Playing Go With a Computer

John M. Hall

December 10, 2003

Abstract

Ever since a computer beat the world champion at chess, the next target has been a game called Go. Go has very simple rules but a very complex game-play. In addition, it seems that Go favors skills that humans are naturally good at. This report presents the fundamentals of Go. It also describes historical game playing methods. Finally, this report discusses several recent advances in artificial intelligence that have been applied to Go. These include improved search tree methods as well as move generation methods. As computer Go programs advance, there is every reason to believe that within the next 20 to 30 years, a computer will beat the world champion at Go.

1 Introduction

The game of Weiqi is believed to develop in China over 4000 years ago [2]. This game is known as Go in the western world. The rules of Go are deceptively simple, but game play quickly becomes quite complex. In addition, the game favors skills such as pattern recognition that humans are quite good at. As a result, this is a difficult game for computers to play. In fact, today's most advanced computer Go programs only play about as well as an average

amateur player [4]. In order to understand the current artificial intelligence work, it helps to understand the basics of Go. A brief introduction to Go is included in Appendix A. For more in-depth introduction, see [2].

1.1 Computer Game Playing

Games are quite often used to test new artificial intelligence concepts. To paraphrase a list of reasons from [8]:

- Games have clearly defined rules.
- Games have a clear concept of winning and losing.
- Games are available in many difficulties.
- There are many human experts to compare results with.

I would add to this list that games are also more fun to investigate than many other theoretical constructs.

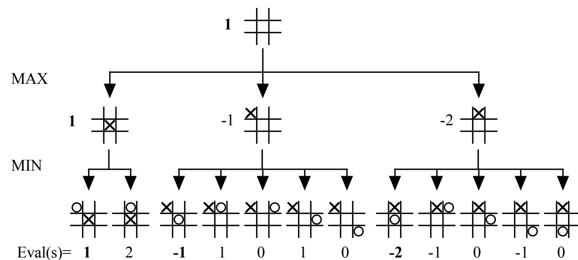


Figure 1: First Two Levels of the Min-Max Tree for Tic-Tac-Toe

Currently the core algorithm used in game playing is the min-max search. The idea of a min-max search is to look at all the possible moves for the

current player. In order to evaluate each of these moves, we find the best move available to our opponent for each of our potential moves. This cycle continues in a tree fashion until either the end of a game or a maximum depth is reached. These leaf nodes are assigned a numeric value that represents how good they are. As we work back up the tree, we take the best move available to us and assume that the other player takes the best move available to them. In a sense, this is a depth-first search where we are trying to find the move that will lead us to the best outcome assuming that our opponent is trying to do the same. Figure 1 shows the min-max tree for tic-tac-toe to a depth of 2 ignoring symmetries. The time complexity of a min-max search is $O(b^d)$ where b is the branching factor or number of choices available to each player, and d is the depth the search is performed to.

The performance of a min-max search can be improved by incorporating $\alpha\beta$ pruning. This works by recognizing that once the value of one sub-branch of a node is either large enough or small enough, then the parent of the node will not choose the node no matter what the other sub-branches evaluate to. As a result, there is no need to evaluate them. In the best case, we will select the optimal choice first each case and the time complexity will be $O(b^{\frac{d}{2}})$. If we choose the optimal choice last, the time complexity is the same as the original min-max search since all nodes are evaluated. However, in most cases, neither of these extremes is realistic. The average performance of a min-max search with $\alpha\beta$ pruning is $O(b^{\frac{3d}{4}})$.

Min-max search in general has several weaknesses including the horizon effect, large errors from the evaluation of unstable positions, and the fact that most of the states reached are obviously terrible solutions [9]. Despite this, min-max search with $\alpha\beta$ pruning is the basic mechanism used in most current game playing algorithms. For example, IBM's Deep Blue used these concepts to play chess. On May 11, 1997, Deep Blue beat the human world

chess champion [8].

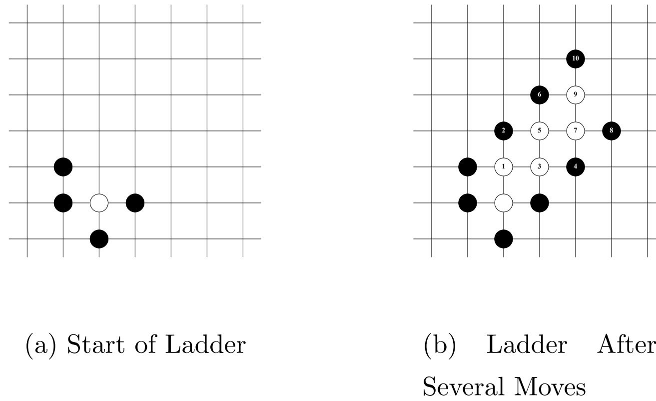
1.2 Computer Go

This discussion is based on the brief history of computer Go programs in [4]. The first computer program to play Go was written in 1960 by David Lefkovitz. Nine years latter, Al Zobrist wrote the first program that beat an actual human player. The player was an absolute beginner. Today, the best Go playing programs rank about the same as an average amateur.

Since computers have beaten the best chess player in the world, it is logical to ask why the same methods have not been successful with Go. The three main reasons why min-max search using $\alpha\beta$ pruning have not been successful are that there are a large number of possible moves for each player, the game is long, and it is difficult to develop an evaluation function for Go [10]. This results in large values of b and d in the above time complexity formulas. The entire game tree complexity for chess contains approximately 10^{123} nodes, but the game tree for Go contains 10^{360} nodes [1, 8]. This may not be a completely fair comparison, but it does show why Go is more complicated than chess.

As discussed above, players of moderate skill can beat the best computer Go programs available. In fact, human players tend to learn and exploit the weaknesses of these programs [4]. On example of this was in 1999 when Müller played the program the Many Faces of Go. After giving the computer a 29 stone handicap, Müller managed to win by 6 [4, 11]. When you consider that the difference in levels between ranked amateurs amounts to about a one stone handicap, this makes the computer player look quite bad.

In fact, it is more than just the added complexity that make computer Go a difficult problem. Humans beat computers about as easily on a 9×9 board even though such a game has a complexity roughly equivalent to chess [4].



(a) Start of Ladder

(b) Ladder After Several Moves

Figure 2: A Simple Ladder

There are many other reasons why Go is such a difficult game for computers. A list of some reasons is show below:

1. Go has a large branching factor
2. There is no known reliable evaluation function [4, 11]
3. Humans can often see many moves ahead as in a ladder (See figure 2 for an example of a ladder)
4. Computers are not as good at recognizing patterns as computers
5. In games of Go, the time between moves is typically short
6. It is hard for computers to group related pieces
7. It is hard to identify live and dead groups
8. Patterns in Go may be very large
9. Non-repeating board state rule makes states more complex [11]

10. It is difficult for a computer to know when the game is over [10]

11. It is hard to deal with repetition locally [10]

These other issues, particularly the difficulty in evaluating a state, are so significant that research has not yet been able to relate program strength to the depth of the search [11].

2 Research

The current Artificial Intelligence research that is applied to computer Go tends to deal with finding more effective min-max search techniques. In most cases, these techniques are used to further prune the search tree. In many of these cases, the techniques were first used in other games. There has also been a fair amount of research into move generation techniques. These techniques limit the branching factor by focusing on a small set of reasonable moves and ignoring the rest. This section will discuss three improvements that have been made to the basic min-max search process as well as two methods that may be used for move generation.

Probcut, introduced in [5], uses a linear probability function to determine if a deeper search from a given node would improve the game. This method was originally designed for the board game Othello. The probability distribution function is determined by running a shallow search from the questionable node. In a sense, this method may be considered as simply a new version of selective evaluation. One interesting note from this paper is that the true probability function closely matches the linear model used. One other advantage to this approach is that like the min-max and $\alpha\beta$ pruning algorithms, probcut does not require extra memory.

A more selective search algorithm known as B* search is introduced in [3]. This method was originally designed for chess. This method also uses a linear

probability function to determine which nodes to expand. Like the A* search, which is presumably where the search gets its name, a B* search saves the current state of those nodes that it is currently in the process of expanding. The search then focuses on the state that is most likely to get the search to a stopping point. Like the probcut method, B* uses a linear probability density function. This method has the interesting effect of it being possible to decide on a move without ever expanding the opponents options from that state. In general, this focused search greatly reduces the number of foolish branches evaluated.

An approach called Partial Order Bounding is introduced in [9]. This paper points to the fact that there is no obvious way in go to evaluate a state with a single number. In general, they point out that any method that does use such a utility function loses information about the state. Instead, in many problems such as Go, it make more sense to evaluate a state relative to another state. At the same time, this paper points out that it is not feasible to carry such comparisons up the evaluation tree. Instead, the search is done to try to meet a certain bound. If this succeeds, a true value is propagated with other information. If this fails, a false value is propagated and the bounds are adjusted. For the Go example, this paper gives an example where they are trying to evaluate the result of a relatively small group. The first question is if the current player can win in this situation. If that goal can not be met, the next goal is to find a way not to lose in the situation. The real strength of this method is that it is much easier to create a comparison heuristic than one that maps to a single value.

Despite these approaches to searching, it should be clear that the average branching factor of 250 [8] in Go makes a complete tree evaluation infeasible. As a result, most methods limit each step of the search to a few number of moves. The process of choosing the moves to explore from a given states is

known as move generation.

The results in citechurchill-new, cant-using attempt to use “soft AI” approaches to generate the proposed moves from a given Go state. In these papers, they propose using neural nets to perform move generation. They then use “hard AI” approaches to evaluate those moves. In this example, they mean this to mean a min-max search with $\alpha\beta$ cutoffs. They show that the min-max search improves the strength of the player. However, the results did not seem quite mature.

The method proposed in [8] propose defining rules using a deductive method. These rules are built by a unique evolutionary algorithm that creates new rules when no applicable rule exists and splits existing rules when two different cases when a matching rule contradicts the earlier rule. Rules are strengthened (fed) when they are found in more circumstances. In general this method could produce rules that match any arbitrarily shaped pattern. The rules database was built with this algorithm by inputing expert games. The algorithm then tried to find patterns that explained why each move was made. In order to evaluate the rules generated, a subset was given to two expert players who were asked to evaluate the rules as good, average, or bad. The experts evaluated the rules toward the low end of the range with 42% being identified as bad and only 9.2% being identified as good. By adding a concept of visibility, which limited the inputs that were available to the evolutionary process, the quality and length of rules was improved. With these rules, the first experts rated about the rules approximately evenly among the three classifications. While this method was not actually developed for move generation, such a use is suggested by the authors.

3 Discussion

Despite the challenges, there are many things about Go that lend themselves well to this game. A quick list of these attributes is below:

1. The local quality of a move is often similar to the global quality of the move
2. Ladders are easy to identify
3. The state of the board changes little in a typical turn
4. Subproblems such as controlling a corner can be done with traditional methods
5. Winner of a local combat is typically determined by a comparison of the number of liberties each group has
6. The handicap system allows an uneven game to come out fairly evenly (I believe this could be used to force a computer program to continue to learn)
7. The game can be divided into an opening, a middle game, and an endgame. Determining an evaluation function for one phase of this game is easier than a generic evaluation function for a board state
8. Goal generation allows simpler searches towards easier goals [4]
9. Endgame problems can typically be broken into independent problems [10]

If I were to design a program to play Go, I believe I would try to determine the upper and lower utility bounds of each position in every small pattern. I believe this would give a fairly easy way to ignore many low quality moves.

I would also try to localize the search as much as possible. In this case, evaluate the winner of each local group and look at how remote moves could influence a local battle. In a way, this could almost be considered like a partial order plan. This plan would evaluate what would be needed to make the local group safe and what would be needed to make a remote group safe. By comparing the size of the potential win to the cost of moves, we could focus on those battles that we were likely to win that had the highest payoffs.

4 Conclusion

The earliest computer chess research began in the 1950s, but it was not until 1997 that a computer chess program beat the human world champion [8]. Computer Go research began in the 1960s and it wasn't until 1978 that a program that was better than an absolute beginner existed [4]. Given the progress, particularly in the last 5 years, computer Go research has been improving rapidly. However, it still looks like computer Go research is well behind the results of chess a decade ago. I would predict that computers will not have a chance at beating a human champion at Go until at least 2020.

References

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

[2] K. Baker. *The Way To Go.* The American Go Association, Box 397 Old Chelsea Station, New York, NY 10113, 1986.

[3] H. J. Berliner and C. McConnell. B* probability based search. *Artificial Intelligence*, 86:97–156, 1996.

[4] B. Bouzy and T. Cazenave. Computer Go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, October 2001.

[5] M. Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, 1995.

[6] R. Cant, J. Churchill, and D. Al-Dabass. Using hard and soft artificial intelligence algorithms to simulate human Go playing techniques. *International Journal of Simulation*, 2(1):31–49, 2001.

[7] J. Churchill, R. Cant, and D. Al-Dabass. A new computational approach to the game of Go. In *In Proceedings of the 2nd Annual European Conference on Simulation and AI in Computer Games*, pages 81–86, 2001.

[8] T. Kojima. *Automatic Acquisition of Go Knowledge from Game Records: Deductive and Evolutionary Approaches*. PhD thesis, University of Tokyo, 1998.

[9] M. Müller. Partial order bounding: A new approach to evaluation in game tree search. *Artificial Intelligence Journal*, 129:279–311, 2001.

[10] M.Müller. Not like other games - why tree search in Go is different. In *Proceedings of the Fifth Joint Conference on Information Sciences*, pages 974–977, 2000.

[11] M.Müller. Computer Go. *Artificial Intelligence*, 134:145–179, 2002.

Appendix A - An Introduction to Go

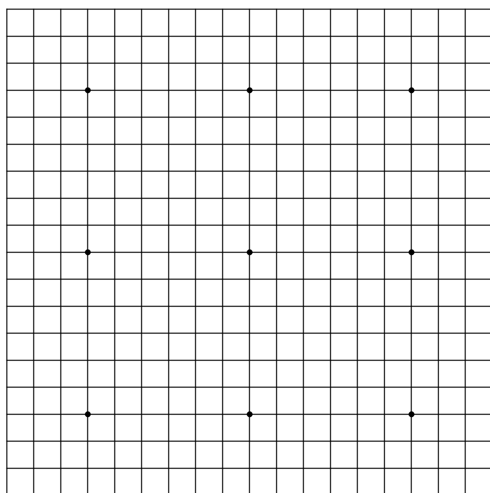


Figure 3: The 19×19 Go Board

Go is played on a board with 19×19 lines such as is shown in figure 3. On such a board, there are 361 intersections. There are two players, black and white. These players alternate placing tokens on these intersections with black going first. When a token is placed on an intersection, the lines connecting that intersection to neighboring intersections are called liberties. Figure 4 shows the 2, 3, and 4 liberties available to a token played on a corner, edge, and middle location respectively. Neighboring tokens share liberties. For example, in figure 5, the group has 6 liberties. To capture a group of tokens, all of the liberties of the group must be eliminated by the other players tokens. Figure 6 shows a group of white tokens before and after it is captured. Once placed, a token is not moved unless it is captured.

There are only three rules that limit where a token may be played. The

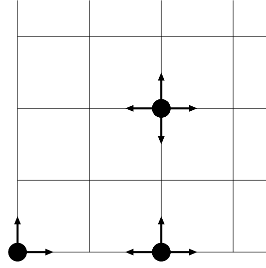


Figure 4: Liberties for Single Tokens

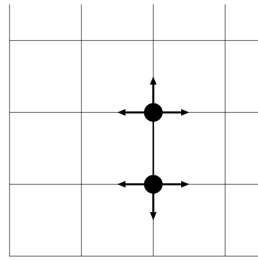
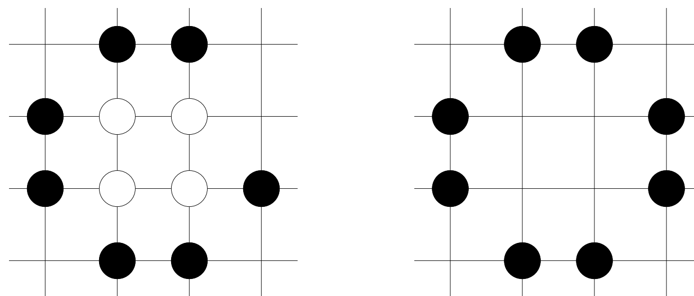


Figure 5: Liberties for Multiple Tokens



(a) Before Capture

(b) After Capture

Figure 6: Capturing a Group of Tokens

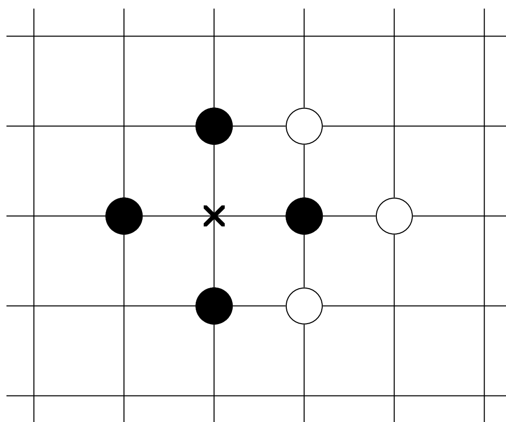


Figure 7: A Situation Leading to a Repeated Board Position

first rule, which is hopefully obvious by now, requires a token to be placed only on an empty intersection. The second rule does not allow a token to be placed where it cannot survive after the move. However, a piece may be placed where it is completely surrounded by the other player's tokens if at least on of the surrounding tokens will be captured by the move. This means that the token will have an available liberty by the end of the term. The final rule is called ko. Ko states that a previous board position may not be revisited. For example, in figure 7 white can play at the $\times$. The rule of ko states that black cannot capture the played white token immediately. Instead, black must first play a token somewhere else on the board. This rule prevents games from continuing forever.

Instead of playing a token, a player may choose to instead pass. If the other player also passes, then the game is over. There are two different methods for determining the score. Both methods rely on counting territory

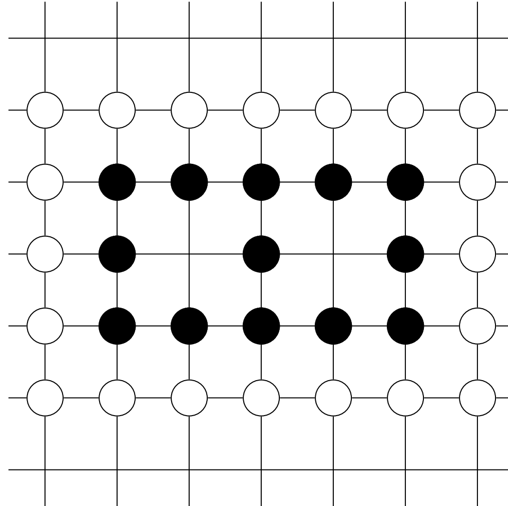


Figure 8: A Stable Formation

controlled. Territory is controlled when it is surrounded by one player and is stable. A group is stable if it cannot be taken by the other player. In practice this means that the group either has at least two completely surrounded independent liberties or eyes, or that the other player could not stop the player from forming two eyes. A simple example of eyes is shown in figure 8. Since white cannot play on either of the empty locations in the black formation, the black formation is stable. All unstable groups that are in the other players territory are captured as prisoners. In figure 8, back controls two locations of territory, the two eyes. The Japanese method of scoring counts territory controlled and the number of prisoners captured. The Chinese method counts territory controlled and the number of pieces controlling it. In order to compensate for black's advantage of moving first, the white player receives an additional $6\frac{1}{2}$ points known as komi. In addition to balancing the game, this guarantees that the game will not result in a tie.