A Survey of Monte-Carlo Techniques in Games
Master's Scholarly Paper

Prahalad Rajkumar, bridgenbarbu@gmail.com

Abstract

The way computer programs play strategy games is quite different from the way humans play. In perfect-information games like chess and checkers, a game-tree search is the core technique in a computer program's arsenal, augmented by good evaluation functions and clever secondary strategies. In other perfect-information games such as go and clobber, there is very little intuition as to how good a position is, and consequently constructing a good evaluation function is not easy. Furthermore, go has a high branching factor. It turns out that Monte-Carlo simulations, i.e. producing repeated random samples and considering their average in making a decision, work surprisingly well in these games. In imperfect-information games such as bridge and scrabble (the latter game has inherent randomness associated with it as well), Monte-Carlo simulations once again turn out to be useful. This paper examines the use of Monte-Carlo simulations in bridge, scrabble, go, clobber, and backgammon, and reports on how this technique impacts each of these games.

1. Introduction

The way computer programs play strategy games is quite different from the way humans play. Humans apply logical reasoning to make the best possible play. In perfect-information games like chess and checkers, a game-tree search is the core technique in a computer program's arsenal, augmented by good evaluation functions and clever secondary strategies. However, brute force alone is not sufficient for a variety of reasons. In perfect information games such as chess, checkers, and go, the game tree grows exponentially, thereby constraining the usage of brute force techniques such as alpha-beta search. In imperfect information games such as bridge and scrabble, the missing information renders it impossible to build the game tree in order to perform a brute-force search. A variety of techniques are used to counter this encumbrance. In other perfect-information games such as go and clobber, there is very little intuition in evaluating how good a position is, and consequently constructing a good evaluation function is not easy. It turns out that Monte-Carlo simulations, i.e. producing repeated random samples and considering their average in making a decision, work surprisingly well in these games. In imperfect-information games such as bridge and scrabble (the latter game has inherent randomness associated with it as well), Monte-Carlo simulations once again turn out to be useful. This paper examines the use of Monte-Carlo simulations in bridge, scrabble, go, clobber, and backgammon, and reports on how this technique impacts each of these games.

Game	Performance against humans
Chess	(Deep Blue) Better than the best human players
Checkers	(Chinook) Better than the best human players
Backgammon	(TD-Gammon) Better than or equal to the top human players
Bridge	(Bridge Baron) Above-average caliber; worse than the best human players, but able to play many complex bridge hands to perfection.
Go	(MoGo) Defeated a human professional in a 9x9 go game with a 9 stone handicap
Scrabble	(Maven) Better than the best human players
Clobber	Unknown

Fig 1. The caliber of computer programs in comparison to humans. The games shown in bold are discussed in this paper.

The remainder of this paper is organized as follows. Section 2 talks about the game of contract bridge in detail. Section 3 examines the anatomy of the scrabble program MAVEN [6]. Section 4 deals with the issues in computer go and the success of programs making use of Monte-Carlo techniques. Section 5 covers the fledgling game of clobber. Section 6 briefly examines a Monte-Carlo approach in Backgammon, while section 7 concludes this paper.

2. Bridge

Bridge is played between two partnerships with a standard deck of cards, each player being dealt 13 cards. There are two phases in the game, the auction (bidding) and the card play. In the auction, players bid for the final contract, i.e. the number of tricks that a partnership contracts, along with the trump suit. If for example one partnership contracted to score 12 tricks with spades as trumps, in the card play phase, this would precisely be the goal of this partnership – while the goal of the opposing partnership would be to take two tricks, and defeat the contract. In the initial rounds of bidding, the bids made are seldom suggestions for a final contract, but rather have the purpose of conveying information about one's hand to his partner (the opponents are entitled to this information as well).

Both human players and computer programs need to have a *bidding system* that assigns meanings to bids. A typical bridge program consists of a large database of bidding rules, thereby assigning meanings to a variety of bidding situations. While a computer program's bidding is largely based on rules, close decisions are made by performing simulations on the candidate set of bids [5]. Consider a situation where your partnership contested up to 4♥, an opponent bids 5♦ as a sacrifice and the next two players pass. The available choices are doubling the

opponents, and bidding 5♥. A bridge program in this situation would project the remainder of the auction for the candidate actions, i.e. double and 5♥. For each of the final contracts (almost certainly 5♦ doubled, and 5♥), the program would deal out many random hands and simulate the card play. The average of the scores obtained on the simulations performed corresponding to each bid is a predictor of how good the bid is. Consequently, the bid that obtains the highest average score is the one that will be chosen by the program.

2.1 Advantages of the Monte-Carlo Approach

The approach that is based on Monte-Carlo simulations has some advantages over the way humans approach the game:

1) An advantage of adopting a statistical Monte-Carlo approach is that computers have no notion of intuition. Consider the following deal from [1]:

♠ 8 6 5 2
♥ K J 2
♦ A J 5
♣ K 6 4

♠ 7
♥ A Q 5 4 3
♦ K 4 2
♣ A Q J 5

South is the declarer in 6♥. West leads the ♠A and continues with the ♠K, ruffed by South. The only way to succeed on this deal is to execute a dummy reversal, i.e. take three spade ruffs in hand, and use the hand with the fewer trumps (dummy) to draw the outstanding trumps. Humans are taught to ruff in the hand containing the longer trumps, and a dummy reversal is therefore counterintuitive. Bridge programs are not taught such principles, and Bridge Baron was able to make the right play with relative ease.

2) When the bidding provides sufficient information, programs will make the right play with a high success rate. This is because, with accurate information, simulations produce game trees that closely depict the actual hand, and therefore the play suggested by the simulation is likely to be the correct play. As an example, Bridge Baron executes complex plays such as the three-suit strip-squeeze, and the esoteric one-suit squeeze to perfection, while most humans would have trouble with these deals.

3) Continuing on the issue of computers playing squeeze deals, there is another point to consider. When there is sufficient information to construct the missing hands, we saw above that computers would then solve the deal with ease. Even when there is not sufficient information to construct the missing hands, a Monte-Carlo approach yields very good chances to succeed. In simulating several hands, the program would observe that the contract cannot be made on most layouts, and may come across a particular layout where the double-dummy analysis reports that the contract can indeed be made. The program would duly adopt that play, and execute the squeeze. Intermediate human players would simply give up on such deals. Expert players recognize such positions correctly, and reason that a particular lie of the opponents' hands is required for the squeeze to succeed. Having made that assumption, they execute the squeeze.

2.2 Disadvantages of the Monte-Carlo Approach

The approach based on Monte-Carlo simulations has certain inherent disadvantages.

1) Consider the following example, along the lines of the one suggested by Smith and Rosenfeld in [2].

♠ A K 5 3
♥ A 4
♦ 6 5
♣ A Q J 10

♠ 6 4 2
♥ K 7 6 2
♦ 9 7 4 3 2
♣ K

West	North	East	South
Pass	1♣	Pass	1♠
Pass	3♠	Pass	4♠
All pass			

The auction is as shown, and West leads the ♦A followed by the ♦K, declarer follows both times, and leads the ♥10, declarer plays low from dummy East's king wins the trick. Now, human players in the East position would simply exit with a heart or a trump, and sit back to win a trick with the ♣K when declarer makes the normal play of taking the club finesse. A computer program in the East position would believe that declarer is omniscient and will play the ♣A to drop the ♣K. Consequently, believing that there is no way to defeat the contract the computer

program may choose to play a card at random, which might turn out to be the ♣K! This type of shortcoming can be rectified by adding appropriate rules to the basic engine.

2) Plays are made at an earlier stage, with the program believing that it would get the play right at a later stage. Consider the very first deal from [1].

♠ A 5
♥ A 8 5 2
♦ A K 10 8 5 3
♣ 4

♠ K Q J 10 4
♥ K 10 4
♦ J 7
♣ A 6 3

South is the declarer in 6♠ and West leads the ♣Q. The key play should be made as early as trick two – declarer must cash the ♦A and follow up by leading a low diamond off of dummy. This is a safety measure, catering to the scenario where both spades and diamonds divide 4-2. Bridge Baron did not find the right play on this deal. Bridge Baron won the ♣A at trick one, ruffed a club low at trick two, cashed the ♦A at trick three, unblocking the jack from hand, cashed the ♠A at trick four, then played a heart to the king and drew the outstanding trumps. When diamonds divided 4-2, Bridge Baron could not recover.

Bridge Baron went wrong because it did not play based on a global plan. It kept delaying its decision on how to play the hand, until it forced itself to play for diamonds being 3-2. On this deal, with several possible options available (e.g. the diamond finesse, dropping a singleton or doubleton ♦Q, hearts being 3-3 and pitching a club loser on the thirteenth heart), Bridge Baron doesn't understand that its plays are eliminating some of its chances. Instead, as its declarer play is based on Monte-Carlo simulations, it "knows" it will always go right later in the hand, and therefore considers all of its plays to be essentially equal. In other words, a series of technically "double-dummy correct" plays forced Bridge Baron into an inferior and single-dummy incorrect line.

3) The Monte-Carlo approach has the disadvantage of not being able to take inferences from an opponent's play. There is a heavy psychological component in bridge that computer programs don't consider. Take the following deal:

♠ A K 5 3
♥ K J 7 2
♦ 6
♣ A Q J 10

♠ Q J 10 9 2
♥ 5 3
♦ A K
♣ K 7 6 2

West leads the ♦Q against 6♠. Declarer needs to guess hearts to make the slam. A human declarer might lead a heart at trick two, putting the West player to the test. Few players in the West seat would duck smoothly, for if East can produce a trick, it is correct to win the ♥A. If West is known for his coffeehousing skills, a hesitation would imply that he *does not* have the ♥A. A computer program cannot make use of this kind of psychological inferences and has to rely on a successful guess – a 50% chance. Simulations would not help in this deal, for on roughly half of the hands West would have the ♥A and on the remaining half of the hands East would have the ♥A; similarly, half the hands would show West to hold the ♥Q and half the hands would show East to have the ♥Q. It is worth noting that the discovery play mentioned on this deal is a psychological one – bridge programs have difficulty executing even straightforward discovery plays [1], where the location of one honor provides information about another key honor. Bridge programs "know" where each card is located, and therefore do not try to seek this information.

4) Similar to the previous point, a program using a Monte-Carlo approach cannot make discovery plays. Consider the following deal:

♠ A Q J T 9 3 2
♥ A 7 4
♦ 6 4 3
♣ -

♠ 8 6 5
♥ K Q J 5
♦ 7 5 2
♣ K 5 3

West opens a 12-14 1NT, after which South becomes declarer in 4♠. West cashes the three top diamonds, East following all three rounds. West then switches to the ♥2, East playing the ♥9, South winning with the ♥K. Now, an expert human declarer would lead the ♣K. If West plays the ♣A, that give him 13 points (9 in diamonds, and 4 in clubs), and there is no room in his hand to hold the ♠K, and therefore the correct play is a spade to the ace, hoping that East holds the singleton ♠K. If West follows smoothly with a low club, then there is no reason to reject the spade finesse, so declarer will come back to hand with a heart and run the ♠8. Using Monte-Carlo simulations, such discovery plays are not feasible. It is to be noted that the discovery play mentioned in this deal is of a psychological nature. Computer programs are not capable of making discovery plays that are straightforward, such as finding out the location of a non-critical ace, to infer the location of a critical honor [1]. This is because, using a Monte-Carlo approach, the computer program "knows" where the missing cards are, and hence does not try to discover any new knowledge.

3. Scrabble

Scrabble is one of the games where a computer program has achieved supremacy over human players. MAVEN [6] is a program by Brian Sheppard, that can play close to perfection, and is better than the best human players. Scrabble is an imperfect information game. It is played on a 15x15 scrabble board, with each player having a rack that can hold seven tiles. Each player plays a word during his turn, and earns a score based on the values of his tiles along with possible bonuses that the board provides (i.e. double-letter score, double-word score, triple-letter score and triple-word score).

3.1 Overview of Maven

The foremost component of MAVEN is its comprehensive vocabulary, comprising of all the words in the Official Scrabble Player's Dictionary (OSPD). In order to enable effective move generation, the dictionary is represented as a Directed Acyclic Word Graph (dawg). Then comes the aspect of move generation – MAVEN's move generator is responsible for generating and each move and scoring it. Scoring moves is important, but equally important is to score the rack that is left after a move is made, and MAVEN's rack evaluator performs precisely that function. Then come search and evaluation, the bread and butter of most games, scrabble being no exception. MAVEN uses different search algorithms, one for the normal game, one for the pre-endgame, and one for the endgame. Each stage of the game has different goals – in the normal game, the goal is to strike a balance between maximizing the score for the current move, and leaving a balanced rack for future moves; the pre-endgame arises when there are 16 unseen tiles, and the goal here essentially is to prepare gracefully for the endgame. The endgame goal is to try and play off as many tiles as possible, particularly high-scoring tiles, and in the ideal scenario, try and play off all the tiles. The most significant tool in MAVEN's arsenal is arguably

its use of simulations, which go hand in hand with searching. The next section discusses simulations in detail.

3.2 Simulations

Sheppard received the inspiration to use simulations in MAVEN by observing scrabble expert Ron Tieckert analyze a particular position. Dealt the opening rack AAADERW, conventional wisdom pointed towards AWARD, a choice that MAVEN agreed with. Tieckert however believed that AWA was a better move, for it left behind a balanced rack, and it also did not open up double-word squares. He tested his hypothesis by playing out fifty parallel games, half of them starting out with AWA and the remaining starting with AWARD, observing that AWA produced better results than AWARD. Sheppard was inspired by the simulation performed by Tieckert, and incorporated this technique into MAVEN. The obvious next question is, up to what search depth should simulations be performed? In general, MAVEN performs simulations only up to two plies, because an average scrabble move takes 4.5 moves, therefore after two plies there would typically be no tiles left behind in the rack.

4. Go

Go is a perfect-information game. The standard size of a go board is 19x19. 13x13 and 9x9 are also possible sizes for non-tournament play. While it may appear that a 9x9 go board should be comparable to chess, that turns out not to be the case. There are two main reasons that contribute to this:

- Constructing an evaluation function for go has been very difficult. In chess, there are a variety of parameters that could be used to write an evaluation function. For starters, the simplest evaluation function could be based on the value of each piece – if white has a bishop more than black but a rook less, then black has an advantage over white. Other bonuses could be considered as well – having a passed pawn is an advantage; having a doubled pawn is a disadvantage; if a pawn is in the seventh rank and is about to queen, that is an enormous advantage; if the pieces control the center, that would be an advantage, etc. In go however, there is no apparent intuition that can contribute to a formal notion of an evaluation function.

- The high branching factor in go, in comparison to chess and checkers, is the other key reason contributing to the difficulty in computer go.

The foundation for Monte-Carlo go was laid by Bruügman [8], who used simulated annealing in his go-playing program gobble. A description of Bruügman's work is described in the appendix. Bruügman's work initiated several enhancements to Monte-Carlo go, including shallow and selective global tree search [10], applying reinforcement learning in addition to using Monte-Carlo simulations [11], combining Monte-Carlo based go with the UCT algorithm [11] based on

the multi-armed bandits problem from game theory [13]. MoGo [12] is a Monte-Carlo go program that uses UCT algorithm along with Monte-Carlo simulations. The next few sections discuss MoGo and the UCT algorithm.

4.2 The Multi-Armed Bandit Problem

The multi-armed bandit problem is based on a slot-machine, with multiple levers. Each lever (arm) has a probability distribution associated with it. The objective is to maximize the total earnings through iterative plays. The factor of interest in such problems is to strike a balance between knowledge that has already been acquired, and exploring new arms to further increase knowledge. Note that the probability distribution associated with the levers is unknown at the outset.

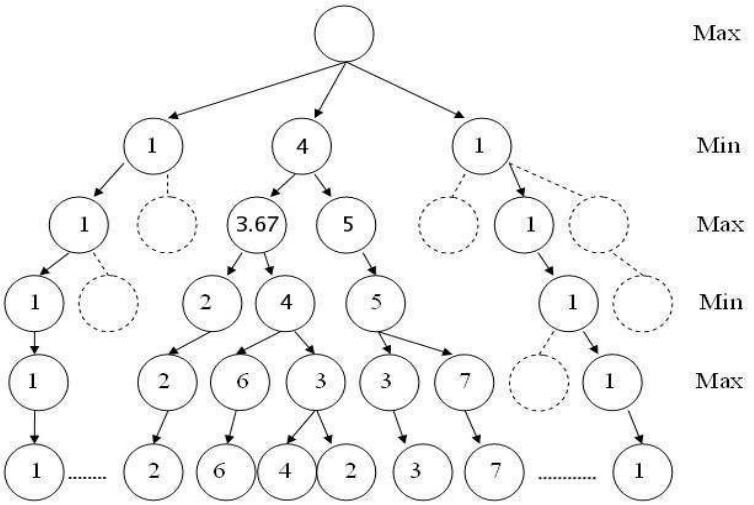

Fig 2. A UCT search highlighting the asymmetric growth of the tree. The value of each node is the arithmetic mean of the value of each child weighted by the frequency of visits.

4.3 The UCB1 Algorithm

Initialization – Play each arm once

Loop – Play arm *j* that maximizes the formula $\overline{X}_j + \sqrt{\frac{2\log n}{T_j(n)}}$

where *n* is the overall number of plays so far, T_j is the number of times arm *j* has been played after the first *n* plays. Note that $\overline{X}_i = \overline{X}_{i,T_i(n)}$ and $\overline{X}_{i,s} = \frac{\sum_{j=1}^{s} X_{i,j}}{s}$.

UCT is the extension of UCB1 to the minimax setting. The concept involves considering each node as an independent bandit, with its child node as independent arms. It plays a sequence of bandits each time.

4.4 Salient features of the UCT Algorithm

The UCT algorithm can stop at any stage, and still yield a good estimate of the game-tree. Another advantage of UCT is that when one child node has a much higher value than others, this node would be explored more often than other nodes. Finally, the tree grows asymmetrically so as to explore the good moves in depth, as shown in Fig. 2.

4.5 Application of UCT to go

MoGo comprises of two parts, the tree search and the Monte-Carlo simulations. Each node of the tree represents a go position, and children nodes represent the position after a corresponding move has been made. The UCT algorithm is applied on the basis that each node (position) is a bandit problem, with each move being an arm with unknown reward, associated with a probability distribution. In the go scenario, there are only two types of arms, winning arms and losing arms, with rewards 1 and 0 respectively. It is to be noted that there is only one tree that is stored in the memory at a given stage. During each simulation, MoGo begins at the root of the tree. The key idea is that it tries to get down to the bottom of the tree and evaluate the position by performing random simulations. To get to the bottom of the tree, the following steps are done:

- At each node, MoGo selects a move based on the UCB1 algorithm.
- If the position arrived by making the move exists, then the program descends to the child node corresponding to the move, and once again selects another move based on the UCB1 algorithm.
- This process stops when the position arrived by making a move does not exist in the tree – in that case, a new node comprising this position will be created. Note that this will be a leaf node.

Monte-Carlo simulation in go typically consists of playing random games, with no go knowledge. The authors explore the possibility of associating go knowledge to simulations. Using 3X3 go patterns, the authors claim that MoGo simulations are likely to produce more meaningful games in comparison to performing entirely random simulations. The main disadvantage of this approach is that the effect of considering 3X3 go patterns does not produce a global benefit. In particular, the 3X3 patterns are obtained from the last played move, to benefit simulations to aid the next move

4.6 Success of MoGo

In August 2008, MoGo defeated professional go player Kim Myungwan in a 9 stone handicap game [14]. This was the first occasion in which any go program has defeated an expert go player – a significant progress from the scenario where even go programs would routinely defeat intermediate players. Myungwan had high praise for MoGo, saying that the program's level was around two dan to three dan, and it made five dan moves on occasions. Myungwan also mentioned that whenever he played aggressively, MoGo played safely, going for the safest route to victory rather than try to maximize the extent of the victory.

5. Clobber

Clobber is a board game created by Albert et al. [15]. The rules of clobber are very simple - the game is played on a chessboard, with all the squares in the board filled with pieces; white squares have white pieces and black squares have black pieces. A move is made by capturing an adjacent opponent piece, "clobbering" it off the board. The game ends when no more moves can be played, and the player who makes the last move wins. Like go, there is no good intuition as to what an optimal strategy should be in clobber, which makes it an ideal candidate for a Monte-Carlo approach.

Albert, Grossman, Nowakowski, and Wolfe [15] have proved that determining who wins from a clobber position is NP-hard. Therefore, a tree-search alone is not sufficient to play clobber. Furthermore, there is no inherent intuition as to what qualifies to be a good position in clobber, rendering it hard to produce a good evaluation function. Given these factors, the Monte-Carlo approach playing random games and arriving at a move that works well on a majority of the games appears to work best. In the 2006 World games Olympiad, Clobber had two entries – MILA and ClobberA, which used Monte-Carlo simulations throughout the game [16]. Willemson and Winands, programmers of MILA, concur that Monte-Carlo methods work best in the middle game. Indeed, the performance of ClobberA in the middle game is of pretty high standard. Willemson and Winands aver that MILA routinely defeats ClobberA in the endgame – an indication that basing an endgame strategy entirely on Monte-Carlo simulations is a mistake.

6. Backgammon

Backgammon is one of the games where a computer program is equal in caliber to humans. TD-gammon [17], a program by Gerald Tesauro, has had the distinction of ranking alongside the top three human experts. TD-gammon consists of a neural network that trains itself to be an evaluation function by repeatedly playing against itself. In essence, the neural network is learning from the outcome of playing against itself. TD-gammon starts out with a random initial strategy, plays out several games using a neural network, and at the end of each game, it revises its neural network weights. To evaluate the value of a position, TD-gammon makes use of a

technique called rollout, which is based on a Monte-Carlo approach. A rollout is a Monte-Carlo evaluation of a position, where a computer plays a position to completion several times, using different random dice rolls. The rollout score is the average of the outcome of all trials. The rollout scores of all candidate moves are compared against each other, to determine what the best move is. This information is used to update the weights of the neural network.

In [18], Tesauro and Galperin study the impact of Monte-Carlo simulations in backgammon. They consider three base players, Lin-1, Lin-2 and Lin-3, all of which are single-layer neural network. Lin-1 is a single layer neural network with only the raw board description. Lin-2 is Lin-1 along with some random noise, added intentionally to weaken its capacity. Lin-3 is built on Lin-1, with additional features and no extra noise, making it to be the strongest candidate. Corresponding to each of these three base players, corresponding Monte-Carlo players are created. For example, the Monte-Carlo player corresponding to Lin-1 would use Monte-Carlo simulations in addition to the base capability of Lin-1.

Network	Base Player	Monte-Carlo Player
Lin-1	-0.52 ppg	-0.01 ppg
Lin-2	-0.65 ppg	-0.02 ppg
Lin-3	-0.32 ppg	+0.04 ppg

Fig 3. The performances of the base players and the corresponding monte-carlo players of three networks

Tesauro and Galperin performed evaluations on these six players - three base players and the corresponding Monte-Carlo players. The results of the experiments showed that the Monte-Carlo players produce considerable improvement over the base players (Fig 4). The performance of the base Lin-1 network was -0.52 points per game (ppg), while the corresponding Monte-Carlo player had a score of -0.01 ppg. Lin-2's base player had a score of -0.65 ppg, and the corresponding Monte-Carlo player's score improved to -0.02 ppg. Finally, Lin-3's Monte-Carlo player produced a score of +.04 ppg, which is better than TD-Gammon 1-ply, in comparison to its base player score of-0.32 ppg. These results strongly suggest that using truncated Monte-Carlo rollouts can significantly aid in improving a backgammon program.

7. Conclusion

As mentioned in the introduction, Monte-Carlo simulations are useful in games where there is missing information (bridge, scrabble), an element of chance (backgammon), or perfect information games where there is no intuition of evaluating a given position (e.g. go, clobber).

This study shows that Monte-Carlo simulations play an integral part in bridge, go and scrabble. In bridge, Monte-Carlo simulations serve as the bread-and-butter technique, over which other techniques are added, and the end product is a competent bridge program. The best bridge

programs at the moment are no match against expert bridge players, which might be the case for at least a few more years.

Using Monte-Carlo methods to enhance the skill of backgammon programs remains highly promising, as shown by Tesauro and Galperin. Today, with the increase in computing power, it is worth conducting further research in this direction.

In bridge, addressing the disadvantages outlined in section 2.2 is an essential first step to produce a champion-caliber program. In go, while the progress shown by MoGo in defeating a professional go master is highly encouraging towards the pursuit of producing an expert program, this effort was achieved with a 9 stone handicap. Nevertheless, this is the first occasion in which any go program has defeated an expert go player – a significant progress from the scenario where even intermediate players would routinely defeat the strongest go programs.

In clobber, the difficulty lies in constructing a good evaluation function, and Monte-Carlo simulations proved to work out well. The caveat here is that Monte-Carlo simulations did not work well in the endgame. This was demonstrated by the match between MILA and ClobberA - ClobberA based its endgame

8. Acknowledgements

I thank Dr. Clyde Kruskal not only for his supervision and advice on this paper, but for his encouragement and inspiration throughout the period I spent at University of Maryland, College Park. I would also like to thank Dr. Stephen Smith for giving me his perspective on Monte-Carlo approach in games.

9. Appendix: History of Monte-Carlo go

9.1 Introduction

Bruügman [8] was the first researcher to apply Monte-Carlo techniques to go. He developed a program called *gobble* that uses Monte-Carlo simulations to play go. Deriving inspiration from nature, he used simulated annealing to play go games to remarkable effect. Today, many go programs make use of Monte-Carlo simulations, but the use of simulated annealing is very rare.

9.2 Simulated Annealing

As a motivating example from physics, Bruügman considers a metal in its transition from a liquid state to a solid state. He defines a configuration in this scenario by giving the position and velocity of each atom. At high temperatures, the atoms move randomly. However, as the metal cools, the atoms move less, and once the metal attains the solid state, the atoms cease to move. This process is called annealing.

Monte-carlo simulations are simulations of evolution of a statistical system (i.e. annealing) using a computer program. First, for the statistical system in consideration, a description of all possible configurations should be chosen. Then, a set of moves for the configurations should be arrived at. Finally, moves are chosen at a random for a given configuration. Based on whether the move increases or decreases the action in the system, a score is assigned to the move. This process is repeated, with "cooling" taking place over time.

4.3 Applying simulated annealing to Monte-Carlo go

Bruügman observes that since simulated annealing provides a good approximate solution to combinatorial hard problems such as the traveling salesman problem, it is tempting to try and use it to solve tree search, which is also a combinatorial hard problem. He however points a couple of problems in doing so. First, the game tree of go is large even if only two moves are considered at each level. Also, some patterns are good because they have been proven so in earlier games, and therefore go knowledge seems essential. Most significantly, game tree search has a key difference from problems such as the traveling salesman problem in that there are two competing players playing a game. Therefore, any local change in the order of moves made by one player has a non-local effect on the game tree. Simulated annealing therefore cannot be used for game-tree searches.

Simulated annealing, however, can be used in a different role. Bruügman makes an important observation that there are some moves which are good, no matter when they are played. This leads to the following strategy for playing random games: each player decides ahead of time the order in which he plays his moves. A game is played to completion using the above set of moves – if a given move cannot be made, the next move in the list is played. Several such games are played, and each move is assigned the average value of all the games in which it occurred.

Bruügman's simplistic approach proved surprisingly effective, and paved the way for many other Monte-Carlo go programs. Bruügman tested gobble against Many Faces of Go, the North American computer go champion at that time. The two programs played twenty games against each other, playing at various handicaps, and Gobble won 13 out of the 20 games.

References

[1] Jason Rosenfeld and Prahalad Rajkumar. *A Computer's Twist: Play Problems for you with Bridge Baron Analysis.* Great Game Products, 2007.

[2] Jason Rosenfeld and Stephen J.J. Smith. *How computers play bridge.* http://www.greatgameproducts.com/articles/comp_play_bridge_article.html.

[3] Stephen J.J. Smith, Dana S. Nau, and Tom Throop. Total-order multi-agent task-network planning for contract bridge. *Proceedings of the Thirteenth National Conference on Artificial Intelligence,* Stanford, California, 1996.

[4] Stephen J.J.Smith, Dana S. Nau, and Tom Throop. Computer Bridge: A Big Win for AI Planning. *AI Magazine, 19(2):93-105,* 1998.

[5] Matthew Ginsberg. GIB: Steps toward an expert-level bridge playing program. *Proceedings of IJCAI-99, pp.584-589,* 1996.

[6] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134:241-275, 2002.

[7] David Lichtenstein and Michael Sipser. Go is polynomial-space hard. *Journal of the ACM, 27, (2), pp. 393-401,* 1980.

[8] Bernd Brügmann. Monte Carlo Go. 1993, www.joy.ne.jp/welcome/igs/Go/computer/-mcgo.tex.Z

[9] Bruno Bouzy and Tristan Cazenave. Computer Go: An AI Oriented Survey. *Artificial Intelligence*, 132: 39-103, 2001.

[10] Bruno Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9X9 go. *Proceedings of 4th Computer and Games Conference,* 2004.

[11] Bruno Bouzy and Guillaume Chaslot. Monte-Carlo Go Reinforcement Learning Experiments. *In IEEE 2006 Symposium on Computational Intelligence in Games,* 2006.

[12] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go.

[13] Peter Auer, Nicol`o Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. 1995

[14] *Supercomputer with innovative software beats Go Professional.* http://www.cs.unimaas.nl/g.chaslot/muyungwan-mogo/

[15] Michael H. Albert, J. P. Grossman, Richard J. Nowakowski, and David Wolfe. An Introduction to Clobber. *Integers: Electronic journal of combinatorial number theory 5(2),* 2005.

[16] J. Willemson and Martin H. M. Winands. MILA wins Clobber tournament. *ICGA Journal, 28(3): 188-190,* 2005.

[17] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence 134, pp. 181–199,* 2002.

[18] Gerald Tesauro and Gregory R. Galperin. On-line policy improvement using Monte-Carlo Search. In *Advances in Neural Information Processing Systems, 1068-1074,* Cambridge, MA, 1996.