Monte Carlo Tree Search with Bayesian Model Averaging for the Game of Go

John Jeong You

A subthesis submitted in partial fulfillment of the degree of
Master of Computing (Honours) at
The Department of Computer Science
Australian National University

May 2012

Typeset in Palatino by TEX and LATEX 2_{ε}.

Except where otherwise indicated, this thesis is my own original work.

John Jeong You
31 May 2012

To Hwa, Me and Wu

Acknowledgements

Most of all, thanks to my supervisor Scott Sanner. I'm rather certain that I won't be able to get more support than what I've had throughout this year. Thanks to his patience and care. Probably not the best outcome I have, but I know that it was the best opportunity I can imagine. Thanks to Scott for guiding me through this wonderful task with vast knowledge and kindness.

Thanks to Joel Veness for the friendly conversation on the implementation issue of MCTS based Go engine. Also thanks for letting me have access to his code of Goanna, a well formulated MCTS based Go engine with great performance which have helped me a lot in the process of implementing GoYui.

Thanks to Hendra and Zhe, my friends of whom I got courage to work through this year.

Finally, thanks to my parents, who have fully supported me throughout this program with love.

Abstract

Computer Go is the next grand challenge for AI games research and in recent years, Monte Carlo tree search (MCTS) algorithms have been used in the top-performing computer Go systems. MCTS is a sampling method for sequential processes that incrementally builds a tree to guide the sampling process. This thesis presents three novel contributions for MCTS in computer Go.

In the computer Go literature, there has been little effort to learn from local similarities between search tree nodes (i.e., Go boards) thus leading to potential sampling inefficiency. In this thesis, we first contribute locally weighted regression (LWR) as an update method in MCTS to address this problem. LWR uses similarity kernels to perform a weighted average of sampled outcomes from similar sequences of play. We evaluate a variety of kernels in LWR for MCTS and as our second major contribution, we show that the top-performing "RAVE" variant of MCTS can be interpreted as LWR with a novel data-dependent kernel that does not require a priori knowledge of Go.

As our third contribution, we note that MCTS algorithms have derived great benefit from combining or mixing multiple models; i.e., multiple models are proposed to predict the outcome of play from a given Go board and these models are averaged together to provide a more accurate and lower-variance prediction. In this thesis we propose Bayesian model averaging (BMA) as a theoretically well-founded alternative mixing model for MCTS. BMA shows reasonable performance compared to other previously proposed mixing methods but crucially has a clear semantics for its tuning parameter that can be interpreted as a model prior. Furthermore, compared to similarly principled mixing methods such as minimum mean square error (MSE), BMA has more expressive power concerning the classes of mixture models it can support. Hence we claim that BMA is an attractive alternative to existing mixing models for MCTS.

Finally, we note that while this thesis presents novel algorithms, theoretical justifications, and insights for MCTS in computer Go, we conjecture these ideas may be useful in the wider context of goal-oriented search, planning, and learning that is pervasive throughout AI.

Contents

Chapter 1

Introduction

1.1 Importance of Go for Artificial Intelligence (AI) research

Go is a 2000 year old board game from ancient China. It is a two player, adversarial game. Each player takes turns to place a stone of their own color on a 19x19 grid board.

From the early days of AI research, games have served as a test bed for testing ideas. This have been compared to using fruit flies in genetics research [Marsland and Schaeffer 1990]. Fruit flies are cheap and easy to breed and each generation only takes 2 weeks. As such a test bed, chess was the drosophila of AI until late 1990s. But in 1997, Garry Kasparov, the world champion chess player of that time, lost to Deep Blue. And the journey of conquering chess with machines has met its destination. In these days, even the programs running on a mobile phone can play at a grandmaster level. And now, Go is considered to be the next drosophila of AI.

Here are some of the properties that were not so common in other previously challenged board games.

1. **High branching factor.** Even from 1970s the complexity of Go was recognized to have a significant difference from chess [Berliner 1978]. In the early game, there are around 300 possible moves for each player on a 19x19 board.

2. **Difficulties in evaluating the mid game.** Unlike chess or checker, there is no known hueristic, or feature to evaluate the mid-game board of Go. Deep Blue makes extensive use of mid-game hueristics [Campbell et al. 2002]. This is due to the nature of the game. In chess and checkers the mid-game can be evaluated to a certain degree with how many pieces one has on the board. This is not true with Go. A stone's effectiveness depends heavily on how the game progresses and this is hard to predict.

For these reasons, traditional alpha-beta search is not feasible for Go. On a 19x19 board, Go has roughly 10^{170} possible cases [Gelly and Silver 2011]. And these factors make computer Go an important and difficult challenge.

1.2 Technical Contribution

Before 2007, computer Go programs did not have reasonable performance. Even the best programs were weaker than amateur level players. The main focus of research in these days was to encode the human expert knowledge to create stronger heuristics. But these heuristics were not easy to create and the resulting programs didn't have much strength.

After 2007, Monte Carlo tree search (MCTS) became popular in Go programs. MCTS is a Monte Carlo method with a tree structure to store and guide the sampling process. MoGo was the first program to have competitive strength against professional players in 9x9 board [Gelly et al. 2006]. In 2009, Fuego became the first program to conquer a win against top-level 9 dan professional player in 9x9 board [Enzenberger et al. 2010]. Currently, Go programs can play almost flawless games on 7x7 board. They are about at the same strength with top level human players on 9x9 board [Enzenberger et al. 2010]. Still, Go programs are much weaker than professional players on a 19x19 board.

Following is the list of technical contributions of this thesis:

1. A Bayesian update process with Gaussian prior for Monte Carlo tree search (MCTS) is derived. Current literature does not have strong emphasis on the similarities within the positions. Bayesian update process with Gaussian prior provides a possible method to update the tree structure of MCTS with consideration on the similarities between the positions.

2. Locally weighted regression (LWR) is tested as an update process for MCTS. With the intractability of Bayesian update process with Gaussian prior for MCTS, LWR is investigated as a simpler model to incorporate the similarities within the board. LWR is a kernel based method which deals with local similarities. Kernel simply means a certain similarity measure. Some primitive kernels were tested to see the effectiveness of LWR in MCTS. Empirical results show that good kernel design is the deciding factor for the performance of LWR in MCTS.

3. Rapid action value estimation (RAVE) is explained within the LWR framework. RAVE is the dominant MCTS variant in the current computer Go literature. But it is not given enough theoretical backgrounds. In this thesis, RAVE is explained within the LWR framework. RAVE is basically a LWR method with a simulation based kernel. This brings us two possible directions to look for. One for kernel based MCTS. We now know kernel based MCTS does work which is shown by RAVE. And the other one for simulation based kernels, which can be RAVE variants.

4. Bayesian model averaging (BMA) process for binomial data is derived. BMA is a model mixing method with a Bayesian setting. As far as it is known to us, this is the first time BMA for binomial data is being presented.

5. BMA is proposed as an alternative model mixing method for MCTS. Empirical results show that the model mixing method is a crucial factor for the performance of MCTS. In this thesis we investigated the potential of BMA as a model mixing method. Empirical results show that BMA can perform on par with other mixing methods used in the current literature. Also, BMA resembles one of our common assumptions on the MCTS, which is the expectation that the plain Monte Carlo estimation will give us the most precise evaluation when we have large number of samples. Also the free variable of derived BMA has a clear semantics.

   These factors show the possibility of BMA as a model mixing method for MCTS.

1.3 Outline

Following is an outline for the rest of this thesis:

- Chapter 2 presents basic game ruless of Go.
- Chapter 3 talks about theoretical background of MCTS and its application on computer Go. RAVE and UCT are explained as enhancements on MCTS. Considerations on developing and testing a simple MCTS Go engine, GoYui, is also briefly discussed.
- Chapter 4 is about application of LWR on MCTS. Several primitive kernels for LWR on computer Go is suggested and tested on GoYui against GNU Go. Also, strength of RAVE is explained within the framework of LWR. Also, Bayesian update process is discussed and a possible update process is derived for MCTS.
- Chapter 5 introduces BMA as an alternative mixing method for MCTS. Derivation of BMA for binomial data set is presented. BMA is applied to MCTS based Go engine, GoYui, and tested against GNU Go. Analysis of the performance and the behaviour of BMA is also given in this chapter.
- Chapter 6 summarizes the thesis with emphasis on contributions and gives future directions.

Chapter 2

Game rules of Go

The name Go comes from the Japanese name of the game, 'Igo'. The Chinese name of the Game is 'Weiqi' and the Korean name is 'Baduk'. Japanese name is widely used in rest of the society as the game was introduced to the other part of the world by Japanese. Go is popular in China, Japan, and Korea. There are professional players and some dedicated cable channels as well. There are couple of annual tournaments in each country [1].

This chapter presents basic rules of Go. Terminology and contents are based on [Baker 2008], [van der Werf et al. 2005] and [Spight 2001] .

2.1 Setting of the board

A normal Go board consists of 19x19 intersections, or positions. Due to its complexity, beginners in Go and basic computer Go programs usually play on a smaller board - 9x9 is a common size of the board for such purposes.

2.2 Progress of Game

Figure 2.1 shows an example of how Go is played.

1. **Turn :** Each player takes turn in Go. In each player's turn, she can place a stone of her color on one of empty intersections on the board. At the start of the game, the player with black stone starts the game and then the opponent with white stone takes turn.

2. **Pass :** On each turn, the player can decide to pass his move and let the opponent to take his turn again. Passing occurs when the player admits his lose, or when the player judges that additional move will not benefit him.

3. **End of game :** The game ends when both player passes. A possible end game is shown in Figure 2.7.

[1]http://en.wikipedia.org/wiki/List_of_professional_Go_tournaments

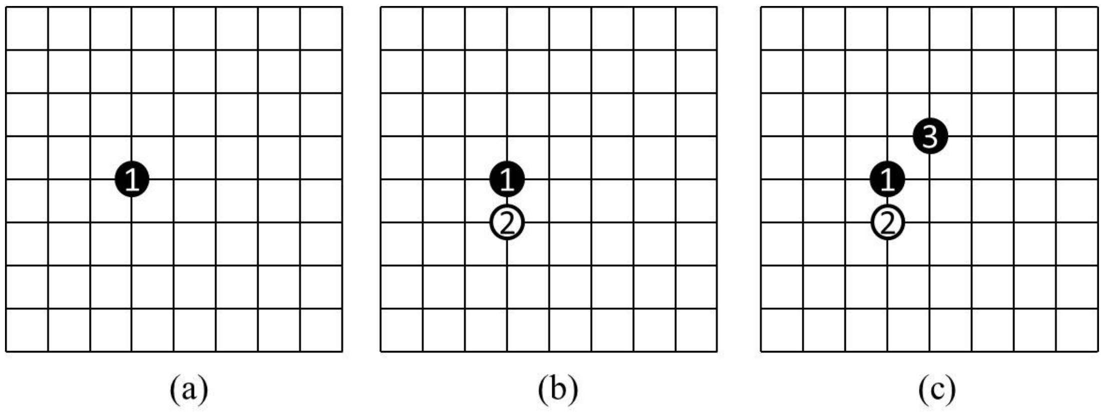


Figure 2.1: How Go is played. First few moves are shown in this figure. Each player takes turn to place a stone of their color on the board. (a) Game always starts with black's turn. Black starts by placing stone 1. (b) White places stone 2 (c) Each player takes turn.

2.3 Chain, Liberty and Captures

- Chain refers to the group of stones of same color on a Go board which are connected to each other. When two stone is right next to each other, they are considered to be within a same chain.
- Liberty of a chain refers to the number of empty spaces surrounding the chain.
- Capture occurs when one's opponent reduces the player's chain's liberty to zero by placing a stone. Suicide is inhibited in Go, so one cannot reduce one's own chain's liberty and get them captured. Figure 2.3-(c) shows an example of capture move.
- Suicide can occur when a player's move results in a chain with 0 liberties. When this move captures opponent stones, it is a legal move. But if it does not it is illegal and black can't play on that position. Figure 2.3-(f) shows a case where black can't play on position 3 which results in a suicide.

2.4 Ko and Superko

Ko and superko are rules for ensuring the progress of game.

1. **Ko :** A player can not play to make the current board identical to the board before opponent's last move. Figure 2.4 shows a typical case. This occurs when you can capture opponent's last stone by current move. Ko rule prohibits such moves. Figure 2.5 shows how the position can be resolved.
2. **Superko :** A player can not play to make the current board identical to any of previous boards. When there are more than 2 ko points on the board, there is a

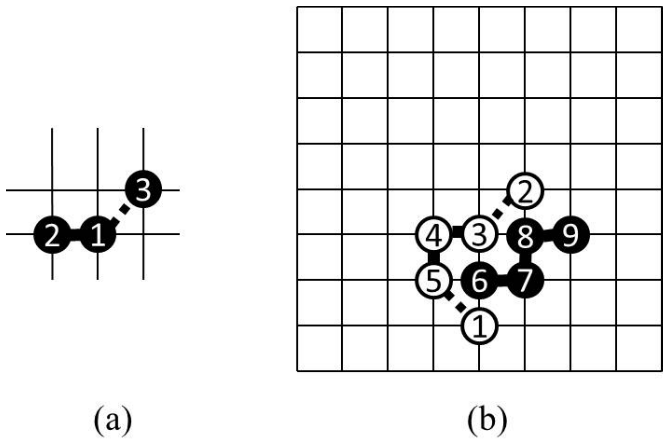


Figure 2.2: Chain. (a) Stone 1 and 2 are in a chain. But stone 3 is not in a same chain with 1 and 2. (b) Stones 3,4,5 are in a chain. But 2 and 1 are not in the chain. Stones 6,7,8,9 are in a chain. A chain only consists of same color.

possibility that the board can repeat itself even with ko rule. The superko rule prohibits such situation. Figure 2.6 shows a example of repeating board with 3 ko points.

2.5 Scoring Scheme

There are two popular methods in Go community to evaluate the end game board. In most cases these two methods give identical results.

- **Territory scoring.** Counts the number of empty positions surrounded by the player and number of opponent's stones captured.
- **Area scoring.** Counts the number of empty positions surrounded by the player and the number of player's alive stones on the board.

Territory scoring is common in computer Go literature. But both scoring methods usually give identical results.

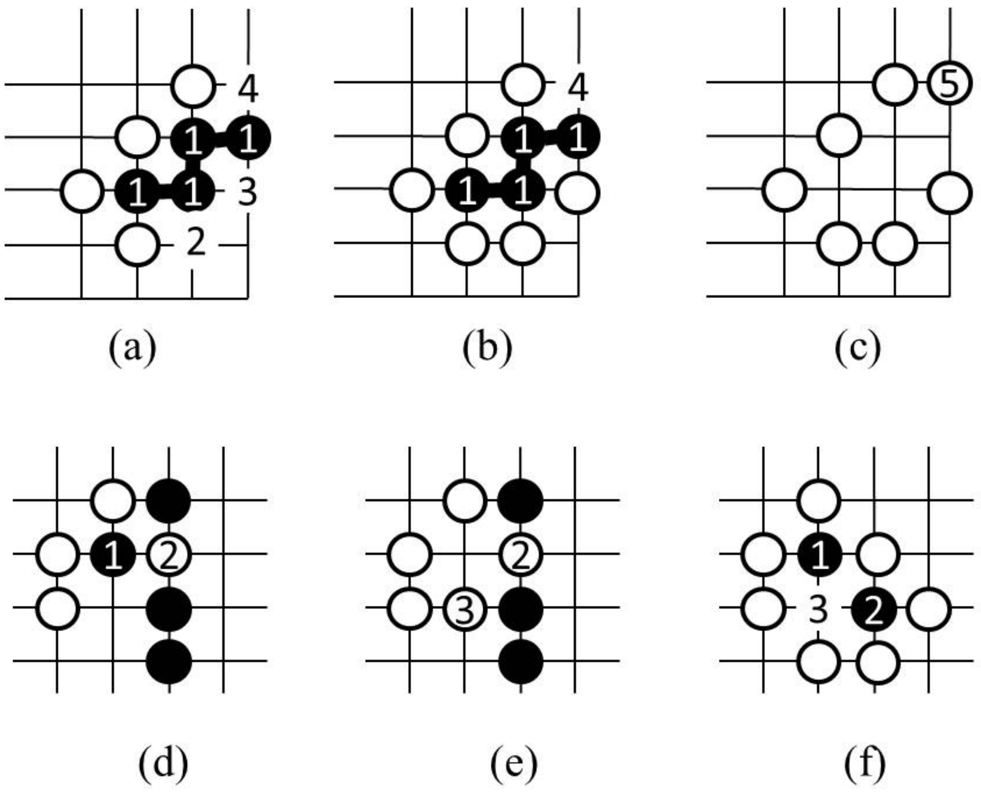


Figure 2.3: Liberty and Capture move. (a) Chain 1 has 3 liberty points, positions 2,3, and 4. When all of these liberty points are occupied by the opponent, the chain gets captured. (b) Chain 1 has 1 liberty point, position 4. In this case we call chain 1 is in Atari. This is a situation where the opponent can capture the chain with next move. (c) White captures chain 1 with move 5. (d) Stones 1 and 2 are in Atari. (e) Stone 1 is captured by move 3. (f) Stones 1 and 2 are in Atari. In this case black can't play on position 3, which will result in a suicide.

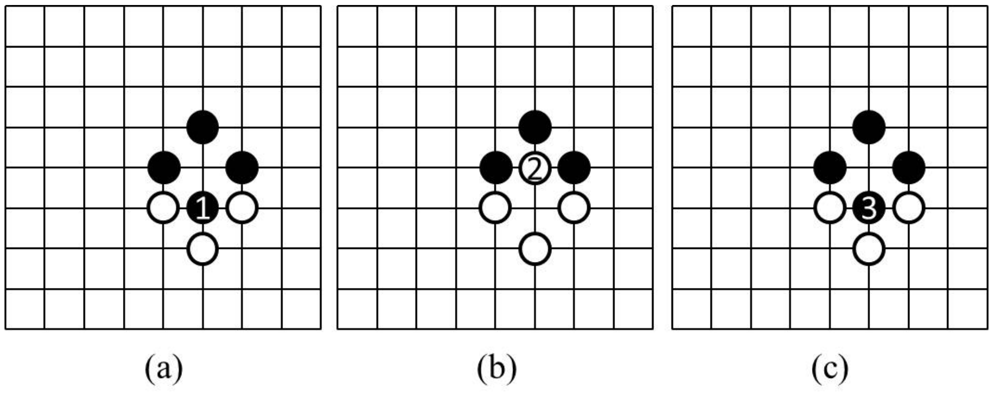


Figure 2.4: Repeating board, a violation of Ko rule. When we don't use Ko rule the game can continue indefinitely. Boards (a) and (c) are identical. And this can repeat again. Ko rule makes move from (b) to (c) an illegal one and ensures the progress of game.

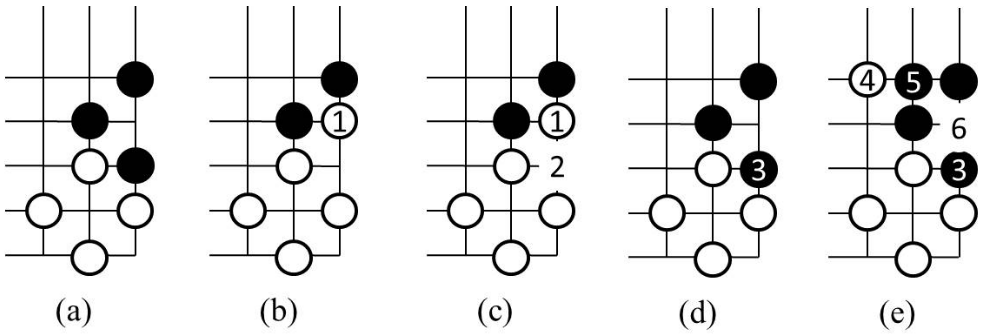


Figure 2.5: How a Ko position can be resolved. After move 3 on (d), white can't repeat the capture and return to (a). Still white can play on other places and come back to the previous position. In (e), position 6 is not illegal for white even if it had (a) in its previous configuration.

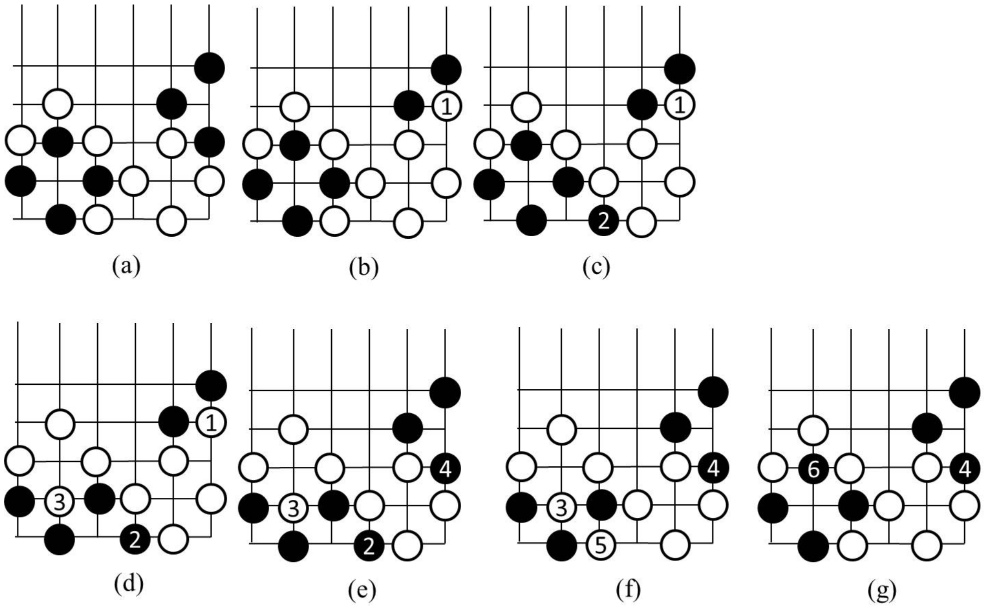


Figure 2.6: Repeating board, a violation of Superko rule. There are cases where the board can repeat itself even with Ko rule. In this figure, (a) and (g) are the same resulting from multiple ko points. Superko rule prohibits this and the move 6 at (g) is illegal.

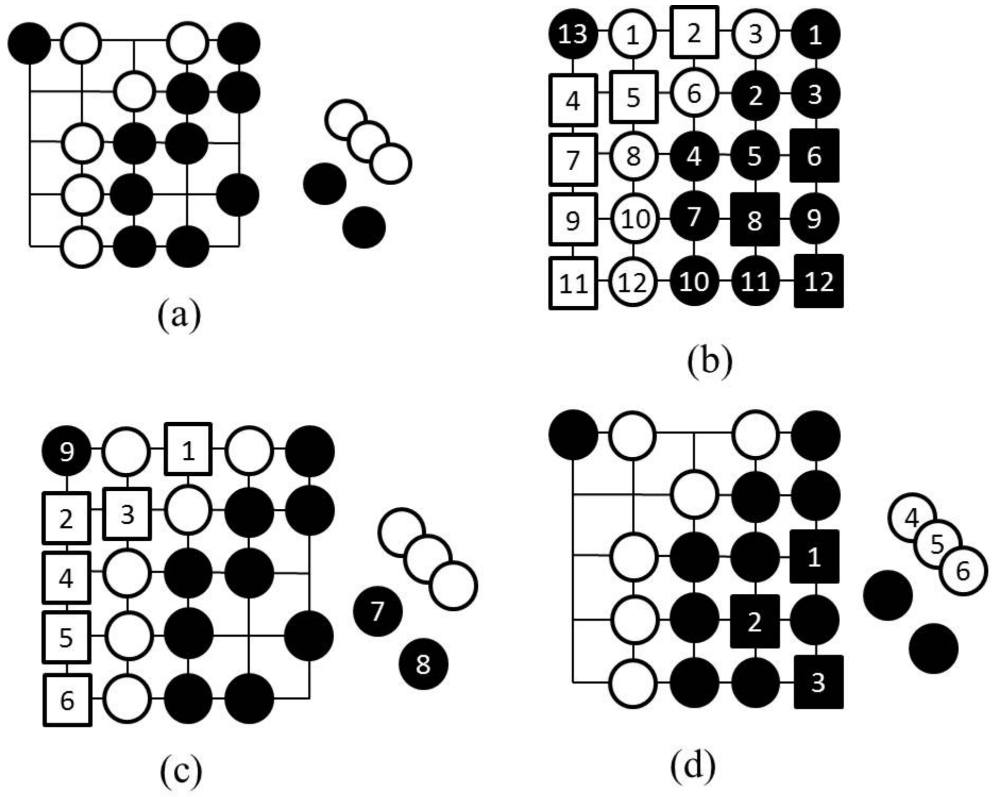


Figure 2.7: Scoring schemes. (a) A possible end game. There are 3 white stones and 2 black stones captured. (b) Area scoring. Empty positions surrounded by the player and the player's alive stones on the board is counted. (c) Territory scoring for white. Empty positions surrounded and captured black stone is counted. (d) Territory scoring for black.

Chapter 3

Background

This chapter talks about the dominant algorithms in current computer Go literature. Issues related to implementing a simple MCTS based Go engine are also discussed. Section 3.1 talks about MCTS and its enhancements, RAVE and UCT. RAVE and UCT combined with MCTS have shown to be very effective approach for computer Go. Section 3.2 talks about methods to combine models for MCTS in the current literature. These methods are important but have not been given enough theoretical emphasis. Finally Section 3.3 discusses issues on implementing and testing a MCTS based Go engine. This section describes the approach taken in creating a simple MCTS based Go engine, GoYui. GoYui has been used as a research platform for testing various ideas on MCTS throughout this thesis.

3.1 Monte Carlo Tree Search (MCTS)

3.1.1 Basic Framework of MCTS

Monte Carlo tree search (MCTS) is a widely used search algorithm which bases its search on Monte Carlo (MC) sampling. MC sampling simply means a random sampling. The basic idea of MCTS is to store the sampled data from MC sampling into a tree structure and to use this information on guiding the direction of future sampling processes. By its nature, MCTS produces biased samples. MC sampling is useful on the domains where exact evaluation is hard but samples are easy to get.

The game of Go fits very well with MCTS. Evaluating a mid-game board is hard in Go. Creating an optimal move is also hard due to its innate complexity. But end game boards are rather trivial to evaluate. As it has simple rules, creating reasonable random samples are relatively doable in Go.

The basic algorithm for MCTS is presented in Algorithm 1.

MCTS can be represented with four major parts [Chaslot et al. 2008]:

1. **Tree Search.** This part of MCTS defines and distinguishes the variants of MCTS. Tree search part searches down the tree by selecting the best child node of each node. The definition of 'best' child node determines how the whole MCTS will operate. The goodness of each child node can be represented with Q-value associated with it. (3.1) shows basic Q-value used on MCTS. It is simply the

Algorithm 1: Monte Carlo Tree Search

```
/* $n_{max}$ : maximum number of iterations                                  */
/* $RandomPlayout(s)$ : Randomly generated next node which
   only follows default policy                                               */
/* $I_i(s,a)$ : indicator function, equal to 1 if action $a$ is
   selected from node $s$ during $i$th sampling, equal to 0
   otherwise                                                                 */
procedure $MCTS(s_{root}, n_{max})$
$T = newEmptyTree()$
$T.root = s_{root}$
for $i \leftarrow 0$ to $n_{max}$ do
    // Tree Search
    $s = T.root$
    while $s.childcount \neq 0$ do
        $s = BestChildNode(s)$
    // Expanding the Tree
    $expand(s)$
    // Monte Carlo Sampling
    while $EndofGame(s) \neq true$ do
        $s = RandomPlayout(s)$
    // Tree Update
    $z = Evaluate(s)$
    $Update(T, S_i, I_i(s,a), z)$
return $T$
end procedure
procedure $BestChildNode(s)$
$c = s.child(0)$
for $i \leftarrow 1$ to $s.childcount$ do
    // i th child of node s
    if $c.Q < s.child(i).Q$ then
        $c = s.child(i)$
return $c$
end procedure
procedure $expand(s)$
// Returns the set of all legal actions from state s
$L = LegalActionFromState(s)$
for $i \leftarrow 0$ to $L.count$ do
    // Returns a Node generated by action L(i) from state s
    $d = newNode(s, L(i))$
    // May have optimistic initialization
    $d.visit = 0 + randomtiebreaker()$
    $d.Q = OptimalValue + randomtiebreaker()$
return $c$
end procedure
```

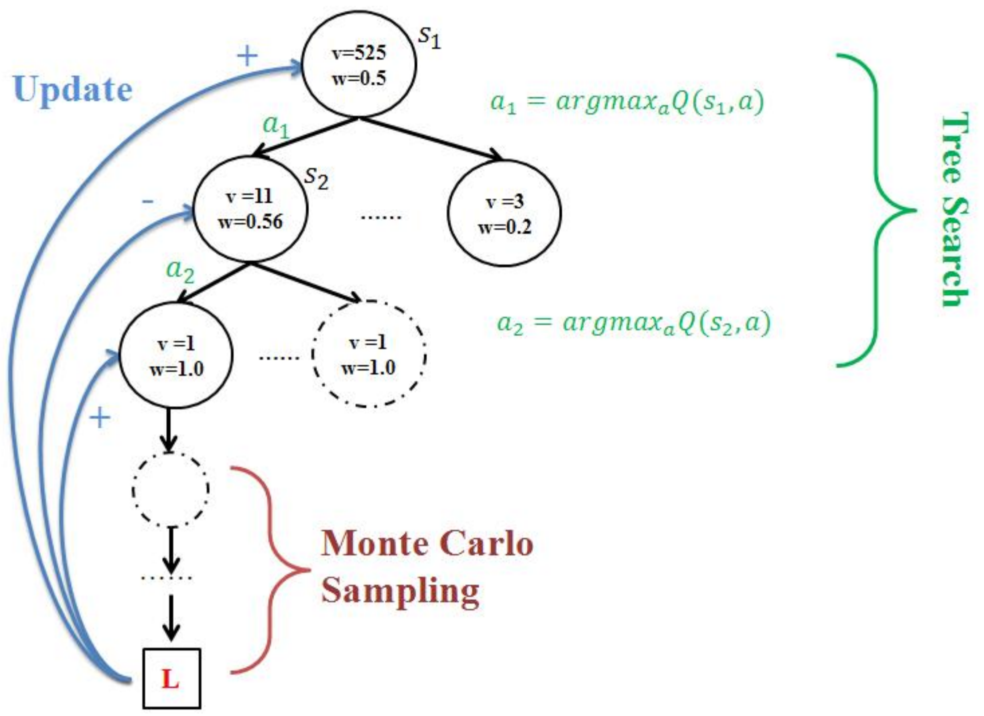


Figure 3.1: Monte Carlo Tree Search. Tree search is done with the current tree. Then MC sampling is performed from the leaf node of tree. Finally we expand the leaf node and update the tree. Nodes with bold lines are expanded nodes within the tree. Dashed nodes are possible nodes that have not been expanded. Square node represents the terminal state, or the end game board for computer Go. Since 2007, MCTS have been dominant in the computer Go literature. Its main strength comes from the statistical power of the Monte Carlo sampling process.

mean value of each state s and action a pair from previous samples. The tree search is basically a greedy search which picks the child node with the best Q-value defined.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} I_i(s,a) z_i \tag{3.1}$$

$N(s)$: the number of visits to the node s.

$N(s,a)$: the number of visits to the node which can be reached by taking action a from node s.

z_i : the evaluation of the terminal state of ith sample.

$I_i(s,a)$: an indicator function which is equal to 1 if the node (s,a) was visited in the ith sample and 0 otherwise.

We are going to use this Q-value as a default Monte Carlo (MC) estimate for the rest of this thesis.

2. **Expanding the tree.** On each iteration, we expand the search tree from the leaf node found in the tree search phase. The leaf node gets expanded with all possible actions that can be taken from that leaf node.

   There are various ways to initialize this new nodes. As a default we can have 0 visits and optimal Q-value to ensure initial exploration. Other more elaborate initialization methods can be found in [Gelly and Silver 2011].

   When the memory is an issue, we can have additional constraint to the expanding process. A common method is to prohibit the expansion until the node gets visited up to certain number of times [Gelly and Silver 2011].

3. **Random Sampling.** In general, a sample in MCTS for computer Go means the end game board and its evaluation as a win or loss. This end game board is obtained by series of randomly generated moves from an initial board. The fraction of samples which are evaluated as a win represents the strength of taking the action that have led to the initial board where the sampling has started.

   This part of MCTS is usually identical throughout its variants. The main idea is to generate the random sample which only follows a default policy. A default policy guides random sampling by making sure the samples does not choose actions which are obviously bad. However, the strength of this default policy determines large part of how strong the resulting MCTS will be.

4. **Tree Update.** Updating the tree totally depends on how we define the Q-value, or the definition of best child node. Based on those definitions, we update the required information to the corresponding parts of the tree for next iteration.

   The only concern for the update process is its efficiency. Most of MCTS's efficiency comes from its constant time update on each iteration. On creating a variant of MCTS the complexity of this update process should be considered with care.

From the following, for convenience, the term *Monte Carlo (MC) estimate* will refer to MCTS with (3.1) as its Q-value.

A node in MCTS generally can be represented in two notations when the actions are deterministic. We can use a single state name, s or a state-action pair (s,a). These are identical to each other when the actions are deterministic. And each state can be represented in either way.

3.1.2 UCB1

Upper Confidence Bound 1 (UCB1) is an allocation strategy for the *multiarmed bandit problem* [Auer et al. 2002]. A multiarmed bandit problem is a simple learning problem with n-arms. Each arm represents a possible action that the player can choose from in each iteration. The player pulls one of the n arms and gets reward from it. Each arms's reward is not previously known to the player and the main question is to figure out the best arm to pull. An allocation strategy is a method to choose next arm to pull based on the previously pulled arms and obtained rewards to minimise regret. Regret simply means the amount of reward lost by not pulling the optimal arm. Regret in the multiarmed bandit problem is defined as,

$$\mu^* n - \sum_{j=1}^{K} \mu_j E[T_j(n)]$$

μ^* is the highest reward expectation among arms, n is the total number of pulls made up to current time step, μ_i is the reward expectation of arm i, E stands for expectation and $T_j(n)$ stands for the number of times arm j is pulled in the first n pulls.

UCB1 has a logarithmic upper bound on regret with respect to the total number of pulls made up to current time step. On each pull, UCB1 chooses an arm which maximizes following formula,

$$\bar{x}_j + \sqrt{\frac{2\ln n}{n_j}} \tag{3.2}$$

$\bar{x}_j$ is the average of rewards from arm j, n_j is the number of times arm j has been pulled, and n is the total number of pulls made up to current time step. With UCB1, we get $O(\ln n)$ as an upper bound on regret, shown in (3.3).

$$O(\ln n) = O\left(\left[8 \sum_{i:\mu_i<\mu^*} (\frac{\ln n}{\Delta_i})\right] + (1 + \frac{\pi^2}{3})(\sum_{j=1}^{K} \Delta_j)\right) \tag{3.3}$$

K is the number of arms, μ_i is the reward expectation of arm i, μ^* is the highest reward expectation among arms, and $\Delta_i = \mu^* - \mu_i$. Note that in (3.3), $\left[8 \sum_{i:\mu_i<\mu^*} (\frac{\ln n}{\Delta_i})\right]$ shows that regret in UCB1 scales linearly with the number of arms. This is the case if we assume the arms are independent.

UCB1 provides an efficient scheme for choosing arms with reasonable regret over time on bandit problems. By using additional UCB1 formula on each node of search tree, we can expect each node to have logarithmic upper bound on the tree search. And by applying this to tree search part of MCTS in computer Go, we can expect each node to have logarithmic upper bound on the regret. In the sense of computer Go this means each node, or move, estimation has certain guarantees which only depends on the number of visits made to that node. With large number of samples, this guarantee in regret helps to solve the exploration-exploitation problem in MCTS.

When applying UCB1 to MCTS computer Go, each arm generally stands for each legal move from current board of the node. And regret generally stands for the amount of difference in expected number of win from the best move and the number of samples marked as a win both within current number of samples.

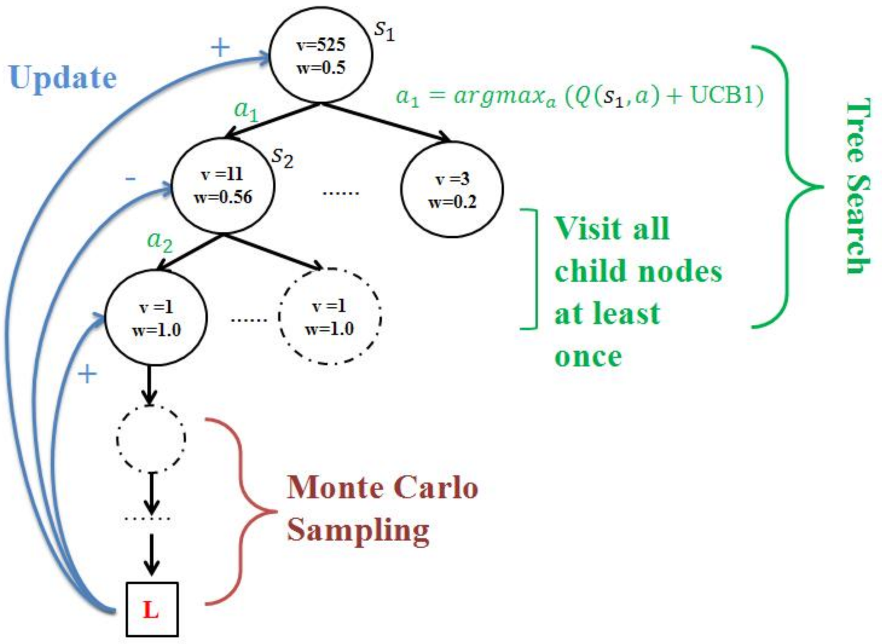


Figure 3.2: Overview of UCT in MCTS. The only additional part from plain MCTS in Figure 3.1 is the UCB1 term used in selecting the next child node (3.4). UCT requires to visit all child nodes at least once before further expanding the tree. This is done with optimistic initialization. With UCB1 term in MCTS, each node in the tree has the logarithmic upper bound in the regret. Regret in MCTS for Go can be described as the difference between the number of wins in two measures. One is the expected number of wins if the best child node is selected on all the visits to the node up to current time step. The other one is the number of wins in the gathered samples up to current time step. UCB1 term provides a way to balance between exploration and explotation in MCTS.

3.1.3 Upper Confidence Tree Search (UCT)

UCT is a tree search algorithm which uses UCB1 formula for choosing the child node [Kocsis and Szepesvári 2006]. Calculation of UCB1 can be done efficiently with constant time for each node. Also, the logarithmic upper bound from UCB1 provides the same upper bound to the each nodes within the tree. In the computer Go literature, UCT usually means a variant of MCTS which uses additional UCB1 term for defining the Q-value in the tree search phase.

UCT has shown to be very effective for computer Go [Gelly et al. 2006]. It is still the dominant method. However, with a smaller number of samples, having UCT does not have much effect on the performance when compared to the MCTS without UCT [Gelly and Silver 2011]. This is particularly the case when we are using the RAVE estimate.

When there are not enough number of samples, the effectiveness of UCT diminishes. The main strength of UCT comes from the logarithmic upper bound from UCB1. However, we can not gather enough samples for most of nodes in the game tree. The main reasons can be listed as follows,

- **Small number of samples.** When the overall number of samples is low, even the root node of the tree can not benefit much from UCT.
- **Deep parts of the tree.** In deeper parts of the tree, most of the nodes will be visited only once. And the upper bound will not help much.
- **Large branching factor.** This is one of the main characteristics of Go. Most of the empty positions on the board is a candidate of next position to place a stone. And there are 19x19 positions at the start of the game for 19x19 board. Also, this branching factor reduces only by 1 for each move on most cases. By this high branching factor, which does not reduces fast, most of the nodes will not be visited more than once even at the shallow part of the tree.

There are two slightly different consequences of having a small number of samples in UCT.

- The upper bound on each node will be large and it will not have much meaning.
- Each sub-tree will not have enough information. The sub-tree might not be able to capture the obvious consequence that can be observed in deeper tree.

In [Gelly and Silver 2011] UCT actually performs worse than pure MC method when using 3000 samples per move. This is also the trend observed on a sandbox implementation of the Go engine, GoYui. Still, [Gelly and Silver 2011] also suspects that when the number of samples increase, UCT will have positive effect.

Figure 3.3 shows number of visit graphs for UCT with various c value. When tested on GoYui, UCT does not help when the number of samples is small.

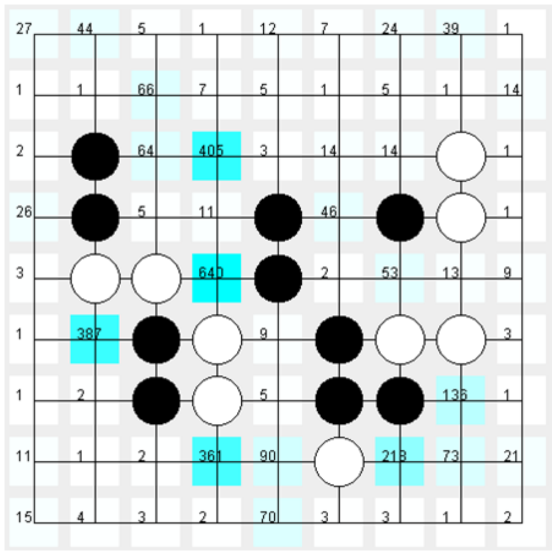
(3.4) shows a simple case where UCT is combined into MCTS. One can simply add the $c\sqrt{\frac{2\ln N(s)}{N(s,a)}}$ term to get the logarithmic upper bound on each child node selections.

$$Q_{UCT}(s,a) = Q(s,a) + \underbrace{c\sqrt{\frac{2\ln N(s)}{N(s,a)}}}_{UCB1} \tag{3.4}$$

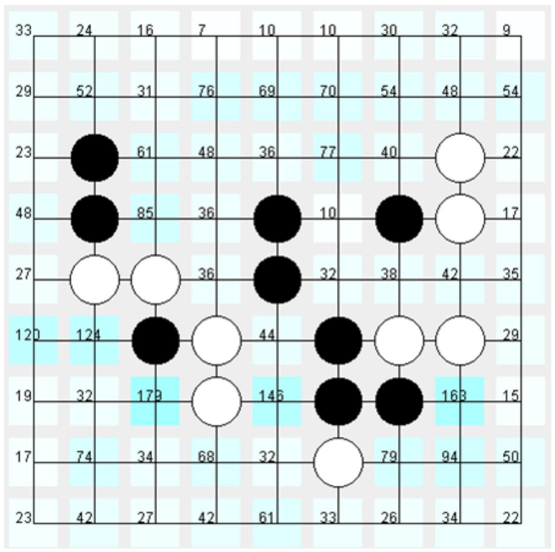
The constant c in (3.4) determines the balance between exploration and exploitation. Larger c gives more weight to the exploration part, and $Q_{UCT}(s,a)$ value will be higher for less visited nodes. Smaller c will result in more exploitation.

3.1.4 Rapid Action Value Estimation (RAVE)

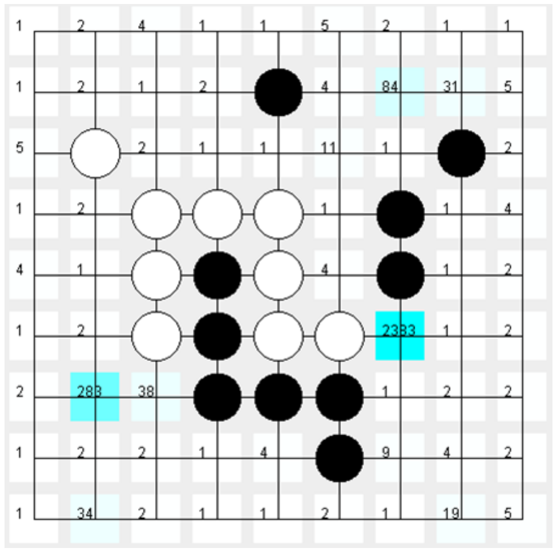
RAVE is a variant of MCTS which uses the all-moves-as-first(AMAF) heuristics [Gelly and Silver 2011]. AMAF heuristic treats same actions within the same search tree as



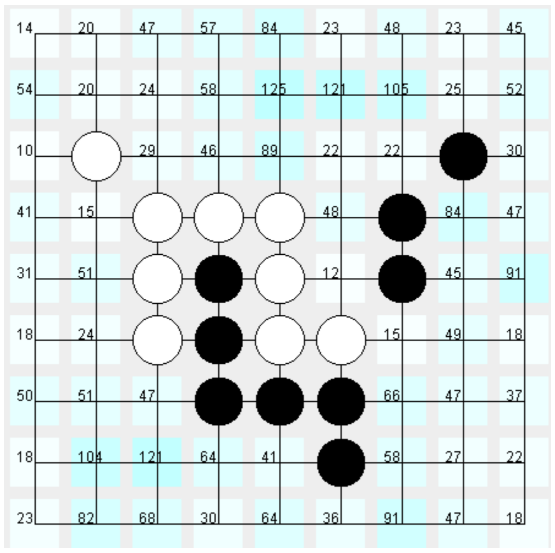
(a) UCT-RAVE, c=0.1



(b) UCT-RAVE, c=0.4



(c) MC-RAVE, c=0.0



(d) UCT-RAVE, c=0.4

Figure 3.3: Graph shows the number of visits vs. the weight c placed on the UCB1 term in equation (3.4). Each number represents the number of visits made to each positions from the root node. High c generally results in a shallow and wide tree. Small c generally results in a very deep tree. For UCT-RAVE, c=0.1 performs better than c=0.4 (tested on GoYui). The main advantage of smaller *c* comes from having more accurate tree. [Gelly and Silver 2011] claims that c=0.0 performs better on small number of samples. Basically, when we have small number of samples, small c can be understood as an intention to avoid the bad moves rather than to look for the best move.

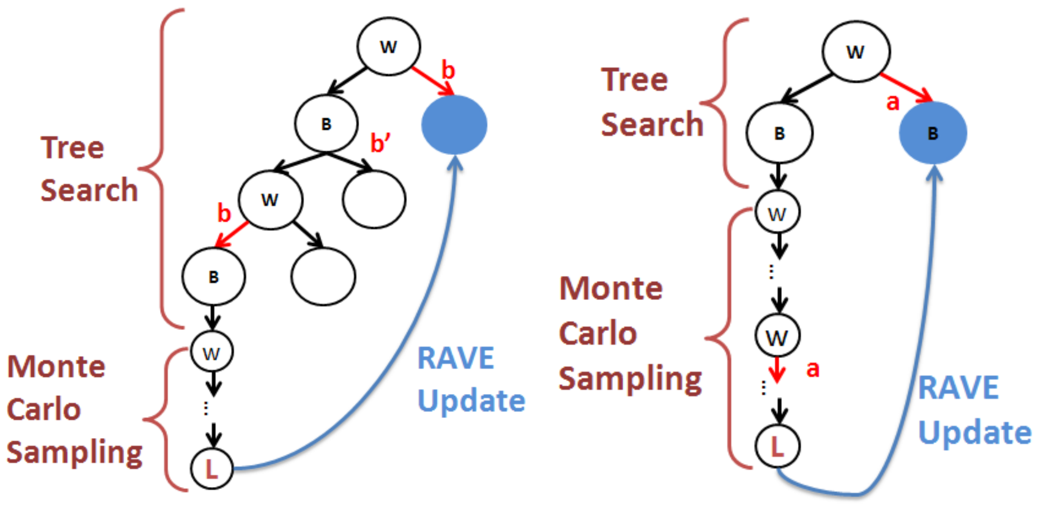


Figure 3.4: RAVE update scheme 1. Black 'w', 'b' letters within each node stands for 'white's, or black's, turn to move'. On each sampling iteration, unlike MC estimation, RAVE also updates actions that was not taken from the current state. By updating unvisited actions, RAVE provides much stronger estimates with small number of samples. Selection of actions for this additional update is done with AMAF heuristic. Action b of root node is not visited in current iteration. But it gets updated as it was encountered in current iteration. Note that action b' will not be updated as it was not encountered in current iteration. Similarly action a of root node will be updated as it was encountered in current iteration. This 'encountered' includes the tree search and the Monte Carlo sampling phase.

identical actions regardless of their positions in the search tree. By being assumed as identical actions, each action is also expected to have the same, or similar, outcomes. In computer Go, this means the same action across the whole search tree is expected to give similar win rates.

RAVE can be understood as a MCTS with (3.5) as its Q-value:

$$\tilde{Q}(s,a) = \frac{1}{\tilde{N}(s,a)} \sum_{i=1}^{N(s)} \tilde{I}_i(s,a) z_i \tag{3.5}$$

$$\tilde{N}(s,a) = \sum_{i=1}^{N(s)} \tilde{I}_i(s,a)$$

The only difference between RAVE and the plain MC estimate, (3.1) is the indicator function, $\tilde{I}$.

- Indicator function I in MC: equal to '1' only when the state action pair was selected during the corresponding sampling process.
- Indicator function $\tilde{I}$ in RAVE: equal to '1' only when the action was taken during the corresponding sampling process in one of descendant states or in

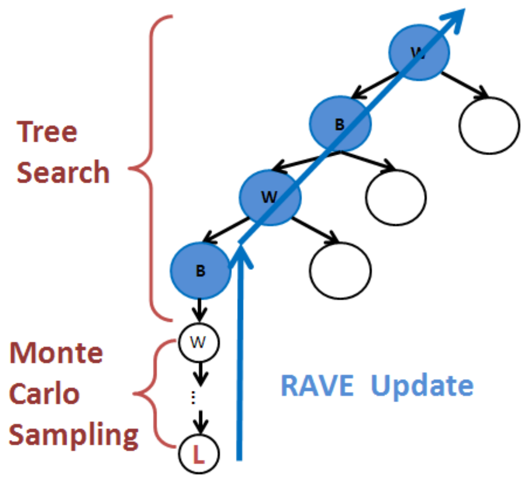


Figure 3.5: RAVE update scheme 2. Same as MC estimation, RAVE also updates actions that was visited in current iteration. RAVE is essentially a MC estimation with additional updates illustrated in Figure 3.4.

the random sampling process.

AMAF heuristic assumes two identical actions on different nodes to have similar effects when considered from their common ancestor. In other words, the AMAF heuristic focuses on the set of actions that have been taken without considering their specific order that have been taken. The AMAF heuristic fits well with incremental games, like Go [Gelly and Silver 2011].

3.2 Combining Models for Monte Carlo Tree Search

According to [Gelly and Silver 2011] RAVE itself does not necessarily produce better results than MC estimate. In fact, as we gather more samples, the direct estimation based on the plain samples represents the true value more accurately. And this makes the Q-value of MC estimate to be more accurate than any other estimates (Q-value stands for the value of taking an action in a state). Still, RAVE is effective when there are less number of samples. For these reasons, RAVE is combined with MC. (3.6) shows a possible framework. The $\beta(s,a)$ function is the combining method which depends on the each state-action pair.

$$Q_\star(s,a) = (1-\beta(s,a))Q(s,a) + \beta(s,a)\tilde{Q}(s,a) \tag{3.6}$$

$Q(s,a)$: a Monte Carlo estimate, (3.1)
$\tilde{Q}(s,a)$: a RAVE estimate, (3.5)

In this section, two β functions from [Gelly and Silver 2011] will be presented. In Chapter 5, another β function with different theoretical back ground with similar performance is presented.

3.2.1 Hand Selected Heuristic

In [Gelly and Silver 2011], a simple heuristic for combining RAVE and MC is presented as (3.7):

$$\beta(s,a) = \sqrt{\frac{k}{3N(s)+k}} \tag{3.7}$$

k : a free variable. k is the number of samples that makes $\beta(s,a) = 0.5$.
$N(s)$: number of visits to node s.

The main rationale given to this heuristic is the expectation that RAVE will perform better on smaller number of samples. And that MC will be closer to the true value as the number of samples increases.

3.2.2 Minimum Mean Squared Error

Another combining method is given in [Gelly and Silver 2011] with better theoretical background which shows better performance than the hand selected heuristic. They call it 'Minimum MSE schedule', and present it as (3.8):

$$\beta(s,a) = \frac{\tilde{n}}{n + \tilde{n} + 4n\tilde{n}\tilde{b}^2} \tag{3.8}$$

n : number of visits to node (s,a)
$\tilde{n}$: number RAVE visits to node (s,a)
$\tilde{b}$: a free variable. Defined as a bias of RAVE estimate

There is a free variable $\tilde{b}$ in this β function, which is defined as the bias of RAVE model (3.9). π stands for an assumption which tells that the samples are drawn from a constant policy.

$$\tilde{b} = \tilde{Q}^{\pi}(s,a) - Q^{\pi}(s,a) = \tilde{B}(s,a) \tag{3.9}$$

3.2.3 Mixing in UCT

UCT can be thought as an external term which balances the exploration. Usually it is added to the combined models. The only concern is the constant c. Empirically, it is better to have small exploration if the best known action can not be investigated to a sufficient level. This simply means smaller 'c' when the number of sample is low. (3.10) shows a typical way of combining UCT. Note that constant $'c'$ is not associated with β function.

$$Q_{UCT*}(s,a) = (1-\beta(s,a))Q(s,a) + \beta(s,a)\tilde{Q}(s,a) + c\sqrt{\frac{2\ln N(s)}{N(s,a)}} \tag{3.10}$$

3.3 Implementing and Testing a Go engine

In this thesis, a Go engine with basic default policy is implemented to test various ideas. Figure 3.6 gives a basic framework of using MCTS for a Go engine. This section addresses some of the issues in implementing a Go engine.

3.3.1 Default Policy

The default policy has huge impact on the overall performance of a Go engine. The main purpose of default policy is to avoid obviously bad moves and to take obviously good moves during the random play out (sampling) process. Following is a list of commonly used heuristics for building a default policy:

- **Don't fill in the eyes.** This heuristic is to avoid obviously bad moves. In the game of Go, there is a problem called 'life and death'. This is to predict whether a set of stones will be captured by the opponent or not. Usually, eye is a territory that is surrounded by one player which does not have danger of being occupied by the opponent. By this property, eye assures the adjacent chains to be alive. Filling in the eye simply is a obviously bad move in general. But not all eyes are secure positions. Some eyes should be filled in to avoid certain threats. As classifying this is essentially solving the life and death problem, simpler rule can be applied in allowing eye filling; we look at the diagonal position of an eye and allowing the eye filling when two or more of the diagonal positions are empty or occupied by the opponent.

- **Capture moves.** This heuristic is to perform obviously good moves first. In general, MC sampling for Go does not include planning for neither players, it is random. As this is the case, performing the capture moves first may help the following playout of the board to result in more reasonable end game.

- **3x3 pattern.** This heuristic is introduced in [Gelly et al. 2006] and is to take better moves first. If the pattern and the board position is matched, the random playout will take the corresponding move. This pattern only looks at the 3x3 positions around an empty position. These patterns are basically hand crafted patterns which is human knowledge based. These patterns do help improving the performance of Go engine (it almost doubles the winning ratio according to the result given in [Gelly et al. 2006]).

The default policy is a quick check list which makes the random playout to create a reasonable end game from current board without taking too much overhead.

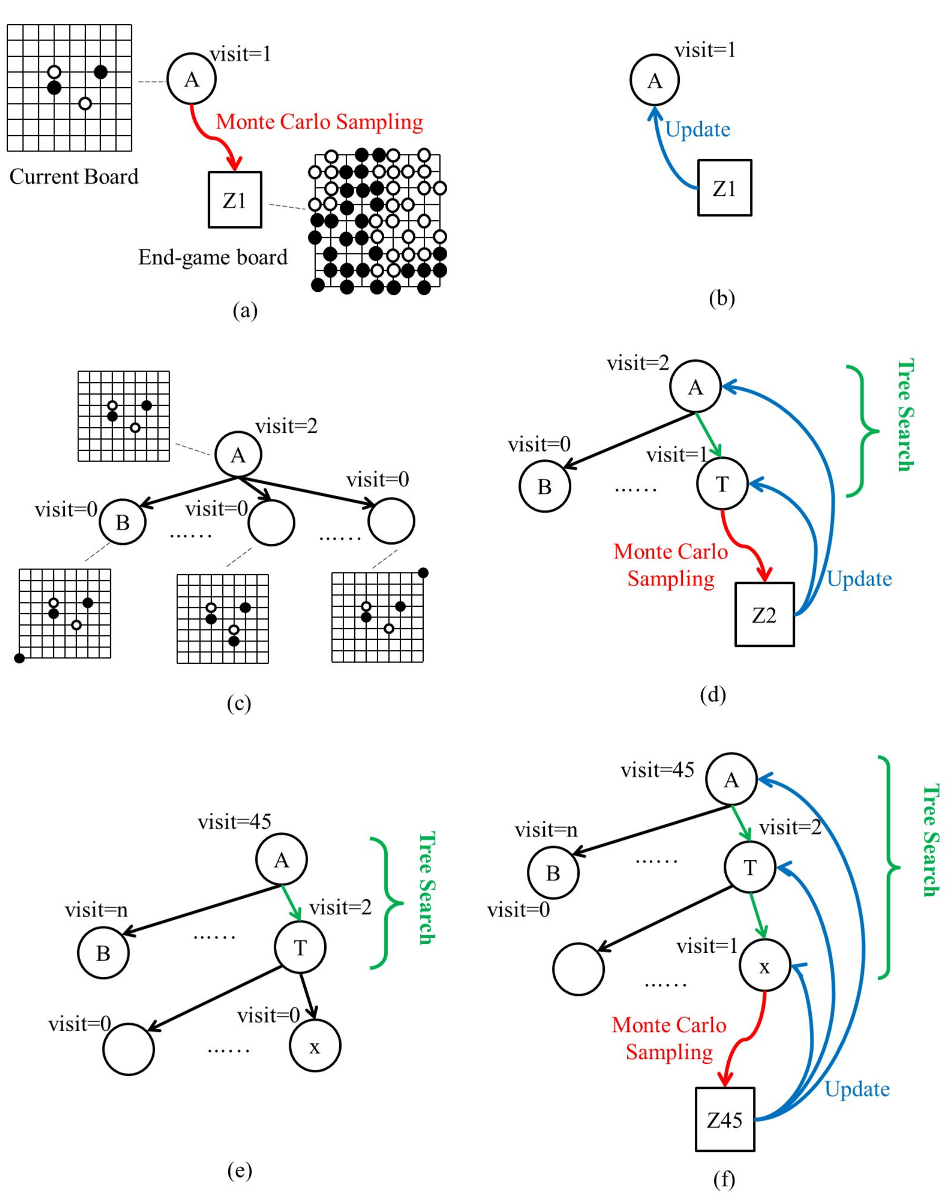


Figure 3.6: Overview of MCTS for Go. Go fits very well within MCTS. (a) Each node represents a board position. On the leaf node we perform Monte Carlo Sampling. This sampling process proceeds until the endgame board. End game board is relatively easy to evaluate in Go. (b) After each Monte Carlo Sampling, we update the tree. Update process depends on the definition of best child node and provided necessary information for next iteration. (c) On second visit, or on n-th visit depending on how expanding is defined, to the node, the node is expanded with all legal moves. (d) When the node is not a leaf node, we choose the child node based on the defined Q-value. After maximum number of iteration is reached, we pick the node with most visits as return it as the move to perform.

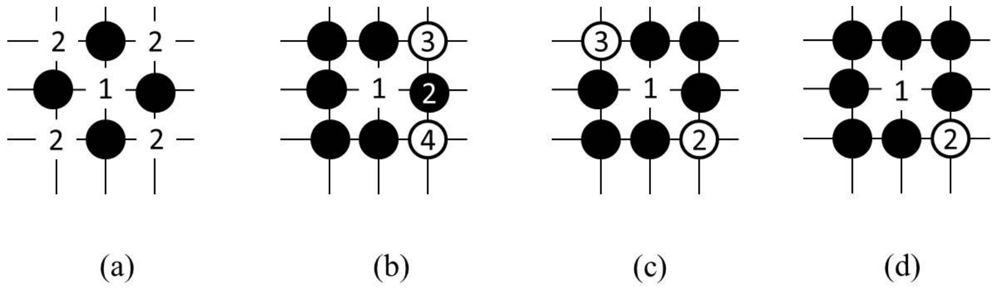


Figure 3.7: Eye. Filling an eye does not provides any advantage on the board, and can be ruled out from the set of legal moves for computer Go. (a) Positions like 1 where all four adjacent positions are filled with one color can be considered to be a candidate of an eye. (b), (c) when more than two diagonal positions are empty there is a possibility of position 1 being playable for the opponent. In these cases filling in position 1 can actually give black advantages on the board. And it is not considered to be an eye. (d) When there are at most one nonblack position around position 1, black does not benefit from playing on position 1. And it is considered to be an eye.

Generally in computer Go, the strength of the engine scales with the number of samples, or time it takes to generate a move. So there is a trade-off between how many samples one can get and how accurate each sample can be.

One thing to note is that researchers have failed to encode the knowledge of professional Go players for a few decades. This tells us how hard it is to generate an accurate estimation of each move in a Go board. Without having an efficient and correct estimation of each move, the rule of thumb will be to use as many samples as possible with a basic and fast default policy.

3.3.2 Data Structure: Representing the Go board

One of the main requirements for a Go engine is to have fast sampling speed. For a typical Go engine with Monte Carlo tree search, its performance rises with the number of samples. And having a large number of samples in a limited time is a crucial factor for a Go engine.

The game of Go evolves around the chains and their liberty points. A simple way to handle these is to store the chains on the current board as in Figure 3.8. Then each move can be handled just by looking at neighbouring chains. By this each capture can be detected and processed efficiently.

3.3.3 Testing and evaluating a Go engine

There are several ways to evaluate or test a Go engine.

- **Human opponent.** Playing against human provides ultimate estimation for an algorithm. But human opponents are noisy and expensive. Still, defeating a human grand master is most likely the ultimate goal of computer Go research

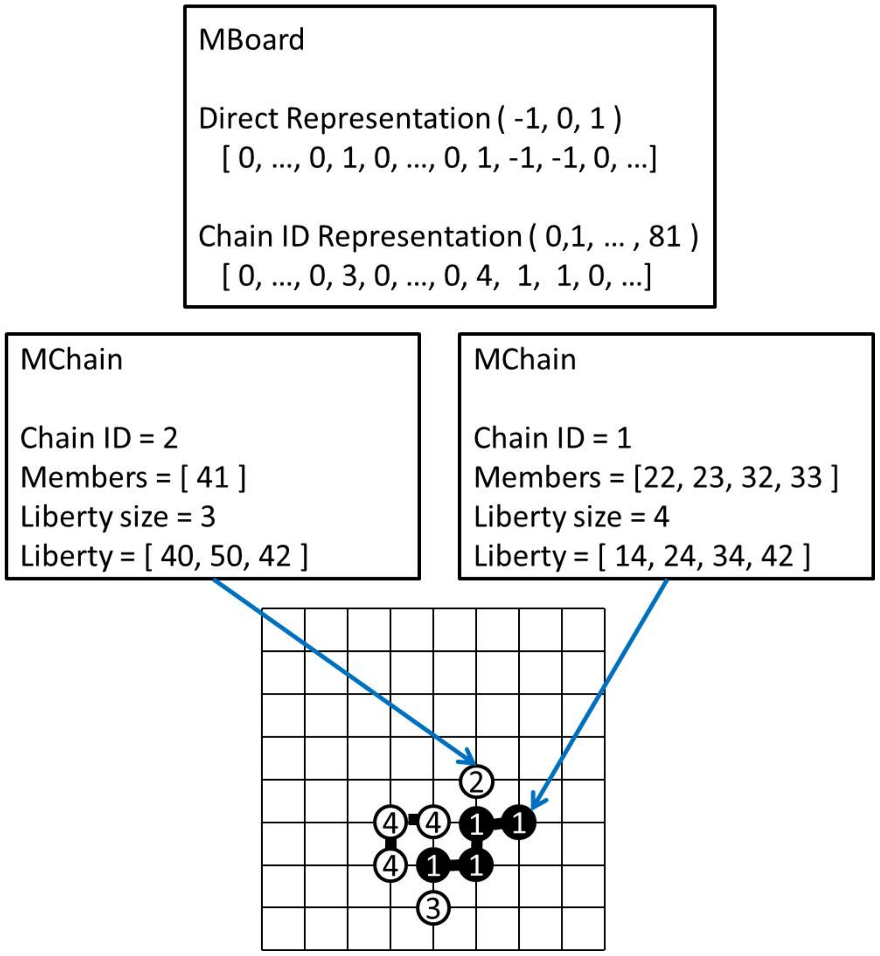


Figure 3.8: Board representation used in GoYui. GoYui acquires faster sampling through efficient board representation. For this purpose, dual representation is used to represent the board. Each stone is represented in the stone board and in the chain ID board. Chain ID board enables efficient capture detection by storing the liberty data of each chain.

as it was for computer chess. KGS Go server[1] is a common online medium to test a Go engine against human opponents.

- **Computer Go Server (CGOS).** CGOS is a common way to test a Go engine against another Go engine. Most of the state of the art Go engines compete in CGOS. Elo in CGOS is one of major way to estimate the power of a Go engine.

- **GNU Go.** The simplest way to test a Go engine is to run against GNU Go[2]. GNU Go is an open source Go engine which was one of the best engines around 2002. GNU Go can be thought as the last engine to have huge impact on the computer Go literature before Monte Carlo tree search became popular.

[1]http://www.gokgs.com/
[2]http://www.gnu.org/software/gnugo/

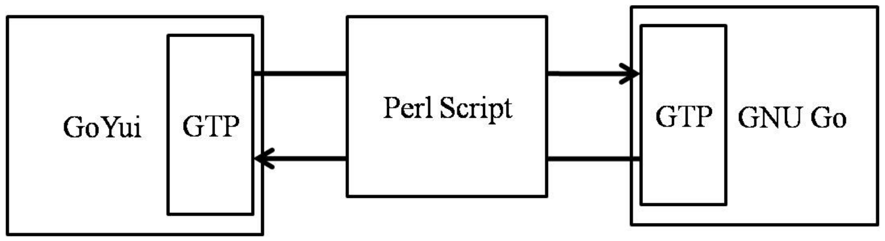


Figure 3.9: Test setting for GoYui. Typical way of testing a Go engine is to run against GNU Go. GNU Go supports Go Text Protocol (GTP). By implementing GTP into the Go engine, it can be easily run against GNU Go. GoYui is tested against GNU Go using this protocol.

GNU Go supports Go Text Protocol(GTP) and comes with several basic scripts to manipulate it.

3.3.4 GoYui

GoYui[3] is a Java based open source Go engine generated with basic default policies to test various ideas on MCTS for computer Go. The strength of GoYui is not close to the state of the art Go engines. But it provided enough strength to test ideas presented in this thesis.

GoYui is composed of around 3000 lines of Java code. 1500 lines are used for MCTS, RAVE, UCT, LWR, and BMA implementations. 500 lines are used for basic a GTP interface and some simple visualisations. 1000 lines are used for Go data structure and basic game engine for faster simulation.

GoYui achieves around 22% win ratio against GNU Go 3.8 level 10 with 3000 samples per move on 9x9 board. GoYui can perform around 10,000 random play outs per second on a 2.0 GHz commodity machine for a 9x9 board.

GoYui is played against GNU Go version 3.8 level 10 on a 9x9 board throughout this thesis. Here are some of the basic settings used for GoYui.

- **Number of samples per move.** 3000 samples are used for each move.
- **'c' constant for UCB1 term in equation (3.4).** c=0.1 is used. This value is obtained empirically.
- **Optimistic initialisation.** A node gets initialized with a optimal value, the value for win. To maintain randomness, small noise is added as a tie-breaker.
- **Slow expansion.** A node does not get expanded until its 6th visit.
- **Superko.** GoYui stores previous boards and tests for the superko on each move.
- **Early termination.** A PASS move is used when final best move from search shows less than 10% win rate expectation.

[3]http://code.google.com/p/go-yui/

Chapter 4

Update Models for Monte Carlo Tree Search (MCTS)

One of the basic observation of MCTS is that the arms, or the child nodes of each node, can be dependent of each other. Each move may have certain similarity with each other. This can be rather obvious in the MCTS for Go for the following reasons:

- The board can result in a same configuration even with having different intermediate states. Though this is rather unusual, it can be very possible in some situations. Figure 4.1 shows how two different actions can result in a same state in the game of Go.

- Local similarity and effect of positions. It is certain that each position in a Go board has their own unique consequences. This is one of the main reasons for Go's high complexity. Each move will have huge effect not only on immediate board, but also on far future board. Still, when we look at the local part of the board, there is local symmetry among the positions. Also, the effect of a move on similar positions will have rather similar consequences.

Still, most work has assumed independence of the arms. This tendency can be seen from one of recent paper as well [Gelly and Silver 2011]. As mentioned in Chapter 3, RAVE and MC estimates have been the method of choice for selecting the best child node on current computer Go literature. RAVE does handle the node dependency, but it is not well described in the current literature.

In this Chapter, some update methods are investigated as alternative update models for the MCTS. These methods have specific focus on handling node, or arm, dependencies. Section 4.1 discusses Bayesian update as an update model for MCTS. Section 4.2 tests Locally Weighted Regression (LWR) as a heuristic for updating the tree for MCTS. Along with RAVE, some similarity measures, or kernels, for the update process is suggested and tested with LWR framework in section 4.3. Finally, in section 4.4, RAVE heuristic is described within the framework of LWR with some description for its superior performance.

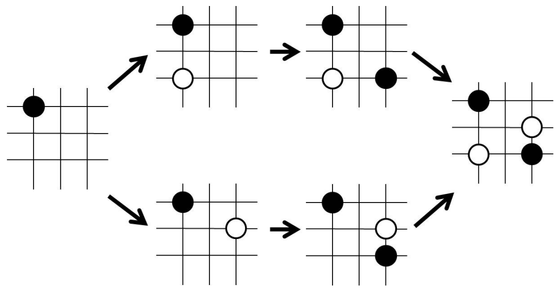
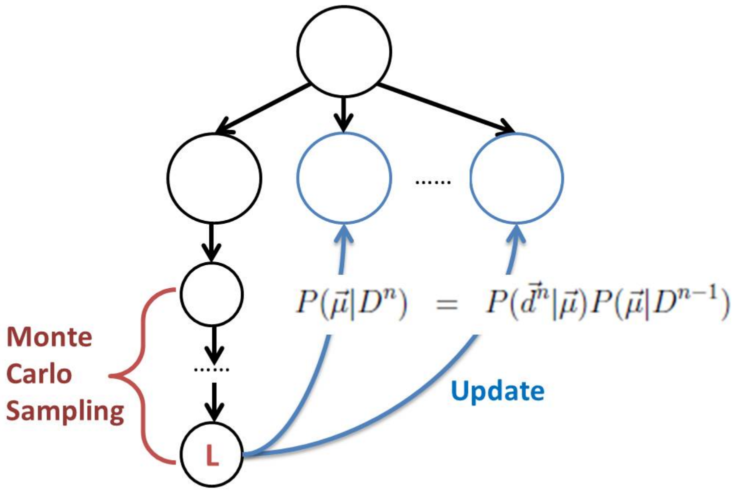


Figure 4.1: Two actions resulting in a same state. Recall that each node in the MCTS for Go is a whole board. This is one of the cases which tells us that two different nodes in the search tree can have certain similarities with each other.



$P(\vec{\mu}|D^n) = P(\vec{d}^n|\vec{\mu})P(\vec{\mu}|D^{n-1})$
Monte Carlo Sampling
Update
L

Figure 4.2: Basic framework of Bayesian update process for MCTS. Each node is updated based on the likelihood of the data observed.

4.1 Bayesian Update with Gaussian Prior

In a search tree for the game of Go, we can assume the child nodes are correlated with each other. Even when each child nodes are correlated with each other, each sample data can be independent of each other. From these two assumptions, we can derive a Bayesian update method for the MCTS. The main feature of this method will be its use of kernels as a similarity measure for each pair of child nodes.

A Bayesian update by itself is simple a method of manipulating beliefs in certain measure and updating it with each given data. Gaussian processes provide a framework for combining kernel methods with Bayesian methods. [Rasmussen and Williams 2006]

A Bayesian update with Gaussian prior is a way to incorporate the similarity between the child nodes with a kernel matrix. It will update the child nodes based on a kernel, which will adapt to the actual sample data obtained. The basic framework of this process applied to MCTS is illustrated on Figure 4.2.

4.1.1 A derivation of the process

A mathematical derivation of a possible Bayesian update process with Gaussian prior for the game of Go is worked out in this section.

Let's assume we have k armed bandit with prior mean distribution

$$P(\vec{\mu}) = N(\vec{\mu}, \vec{0}, K), \tag{4.1}$$

where K is a predefined initial Kernel. For each data, d^i from i^{th} iteration, we can update $P(\vec{\mu})$ with Bayesian method,

$$P(\vec{\mu}|D^i) \quad = \quad P(d^i|\vec{\mu})P(\vec{\mu}|D^{i-1}) \tag{4.2}$$

Each data in a play-out for Go can be represented as a win or lose for selecting a certain intersection. This means we only need one free variable, the μ for selected arm, for representing the likelihood.

$$\begin{aligned} P(d^i|\vec{\mu}) &= \int_{s(i)} N(\vec{\mu}, \vec{v}_i, C_L) d\vec{v}_i && (4.3) \\ &= N(\mu_{s(i)}, d^i, \sigma^2) && (4.4) \\ &= N(d^i, \mu_{s(i)}, \sigma^2) && (4.5) \end{aligned}$$

$$\vec{v}_i = \begin{bmatrix} 0 \\ \vdots \\ d^i \\ \vdots \\ 0 \end{bmatrix}$$

$$C_L = \begin{bmatrix} \sigma^2 & 0 & \dots & 0 \\ 0 & \sigma^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma^2 \end{bmatrix}$$

$s(i)$ = index of the arm selected in i^{th} iteration

$\vec{v}_i$ = observed data from i^{th} iteration, only one element can be nonzero

From (1), (2), and (4) we get

$$\begin{aligned} P(\vec{\mu}|D^n) &= N(d^n, \mu_{s(n)}, \sigma^2) \dots N(d^1, \mu_{s(1)}, \sigma^2) N(\vec{\mu}, \vec{0}, K) && (4.6) \\ &= N(\vec{\mu}, \vec{a}, C^n) N(\vec{\mu}, \vec{0}, K) && (4.7) \end{aligned}$$

$$\vec{a} = \begin{bmatrix} \frac{\sum_{i=1}^{n} d^i Arm(d^i,1)}{T_1(n)} \\ \frac{\sum_{i=1}^{n} d^i Arm(d^i,2)}{T_2(n)} \\ \vdots \\ \frac{\sum_{i=1}^{n} d^i Arm(d^i,k)}{T_k(n)} \end{bmatrix}$$

$$C^n = \begin{bmatrix} \frac{\sigma^2}{T_1(n)} & 0 & \dots & 0 \\ 0 & \frac{\sigma^2}{T_2(n)} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\sigma^2}{T_k(n)} \end{bmatrix}$$

$$T_i(n) = \sum_{j=1}^{n} Arm(d^j, i)$$

$$Arm(d^j, i) = \begin{cases} 1 & \text{if j-th data is drawn from pulling i-th arm} \\ 0 & \text{otherwise} \end{cases}$$

The initialization part of UCB1 involves visiting each arm at least once. This means any unvisited arms will be visited before visiting other arms twice. Having $T_i(n) = 0$ causes a problem only when we need to take the inverse of $T_i(n)$. But we take this inverse only after the initialization is done. Because during the first k plays, we need to visit each arms at least once. (This is for using UCB1) So, we don't have to worry about the case $T_i(n) = 0$.

To take this in account, let's assume we do the initialization first. Then,

$$P(\vec{\mu}|D^{n+k}) = N(\vec{\mu}, \vec{a}, {C_I}^{n+k}) N(\vec{\mu}, \vec{0}, K) \quad (4.8)$$

$${C_I}^{n+k} = \begin{bmatrix} \frac{\sigma^2}{R_1(n)+1} & 0 & \dots & 0 \\ 0 & \frac{\sigma^2}{R_2(n)+1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\sigma^2}{R_k(n)+1} \end{bmatrix}$$

$$R_i(n) = \sum_{j=k+1}^{n} Arm(\vec{d^j}, i)$$

Let

$$P(\vec{\mu}|D^{n+k}) = N(\vec{\mu}, \vec{m_n}, \Sigma_n) \quad (4.9)$$

When $n = 0$,

$$\begin{aligned} P(\vec{\mu}|D^k) &= N(\vec{\mu}, \vec{a}_0, {C_I}^k) N(\vec{\mu}, \vec{0}, K) && (4.10)\\ \Sigma_0 &= ((C_I^k)^{-1} + K^{-1})^{-1} && (4.11)\\ \vec{m}_0 &= \Sigma_0((C_I^k)^{-1}\vec{a}_0 + K^{-1}\vec{0}) = \Sigma_0((C_I^k)^{-1}\vec{a}_0) && (4.12)\end{aligned}$$

For $n = l$ $(l > k)$,

$$\begin{aligned} P(\vec{\mu}|D^{k+l}) &= N(\vec{\mu}, \vec{a}_l, {C_I}^{k+l}) N(\vec{\mu}, \vec{0}, K) && (4.13)\\ &= N(\vec{\mu}, \vec{v}_l, C_L) N(\vec{\mu}, \vec{a}_{l-1}, {C_I}^{k+l-1}) N(\vec{\mu}, \vec{0}, K) && (4.14)\\ &= N(v_{s(l)}, \mu_{s(l)}, \sigma^2) N(\vec{\mu}, \vec{a}_{l-1}, {C_I}^{k+l-1}) N(\vec{\mu}, \vec{0}, K) && (4.15)\end{aligned}$$

Now if we look at how $({C_I}^{k+l})^{-1}$ changes from $({C_I}^{k+l-1})^{-1}$,

$$({C_I}^{k+l})^{-1} = ({C_I}^{k+l-1})^{-1} + Q^{s(l)} \qquad (4.16)$$

$$Q^{s(l)} = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 0 & \vdots & \vdots & \vdots & 0 \\ 0 & \vdots & q & \vdots & 0 \\ 0 & \vdots & \vdots & \vdots & 0 \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

$$\begin{aligned} q &= {Q^{s(l)}}_{s(l)s(l)} && (4.17)\\ &= ({C_I}^{k+l})^{-1}_{s(l)s(l)} - ({C_I}^{k+l-1})^{-1}_{s(l)s(l)} && (4.18)\\ &= \frac{R_{s(l)}(l) + 1}{\sigma^2} - \frac{R_{s(l)}(l-1) + 1}{\sigma^2} && (4.19)\\ &= \frac{1}{\sigma^2} && (4.20)\end{aligned}$$

This is because only $(s(l), s(l))$ component of ${C_I}^{k+l-1}$ changes and ${C_I}^{k+l-1}$ is diagonal.

From (3.12),

$$\begin{aligned} \Sigma_{k+l} &= (({C_I}^{k+l})^{-1} + K^{-1})^{-1} && (4.21)\\ &= (({C_I}^{k+l-1})^{-1} + Q + K^{-1})^{-1} && (4.22)\\ &= (Q^{s(l)} + ({C_I}^{k+l-1})^{-1} + K^{-1})^{-1} && (4.23)\\ &= ((Q^{s(l)} + (\Sigma_{k+l-1})^{-1})^{-1} && (4.24)\\ Q^{s(l)} &= \vec{u}_{s(l)}\vec{v}^T_{s(l)} && (4.25)\end{aligned}$$

$$\vec{u}_{s(l)} = \vec{v}_{s(l)} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \frac{1}{\sigma} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

($s(l)$-th term is the only nonzero term.)

From the Sherman-Morrison formula for efficient rank-1 matrix inverse updates [Sherman and Morrison 1950] and for $l > k$ and $s(l) = b$, the derivation for the covariance update is

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1}u}$$

$$\begin{aligned} \Sigma_l &= ((\Sigma_{l-1})^{-1} + (Q^b)^{-1} && (4.26) \\ &= ((\Sigma_{l-1})^{-1} + u_b v_b^T)^{-1} && (4.27) \\ &= \Sigma_{l-1} - \frac{\Sigma_{l-1} u_b v_b^T \Sigma_{l-1}}{1 + v_b^T \Sigma_{l-1} u_b} && (4.28) \end{aligned}$$

As the whole Σ_{l-1} changes, the mean should be updated as follows:

$$\vec{m}_l = \Sigma_l((C_I^{l-1})^{-1}\vec{a}_0 + K^{-1}\vec{0}) = \Sigma_l((C_I^{l-1})^{-1}\vec{a}_0) \quad (4.29)$$

4.1.2 Inherent inefficiency

The main drawback of Bayesian update is its theoretical complexity. In the previous section, we derived a method with $O(N^2)$ to perform Bayesian update with Gaussian prior for each node. N is the number of child nodes. In fact, this is as efficient as it can get. We actually need to update relationship between each pair of child nodes on each iteration and this requires $O(N^2)$. Compared to this, default Monte Carlo method requires only constant time on each node. This difference makes Bayesian update method infeasible in practice (especially for go which has around $N = 300$ on 19x19 board).

4.2 Locally Weighted Regression for MCTS

In the previous section, we have briefly witnessed the intractability of Gaussian process approaches to regression in multi-armed bandit problem. For this reason, a more tractable kernel regression method is investigated in this section.

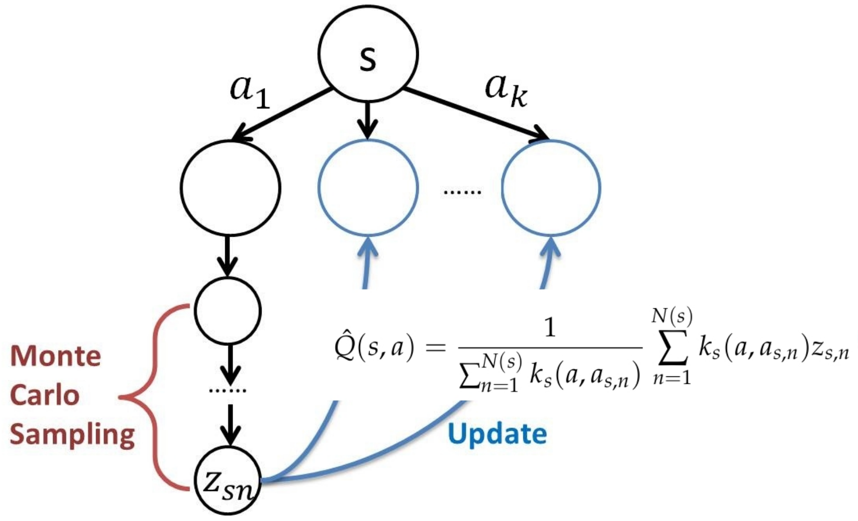

s
a_1
a_k
......
Monte Carlo Sampling
$\hat{Q}(s,a) = \frac{1}{\sum_{n=1}^{N(s)} k_s(a, a_{s,n})} \sum_{n=1}^{N(s)} k_s(a, a_{s,n}) z_{s,n}$
Update
z_{sn}

Figure 4.3: Basic framework of LWR for MCTS. Each node is updated based on the similarity with the selected node. In this figure, action a_1 is selected in the current iteration. Other siblings of a_1, for example a_k, is updated by the result of current MCTS sampling which uses a_1. This update is weighted with the similarity between the nodes. Such similarities can be represented, or approximated, by a similarity kernel. The $\hat{Q}(s,a)$ represents the LWR estimate on the state-action pair, or the node, (s,a) (4.30).

4.2.1 Locally Weighted Regression

Locally weighted regression (LWR) had been a method of choice for simple and robust regression [Cleveland and Devlin 1988]. However, Bayesian methods have better theoretical support and are much more preferred than LWR in recent days. Still, LWR is often used as an alternative when the complexity is an issue. For the game of Go, due to its high branching factor, Bayesian update with Gaussian prior is infeasible. This makes the LWR method as one of the few computationally efficient alternatives.

4.2.2 LWR as an update process for MCTS

The Basic framework of LWR as an update process of MCTS is given in Figure 4.3. Basically, we use a kernel similarity measure k and update the siblings of the selected action.

LWR can be used in MCTS to estimate the Q-value of a node based on the similarity of the node with its siblings. We will use $\hat{Q}(s,a)$ to represent such Q-value.

$\hat{Q}(s,a)$ can be represented as follows,

$$\hat{Q}(s,a) = \frac{1}{\sum_{n=1}^{N(s)} k_s(a, a_{s,n})} \sum_{n=1}^{N(s)} k_s(a, a_{s,n}) z_{s,n} \tag{4.30}$$

$$a_{s,n} : \text{the action taken on } n^{th} \text{ visit to node } s$$

$$z_{s,n} : \text{the sampling result of } n^{th} \text{ visit to node } s$$

This estimate can be mixed with MC estimate for MCTS using following $Q_\star$ as its Q-value.

$$Q_\star(s,a) = (1 - \beta(s,a))Q(s,a) + \beta(s,a)\hat{Q}(s,a) \tag{4.31}$$

The main aspect of this method comes from updating a node's siblings based on a similarity measure, or kernel. Obviously, the strength of this kernel decides the overall performance of resulting MCTS.

4.3 Kernels in LWR MCTS

Choice of the kernel drastically affects the performance of resulting LWR MCTS. In this section we will witness the extreme importance of good kernel designs in kernel-based approaches for MCTS Go.

4.3.1 Kernels

A kernel is a similarity measure. In LWR for Go, a kernel simply represents how similar two board positions are for a given board. Here are some kernels which have been tested with GoYui.

1. Manhattan distance between positions.
   (a) Assumption : Two positions that are close to each other will have similar effects on the board.
   (b) Kernel : $k_s(a_i, a_j) = (16 - |a_i.x - a_j.x| + |a_i.y - a_j.y|)/18$
2. Win ratio similarity.
   (a) Assumption : In the bandit point of view, we only consider the win rate of each arm. If we apply this view to computer Go, we can consider two nodes with similar win ratio to be similar to each other.
   (b) Kernel : $k_s(a_i, a_j) = 1.0 - |(s, a_i).winrate - (s, a_j).winrate|$
3. RAVE estimate as a similarity measure. [1]

[1]This is rather a pseudo kernel. It is different from RAVE.

Kernels	Win rate	Std. Error	Wins	Number of Games
Manhattan distance	3.4%	0.57%	34	1000
Winrate similarity	1.2%	0.34%	12	1000
RAVE based	9.3%	0.92%	93	1000

Table 4.1: Performance of LWR with 3 proposed kernels. The kernel for LWR is the major factor deciding its performance.

(a) Assumption : RAVE can be thought of as a similarity measure learned from MCTS simulations. We use RAVE estimate on the root node in the update process. This 'kernel' is not exactly in a correct format. Still, in terms of update process, when a_j is selected on current search, we update a_i based on their similarity measure, $k_s(a_i, a_j)$.

The similarity measure used here is not a kernel between two nodes, it is rather a kernel between the node and the overall samples collected. When we search down the tree in MCTS, we choose the nodes with best expectations, or Q-values, with the method in use. The RAVE value at the root node is simply about how good are each actions. Having this value as kernel means we give more trust on the actions which have been proven to be good in the random roll outs.

(b) Kernel : $k_s(a_i, a_j) = \tilde{Q}(root, a_i)$

4.3.2 Analysis of results

Empirical experiment is done to see the potential of LWR as an update model for the MCTS. The main questions are,

- Does the LWR process actually provides improvement to the resulting MCTS?
- Which types of kernels show the most improvement in the MCTS for computer Go?

LWR is implemented in GoYui and tested against GNU Go v3.8. Hand selected schedule from Section 3.2.2 with $k = N/2$ is used for $\beta(s, a)$ function in the equation (4.31). N is the maximum number of samples to gather for each move. Also, UCB1 value is added to $Q_\star$ with c = 0.1. Result of simulation is given in Figure 4.4.

Compared to plain UCT, LWR does not seem to perform well. Main reason comes from the kernel, or the similarity measure used.

- Though within error range, Manhattan distance shows similar win rate as plain MC-UCT. This may indicate that two positions that are far apart actually do not influence each other much.

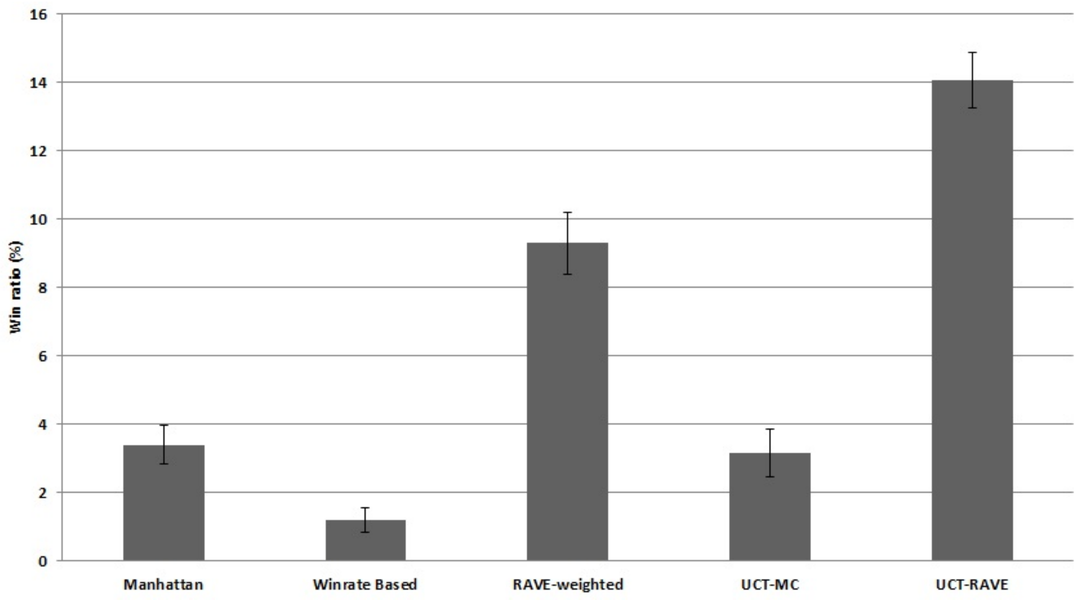


Figure 4.4: Win rate vs. tested kernels. Empirical results show that the proposed primitive kernels are not performing well. This tells us the difficulties of designing good kernels for LWR. Still, as in Section 4.4, RAVE is mathmatically identical to LWR. With decent kernels, LWR can actually bring improvements to MCTS as shown by the performance of RAVE in this figure. UCT-MC and UCT-RAVE is included for comparison. Each setting is tested against GNU Go v3.8 level 10 with 3000 samples per move. UCT is used on all methods with c = 0.1 with the equation (3.4)-we are mixing LWR and MC estimates in this simulations.

- Win ratio similarity shows a worse winrate than MC-UCT. Again, this is within the error range. Still, This result might be showing that the similar win ratio does not necessarily mean similar position.

- RAVE-based kernel shows better performance than MC-UCT, and the increase in win rate is notable. This tells us that our assumption for having this similarity measure has some aspect which actually results in a better performance.

As the results shows, it is not so easy to design a kernel for Go. Even the kernels which seems obvious to provide improvement, like Manhattan distance, does not. For many years before MCTS, these kind of manual encoding of knowledge was a mainstream of Go research. And we know they have not been so successful compared

Win rate	Std. Error	Wins	Number of Games
3.22%	0.70%	20	620

Table 4.2: Performance of UCT-MC. GoYui does not have decent performance with plain MCTS-UCT.

to MCTS. This tells us how hard it is to encode knowledge of Go. And also gives us the question why RAVE, which has very similar update process as LWR, works so well.

4.4 Locally Weighted Regression and RAVE

The RAVE algorithm is mathematically identical to LWR with a special kernel,

$$k_{ns}(a_i, a_j) = \begin{cases} 1 & \text{if } \tilde{I}_n(s, a_i) = 1 \text{ and } \tilde{I}_n(s, a_j) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.32}$$

$\tilde{I}_n(s,a)$ is an AMAF indicator function. $\tilde{I}_n(s,a)$ returns 1 if the action a is selected on any part of its sub tree including current node s. This includes the MC-sampling process. Basically it tells us whether the action is used in any consequent part of the whole MCTS process. Subscript n denotes each iteration n.

LWR for MCTS is defined as,

$$\hat{Q}(s,a) = \frac{1}{\sum_{n=1}^{N(s)} k_s(a, a_{sn})} \sum_{n=1}^{N(s)} k_s(a, a_{sn}) z_{sn}$$

With having a_{sn}, the $\tilde{I}_n(s, a_j) = 1$ part of kernel (4.32) becomes $\tilde{I}_n(s, a_{sn}) = 1$. And this is always 1 because a_{sn} is the action selected on i^{th} visit to the node s. Thus the LWR for MCTS with kernel (4.32) becomes,

$$\hat{Q}_R(s,a) = \frac{1}{\sum_{n=1}^{N(s)} \tilde{I}_n(s,a)} \sum_{n=1}^{N(s)} \tilde{I}_n(s,a) z_{sn}$$

And this is identical to RAVE which is defined as,

$$\tilde{Q}(s,a) = \frac{1}{\tilde{N}(s,a)} \sum_{i=1}^{N(s)} \tilde{I}_i(s,a) z_i$$

In this regard, RAVE is basically a LWR where its kernel depends on how current iteration of MCTS rolls out. This kernel (4.32) is just asking the MCTS process whether the two actions from current node is similar to each other. If the current iteration observes those two actions, it will say 'yes they are similar' by returning 1. If one of the action does not occurs in current iterations, this kernel will say 'no they are not similar' by returning 0.

And this tells us why RAVE works so well. Even without good amount of knowledge about Go, RAVE has very effective kernel which is based on random roll outs with default policy. The kernel is coming from the simulation. Two actions are more similar if they occur more often in the same roll out. As the iterations gets larger, the randomness from discrete values 0 and 1 will not have much effect. And at the end we effectively have a kernel telling us the similarity of two nodes, averaged over random sampling.

Still, we need to remember that the subtree is being built in this process. This simply makes the kernel to have more bias towards the results from previous iterations.

This slightly different view point on RAVE heuristic can provide possible future direction for the computer Go research. Now we know that a good kernel can help the performance. And this means we can actually look for more specific kernels based MCTS. Some kernels may provide better estimation on similarities of two nodes. And this may result in a better performance.

4.5 Summary

In this chapter we have investigated a variant of Monte Carlo tree search which uses locally weighted regression as its Q-value. The main purpose of LWR is to use the results from other similar nodes. This is to reduce the required number of samples to get a good estimation. Several primitive kernels were tested but did not show better performance than RAVE. A possible explanation for the high performance of RAVE is given in LWR framework.

RAVE shows that a good kernel can greatly improve the performance of MCTS. But what should be done if we end up with 'multiple estimators' for the quality of an action? For example, LWR models using different kernels can result in vast amount of models. How should these models be combined? Should we take the best model, or average them? In the next chapter we discuss this problem of combining various models for MCTS.

Chapter 5

Bayesian Model Averaging for Monte Carlo Tree Search

In the previous chapter we have discussed about the potential of kernel based MCTS. By the nature of kernels, this may lead us to large amount of additional models. Also, it is empirically shown that the method of mixing the MC and RAVE models is an important factor in deciding the strength of resulting MCTS algorithm. In this chapter we present Bayesian model averaging (BMA) as an alternative model mixing method for MCTS. Experiment and analysis of BMA on MCTS-based Go show the potential of BMA as an effective model mixing method for MCTS.

Section 5.1 talks about Bayesian model averaging and presents a derivation of BMA process for a binomial data set. Section 5.2 deals with applying BMA on MCTS in general. Section 5.3 talks about applying BMA on MCTS based computer Go using the two models, MC and RAVE, from the current literature. Section 5.4 provides some analysis on the performance and behaviour of BMA on MCTS Go. Finally, Section 5.5 summarises the chapter with mentioning the contributions made in this chapter.

5.1 Bayesian Model Averaging (BMA)

Bayesian model averaging (BMA) is a method of averaging predictions from multiple models when some data is given. A general case of BMA can be represented as follows.

$$\begin{aligned} E[q_{s,a}|D] &= \int_{q_{s,a}\in R} q_{s,a} P(q_{s,a}|D) dq_{s,a} \\ &= \int_{q_{s,a}\in R} q_{s,a} \sum_{m\in M} P(q_{s,a}|m,D)P(m|D) dq_{s,a} \\ &= \sum_{m\in M} E[q_{s,a}|m,D]P(m|D) \\ &= \sum_{m\in M} E[q_{s,a}|m]P(m|D) \end{aligned} \tag{5.1}$$

$q_{s,a}$ is a random variable of data for a state-action pair (s,a). D is collection of given data. And M is a set of models. $E[q_{s,a}|D]$ means the expected value of $q_{s,a}$

based on given data set D. In other words, $E[q_{s,a}|D]$ simply means how well the action a will perform on state s given data D.

From (5.1),

- $E[q_{s,a}|m]$ represents the prediction of model m. This value depends only on the model that is trained over data D.
- $P(m|D)$ represents the weight on model m given data D.

Thus, BMA can be understood as a method where predictions from each model,$E[q_{s,a}|m]$ are weighted with model probability, $P(m|D)$ in predicting the expected value of a random variable, $q_{s,a}$ based on data D.

5.1.1 Assumptions on Data and Model Probability

From (5.1), $E[q_{s,a}|m]$ term is just the predictions from each model with data D. And it is straight forward to use. But the model probability $P(m|D)$ term is not so obvious. With some assumptions on data D, we can derive the model probability $P(m|D)$ as follows.

1. Assume independence of each state-action pair (s,a), and let $D_{s,a} = \{q_{s',a'} \in D | s' = s, a' = a\}$

$$P(m|D) = P(m|D_{s,a}) \tag{5.2}$$

2. Bayes rule. Omit the shared constant $\frac{1}{P(D_{s,a})}$.

$$\begin{aligned} P(m|D_{s,a}) &\propto \hat{P}(m|D_{s,a}) && (5.3) \\ &= P(D_{s,a}|m)P(m) && (5.4) \end{aligned}$$

3. Assume independence of each data. Let $P(m)$ to be a prior weight to model m.

$$\hat{P}(m|D_{s,a}) = \prod_{q_{s,a} \in D_{s,a}} P(q_{s,a}|m)P(m) \tag{5.5}$$

Finally, we get $P(m|D_{s,a})$ as,

$$\begin{aligned} P(m|D_{s,a}) &= \frac{\hat{P}(m|D_{s,a})}{\sum_{m' \in M} \hat{P}(m'|D_{s,a})} \\ &= \frac{\prod_{q_{s,a} \in D_{s,a}} P(q_{s,a}|m)P(m)}{\sum_{m' \in M} \prod_{q_{s,a} \in D_{s,a}} P(q_{s,a}|m')P(m')} && (5.6) \end{aligned}$$

Notice in (5.6), we no longer have the $\hat{P}(m|D_{s,a})$ term. We have used two assumptions on the way,

- independence of each state-action pair.

- independence of each data.

From (5.6), the expression can be further specified if the type of the function for $P(q_{s,a}|m)$ is known.

5.1.2 Gaussian BMA

In the case where each data observation is modelled from a Gaussian distribution, (5.7) can be derived as the weight on each model. Derivation of this formula can be found in [Downey and Sanner 2010].

$$\hat{P}(m|D_{s,a}) = e^{-\frac{|D_{s,a}|}{2}} N(q^m_{s,a}; \mu_{s,a}, \sigma^2_{s,a})^{|D_{s,a}|} \tag{5.7}$$

5.1.3 Binomial BMA

In this subsection a derivation of BMA for binomial data set is given. Different from Gaussian BMA which deals with real-valued outcomes, BMA for binomial data set deals with Bernoulli, or binary, outcomes. As far as it is known to us, this is the first time BMA for binomial data set is presented in this form.

We have binomial distribution for the set of data when both of the following holds.

- Each data item can be modelled from a Bernoulli distribution.

- Each data items are independent of each other.

This second condition matches with the second assumption we used to derive (5.6).

From (5.6), we further specify it for the case where $P(q_{s,a}|m)$ is a Bernoulli distribution. Let's assume each data $q_{s,a}$ is a Bernoulli distribution.

$$q_{s,a} \in \{0,1\}$$

Let's define $w_{s,a}$ as the number '1' s observed from (s,a) pair.

$$w_{s,a} = |\{q_{s,a} \in D_{s,a}|q_{s,a} = 1\}|$$

As we are assuming each node is independent of each other, we can focus only on the data which is observed from the current node, $D_{s,a}$.

Then $\prod_{q_{s,a}\in D_{s,a}} P(q_{s,a}|m)P(m)$ of (5.6) can be represented as,

$$\begin{aligned}\prod_{q_{s,a}\in D_{s,a}} P(q_{s,a}|m)P(m) &= P(m) \prod_{q_{s,a}\in D_{s,a}} P(q_{s,a}|m)\\ &= P(m) \prod_{(q_{s,a}=1)\in D_{s,a}} P(q_{s,a}|m) \prod_{(q_{s,a}=0)\in D_{s,a}} P(q_{s,a}|m)\\ &= P(m)P(q_{s,a}=1|m)^{w_{s,a}} P(q_{s,a}=0|m)^{|D_{s,a}-w_{s,a}|} \end{aligned} \tag{5.8}$$

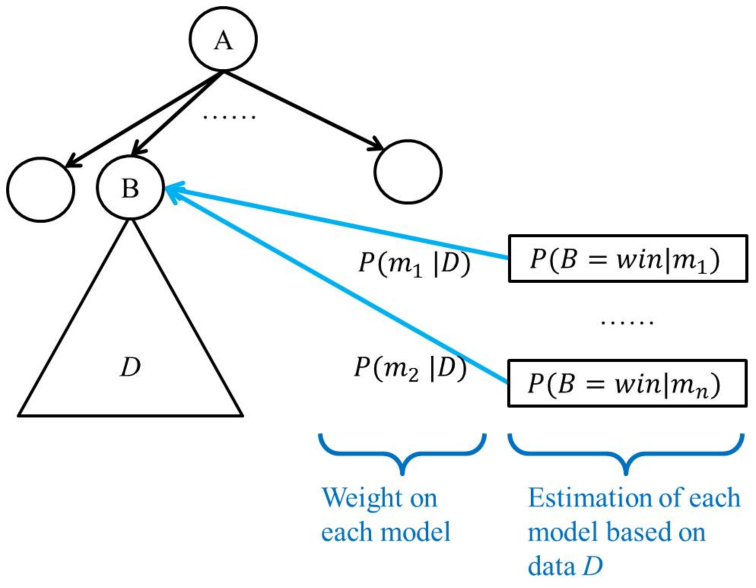


Figure 5.1: BMA for MCTS. BMA can be used as a model mixing tool for MCTS. Each model's estimation gets averaged with the weight on each corresponding model. For each node, notice that the weight on each model depends only the data from its sub tree. Still, this does not necessarily mean each model estimation also depends only on the data from the node's sub tree.

From (5.6) and (5.8) we get

$$P(m|D_{s,a}) = \frac{P(m)P(q_{s,a}=1|m)^{w_{s,a}}P(q_{s,a}=0|m)^{|D_{s,a}-w_{s,a}|}}{\sum_{m'\in M} P(m')P(q_{s,a}=1|m')^{w_{s,a}}P(q_{s,a}=0|m')^{|D_{s,a}-w_{s,a}|}} \tag{5.9}$$

Once again, in (5.9), we can notice that all the terms are either a model prior $P(m)$ or a prediction of a particular model $P(q_{s,a}|m)$.

5.2 BMA for MCTS

BMA can be used as a tool to mix various models in evaluating nodes in MCTS.

Let Q^* stand for the true expected reward value. And $Q^*(s,a|m)$ for the estimate of the true expected reward on node (s,a) with model m. Then if we use BMA, we have (5.10) as a Q-value for evaluating nodes.

$$Q_\star(s,a|D) = \sum_{m\in M} E[Q^*(s,a|D,m)]p(m|D) \tag{5.10}$$

Again, the $E[Q^*(s,a|D,m)]$ factor is the estimation from each model and the $p(m|D)$ factor is the weight on each model.

In Section 5.1.1 we assumed independence of each node and independence of each data. These two assumptions also can be applied to MCTS:

- **Independence of each nodes.** When we are given the data set, each nodes only depends on these data. And thus they are independent of each other.
- **Independence of each data.** Each data items are correlated in MCTS. This is because the earlier data items build the sub tree of current node. This sub tree guides the future search and thus has influence on future data. But this correlation may not be so dominant if the Monte Carlo Sampling at the end is not so deterministic, i.e. not so strong. This is because the samples itself will not be similar to each other. Especially when the Monte Carlo Sampling is not strong, even the sample from a same leaf node will be not so similar from each other. So the tree will guide the future samples, but the correlation on future data can be small.

Thus $P(m|D)$ from (5.10) can be further simplified to (5.6).

One of advantages of BMA for model mixing is its scalability. For each additional model, BMA requires one additional free variable. This tuning parameter is simply the $P(m)$ term, which is the prior of each model m.

5.3 BMA for computer Go with MC and RAVE estimates

In [Gelly and Silver 2011], Q-value of a given state-action pair,(s,a) is represented with weighted sum of two models, MC and RAVE.

$$Q_\star(s,a) = (1-\beta(s,a))Q(s,a) + \beta(s,a)\tilde{Q}(s,a) \tag{5.11}$$

$Q(s,a)$ is the MC estimate. $\tilde{Q}(s,a)$ is the RAVE estimate. And $\beta(s,a)$ is the weight function.

In averaging these two models, we can use BMA.

$$Q_\star(s,a|D) \;=\; \sum_{m\in RAVE,MC} E[Q^*(s,a|D,m)]P(m|D) \tag{5.12}$$

If we assume each node is independent,

$$\begin{aligned} Q_\star(s,a|D) &= Q_\star(s,a|D_{s,a}) \\ &= \sum_{m\in RAVE,MC} E[Q^*(s,a|D_{s,a},m)]P(m|D_{s,a}) \\ &= Q(s,a)p(m=RAVE|D_{s,a}) + \tilde{Q}(s,a)P(m=MC|D_{s,a}) \end{aligned} \tag{5.13}$$

(5.11) is essentially the same as (5.13) if we define $\beta(s,a)$ as

$$\beta(s,a) = p(m=RAVE|D_{s,a}) \tag{5.14}$$

In computer Go, end game score can be represented in various ways. Often it is represented as a binary variable, a win or a loss. This is to have robustness in the algorithm [Gelly et al. 2006]. As each sample has only two possible values, the distribution for a single sample, $P(q_{s,a}|m)$ becomes a Bernoulli distribution. And the whole sample data on a node can be represented with a binomial distribution.

Let's say 1 represents a win and 0 represents a loss on each sample data. We also assume independence of each data. Let,

$$\begin{aligned} w_{s,a} &= (d_{s,a} \in D_{s,a} | d_{s,a} = win) \\ P(MC) &= P(m = MC) \\ P(RAVE) &= P(m = RAVE) \end{aligned}$$

$w_{s,a}$ stands for number of wins from node (s,a). $P(MC)$ and $P(RAVE)$ are model prior for MC and RAVE models. Then from (5.9) we can get,

$$\begin{aligned} & \beta_{BMA}(s,a|D) \\ = \; & p(m = RAVE|D_{s,a}) \\ = \; & \frac{P(RAVE)\tilde{Q}(s,a)^{w_{s,a}}(1-\tilde{Q}(s,a))^{|D_{s,a}|-w_{s,a}}}{P(MC)Q(s,a)^{w_{s,a}}(1-Q(s,a))^{|D_{s,a}|-w_{s,a}} + P(RAVE)\tilde{Q}(s,a)^{w_{s,a}}(1-\tilde{Q}(s,a))^{|D_{s,a}|-w_{s,a}}} \\ = \; & \frac{1}{1 + \dfrac{P(MC)Q(s,a)^{w_{s,a}}(1-Q(s,a))^{|D_{s,a}|-w_{s,a}}}{P(RAVE)\tilde{Q}(s,a)^{w_{s,a}}(1-\tilde{Q}(s,a))^{|D_{s,a}|-w_{s,a}}}} \\ = \; & \frac{1}{1 + \dfrac{P(MC)}{P(RAVE)}\left(\dfrac{Q(s,a)}{\tilde{Q}(s,a)}\right)^{w_{s,a}}\left(\dfrac{1-Q(s,a)}{1-\tilde{Q}(s,a)}\right)^{|D_{s,a}|-w_{s,a}}} \end{aligned} \tag{5.15}$$

And this gives us BMA for MCTS with MC and RAVE models as

$$Q_\star(s,a) = (1-\beta_{BMA}(s,a))Q(s,a) + \beta_{BMA}(s,a)\tilde{Q}(s,a) \tag{5.16}$$

From this point, we are going to call our novel contribution on (5.16) as *BMA* (Bayesian Model Averaging for Binomial data set with MC and RAVE estimate).

5.4 Performance of BMA in GoYui

Empirical experiment is done to see the potential of BMA as a model mixing method for the MCTS. The main questions are,

- The performance of BMA compared to other previous model mixing methods.
- The effect of the tuning parameter of BMA on its performance.
- The behaviour of BMA within the resulting MCTS.

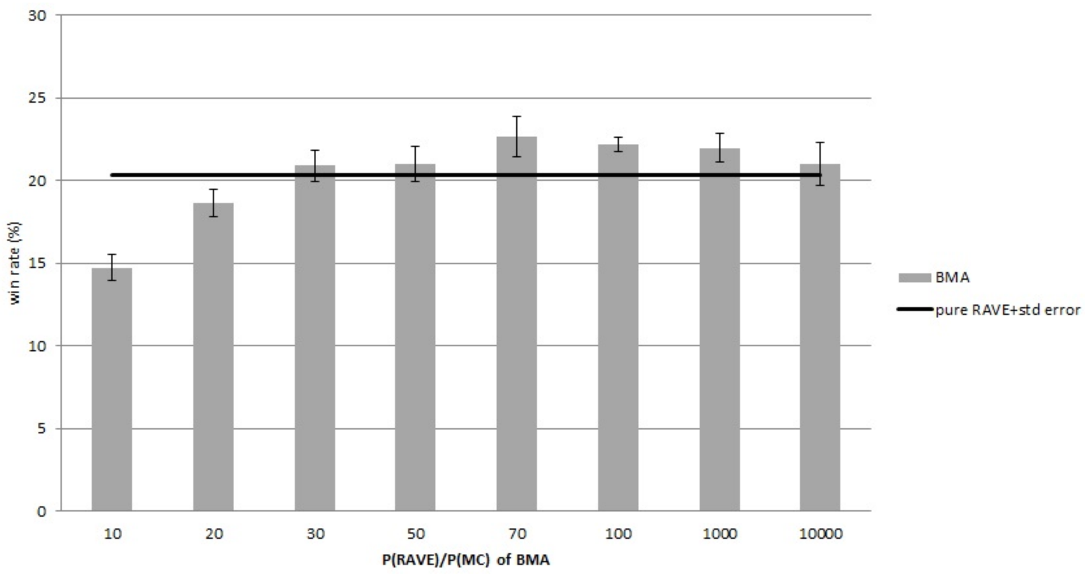


Figure 5.2: Win rate(%) of BMA vs. $P(RAVE)/P(MC)$, the prior of BMA. Black line represents the win rate of pure RAVE plus its standard error. More weight on RAVE estimate generally results in better performance. When we have higher prior on RAVE estimate, BMA becomes almost identical to pure RAVE. And we can see the tendency of win rate converging to pure RAVE with high prior on RAVE estimate.

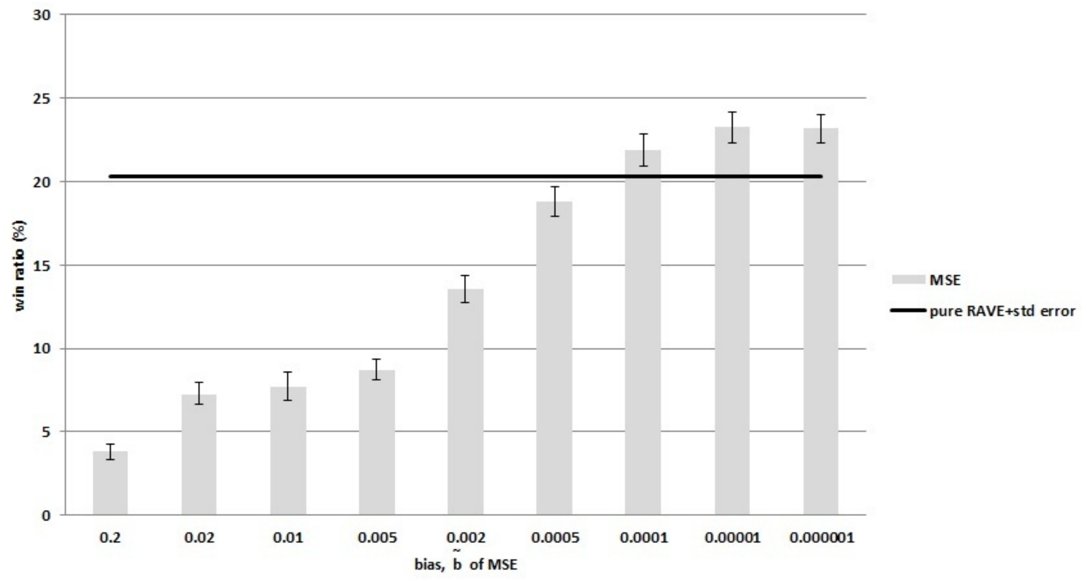


Figure 5.3: Win rate(%) of MSE vs. $\tilde{b}$, the bias of MSE. Black line represents the win rate of pure RAVE plus its standard error. $\tilde{b}$ has huge effect on the performance of MSE. By the definition of $\tilde{b}$, at equation (3.9), smaller $\tilde{b}$ generally means more trust in the RAVE estimate.

k	Win rate	Std. Error	Wins	Number of Games
1500	14.06%	0.82%	254	1806

Table 5.1: Performance of hand selected schedule

$\frac{P(RAVE)}{P(MC)}$	Win rate	Std. Error	Wins	Number of Games
10000	15.41%	0.82%	298	1934

Table 5.2: Performance of BMA for Gaussian data. For MCTS Go with GoYui implementation, BMA for binomial data shows better performance than BMA for Gaussian data.

Win rate	Std. Error	Wins	Number of Games
19.99%	0.33%	2881	14411

Table 5.3: Performance of pure RAVE.

$\frac{P(RAVE)}{P(MC)}$	Win rate	Std. Error	Wins	Number of Games
10	14.75%	0.80%	288	1952
20	18.62%	0.82%	418	2245
30	20.90%	0.93%	401	1919
50	21.03%	1.07%	307	1460
70	22.64%	1.22%	266	1175
100	22.71%	0.43%	1201	5287
1000	21.97%	0.85%	517	2353
10000	21.0%	1.29%	210	1000

Table 5.4: Performance of BMA for binomial data with different priors, $\frac{P(RAVE)}{P(MC)}$

$\tilde{b}$	Win rate	Std. Error	Wins	Number of Games
0.2	3.82%	0.46%	67	1755
0.02	7.2%	0.58%	144	2000
0.01	7.7%	0.84%	77	1000
0.005	8.72%	0.63%	175	2007
0.002	13.55%	0.79%	253	1867
0.0005	18.80%	0.91%	345	1835
0.0001	21.90%	0.98%	388	1772
0.00001	23.26%	0.93%	479	2059
0.000001	23.16%	0.82%	617	2664

Table 5.5: Performance of minimum MSE with different $\tilde{b}$

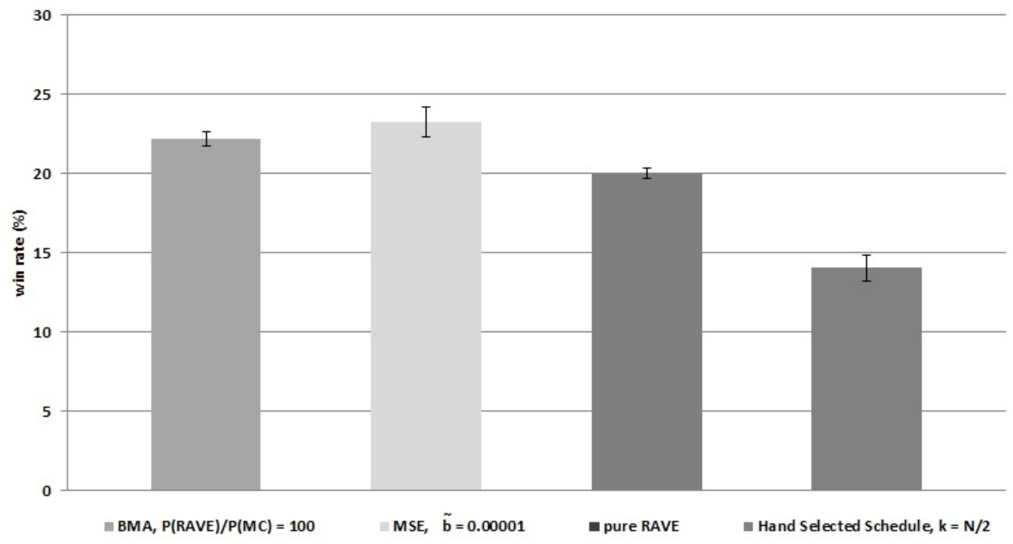


Figure 5.4: Win rate(%) vs. mixing methods with best performing parameters. BMA is on-par with MSE. Both BMA and MSE shows better performance than pure RAVE. Also note that pure RAVE shows higher win rates than hand selected schedule, a mixed model of MC and RAVE estimate. Each parameter is tested with at least 1000 complete games against GNU Go v3.8 level 10 with 3000 samples per move.

5.4.1 Simulation Settings

Four variants of MCTS with MC and RAVE estimate is tested on a sandbox Go engine GoYui. On all of these cases, UCB1 term is used with $c = 0.1$.

1. Pure RAVE. This setting only uses RAVE estimate as its Q-value.

$$\beta(s, a) = 1$$

2. Hand selected schedule. This heuristic gives more weight on RAVE on earlier stages of a game and gradually reduces it. Given in [Gelly and Silver 2011]

$$\beta(s, a) = \sqrt{\frac{k}{3N(s) + k}}$$

   $N(s)$: number of visits to node s, this is the parent node of node (s, a)
   k : a tuning parameter. When $n = k$, $\beta(s, a)$ becomes 0.5.

3. Minimum Mean Squared Error (MSE). A method which minimises squared error between the estimated value and the true value of the node. Given in [Gelly and Silver 2011].

$$\beta(s,a) = \frac{\tilde{n}}{n + \tilde{n} + 4n\tilde{n}\tilde{b}^2}$$

n : number of visits to node (s,a)
$\tilde{n}$: number of all-moves-as-first(AMAF) visits to node (s,a)
$\tilde{b}$: a tuning parameter. Defined as a bias of RAVE estimate,
$(\tilde{b} = Q^*{}_{RAVE}(s,a) - Q^*(s,a))$

4. Bayesian Model Averaging for Binomial data set with MC and RAVE estimate (BMA). Derivation is given in Section 5.3

$$\beta(s,a) = \frac{1}{1 + \frac{P(MC)}{P(RAVE)} \left(\frac{Q(s,a)}{\tilde{Q}(s,a)} \right)^{w_{s,a}} \left(\frac{1 - Q(s,a)}{1 - \tilde{Q}(s,a)} \right)^{n - w_{s,a}}}$$

n : number of visits to node (s,a)
$w_{s,a}$: number of wins observed in samples from node (s,a)
$\frac{P(MC)}{P(RAVE)}$: a tuning parameter. Defined as a model prior.
$Q(s,a)$: estimate of win ratio from node (s,a) by MC model
$\tilde{Q}(s,a)$: estimate of win ratio from node (s,a) by RAVE model

Performance of these UCT-RAVE variants are tested empirically on a sandbox Go engine GoYui. For each parameter at least 1000 games against GNU Go v3.8 with 3000 samples per move was played.

5.4.2 Analysis of BMA in GoYui

Figures 5.2, 5.3, and 5.4 show the simulation results. In the following, the performance of each MCTS variants is analysed.

5.4.2.1 Performance of pure RAVE

One of the characteristics of GoYui is the superior performance of pure RAVE. In Figure 5.4, parameters which places more weight on RAVE generally performs better than more balanced parameters. Here are three observations which support this,

- Hand selected schedule with larger k. $k = N/2$ is empirically selected parameter for GoYui which showed decent win ratio among similar values. $k = N/3$ in [Gelly and Silver 2011] shows best performance for hand selected schedule. In fact, we can simulate pure RAVE with hand selected schedule. If we set k with a large value, large compared to number of samples per move, $\beta(s,a)$ will be almost equal to 1 on all nodes throughout the search. And hand selected schedule becomes, effectively, pure RAVE. From this, we can see that the method which balances MC and RAVE on some parts of its nodes performs worse than the method which only uses RAVE.

- MSE with smaller bias $\tilde{b}$. The bias term $\tilde{b}$ of MSE can be thought, in some sense, as a weight on RAVE estimate. With same n and $\tilde{n}$, smaller $\tilde{b}$ only makes corresponding $\beta(s, a)$ larger. And the simulation result shows obvious relationship between $\tilde{b}$ and the win ratio. The win ratio increases as the $\tilde{b}$ gets smaller. Whether this tendency will continue is not certain. Still, when we set $\tilde{b} = 0.0$ the performance remains around 20%.

- BMA with larger model prior on RAVE estimate. Similarly to $\beta(s, a)$ of MSE, for a given set of data, increasing the relative model prior of RAVE estimate will only increase $\beta(s, a)$. And we can see the winning ratio increasing when the model prior gets larger. Obviously, if we have extremely large model prior for RAVE estimate, $\beta(s, a)$ of BMA will effectively become 1 and the method will become pure RAVE. This gives an approximate minimum for larger model priors.

This observations do not seem to match with other results. In [Gelly and Silver 2011], RAVE is assumed to learn faster. But it is not assumed to perform better than method which combines MC estimates with it. We assume that MC will have much better prediction when the number of samples are greater than certain threshold. This assumption is the main rationale for giving more weights to MC estimate as we get more samples. Also it is the main rationale for expecting combined method to perform better. But, 3000 samples might have not been enough for MC to surpass the estimation of RAVE with GoYui implementation. Then why do we assume RAVE to perform better with a smaller number of samples? Explanations can be,

- AMAF heuristic always has more visits. This is purely related to the number of samples each node get from each estimation. For each actual visit to the node, we give +1 to MC visit and AMAF visit at the same time. As AMAF heuristic also updates other possible nodes, it will always have more number of samples updating it. This difference in number of visit scales with the size of the Go board. For larger board, this effect can be bigger, and the RAVE estimate will have far more visits than MC estimate. Having more sample simply increases the strength in MCTS. When the number of sample is small enough, MC estimate will be very noisy. In this case, the RAVE estimate will have an order of magnitude more samples and will perform better. The only remaining question will be whether RAVE estimate also converges to true estimate with infinite number of samples. And if it does so, how fast it converges compared to MC estimate?

- RAVE estimate can benefit from successes made from outside the current sub tree [Gelly and Silver 2011]. If there is a good structure developed in other sub tree, this will also benefit other part of the tree when we use AMAF heuristic. And small number of samples might not allow MC methods to find such structures on most of local sub-trees.

- Bias vs. variance trade off. By its definition, the RAVE estimate is more biased than the MC estimate. For small number of samples, the RAVE estimate is much more reliable as it has lower variance resulting from its high bias.

5.4.2.2 Comparison between pure RAVE, BMA and MSE

The strength of BMA seems to be on par with MSE. Still, both methods does not surpass pure RAVE by much. Even though the 3% difference between pure RAVE and BMA or MSE is small, one thing to note is that BMA and MSE behaves somewhat differently from pure RAVE.

- MSE obviously, does not converge to pure RAVE with any value of $\tilde{b}$.
- For BMA, there are cases where the weight on MC estimate increases more than 0.5 even with high model prior.

From these, BMA or MSE can not be claimed to work because it naively places more weight on the RAVE estimate. Also, empirical results show that they do seem to perform somewhat better than pure RAVE.

5.4.2.3 Behaviour of BMA

In this section we show that BMA asymptotically places more weight on MC estimate as it gathers more number of samples.

Figure 5.5 shows $(1-\beta(s,a))$, or the weight on MC estimate, of BMA for a node which have been selected at the end. Figure 5.8 also shows $(1-\beta(s,a))$ of BMA for a node which have been selected at the end. In both cases we can see the weight on MC model increasing with iteration.

For a function $f(x)$

$$f(x) = x^{w}(1-x)^{(n-w)} \tag{5.17}$$

maximum of $f(x)$ is at $x = \dfrac{w}{n}$.

Let

$$\alpha_{MC}(s,a) = Q(s,a)^{w_{s,a}}(1-Q(s,a))^{n-w_{s,a}} \tag{5.18}$$

$$\alpha_{RAVE}(s,a) = \tilde{Q}(s,a)^{w_{s,a}}(1-\tilde{Q}(s,a))^{n-w_{s,a}} \tag{5.19}$$

Recall that $Q(s,a)$, the MC estimate, is defined as

$$Q(s,a) = \frac{1}{N(s,a)}\sum_{i=1}^{N(s)} I_i(s,a)z_i$$

And this is effectively

$$Q(s,a) = \frac{w_{s,a}}{N(s,a)} \tag{5.20}$$

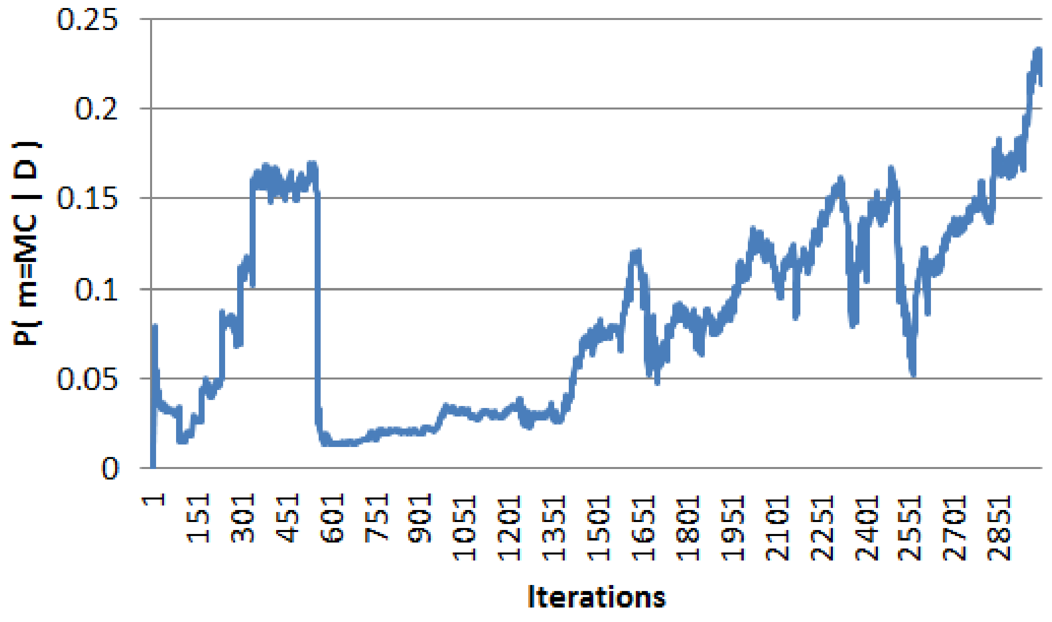


Figure 5.5: The weight of the MC estimate for a fixed node vs. total number of samples at the root node. $(1-\beta(s,a))$, or the weight on MC estimate of BMA with P(RAVE)/P(MC) = 70. This node is a immediate child of the root node and have been selected at the end of 3000 iterations. We can see the weight on MC estimate increasing at the end. Each drop usually means MC and RAVE estimates being closer to each other, by actual visit to this node or by result from RAVE update on this node.

From (5.17), (5.18) and (5.20) we can see that $\alpha_{MC}(s,a)$ is never going to be smaller than $\alpha_{RAVE}(s,a)$ because $Q(s,a)$ gives the maximum for both $\alpha_{MC}(s,a)$ and $\alpha_{RAVE}(s,a)$.

If we set our model prior to be equal to each other as $P(MC) = P(RAVE)$, then $\beta(s,a)$ of BMA becomes,

$$\begin{aligned}
\beta(s,a) &= \frac{1}{1+\frac{P(MC)}{P(RAVE)}\left(\frac{Q(s,a)}{\tilde{Q}(s,a)}\right)^{w_{s,a}}\left(\frac{1-Q(s,a)}{1-\tilde{Q}(s,a)}\right)^{n-w_{s,a}}} \\
&= \frac{P(RAVE)\tilde{Q}(s,a)^{w_{s,a}}(1-\tilde{Q}(s,a))^{n-w_{s,a}}}{P(MC)Q(s,a)^{w_{s,a}}(1-Q(s,a))^{n-w_{s,a}}+P(RAVE)\tilde{Q}(s,a)^{w_{s,a}}(1-\tilde{Q}(s,a))^{n-w_{s,a}}} \\
&= \frac{P(RAVE)\alpha_{RAVE}(s,a)}{P(MC)\alpha_{MC}(s,a)+P(RAVE)\alpha_{RAVE}(s,a)} \qquad (5.21) \\
&= \frac{\alpha_{RAVE}(s,a)}{\alpha_{MC}(s,a)+\alpha_{RAVE}(s,a)} \qquad (5.22)
\end{aligned}$$

As $\alpha_{RAVE}(s,a) \leq \alpha_{MC}(s,a)$ for all cases, $\beta(s,a) \leq 0.5$ when $P(RAVE) = P(MC)$.

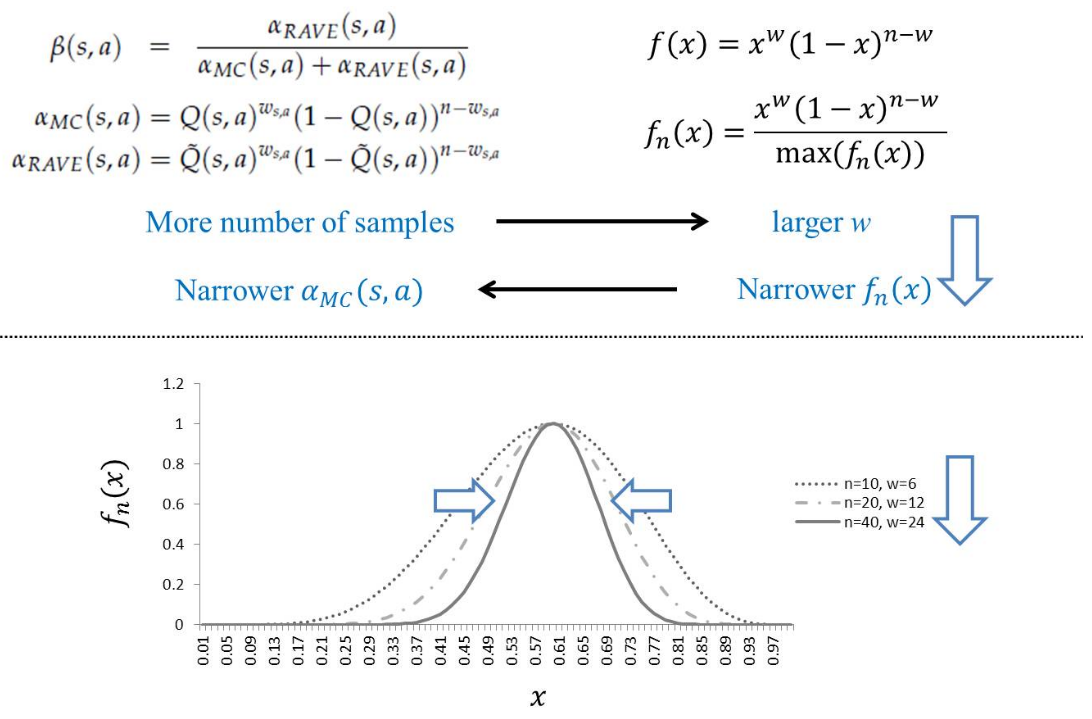


Figure 5.6: Influence of the number of samples on normalized version of (5.17). Larger number of samples results in narrower α_{MC} and α_{RAVE}. This property is used to describe how more number of samples can result in narrower α_{RAVE} which gives smaller α_{RAVE} with the same $\tilde{Q}(s,a)$ in Figure 5.7.

Simply put, this tells us that BMA will always have more weight on MC estimate than on RAVE estimate when we have same prior on both models.

Now let's think about the case where we have $P(RAVE) >> P(MC)$, which is the case with our simulation. In this setting there are two cases,

1. $Q(s,a) \approx \tilde{Q}(s,a)$. This is rather common with BMA in MCTS, especially for the nodes that is selected at the end of search.

   Recall that we select nodes with the most visits at the end of MCTS. This is to have robustness in the search. The main rationale is that we know more about the node that have been visited more.

   MCTS with low exploration tends to visit only a small subset of nodes for the most of iterations. When we have most of iterations visiting a single node, the MC and RAVE estimates of that node become very similar to each other. This is because the update process for the visited node is identical for both MC and RAVE estimates.

   When both estimates are similar, the weight on each model does not have any meanings, we can have any weight, if its not zero for both. And $\beta(s,a)$ becomes ilrelevant. For $P(RAVE) >> P(MC)$, $\beta(s,a)$ will become similar to 1 and this makes $1-\beta(s,a)$ similar to 0.

2. $Q(s,a) > \tilde{Q}(s,a)$ or $Q(s,a) < \tilde{Q}(s,a)$.

   When MC and RAVE estimates are not similar, we know that MC estimate will always have more weight on it. This is observed from $\alpha_{RAVE}(s,a) \leq \alpha_{MC}(s,a)$. For $P(RAVE) >> P(MC)$, this difference is mitigated with high prior on RAVE. But as we can see from Figure 5.8 there are cases where we have $(1 - \beta(s,a))$, the weight on MC estimate, rising. And these cases have very large $\alpha_{MC}(s,a)$ compared to $\alpha_{RAVE}(s,a)$. This simply means that we have learned from current samples that MC estimate is much more accurate than RAVE estimate on current state.

The effect of having more over samples can be seen from Figure 5.7. For same difference in MC and RAVE estimate, weight on MC model increases with more number of samples. BMA resembles our general belief on MC estimate, that MC estimate will get better with the number of samples.

In short, we have more weight on MC estimate when the MC and RAVE estimates disagree. This weight can be reduced with larger prior on RAVE. But even with the prior, there are cases where we have significant weight on MC estimate.

5.5 Summary

In this chapter we have investigated Bayesian model averaging as an alternative mixing method, or schedule, for MCTS. Contributions from this chapter can be listed as,

- A novel method BMA for binomial data set is derived.
- The effectiveness and possibility of BMA as a model mixing method is presented. BMA shows comparable performance with minimum MSE. But they have different behaviour.

BMA showed similar performance as minimum MSE when tested on GoYui against GNU Go. Even though BMA's performance is within the error range of pure RAVE, BMA does not act identical to pure RAVE with the tested parameters. Also, their error range with 1 standard error does not overlap on tuned parameters. More importantly, minimum MSE which is shown to have better performance in [Gelly and Silver 2011] shows similar win rate as BMA. Also, Bayesian framework gives us a decent theoretical background for BMA. Additionally, the tuning parameter of BMA has clear semantics within Bayesian settings and the number of tuning parameters linearly scales with the number of models. These shows possibility of BMA as an effective method for mixing various models in MCTS.

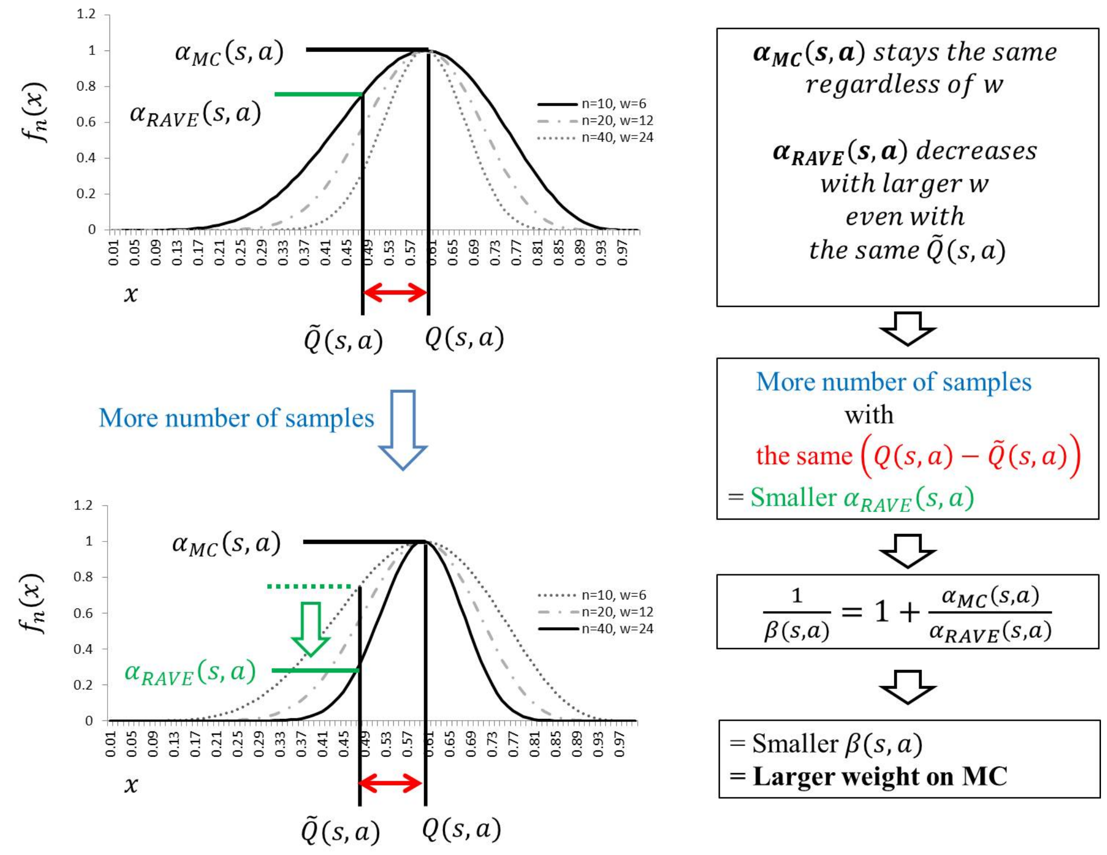

$\alpha_{MC}(s,a)$
$\alpha_{RAVE}(s,a)$
$f_n(x)$
x
n=10, w=6
n=20, w=12
n=40, w=24
$\tilde{Q}(s,a)$
$Q(s,a)$
More number of samples
$\boldsymbol{\alpha_{MC}(s,a)}$ stays the same regardless of w
$\boldsymbol{\alpha_{RAVE}(s,a)}$ decreases with larger w even with the same $\tilde{Q}(s,a)$
More number of samples with the same $\left(Q(s,a) - \tilde{Q}(s,a)\right)$
= Smaller $\alpha_{RAVE}(s,a)$
$\frac{1}{\beta(s,a)} = 1 + \frac{\alpha_{MC}(s,a)}{\alpha_{RAVE}(s,a)}$
= Smaller $\beta(s,a)$
= Larger weight on MC

Figure 5.7: Normalized version of (5.17). Larger number of samples results in more weight on MC estimate. This property closely resembles the common assumption on MCTS : MC estimate will be more accurate when there are large number of samples. Refer to Figure 5.6 or to Section 5.4.2.3 for the description of each variables. α_{MC} is always at the maximum. When the difference between MC and RAVE estimates stay as a constant, α_{RAVE} decreases with the number of samples. Figure 5.6 shows how more number of samples result in narrower α_{RAVE} which gives smaller α_{RAVE} with the same $\tilde{Q}(s,a)$. Smaller α_{RAVE} only results in smaller $\beta(s,a)$ and, thus, larger weight on MC estimate.

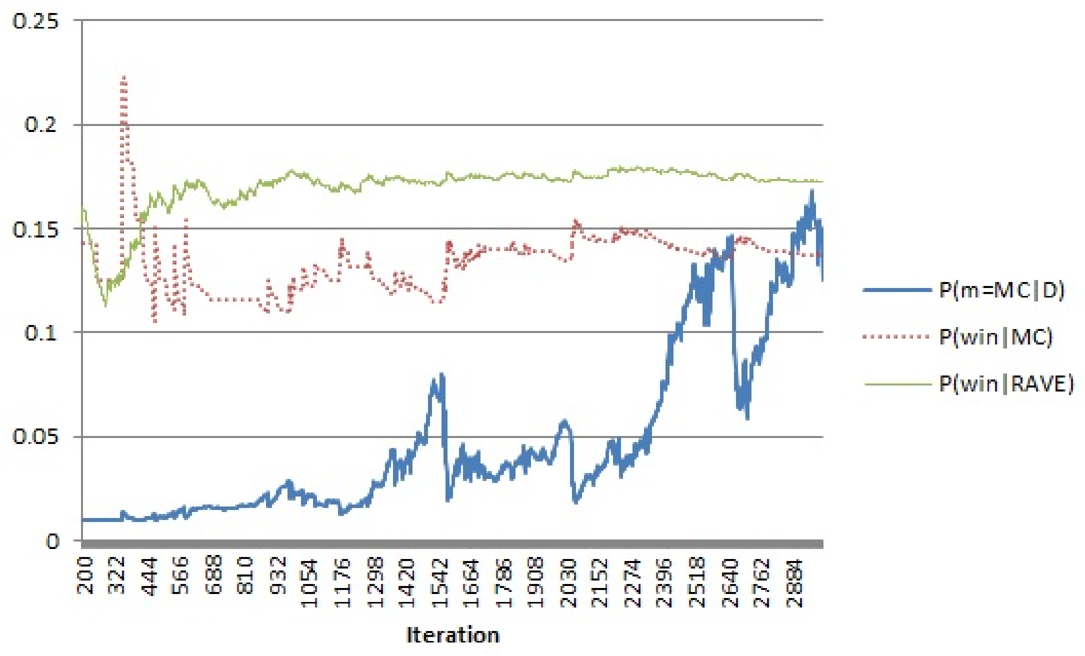


Figure 5.8: BMA with P(RAVE)/P(MC) = 100. This node is an immediate child of the root node and have been selected at the end of 3000 iterations. RAVE estimate does not changes by much as it usually has order of magnitude more visit than MC estimate. Notice both MC and RAVE estimate of this graph is less than 0.2. This simply means the board situation is grim and is expecting a loss from this move. In this case MCTS tends to spread out its visits to nodes. And this results in order of magnitude difference in RAVE and MC visits. And this is the main reason for the disagreement of RAVE and MC values at the end of 3000 iterations. Decrease in the weight for MC estimate is caused by MC and RAVE estimates being closer to each other.

Chapter 6

Conclusion

6.1 Summary

In this thesis the following problems were discussed:

- **Update process of MCTS.** Most of work in computer Go literature does not have strong theoretical analysis on handling the dependencies or the similarities within the search tree. But intuitively, we can, at least, see the local similarities appearing in the Go board. The RAVE algorithm deals with this to good measure but the theoretical background for RAVE as a similarity measure is not well discussed in current literature.

  Our approach for this problem was to investigate possible update methods which are designed specifically to handle the similarities within the search tree. More precisely, the methods which look for the similarities within the legal moves from the same current board. By considering the node similarities, the overall MCTS process can utilize most of its samples rather than throwing them away.

- **Model mixing method for MCTS.** Empirical results show that method of combining multiple models is one of crucial factors deciding the strength of resulting MCTS algorithm. But this portion of MCTS is also not given enough attention in current literature.

  Our approach for this problem was to investigate the possibility of Bayesian model averaging as an alternative, or additional, model mixing method. Bayesian model averaging has a decent theoretical background. Also the tuning parameter has clear very representation within the Bayesian framework. The tuning parameter in BMA represents our initial beliefs on each model when there are small number of samples.

For both of these problems our goal was to come up with a novel MCTS enhancements which can outperform strong MCTS baselines and give additional strength to the state-of-the-art Go programs.

Contributions of this thesis is listed in the following:

- **A Bayesian update process with Gaussian prior for MCTS.**

  A Bayesian update process with Gaussian prior for computer Go, or binomial data set, is derived. The main intention is to investigate the potential of MCTS variant with specific emphasis on incorporating the node similarities. This process provides a possible method to update the tree structure of MCTS with consideration on the similarities between the each pair of nodes sharing same parent node within the search tree. But this process has inherent inefficiency of $O(n^2)$ caused from considering all the pairs of child nodes. n represents the number of child nodes. And for a 19x19 board in Go, n is at around 300.

- **Locally weighted regression (LWR) as an update process for MCTS.**

  LWR is investigated as an update method for MCTS in Chapter 4. Main rationale for this method is the intention to handle the similarities within the search tree. Also, LWR is tractable in MCTS. Some primitive kernels were tested along with RAVE based kernel in order to check the effectiveness of LWR update on MCTS. Empirical and theoretical results show that the performance of LWR strongly depends on the effectiveness of the kernels used.

- **Explanation of RAVE within LWR framework for MCTS.**

  Rapid action value estimation (RAVE) have been shown to be the most effective method in MCTS for Go by the state of the art Go engines [Gelly and Silver 2011] [Enzenberger et al. 2010]. But there has not been much explanation about its strength.

  In Chapter 4, an explanation for the strength of RAVE is given within the framework of LWR in MCTS. The all moves as first (AMAF) heuristic in MCTS is identical to having each iteration of MCTS as an similarity measure of two different state-action pairs. In such regard, RAVE can be thought of as a LWR algorithm using the kernel similarity which bases its evaluation on each random roll outs.

  This explanation gives us two insights on kernel based MCTS.

  1. Kernel based MCTS actually brings some improvements to MCTS.
  2. Simulation based kernels are effective on complex domains such as computer Go.

- **Derivation of Bayesian model averaging (BMA) for binomial data set.**

  In Chapter 5, we have derived BMA for binomial data set. As far as it is known to us, this is the first time where the BMA for binomial data set is derived and used in MCTS. A series of Go games can be thought of as a binomial data if we only consider the win and loss of each game. And having this binary representation actually gives us more robustness in the resulting MCTS for Go [Gelly et al. 2006].

- **BMA as an alternative mixing method for MCTS.**

  In Chapter 5, we have investigated BMA as an alternative mixing method for MCTS. BMA is a method of averaging multiple models within the Bayesian framework. Here are some reasons to consider BMA as a valid option:

  - **Performance.** BMA performs on par with minimum MSE. Minimum MSE is one of the mixing methods proposed in [Gelly and Silver 2011] which shows huge improvement in the performance of MCTS. BMA shows similar performance improvement as minimum MSE with GoYui implementation when tested against GNU Go. This shows the potential of BMA as an effective model mixing method.
  - **Asymptotically higher weight on MC estimate.** By its nature, BMA places higher weight on MC estimate as MCTS gets more number of samples. This behaviour resembles our common assumptions on MC estimate. In general, we expect the MC estimate to provide the most accurate estimation when there are large number of samples. Asymptotically, when the number of samples increases the weight on MC estimate also increases in BMA. With this we may claim that BMA is well suited for MCTS.
  - **Robustness of the tuning parameter.** BMA shows smaller fluctuation in the win rate within the tested parameters. For n models, BMA has $n-1$ tuning parameters. This tuning parameter also has a clear semantics. It represents the prior beliefs on each model. In the experiments, this clear meaning of the tuning parameter made it easy to select the values to test on. And from this simple meaning within the tuning parameter, BMA seems to be more robust than minimum MSE. For minimum MSE, $\tilde{b}$ term, the tuning parameter of minimum MSE, is rather hard to choose the appropriate value for it. And this resulted in more fluctuations in the win rate of the chosen parameters.
  - **Bayesian Settings.** BMA has a decent theoretical background within Bayesian settings. BMA has a quite different derivation from minimum MSE. And this makes BMA a valuable alternative as a model mixing method. If the property of the domain does not fits well within other methods, BMA can be tested with a hope as it has a different theoretical background.
  - **Expressiveness.** BMA can represent from pure MC to pure RAVE with adjusting its tuning parameter. But this is simply not possible with minimum MSE. With large $\tilde{b}$, minimum MSE becomes similar to pure MC. But minimum MSE can not represent the pure RAVE. BMA can balance RAVE and MC from pure RAVE to pure MC. And this is very easy to manipulate.

6.2 Future Directions

- **Kernel based MCTS.** RAVE shows us that LWR update process works very well on MCTS with effective kernels. This observation encourages us to look for more kernel based MCTS. The type of kernels that will be effective on MCTS for computer Go are most likely to be in a similar form with RAVE. This is due to the difficulties within encoding the Go knowledge into a definite form. Still, it seems rather clear that without compensating the flaws within the AMAF heuristic, there will not be much advancements in the computer Go. And one of the obvious ways to look for alternative models will be investigating other possible simulation based kernels for computer Go. Also it will be rather interesting to see how kernel based MCTS can perform in other complex domains where explicit evaluation of each intermediate states are not easy to get.

- **Further testing of BMA.** Performance of BMA is not well tested due to the characteristics of GoYui. In [Gelly and Silver 2011], minimum MSE shows around 97% win ratio against GNU Go where as hand selected schedule shows around 92%. This improvement is also observed within GoYui implementation. But the problem comes from the performance of pure RAVE. In [Gelly and Silver 2011], the performance of pure RAVE is not given specifically and it is speculated to have worse performance than mixed methods. Whether the strength of BMA comes from the special property of GoYui is unknown. Still, it does seem to provide improvement compared to pure RAVE. To further verify the strength of BMA vs. minimum MSE, we need to test BMA with many other default policies as MCTS is very sensitive to the default policy. Investigating the strength of BMA on other applications of MCTS, especially for a case where there are more than 2 models to average over, can be an interesting question.

- **Developing MCTS variants based on the known characteristics of RAVE estimate.** In Appendix A, we show two variants of MCTS among many other possibilities. These are tested briefly on GoYui.

  – Weighted update for the RAVE estimated.

  – Limited number of visit to each node.

  Interestingly these variants shows similar win rate with original RAVE. Further investigating these methods which tries to exploit the property of RAVE estimate can be a possible direction.

6.3 Go, MCTS and AI

In this thesis we have investigated possible enhancements to MCTS within current computer Go literature. Games have served as important test beds throughout the history of AI. In these days, Go have finally become the inarguable 'task par excellence for AI' [Berliner 1978]. Go is a domain where traditional methods does not have much hope. After all, MCTS is just a Monte Carlo method with a tree. But with the observed performance of MCTS in computer Go, now we have an additional example where MCTS is working perfectly well within a complex domain. It is rather obvious that the MCTS methods and its enhancements from the computer Go can be applied to other goal based search, planning, or learning algorithms which are pervasive in AI. AI community still has a lot more to achieve from the computer Go. Simply put, machines have not been able to outperform the human beings in Go. This indicates the existence of missing pieces within computer Go that can lead to the advancements in AI. In such regards, we hope the results of this thesis can also help beyond computer Go to the other important application areas of AI.

Appendix A

Other MCTS variants

A.1 Limited visit MCTS

- Algorithm : When selecting a child node in tree search phase of MCTS, the child node is not selected if it has more than k visits. k is a free variable. Final selection after all iteration can be done randomly if there are multiple child nodes with k visits.
- Rationale : Most of the time, RAVE only focuses on one or two nodes. In those cases, only 2 two nodes has more than 10% of the total iteration. But when a node has more than certain amount of visit, we can expect to reach a limit where we don't get more information from the node. This is crucial especially when we don't have enough iterations to visit most of the nodes. Instead of focusing on having better estimation of 1 2 nodes, having limit on the number of visit may give us more opportunity to explore some other candidates. The limit is set to have enough information, empirically, after k visits.

  By this method, intended result was to have more than 1 2 nodes with enough visits so that we can choose from the nodes we know about.
- Performance : Tested on GoYui against GNU Go v3.8 level 10. GoYui is set with MC-RAVE with BMA using P(RAVE)/P(MC) = 100 and c=0.1 for UCT.

  Win rate was 18.7% with 2.23% standard error.

A.2 RAVE with Weighted update

- Algorithm : following $\tilde{Q}'(s,a)$ is used as a $\tilde{Q}$-value of RAVE estimate.

$$\tilde{Q}'(s,a) = \sum_{i=1}^{N(s)} w_i(s,a)\tilde{I}_i(s,a)z_i$$

$$w_j(s,a) = \frac{K_j(s,a,a_{sj})}{N(s,a_{sj})}$$

$$K_j(s,a,b) = \sum_{i=1}^{j}(\tilde{I}_i(s,a) + \tilde{I}_i(s_c,a))(\tilde{I}_i(s,b) + \tilde{I}_i(s_c,b))$$

a_{sj} is the action selected from node s on jth iteration.

$K_j(s, a, b)$ simply stands for the number of times action a and b both have been selected in the sub tree of s including random rollout.

s_c is a counter part of state s, which is the same board configuration with opponent's turn.

- Rationale : RAVE should be telling us how similar two actions are. By looking at the number of times two actions have been selected in the MCTS process, we can estimate how similar those two actions are. This similarity is used as an extra weight on RAVE update.
- Performance : Tested on GoYui against GNU Go v3.8 level 10. GoYui is set with MC-RAVE with BMA using P(RAVE)/P(MC) = 100 and c=0.1 for UCT.

  Win rate was 20.0% with 6.7% standard error.

Bibliography

Auer, P., Cesa-Bianchi, N., and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning 47*, 2-3, 235–256. (p. 16)

Baker, K. 2008. *The Way to Go*. American Go Foundation. (p. 5)

Berliner, H. J. 1978. A chronology of computer chess and its literature. *Artificial Intelligence 10*, 201–214. (pp. 1, 62)

Campbell, M., Hoane, A., and Hsu, F. 2002. Deep Blue. *Artificial Intelligence 134*, 57–83. (p. 1)

Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. 2008. Monte-Carlo Tree Search: A New Framework for Game AI. *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*. (p. 13)

Cleveland, W. S. and Devlin, S. J. 1988. Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association 83*, 403, 596–610. (p. 35)

Downey, C. and Sanner, S. 2010. Temporal difference bayesian model averaging: A bayesian perspective on adapting lambda. *Proceedings of the 27th International Conference on Machine Learning*. (p. 43)

Enzenberger, M., Müller, M., Arneson, B., and Segal, R. 2010. Fuego - an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games 2*, 259–270. (pp. 2, 60)

Gelly, S. and Silver, D. 2011. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence 175*, 1856–1875. (pp. 1, 16, 18, 19, 22, 23, 29, 45, 49, 50, 51, 55, 60, 61, 62)

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. 2006. *Modification of UCT with Patterns in Monte-Carlo Go*. Technical Report 6062, INRIA. (pp. 2, 18, 24, 46, 60)

Kocsis, L. and Szepesvári, C. 2006. Bandit based monte-carlo planning. *Machine Learning: ECML 2006 4212*, 282–293. (p. 18)

Marsland, T. A. and Schaeffer, J. 1990. *Computers, Chess, and Cognition*. Springer-Verlag. (p. 1)

Rasmussen, C. E. and Williams, C. K. 2006. *Gaussian Processes for Machine Learning*. The MIT press. (p. 30)

Sherman, J. and Morrison, W. J. 1950. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics 21*, 124–127. (p. 34)

Spight, W. L. 2001. Extended thermography for multiple kos in Go. *Theoretical Computer Science 252*, 23–43. (p. 5)

van der Werf, E. C., van den Herik, H. J., and Uiterwijk, J. W. 2005. Learning to score final positions in the game of Go. *Theoretical Computer Science 349*, 168–183. (p. 5)