	14	40	-65.00%	-85.60%	-35.36%
Brügmann	1	300	350	-14.29%	-26.66%	-0.01%
Brügmann	3	36	64	-43.75%	-64.50%	-15.50%
Brügmann	5	442	508	-12.99%	-23.49%	-1.17%

Table 8.1: MC plays the n first moves: results.

8.2 MC Chooses Between the Highest Valued Moves

Here, the strategy is to let Monte Carlo choose from the n highest scored moves, as determined by the GNU Go valuation. This is simply not a good strategy, and using both Abramson's and Brügmann's models, Monte Carlo is weaker. The reason is that urgent moves are ignored. Often, there is one move on the board that is much more important than the others. This is usually recognised by GNU Go by giving it a much higher score, but Monte Carlo is too blunt a tool to judge this effectively, and what happens is that another of the top n moves gets picked instead. What is missing from this strategy is taking into account the relative GNU Go valuation between the moves. Monte Carlo mustn't be

allowed to pick moves if they have a significantly lower score than the best move. This approach is tested in section 8.4.

The results are presented in table 8.2. See figure C.2 in the appendix for an illustration.

Model	n	Won MC	Won GG	RPS	RPS Low	RPS High
Abramson	2	21	49	-57.14%	-77.62%	-28.43%
Abramson	4	13	57	-77.19%	-92.64%	-56.56%
Abramson	6	12	68	-82.35%	-95.79%	-64.94%
Brügmann	2	35	65	-46.15%	-66.31%	-18.84%
Brügmann	4	36	64	-43.75%	-64.50%	-15.50%
Brügmann	6	35	115	-69.57%	-81.89%	-54.37%

Table 8.2: MC chooses between the n highest valued moves: results.

8.3 MC is Weighted with the GNU Go Valuation

Integrating Monte Carlo using this strategy produces a weighted mean of the GNU Go and Monte Carlo scores using normalised[17] values. For a discussion around normalisation of scores see sections 7.1 and 7.4. The formula for producing the weighted score is

$$V = V_{MC}\frac{MAX_{GG}}{MAX_{MC}}w + V_{GG}(1-w), \qquad (0 \leq w \leq 1).$$

This is different to the other methods presented in that it doesn't let Monte Carlo select a move by itself under given circumstances, but instead influences the moves available. The higher the value of w, the higher the influence of the Monte Carlo score. At the extreme points, $w = 0$ plays identical to GNU Go, and $w = 1$ is the Monte Carlo module without any aid from GNU Go. Since Monte Carlo on its own is a very weak player, any positive effect on playing strength should occur for a low w. In theory, plotting RPS vs w should produce a curve which starts at 0, ascends for a while if the positive effect exist, and then drops off into negative territory.

Unfortunately the results as to whether RPS actually increases are inconclusive. There may exist a small such effect, but it's too small to say so with statistical confidence. It wasn't worth pursuing a conclusive answer, since the results in 8.4 are more promising, and it's possible to say that they are statistically stronger than the ones presented in this section, so this is an inferior approach.

The results are presented in table 8.3. See figure C.3 in the appendix for an illustration.

[17] So that scores are positive and the lowest score is 0.

Model	w	Won MC	Won GG	RPS	RPS Low	RPS High
Abramson	0.05	407	423	-3.78%	-16.06%	10.25%
Abramson	0.10	374	386	-3.11%	-15.99%	11.71%
Abramson	0.15	1541	1459	5.62%	-1.68%	13.47%
Abramson	0.20	307	343	-10.50%	-23.36%	4.39%
Brügmann	0.05	1246	1264	-1.42%	-8.85%	6.60%
Brügmann	0.10	1297	1353	-4.14%	-11.18%	3.45%
Brügmann	0.15	202	248	-18.55%	-32.58%	-1.95%
Brügmann	0.20	475	525	-9.52%	-20.14%	2.43%

Table 8.3: MC is weighted with the GNU Go valuation: results.

8.4 MC Selects Within a Range of the Best Move

This strategy defines a minimum value using a percentage of the GNU Go valuation of the best move. Then the Monte Carlo valuation is used to select a move from those larger than the minimum value. This is similar to the strategy of selecting between the n highest valued moves, as described in section 8.2, but with one important difference. This approach takes into account urgent moves. If there is one move that stands out in the GNU Go valuation, that move will be selected no matter what Monte Carlo says. This is important, since GNU Go is a much better player than Monte Carlo on its own, so its valuation should bear more importance. Essentially, what this strategy allows is to let GNU Go perform the first valuation, and in the case there are moves that couldn't quite be distinguished between, Monte Carlo then selects between them.

The results prove the strategy is successful when using the Brügmann model. For Abramson, the difference is too small to determine. It's actually possible to say with confidence that this is the best strategy overall, because the second best strategy, Abramson 15% weight influence (see section 8.3), is statistically weaker.

Let μ_1 and μ_2 be the winning ratios for the two strategies to compare. If the confidence interval for the difference $\mu_1 - \mu_2$ only contains values > 0, μ_1 is statistically higher than μ_2 and strategy 1 is better. The standard deviation for the interval is given by

$$\sigma_D = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} = \sqrt{\frac{\mu_1(1-\mu_1)}{n_1} + \frac{\mu_2(1-\mu_2)}{n_2}}.$$

Choose $z = \phi(0.95) = 1.65$ for a one sided interval $[\mu_L, \infty]$. Then the interval limit is given by

$$\mu_L = \mu_1 - \mu_2 + \sigma_D z.$$

Comparing the most promising strategy, Brügmann 10% range, with Abramson 15% weight influence gives $\mu_1 = 4195/7800$, $n_1 = 7800$, $\mu_2 = 1541/3000$, $n_2 = 3000$. Inserting values shows that the former is statistically the best since

$$\mu_L = 0.006449 > 0.$$

When plotting RPS against the range r it can be seen for both Monte Carlo models that the optimal selection range seems to be around 10-15% of the best move. In theory, as with the other results, the graph should start at RPS 0, ascend, and then turn negative as the Monte Carlo influence grows greater. It's surprising that the results for $r = 5\%$ are so bad, which is reflected by both models. In theory there should be an improvement in playing strength. What this is saying is that too little Monte Carlo influence is actually worse than none at all. A possible explanation could be that the true best move is not so often found among the highest valued 5% but instead among those valued 5-15% lower than the highest valued move. This would suggest GNU Go is making a systematic error in undervaluing the true best move.

The results are presented in table 8.4. See figure C.4 in the appendix for an illustration.

Model	r	Won MC	Won GG	RPS	RPS Low	RPS High
Abramson	5%	394	446	-11.66%	-22.94%	1.15%
Abramson	10%	566	574	-1.39%	-12.22%	10.76%
Abramson	15%	177	223	-20.63%	-35.12%	-3.34%
Abramson	20%	54	96	-43.75%	-61.11%	-21.42%
Abramson	25%	154	196	-21.43%	-36.70%	-3.00%
Brügmann	5%	81	119	-31.93%	-49.46%	-9.78%
Brügmann	10%	4195	3605	16.37%	11.29%	21.69%
Brügmann	15%	3323	2877	15.50%	9.87%	21.44%
Brügmann	20%	493	557	-11.49%	-21.66%	-0.09%
Brügmann	25%	344	406	-15.27%	-26.73%	-2.20%

Table 8.4: MC selects within a range r of the best move: results.

9 Conclusions

The study tested four different strategies for integrating a Monte Carlo module into GNU Go, and for each one the degree of Monte Carlo influence was parameterised and varied to empirically find the optimal level. A script was used to play a large number of games between the unmodified and modified program, and conclusions were drawn from observing the outcomes using confidence intervals for the binomial distribution.

The original questions for the thesis were whether it was possible to improve GNU Go using Monte Carlo statistics, and if so, how it can be integrated most effectively. Both questions have been successfully answered. It is possible, and it can be said with statistical confidence that the most effective integration strategy out of the ones tested is to first value the move alternatives using the original GNU Go valuation, and then use the Brügmann Monte Carlo model to select from the top 10-15% of the moves.

As for the other strategies for GNU Go and Monte Carlo integration tested, they either statistically worsened the performance, or it was inconclusive whether there was an improvement or not. The important point however, is that they were all statistically weaker than the best method mentioned previously.

The results suggest that for a successful integration strategy, there is an optimal level of Monte Carlo influence. This is easy to imagine, since the extreme cases are the original GNU Go (no influence) and the Monte Carlo module alone (maximum influence). Standalone Monte Carlo is clearly weaker than GNU Go. Hence if an improvement exists, it does so at some point between the two extremes, and it's plausible that that point is unique.

There is little point in this study trying to determine a very accurate value for the optimal influence level, since this most likely would vary with the exact nature of the program and Monte Carlo model being integrated. Tweaking the influence would be done as a last step in an integration. However, the overall integration strategy successful here, i.e. to let MC select within a range of the best move, should be a good strategy also for other projects.

It's interesting that the best results were obtained using the Brügmann model, since this is using the all-moves-as-first heuristic, which significantly[18] speeds up the simulation but is regarded as a compromise [BH03]. Across the four types of integrations tested, the performance for the Abramson and Brügmann models is broadly similar for the same parameters, but in the case of the most successful strategy, Brügmann is statistically stronger. The point estimate of the improvement on GNU Go 3.6 is 16.37%, and the computational cost is low.

It seems plausible that results can be further improved using a more refined Monte Carlo model, for which the work undertaken here can serve as a basis.

[18] Brügmann performs 1 simulation to evaluate all moves on the board, whereas Abramson performs 1 per move. This yields a big improvement if many moves are evaluated.

9.1 Future Work

There are two important areas for further research:

9.1.1 A Study Using a 19x19 Board

Although 9x9 is often used for computer Go, this is only really because it simplifies the problem. It's good to try out strategies on a small board, but eventually the aim must be to play on 19x19, which is the official size. There are several difficulties here. The number of random games played needs to increase, and games will be longer. It may also be that the positive effect of Monte Carlo is smaller, calling for a more refined model to achieve results. To date, Monte Carlo programs have been successful on 9x9, but 19x19 has proven more difficult.

9.1.2 Improving the Effectiveness of the Monte Carlo Model

The models used in these tests are the simplest possible, since they consist only of the Monte Carlo heuristic itself. There are many ways of potentially improving on this. One that looks interesting is combining Monte Carlo with tree search [C06]. Another idea could be to use Monte Carlo not on a global scale, but to examine specific situations on the board, such as life & death status of groups.

9.2 Comments on the Results

I asked Gunnar Farnebäck, who works on GNU Go, for comments on my results. This is his reply:

> State of the art for Monte Carlo Go has virtually exploded during the time you've worked on this thesis. Today[19] there is no doubt that Monte Carlo programs are superior on 9x9 board and the best are probably at a level around 3-5 dan[20]. On 19x19 they also seem to have overtaken the traditional programs, although not as markedly. The keys to the success have been tree search with help of UCT (Upper Confidence bounds applied to Trees) and improved rules for move generation in the Monte Carlo simulations. For further reading refer to Sylvain Gelly's articles from 2006 and 2007 and Rémi Coulum's articles from 2007.

This confirms the positive result I arrived at in this experiment, and indeed it also confirms that it's possible to improve Monte Carlo Go much beyond the very simple model that was used here.

[19] January 2008

[20] To give an idea, this is on par with the top amateur players in Sweden, but doesn't reach the level of a Chinese professional.

References

[BC01] Bouzy B., Cazenave T., *Computer Go : an AI oriented Survey*, Artificial Intelligence, Vol. 132 n1 (2001), 39-103

[BH03] B. Bouzy and B. Helmstetter. *Monte carlo go developments*. In Ernst A. Heinz H. Jaap van den Herik, Hiroyuki Iida, editor, 10th Advances in Computer Games, pages 159-174, Graz, 2003. Kluwer Academic Publishers. 2003.

[BDSS02] Billings, D., Davidson, A., Schaeffer, J., and Szafron, D. (2002). *The challenge of poker*. Artificial Intelligence 134, pages 201-240.

[A90] B. Abramson. *Expected-Outcome: A General Model of Static Evaluation*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(2):182-193, 1990

[BC06] Bruno Bouzy and Guillaume Chaslot. *Monte-Carlo Go Reinforcement Learning Experiments*. In G. Kendall and S. Louis, editors, IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA, pages 187-194., 2006.

[B93] Brugmann, B. (1993). *Monte Carlo Go*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 12, pp 182-193.

[C06] Rémi Coulom. *Efficient selectivity and backup operators in Monte-Carlo tree search*. Submitted to CG 2006, 2006.

[CH05] Tristan Cazenave and Bernard Helmstetter. *Combining tactical search and Monte-Carlo in the game of Go*. In IEEE CIG 2005, 2005.

[T00] T. Thomsen. *Lambda search in game trees with applications to Go*. In I. Frank and T. Marsland, editors, Proceedings of the 2nd international conference on computers and games, pages 57-80. Springer-Verlag, 2000.

[WWW1] GNU Go homepage: `http://www.gnu.org/software/gnugo/`

[WWW2] GTP homepage: `http://www.lysator.liu.se/~gunnar/gtp/`

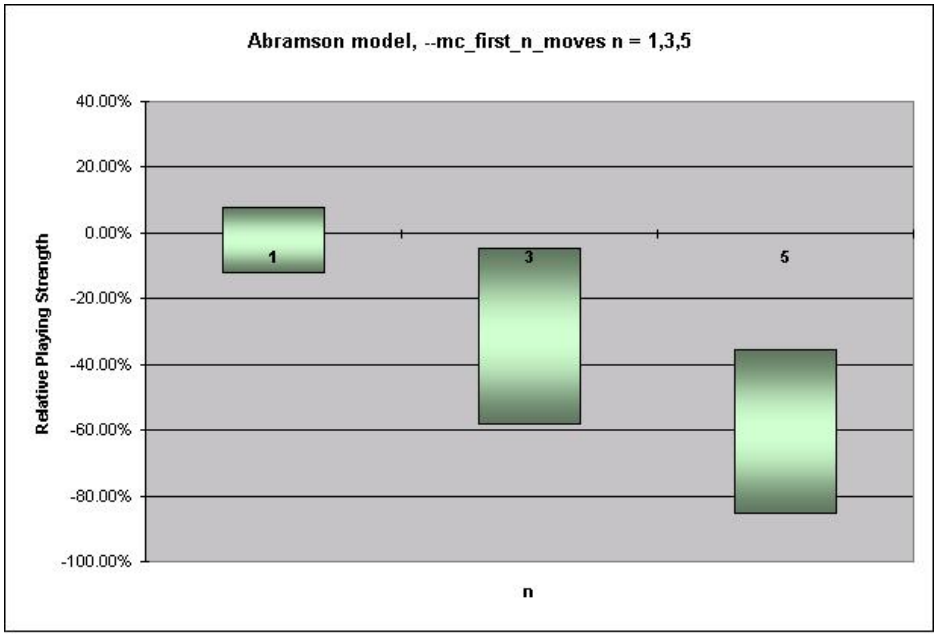
[WWW3] Sensei's Library: `http://senseis.xmp.net`

A Glossary of Terms

`contact play`	A move that is played adjacent to an enemy stone.
`joseki`	Japanese for "Set stones". The term refers to an established sequence of moves which results in a fair outcome for both Black and White.
`eye`	Refers to an empty intersection on the board surrounded by stones of one colour.
`kill`	See life & death. To kill a group of stones means to play a move that renders them dead.
`ko`	A conditional form of life. It's better to live (or kill) unconditionally. Please refer to Go litterature for more information.
`komi`	In Go, Black always moves first, which is an advantage. To counter this advantage, White is awarded extra points, called komi. This is typically 5.5, 6.5 or 7.5 points, depending on ruleset. The half point serves to avoid ties.
`liberty`	An empty adjacent intersection. A group with a low number typically risks capture.
`life & death`	A living group of stones can not be captured no matter what the opponent does. This can be achieved by surrounding two independent areas of territory. A dead group is a group that cannot enter the live state no matter what move is played. Of course however, mistakes happen and living groups die and vice versa.

B Literature

- There are many books that introduce the game. One is the Learn to Play Go series: Kim, Janice and Jeong Soo-hyun, five volumes: Good Move Press, Sheboygan, Wisconsin, second edition, 1997. ISBN 0-9644796-1-3.
- Go Wikipedia page: `http://en.wikipedia.org/wiki/Go_(board_game)`
- To play Go online, visit the Kiseido Go Server: `http://www.gokgs.com/`



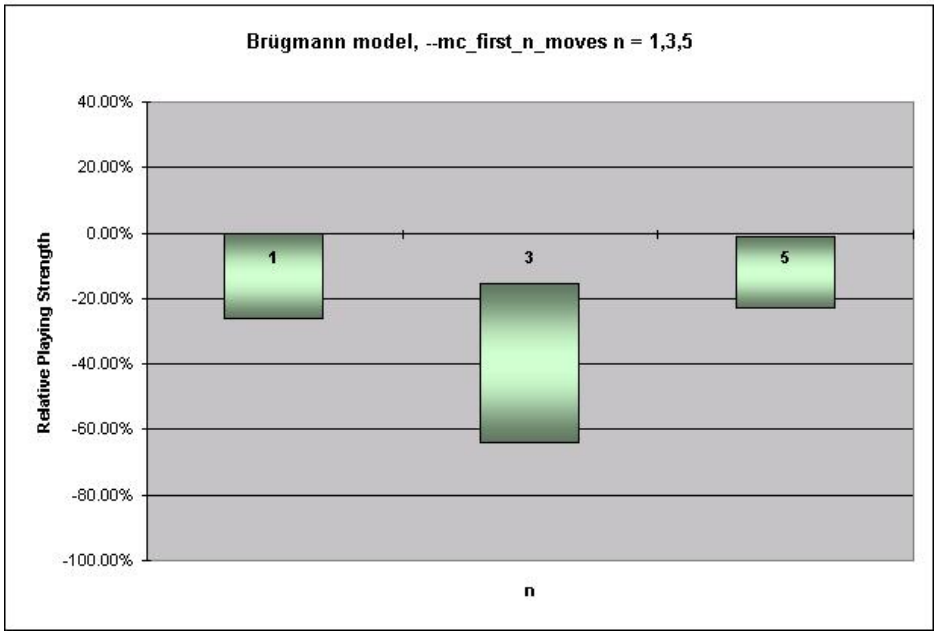
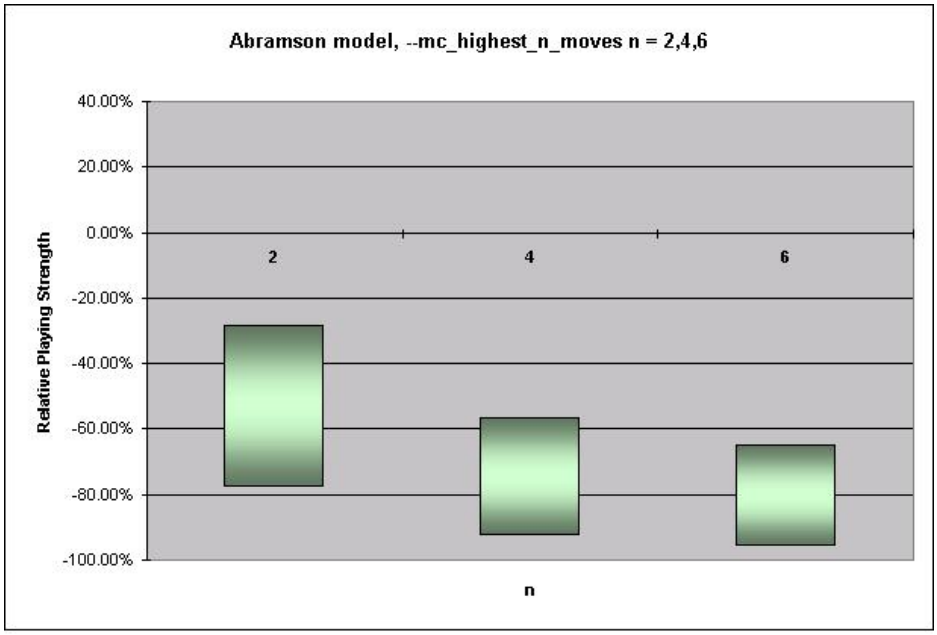


Figure C.1: MC plays the first moves: 95% confidence intervals for the difference in playing strength.



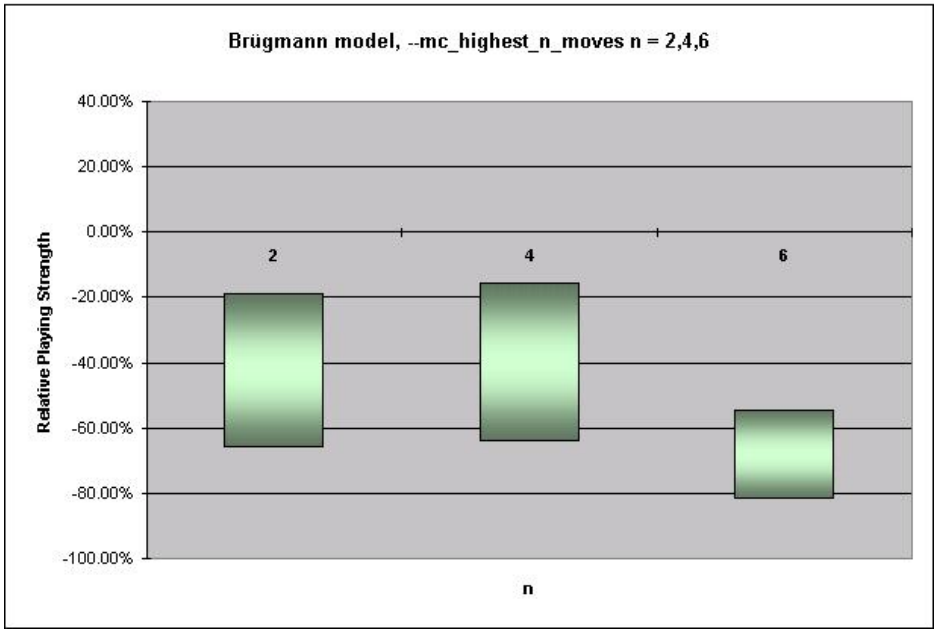
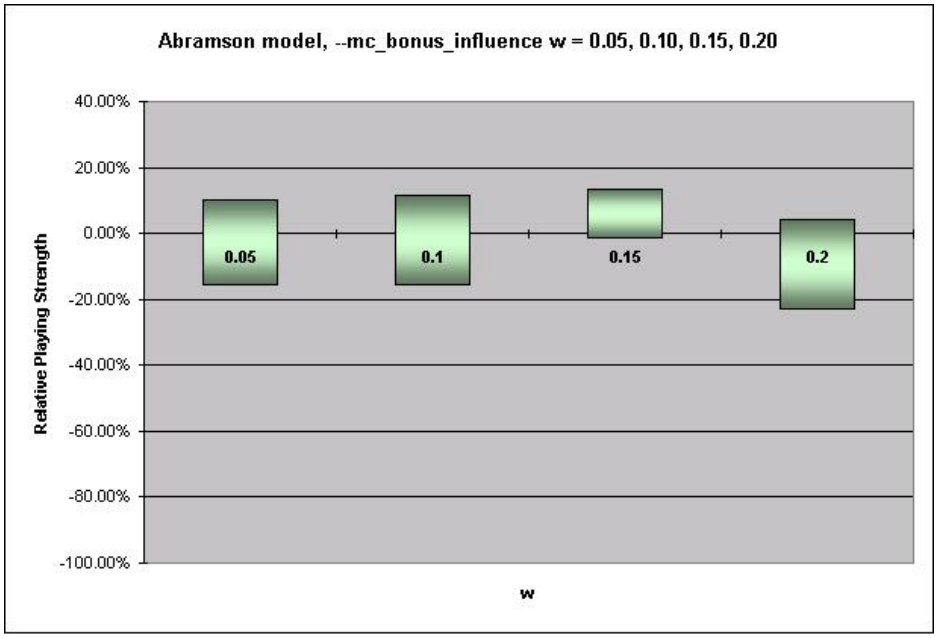


Figure C.2: MC chooses between the highest valued moves: 95% confidence intervals for the difference in playing strength.



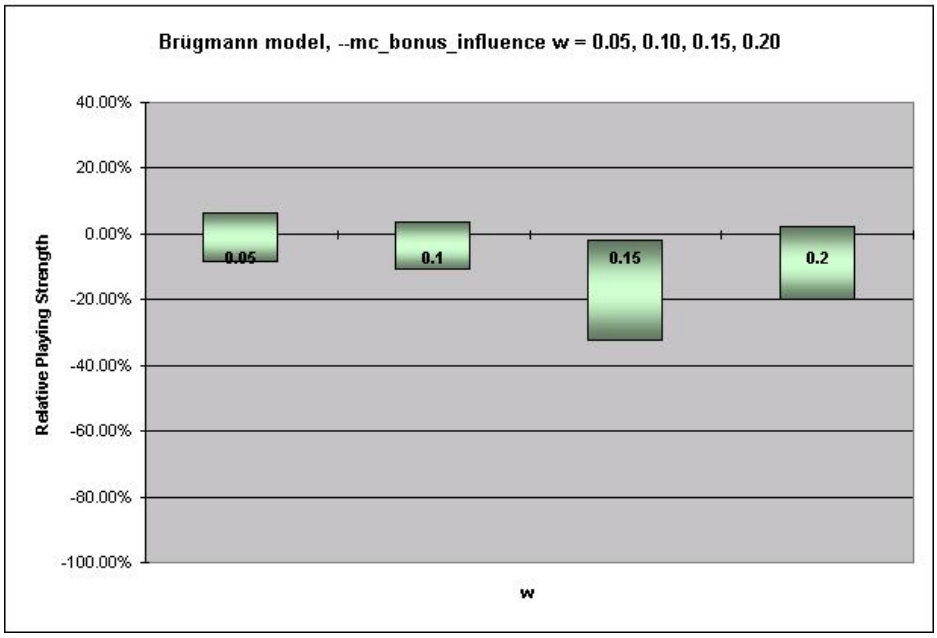
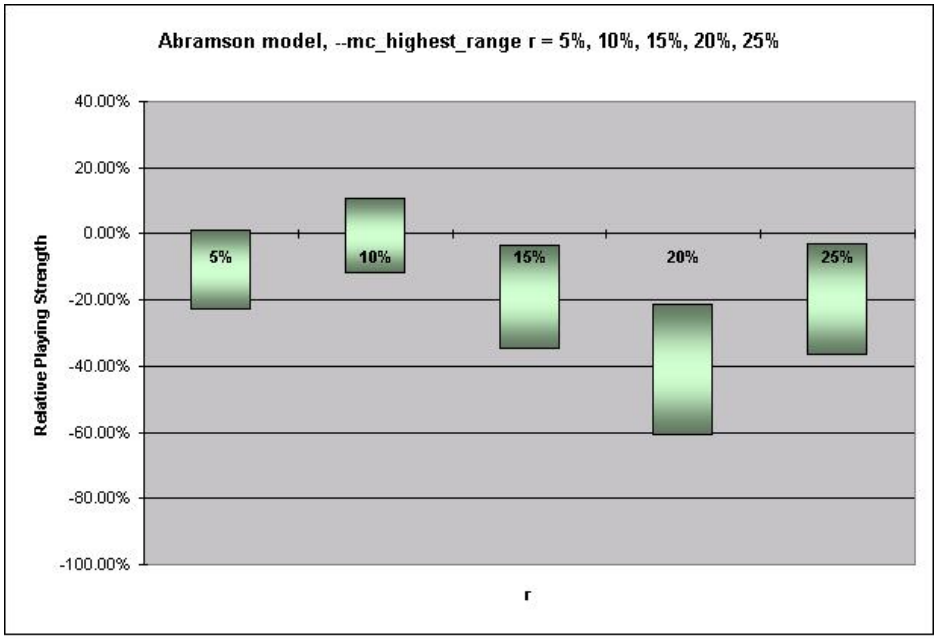


Figure C.3: MC is weighted with the GNU Go valuation: 95% confidence intervals for the difference in playing strength.



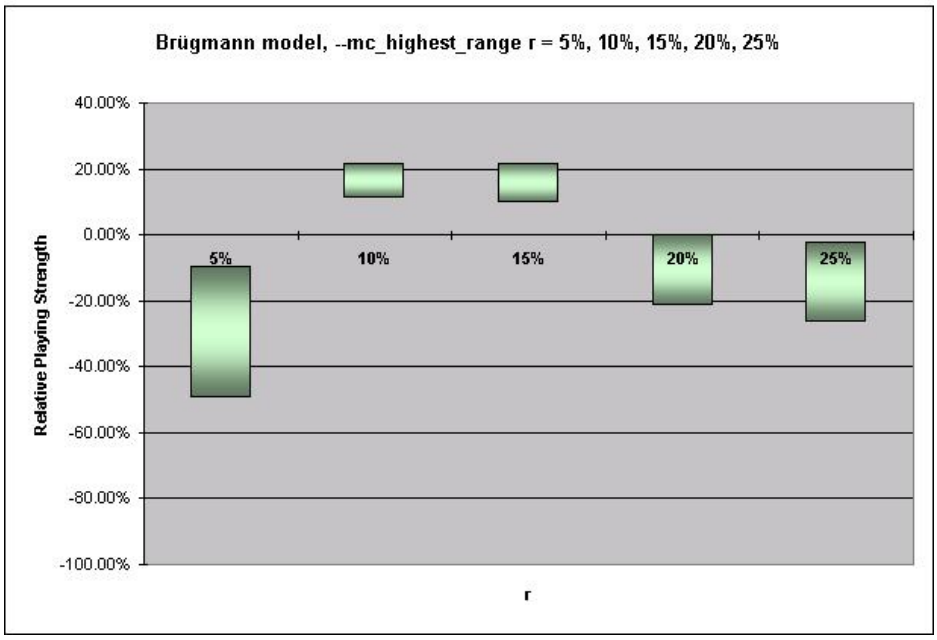


Figure C.4: MC selects within a range of the best move: 95% confidence intervals for the difference in playing strength.