Aamir Shah

Alex Kindel

Jakkree Janchoi

Karl Saldivar

Team 16 – Go Game

Using Search Techniques in the Board Game Go

Abstract  
Implementing Go Game engines has long been a challenging problem in Computer Science. While Chess Artificial Intelligence engines continue to improve from year to year, the progress of Go AIs remain stagnant. This is due to the complex rule set of Go and the large board size. The Chess board size is 8x8 and has a well defined end game. Go, on the other hand, has a board size of 19x19 and a difficult to evaluate end game. Evaluating every Go board state is impossible, many moves evaluate to the same score, and long term strategies are difficult for engines to plan. It is for this reason that implementing AIs in Go is a daunting task. Despite the challenges of Go, one may use a search tree to generate a good move. We present an intuitive method of solving Go using an optimized search strategy and deploy a simple heuristic to maximize the search depth.

1. Introduction

Go AIs remain an active area of Computer Science research. There are many tournaments being held throughout the world where players enter their AIs and play a best of N games against each other: The European Go Congress, the UEC Cup in Japan, or the US/North America Go Championship [1]. Go is well known to be more difficult to solve than Chess despite having simpler rules. While Go’s immediate game play rules are simple to understand, simple heuristics cannot always provide the best possible move. High ranking Go players rely on intuition, pattern matching, and expert domain knowledge of the game to decide on their moves. Chess AIs on the other hand can calculate best moves using quite simple heuristics. If a Chess game tree were to reach its maximum depth, each leaf node would correspond to a checkmate state. However, in the game of Go, these states do not exist in search trees. The end game is often arbitrary. In tournaments, an expert is required to analyze the end game board, and mark stones as “dead”- meaning those stones will eventually, through a series of moves, become captured. No computer program is able to predict if stones will eventually be “dead”, and ones that attempt to do so require extremely expensive computational time. Therefore, employing this heuristic into search trees would greatly slow the game down. In addition, such scoring techniques are irrelevant in the early game as players are still competing for territory and capturing their opponent’s pieces. Heuristics often search for an immediate reward, while ignoring the long term effects of its decision.

By focusing our attention on smaller boards and maximizing the capture game, we reduce the complexity of dealing with end game scoring. Even so, end game scoring is still is a large factor on smaller boards. The Capture Game is a competition between two players where each player tries to capture as many of the other opponent's pieces as possible. Our AI weighs heavily on the capture game, however we also employ additional techniques to maximize territory on smaller boards.

Main Contributions

• We implemented an AI of the Game Go in Java that can play vs. humans or against other AIs.

• We made our framework capable of interchanging between search methods: these search methods include alpha-beta Negamax, Negascout, and iterative deepening Negascout: all with move ordering, a transposition table, and a killer heuristic.

• We developed an intuitive heuristic that relies on captured pieces, liberties, and the Euler number to maximize eyes and connect stones.

• We generated a large amount of statistics to gauge the performance of our AI.

We will first discuss the background of Go and the previous work done in the field, then we will explain in depth our methodology for generating efficient moves. Finally, we will discuss the performance of our program and explain how to improve it.

• Division of Work

Aamir Shah is responsible for the general framework structure, integrating code together, and the search function along with majority of the research and the writing of this paper. Alex Kindel assisted with developing the search, implementing general game rules in the framework, and making clarifications, corrections, and additions to the final report. Jakkree Janchoi and Karl Saldivar worked on the evaluation function and generated statistics vs. other AI’s.

2. Background and Terminology

The rules of Go are relatively simple. The board is a 19x19 sized grid with white and black pieces. Black and white respectively take turns placing pieces on the board. When a piece is surrounded above, below, and to the sides by the opponent’s pieces, it has zero liberties and is thus captured and removed from the board. A liberty is an empty point on the intersections of a piece. This picture is an example of a capture. After white places a stone on point A, black’s liberty count goes from one to zero, and so black’s group is captured.

There are cases where the same set of moves between two players is repeated indefinitely. Such a situation is called KO. In the picture to the left, white can capture black, and then black can capture white -- over and over. Most sets of rules prohibit KO, however when a cycle is repeated over a large span of moves, it often requires an expert witness to call KO. Our framework does not allow KO for small spans of moves.

When both players pass, the game is over. When the game is over, players mark dead stones – stones that will likely be captured in the future, but playing it out would be a waste of time. Often an expert judge is required to mark stones as dead. AI engines struggle to calculate dead stones since it is complex and computationally expensive. Scoring is based on territory (number of liberties each player has) plus the number of captured pieces. White automatically gets +6.5 points since he/she is at a disadvantage for going second (called Komi).

3. Related Work

To date, the largest board size of Go to be solved fully was played on a 5x5 board in 2002 by a computer program called MIGOS [2]. The solution was found at 23 ply deep in about 4 hours. Increasing the size of the board increases complexity dramatically and end game states become more difficult to calculate. Some solutions based on human analysis exist for larger boards but they have not been confirmed by computers. There are a variety of techniques employed to generate moves in the game of Go: knowledge based systems, Monte-Carlo methods, machine learning methods, and minimax search trees.

3.1 Knowledge-based Systems

Knowledge-based systems have become quite popular in solving board games. Time Kinger and David Mecher argue:  "it is our belief that with better tools for representing and maintaining Go knowledge, it will be possible to develop stronger Go programs." They propose two ways: recognizing common configurations of stones and their positions and concentrating on local battles. "... Go programs are still lacking in both quality and quantity of knowledge." [3] Such programs require extremely talented programmers to mimic human action and thought. Knowledge-based systems require advanced pattern matching techniques and hundreds of modules for the different kinds of patterns. Programs that rely on knowledge: Goemate [4], The Many Faces of Go [5], Go Intellect [6], and Go++ [7]. Each of these programs at some point has been considered the world’s best program.

3.2 Monte-Carlo Methods

Monte-Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results [8]. In Go it works like this: generate a list of moves, play a move. Then, generate thousands of random games. The move that generates the best set of random games is chosen. Such methods require large amounts of memory and processing power and low expert domain knowledge as opposed to knowledge based systems. These methods give good move generation and work especially well for smaller boards, however they lack in overall game tactics. MoGo [9] won the 2007 Computer Olympiad using the Monte-Carlo method. They applied a technique called upper confidence bounds applied to trees (UCT) [10]. UCT uses previously generated games to guide the search to better lines of play. Other notable Monte Carlo programs are The Many Faces of Go v 12 [5], Leela [11] and Crazy Stone [12].

3.3 Machine Learning

Machine Learning techniques rely on software to generate rules, patterns and conflict resolution strategies. Using database of professional games or by playing games against opponents, these algorithms learn techniques that can later be applied. This is achieved using a neural network or genetic algorithms. NeuroGo is a notable engine that uses neural networks to evaluate board positions [13]. People play against NeuroGo and train it using temporal-difference learning. NeuroGo then applies previously learned pattern techniques to evaluate the best move. Many engines rely on Machine Learning to achieve small things, such as Crazy Stone [12] which generates moves from several hundred sample games.

3.4 Tree Search

Tree search is not a feasible strategy when dealing with large 19x19 Go Boards without special modifications. It is nearly impossible to evaluate every possible move in a 19x19 board even using the world’s most powerful super computers. A simple estimate for the size of the game tree in Go, which assumes an average

branching factor of 250 and an average game length of only 150 ply (which is quite optimistic because the longest professional games are over 400 moves), leads to a game tree of about 250^150 ≈ 10^360 nodes [15]. In comparison, Chess has about 10^120 nodes [27]. Search is extremely effective in Chess. This is due to the fact that the board is smaller, there are less possible moves due to move constraints, and the evaluation function can be easily defined by the number of pieces on the board. In Go, search is limited due to end game considerations, a high branching factor, and the fact that many moves are evaluated to the same score, but realistically one of those moves will result in a better strategy than the others. GnuGo [14] is a popular program that incorporates tree search to determine the best possible move. It uses pattern matching techniques to select a small set of moves, and evaluates these moves using popular searching techniques, effectively reducing the branching factor significantly. GnuGo has been a strong competitive force in the Go world and continually ranks in top 10 in most tournaments. There are a variety of search techniques that can be used. The most common form of tree search in board games is alpha-beta pruning [16], due to the fact that it speeds up conventional minimax search by pruning branches. Negascout is a variation of alpha-beta pruning that relies on move ordering and null windows to achieve better results. Negascout gives a 10% performance increase from alpha-beta pruning [17]. Negascout is the best search technique in Go Games when combined with Move Ordering, Transposition Tables, and killer heuristics [18]. More details of alpha-beta pruning and Negascout will be discussed in sections 5.2 and 5.4.

In conclusion, much work has been done in the research of Go including machine learning, knowledge based pattern matching, Monte Carlo methods, and tree search research. Research continues to grow and in the future when AI techniques become more advanced, larger Go boards may be solved completely.

4. Framework

We title our software Team16 Go. We run our Go game on an open-source package called GoGui [19]. GoGui is an open source framework that takes care of all the graphical user interface issues in the game of Go and allows our game to play against other AIs. It notifies of any illegal moves, automates end game scoring, and implements a protocol called Go Transfer Protocol (GTP). GTP is a protocol that allows two engines to communicate with each other using bit strings. The main use for GTP is to play GoGui over the internet with a friend; however Go engines can implement GTP. This allows Program vs. Program or Player vs. Program. This is a convenient framework for us because we can gauge our AI strength vs. humans and other popular Go engines.

We also use a framework called Lurgee strategy game framework [20]. Lurgee is an open source package that provides implementations of Negamax and Negascout search. We rely on this framework to take care of the skeleton of our searching, board states and move generation. However, we had to implement the internals to work with the Go game and GTP. We also heavily modified the search to allow for our own move ordering and transposition table.

All code is written in Java and programmed using Eclipse IDE. We also downloaded the engines AmiGo and GNUGo to test our AI vs. other AIs through GNUGo, and GNUGo’s end game scoring to score our games.

In brief, our program works as follows: given a board state and the current player, generate a list of all valid moves. Moves that capture pieces also update the board state. Then, run Negascout tree search to find the best possible move. Apply this move to the current board state, and send it back to the opponent on the other end of the GTP protocol network.

5. Searching

Our current frame work uses iterative-deepening Negascout with killer heuristics, move ordering, and a transposition table. We can turn off each of these features to evaluate performance, but we have found that this particular search strategy is the optimal technique for Go Games. We first examine Minimax and Negamax search techniques which provide a background to understand Negascout. We then talk about the optimizations we applied to Negascout including transposition table, move ordering, and killer heuristics.

5.1 Minimax

In minimax search there are two node types. A max node is where the player tries to maximize his score and the min node where the opponent tries to minimize the score. Nodes at an odd ply are min nodes. Starting from evaluations at the leaf nodes, and by choosing the highest value of the child nodes at max nodes and the lowest value of the child nodes at min nodes, the evaluations are propagated back up the search tree, which eventually results in a value and a best move in the root node[18].

5.2 Alpha-Beta Search

Alpha-beta search is an extension of minimax search. Alpha is the lower bound value: it is the worst possible move for the Max player. Any sub tree lower than alpha is not worth investigating. Beta is the upper bound value: it is the worst possible move for the Min player. Nodes greater than Beta also do not have to be investigated. The result is identical to minimax, however many branches are cut off (left unexplored) thus greatly increasing performance.

5.3 Negamax with alpha-beta pruning

Negamax is a variance of minimax with alpha-beta pruning except it relies on the zero-sum property of a two player game. If black scores 10, then white must score -10. Instead of calling a separate routine for the min and the max player, it passes the negated according to the following mathematical relation:  
  
 max(a,b) == -min(-a,-b)

The psuedocode is simple:

int negaMax( int depth ) {

if ( depth == 0 ) return evaluate();

int max = -oo;

for ( all moves) {

score = -negaMax( depth - 1 );

if( score > max )

max = score;

}

return max;

}

5.4 Negascout

Negascout is built on the foundation of Negamax and is our main search function. Also called Principal Variation Search, Negascout dominates alpha-beta pruning in the sense that it will never examine a node that can be pruned by alpha-beta [21], but it relies on move ordering to capitalize on this advantage. Negascout assumes that the first move explored is the best move. This is accomplished using a good move ordering scheme, which will be discussed in the next sections. Assuming the first move is the best move, it searches all remaining nodes using a null window. A null window is when alpha is equal to beta, which is faster than alpha beta pruning since most nodes will be cut off. If the null window fails, Negascout will re-search the remaining nodes using regular alpha-beta. If a poor move ordering is used, Negascout is slower than alpha-beta pruning (although the use of a transposition table negates much of the penalty as moves will not need to be re-evaluated); otherwise on average, Negascout is 10% faster than regular alpha-beta pruning [21].

function negascout(node, depth, α, β)

if node is a terminal node or depth = 0

return the heuristic value of node

b := β (\* initial window is (-β, -α) \*)

foreach child of node

a := -negascout (child, depth-1, -b, -α)

if a>α

α := a

if α≥β

return α (\* Beta cut-off \*)

if α≥b (\*check if null-window failed high\*)

(\* full re-search \*)

α := -negascout(child, depth-1, -β, -α)

if α≥β

return α (\* Beta cut-off \*)

b := α+1 (\* set new null window \*)

return α

5.5 Move Ordering

A good move ordering scheme is vital when using Negascout search. We achieve this using two measures: a killer heuristic and a transposition table.

5.5.1 Killer Heuristic

Killer moves rely on the assumption that a good move in one branch is likely a good move in another branch of the same depth. It assumes that a move that produced a cutoff on the same depth will likely produce a cutoff in the present position [22]. In our application, we store 2 killer moves. If a move produces a cutoff, it replaces one of the two killer moves. During move ordering, we select these two moves as our first moves, to increase the likelihood of early cutoffs.

5.5.2 Transposition Table

We use a transposition table to store the evaluation of previous moves. This becomes extremely useful when performing iterative deepening, as many nodes are evaluated more than once. When a new move is generated during tree search, we check the transposition table to see if we have evaluated this move at a depth that is greater or equal to the current search depth [23]. If so, we can return the score that is stored in the transposition table. If not, we evaluate the position and then store the result in our transposition table. Our hash table has a limited size and in the case of collision replaces the node that it collided with. Other notable collision techniques are to keep the node with the larger depth: calculating nodes at larger depths are more computationally expensive. Optimally, multiple methods of replacing old entries can be used.

We use a form of hashing called Zobrist hashing [24] which is common in board games such as chess as it is very computationally efficient and generates few collisions. Zobrist hashing assigns a pseudorandom number to each board location for each player. For example, a white piece on board location [0,0] has a unique number, as does a black piece on board location [1,2]. To calculate the hash value for a board state, for each piece on the board, we XOR the current hash with its unique pseudorandom number. Zobrist hashing is useful because once the board is hashed, adding and removing pieces from the board is computationally cheap. To add a piece to the board, simply XOR the piece’s unique value with the current hash value. To remove a piece, you can also XOR the piece’s unique value with the current hash value. Zobrist hashing is a fast and effective technique to store millions of board states with minimal amounts of collisions.

In conclusion, our search incorporates a variety of advanced enhancements. Because the board size of Go is so large, it is important to have an optimal search strategy and a simple heuristic.

6. Evaluation Function

A quick simple evaluation function leads to deeper plied searches. Therefore, we tried to minimize the amount of computation in our evaluation function, and perform an O(N) heuristic to determine the score. Our evaluation function considers the liberties, the number of pieces on the board, avoiding edges, connecting stones, and maximizing eyes similar to the strategy deployed by Erik van der Werf [18]. This strategy works particularly well for small boards (less than 9x9), but lacks in overall territorial strategy near end game.

Our first consideration in our evaluation function is the number of liberties. As mentioned before, the number of liberties is the number of empty intersections around a stone. In this picture, you can see that black has 5 liberties and white has 7 liberties. If white were to place a stone on each of the black dots, black would be captured. Therefore, maximizing one’s own liberties while minimizing the opponent’s liberties is an overall good strategy. For our heuristic, we calculate the number of black’s liberties minus the number of white’s liberties. If the difference between the two players is large, this value is discarded as a large difference will not necessarily benefit the strategic value of the next move. We calculate the number of liberties of the entire board rather than piece by piece for efficiency.

Another aspect of our heuristic is the number of pieces on the board. Maximizing one’s own number of pieces while minimizing the opponent’s number of pieces is a good overall strategy for the capture game. If one move reduces the opponent’s pieces, this value should be weighed heavily. This value will generally be the same for each move, however if a piece is captured it will differ.

Placing pieces on the edge of the board is an overall bad strategy unless there is a good means to an end, such as capturing pieces. The reason being is that stones on edges have fewer liberties than stones away from edges as the “empty” area off the edge of the board does not count as liberties. Stones on edges are relatively easy to capture. We negatively weigh placing pieces on the edge of the board, while encouraging the opponent to place their stone on an edge.

Our last consideration in our heuristic is connecting stones and making eyes.

Connecting stones is an overall good strategy in Go, since connected stones are much harder to capture than isolated stones. An eye in go is a circle formation of stones with an empty area inside. On the game pictured to the left, “Eyes in Go”, the group of white stones on the left hand side of the board is an eye. If black were to place a stone in the center it would immediately become captured; this term is called suicide and is illegal in most forms of Go. However in the group to the right in the figure “Eyes in Go”, black may play in the eye and capture the group. If a group contains two such eyes.Eyes in Go

In the above game, “Two Eyes”, black can never capture the white group as it has two such eyes. A play by black in either of the eyes would be a suicide move. For this reason, forming a group with two eyes is very advantageous.

A heuristic to connect stones and create eyes separately would be computationally expensive. Luckily, there is a method to do both in one sweep. We achieve this using the Euler Number [25]. The Euler number of a binary image is the number of objects minus the number of holes in those objects. Minimizing the Euler number thus connects stones and creates eyes [18]. We slide a 2x2 window across the board and look for three types of occurrences: Q1, Q3 and Qd as shown here:

We count the number of each occurrence of these three quad types, and then calculate the Euler number using this formula:

Our final evaluation function looks like this:

int score =  
 Math.min(Math.max(liberties,-4), 4)  
 + -4\*euler + 5\*numOfPieces  
 + numOnEdge

7. Evaluation and Final Results

Overall our engine is quite strong vs. beginner players on small boards. However, on 19x19 boards, our AI is no match. Quite simply, more time would need to be invested in order to achieve a strong AI on large boards. Most competitive programs have required five to fifteen person-years of effort, and contain 50–100 modules dealing with different aspects of the game [3].

7.1 Performance

By adding all the optimizations we discussed such as Negascout, Transposition Table, Move Ordering, etc. we were able to increase the performance of our AI quite a bit. As a result, we can now search at a depth of four in a reasonable time. In the end, these optimizations allow searching to a greater depth than using regular alpha-beta which leads to better decisions in the engine.

The following tables are based on performing 100 moves on a Go Board.

Negascout with Transposition Table, Move Ordering, and Killer Heuristic - 100 Moves

Depth

Board Size

Average Time Per Move

3

9x9

1.797 sec

4

9x9

11.153 sec

5

9x9

408.684 sec

2

19x19

4.030 sec

3

19x19

313.523 sec

Alpha-beta pruning (without Transposition table, Move Ordering, or Killer Heuristic)  
 – 100 Moves

Depth

Board Size

Average Time Per Move

3

9x9

2.656 sec

4

9x9

14.545 sec

5

9x9

1093.889 sec

2

19x19

5.785 sec

3

19x19

780.931 sec

Adding a Transposition Table, Move Ordering, Negascout and a Killer Heuristic helped improve the performance of our search significantly. Negascout is meant to give a 10% performance increase from alpha-beta, and our results validate this (we get significantly better than 10% on larger depths). However, employing a search of depth 5 on a 9x9 board or a search of depth 3 on a 19x19 board is too long no matter what search method is used. Increasing these depths would make our engine take over an hour to generate a move, which is unreasonable.

7.2 Against other AI’s

We played 10 games vs. GnuGo [14] on a 9x9 board searching at a depth of ply 4. GnuGo, as discussed earlier, uses tree search: it uses pattern matching to choose a small subset of moves to search from. GnuGo is very well optimized for large 19x19 games, but also works well on smaller boards. While we did play 10 games, every game panned out exactly the same. Our AI was able to beat GnuGo on its beginner setting. This is quite impressive and beyond our expectations. The final score was 30 vs 28. It should be noted that GnuGo was dominating us in territory.

We attempted to play an AI named Amigo [26] using a 19x19 board using a ply of 2. Amigo destroyed our AI. It captured most of our pieces, and dominated in territorial scoring. Because our AI could only search 2 moves ahead, it was not able to predict a plan to capture large groups of stones. As a result, it got beat badly. Increasing our search depth made the game too long (sometimes over an hour per move).

7.3 Against Team 16

Each team member played 5 games against the AI. The scores were as follows:

Alex Kindel

5-0

Aamir Shah

1-4

Jakkree Janchoi

1-4

Karl Saldivar

3-2

5-0 means that Alex Kindel beat the Go Engine 5 games out of 5. It should be noted that Alex and Karl are both beginner-intermediate in Go, while Jakkree and Aamir are beginners. To start with, each player was being beat by the AI; however because we understand the inner workings of the algorithm, we were able to intelligently select moves that counter our heuristic and as a result adapt to the game play of our algorithm.

8. Conclusion & Future Work

Our AI had quite impressive results for a 10 week project on 9x9 and smaller boards. It is able to beat beginner level players. By employing additional techniques such as move pruning using pattern matching techniques similar to those that GnuGo uses, we would be able to search at deeper ply. In addition, by expanding our evaluation function to work for larger boards, we would have a decently competitive AI that would beat many intermediate players.

Citations

[1] Computer Go, Wikipedia, http://en.wikipedia.org/wiki/Computer\_Go, March 2010

[2] Solving Go on Small Boards, Erik C D van der Werf, H Jaap van den Herik, Jos W. H. M Uiterwijk, ICGA Journal, June 2003, pg 92.

[3] Müller, Martin. Computer Go, Artificial Intelligence 134 (2002): p151

[4] Goemate, Sensei’s Library, http://senseis.xmp.net/?Goemate, March 2009.

[5] The Many Faces of Go, Smart Games, http://www.smart-games.com/manyfaces.html, Feb 2009.

[6] Go Intellect, Sensei’s Library, http://senseis.xmp.net/?GoIntellect, Jan 2008.

[7] Go++, http://www.goplusplus.com/.

[8] Monte Carlo method, Wikipedia, http://en.wikipedia.org/wiki/Monte-Carlo\_method, March 2010.

[9] Mogo, Sylvain Gelly’s Home Page, http://www.lri.fr/~gelly/MoGo.htm.

[10] Combining Online and Offline Knowledge in UCT, Sylvain Gelly, David Silver, ICML 2007.

[11] Leela, http://www.sjeng.org/leela.html.

[12] Crazy Stone, http://remi.coulom.free.fr/CrazyStone/, Nov 2009.

[13] Evaluation in GO by a Neural Network Using Soft Segmentation, M Enzenberger, Proceedings of the 10th Advances in Computer Games Conference, Graz, 2003.

[14]GnuGO, http://www.gnu.org/software/gnugo/gnugo.html, August 2009.

[15] L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. Artificial Intelligence, 66(1):91–124, 1994.

[16] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. Artificial

Intelligence, 6(4):293–326, 1975.

[17] Negascout, Wikipedia, http://en.wikipedia.org/wiki/Negascout, Aug 2009.

[18] AI Techniques for the Game of Go, Erik van der Werf, PHD Thesis, Universiteit Maastricht, 2005.

[19] GoGui, http://gogui.sourceforge.net/, 2009.

[20] Lurgee, Stregy Game Framework, http://lurgee.net/strategy-game-framework/, 2008.

[21] Negascout, Wikipedia, http://en.wikipedia.org/wiki/Negascout, Aug 2009.

[22] Informed Search in Complex Games, Dissertation, Universitaire Pers Maastricht, 2004.

[23] Transposition Table, Wikipedia, http://en.wikipedia.org/wiki/Transposition\_table, August 2009.

[24] A New Hashing Method With Application for Game Playing, Albert Zobrist, University of Wisconsin Technical Report 88, April 1970.

[25] S. B. Gray. Local properties of binary images in two dimensions. IEEE Transactions

on Computers, C-20(5):551–561, 1971.

[26] AmiGoGTP, http://amigogtp.sourceforge.net/, March 2010.

[27] Claude Shannon (1950). "Programming a Computer for Playing Chess". Philosophical Magazine 41 (314).