Shaun Benjamin
Conner Jevning

Final Report

Motivation:

Go has long been considered a difficult challenge in the field of AI and is considerably more difficult to solve than chess. Unlike chess, the size of the Go board results in a huge branching factor on the order of hundreds per move. Furthermore, most of the moves in Go are legal. For instance, in the beginning chess has only about 20 possible moves, in Go there are 50 (accounting for symmetry) or 361 (not accounting for symmetry) possible moves. So far the largest board Go has been solved for is a 5x5 board in 2002, by a program called MIGOS. Recent developments in Monte Carlo tree search and machine learning have brought the best programs to high dan level on the small 9x9 board. In 2009, the first such programs appeared which could reach and hold low dan-level ranks on the KGS Go Server, also on the 19x19 board.

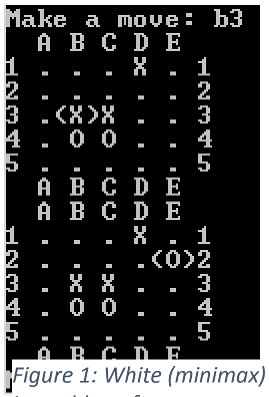
Problem Definition:

We attempt to build a computer Go solver using a multitude of methods and evaluate their successes against each other as well as its efficiency on various board sizes. The board sizes that we consider are 5x5 (small), 9x9 (medium) and the 19x19 (large).

Challenges:

The most significant challenge of Go compared to most other games is the large branching factor associated with the game. Compared to a chessboard, the Go board can be much larger. Furthermore, most moves in Go are legal, unlike in chess where most moves can be pruned as they are illegal. This results in a large branching factor and thus, conventional minimax is not a feasible option.

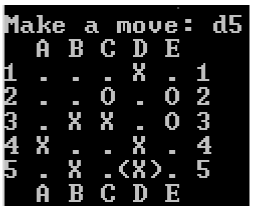
Another challenge of Go is that board positions are more easily evaluated using pattern recognition than computing depth first search over the board. For instance, consider figure 1 where a human plays as Black ('X') and minimax (depth 2) plays as White ('O'). Here, the white chain at row 4 is in danger of being captured by black, yet minimax is unable to see this danger and plays at e2. Note that the newest played piece is placed in brackets for ease of use. A few moves later, white has lost its chain at row 4, see Figure 2. As it turns out, minimax would need a depth setting of 4 or higher in order to recognize the danger in such positions. This means that even if the branching factor was somehow reduced to a manageable size, the depth required to make meaningful decisions will still make the search tree very large.



```
Make a move: b3
  A B C D E
1 . . . X . 1
2 . . . . . 2
3 .(X)X . . 3
4 . O O . . 4
5 . . . . . 5
  A B C D E
  A B C D E
1 . . . X . 1
2 . . . .(O)2
3 . X X . . 3
4 . O O . . 4
5 . . . . . 5
  A B C D E
```

Figure 1: White (minimax) is unable to foresee danger to chain at row 4

The final challenge of Go, similar to many other games, is the need for expert game knowledge. Since searching over the entire search space is impossible, we require domain knowledge in order to prune out entire sections of this tree. Furthermore, the features to extract in order to perform state evaluation require domain knowledge as well. Extracting meaningless features can waste time since we are evaluating an exponential number of states. Simply using the score of each state is also not enough as the score at depth 2 is meaningless. If we were to maximize the score after 4 moves (depth 2) we might play in a greedy fashion that claims a large amount of territory but this territory gain is temporary as this will result in disconnected stones that the opponent can easily capture.



```
Make a move: d5
  A B C D E
1 . . . X . 1
2 . . O . O 2
3 . X X . O 3
4 X . . X . 4
5 . X .(X). 5
  A B C D E
```

Figure 2: White (minimax) loses chain at row 4

Approaches:

1. Random Agent (playable on all board sizes):

This is our simple baseline algorithm. Simply picks a random move over all legal moves with a uniform distribution. It is important to note that the random agent on board sizes of 5x5 or smaller can make a good move by pure chance. On larger boards, random agent performs poorly in terms of the strength of each move but it is the fastest of all agents and thus testing on random agents is much quicker. Since this agent is able to play quickly on all board sizes, we chose random agent as our baseline algorithm.

2. Minimax Agent (playable on board sizes 5x5 and smaller):

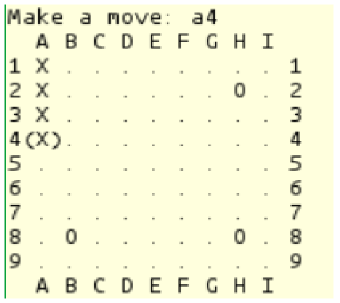
Ordinary minimax agent with alpha-beta pruning. This agent is unusable on board sizes 6 or larger due the branching factor. Furthermore, depth search of 2 – 4 is still not very strong against a human player with a very weak understanding of the game. The simple minimax agent is only stronger than the random agent. The evaluation function that we used to evaluate favorable states take into consideration the following features: number of friendly pieces in atari (a chain which only has one liberty), the number of captures, the number of our pieces on the board and the current score.

3. Minimax Agent with Heuristics (playable on all board sizes):

There were two main concerns of the straight forward minimax that needed to be dealt with. The first was the large state space.

In order to cut down the state space there are a few optimizations that were done. The first is that during the search, as we are expanding the states that the opponent can take us to, we only consider moves of the opponent that are close by to our pieces on the board. The intuition is that it is possible that the opponent might play on locations far away from our pieces but is unnecessary to consider these possibilities now. Instead, we can consider those possibilities later when the opponent does actually make such a move. In doing so we only consider moves that could potentially attack our own pieces. This reduced the state space significantly but it was still not enough on a 19x19 board with only depth search of 2.

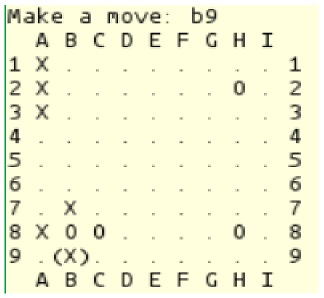
A second optimization was used where during the search when it is the agent's turn to make a move, we only consider moves that are neighbors of pieces (both black & white pieces) that have already been placed. The intuition for why the agent can get away with this is that during the first few moves of the game, the agent places stones on every available corner of the board, see Figure 3. The idea being that the agent wants to cast a wide net over a large territory and strengthen its position on those territories later. Since we consider moves over all neighbors of pieces that have already been placed, this means that the agent in theory is able to claim all areas of the board if the opponent plays poorly. This reduces the state space dramatically and we are able to use minimax over this pruned state space for a 19x19 board. The final concern of minimax, which was mentioned earlier, is that the ability for depth search to evaluate stones which are in danger of being captured is very poor.



```
Make a move:  a4
  A B C D E F G H I
1 X . . . . . . . .  1
2 X . . . . . . O .  2
3 X . . . . . . . .  3
4(X). . . . . . . .  4
5 . . . . . . . . .  5
6 . . . . . . . . .  6
7 . . . . . . . . .  7
8 . O . . . . . O .  8
9 . . . . . . . . .  9
  A B C D E F G H I
```

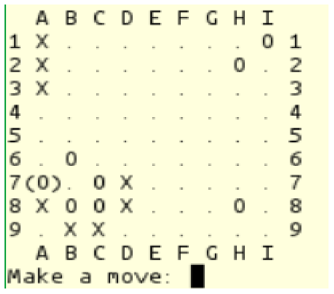
Figure 3: White ('O') secures the corners

In order to address positions where a chain of our stones are being threatened we developed a method for detecting such positions and instead of simply doing depth search with some heuristics, we modify the search to only consider paths where the first move either saves our chain of stones or attacks the enemy stones. The agent first starts by first identifying all friendly chains who have at least half of their liberties occupied by enemy stones. If the chain has two or more eyes then the chain is considered to be safe. If the chain does not have two or more eyes, then the enemy chains which are attacking this friendly chain are found. Finally, we do depth search over all moves that either extend the number of liberties of this friendly chain or reduce the number of liberties of the enemy chain. This allows minimax with heuristics to identify potential threats and save itself accordingly. See Figures 4 and 5 for an example in which White, the heuristic minimax agent, saves itself by attacking the opponent's stones. By using these heuristics minimax is able to play moderately well on all board sizes from 5x5 to 19x19 with less that 30 seconds per move.



```
Make a move:  b9
  A B C D E F G H I
1 X . . . . . . . .  1
2 X . . . . . . O .  2
3 X . . . . . . . .  3
4 . . . . . . . . .  4
5 . . . . . . . . .  5
6 . . . . . . . . .  6
7 . X . . . . . . .  7
8 X O O . . . . O .  8
9 .(X). . . . . . .  9
  A B C D E F G H I
```

Figure 4: White ('O') is being attacked at lower left of the board



```
  A B C D E F G H I
1 X . . . . . . . O  1
2 X . . . . . . O .  2
3 X . . . . . . . .  3
4 . . . . . . . . .  4
5 . . . . . . . . .  5
6 . O . . . . . . .  6
7(O). O X . . . . .  7
8 X O O X . . . O .  8
9 . X X . . . . . .  9
  A B C D E F G H I
Make a move: █
```

Figure 5: White captures Black at b7

4. Monte Carlo Tree Search:

Monte Carlo Tree Search is essentially minimax using depth 2, with the evaluation function being the results of simulations. These simulations consist of a "depth charge" which is a simulation of random moves to the end of the game. The algorithm then evaluates how favorable each possible move is, based on the results of simulations using that move, like whether it won or lost and by how much.

5. Monte Carlo Tree Search with Upper Confidence bounds applied to Trees (UCT):

Monte Carlo Tree Search with UCT ended up being our most sophisticated and best performing algorithm. Essentially the algorithm builds up a tree in memory, where each node in the tree is a possible move that can be made given the current state of the board. MCTS with UCT relies on being able to run multiple full game simulations, usually the more the better. Thus when starting a game, a certain time limit per turn needs to be set, usually around 30 seconds per turn in most professional implementations, which we also used in ours.

Upon starting its turn, the Monte Carlo UCT agent sets a dummy node to act as the root of the tree. Each node in the tree keeps track of the number of times a simulation has been run from it and its children, how many of those simulations have resulted in wins, and which node is its child (and sibling, if one exists). Next, all possible moves from the current board state are attached as children of the root. We then start running simulations. Each simulation consists of one “depth charge” which simulates random moves for each side until the very end of the game.

When a node is selected to be simulated, we check to see if it has been expanded to its children yet and how many times a simulation has been run from it. If it has not been expanded and has been simulated less than a set number of times (we used ten) we run a depth charge. If either of these conditions aren’t met, we move on to simulating its children. If its children have not yet been initialized, we attach them. We then select a child to simulate.

The method to select which child to simulate is one of the key parts of the UCT algorithm. We loop over all of the children. If any of them have not yet been visited and had a simulation run from them, we force one of them to be chosen at random so they can be simulated, to ensure that everything will be simulated at least once. As long as they have all had at least one simulation run from them, then we use a mathematical formula to select the node. The formula is: sqrt(1 / 5) * sqrt(log(total visits to parent) / total visits to child). We then choose the child that results in the highest value from this formula.

Upon selection of a node to simulate and the completion of a depth charge, we then update the statistics for that node. We add one to the total number of visits to the node, and add one to the number of wins from this node if and only if the depth charge resulted in a win. We then backpropagate, doing the same for the node’s parent, the parent’s parent, and so forth all the way up to the root.

Once the algorithm runs out of time for its turn, it ceases running simulations. Then from the root, it selects the node with the highest number of simulations run from it and its children, rather than the node with the highest win rate. It then makes the move associated with this node, and the process starts over again on its next turn.

Results (See last page for result graphs)**:**

As an aside, please note that these numbers are the average of the win rates for each of the possible board sizes (5x5, 9x9, 19x19) that we were able to run each of the algorithms on. For example, any numbers resulting from an algorithm vs. minimax are only from a 5x5 board. Similarly, for UCT, the numbers are the average of simulations on 5x5 and 9x9 (issues with 19x19 and some results of simulations on it are covered later). Some simulations were able to use all three board sizes, and thus the numbers listed are the average of those numbers.

1. **Random Agent:**

   As expected, the random agent was an awful Go player, losing the vast majority of games it played. Its futility ranged from losing 82% of games to minimax to losing 98% of games to UCT, with a 93% loss rate to minimax with heuristics and 96% loss rate to Monte Carlo.
2. **Minimax Agent:**

   Minimax was only able to be run on small board sizes (5x5 or smaller) and due to this the numbers may be skewed a bit. It of course did well against random, winning 82% of the time. It lost the (very) small majority of games to minimax with heuristics and Monte Carlo, with win rates of 48% and 47% against each of them, respectively. Its efforts against UCT were quite pathetic however, winning only 17% of the time.
3. **Minimax Agent with Heuristics:**

   Minimax with heuristics ended up as probably our second best algorithm. It predictably crushed the random algorithm 93% of the time. Versus regular minimax it won a bit more than half its games, at 52%. When run against Monte Carlo, it did extremely well, winning 76% of its games. Unsurprisingly, however, it was no match for UCT, as it was only able to pull out a win in 22% of its games.
4. **Monte Carlo Agent:**

   Monte Carlo had an up and down performance. It did very well against the random algorithm, winning 96% of the time. Against minimax, it was able to win 53% of its games. That was the end of the good news though, as it was only able to win 24% of its games against minimax with heuristics, and a terribly poor 19% of its games versus UCT.
5. **Monte Carlo with UCT:**

   When given enough time per turn to run its necessary simulations, Monte Carlo with UCT did extremely well, as predicted. It won a decisive 98% of games versus the random algorithm. Against minimax, it continued its domination by winning 83% of the time. Adding heuristics to minimax didn't make things much tougher, as it was able to win 78% of the time. And finally, versus Monte Carlo, it finished its consistently dominant performance by winning 81% of its games.

Analysis:

1. **Random Agent:**

   The simplest and fastest agent, the random agent is most successful against the minimax agent by winning 18 out 100 games. Against all other algorithms it did much worse. The reason being that minimax can only run on small boards and thus the random will sometimes get lucky and play strong moves. As a result it's no surprise that it was able to perform best against minimax agent.
2. **Minimax Agent:**

   Minimax agent splits most of its games almost evenly against minimax agent with heuristics and the Monte Carlo tree search agent. It is no surprise that the minimax agent has a slight disadvantage over the minimax agent with heuristics since the heuristics make the algorithm faster, but do not provide much of an advantage on the small board size. The results between the Monte Carlo tree search agent and the minimax agent are more interesting and will be discussed in the Monte Carlo Agent section.

3. **Minimax Agent with Heuristics:**

   Minimax agent with heuristics does well against the random agent but as discussed previously essentially splits games against minimax agent. It is important to note that in its games against minimax agent, it only played on a 5x5 board since minimax cannot function on a larger board. The most interesting result for the minimax agent with heuristics is in its games against the Monte Carlo tree search. From the tests we saw that on smaller board sizes (5x5) the two agents would split wins evenly. As soon as the board size was moved up to a 9x9, the minimax agent with heuristics would win 90% of its games. Here we see that the heuristics allow the minimax agent allow it to maintain a significant portion of its optimality. Not only that, but since it outperforms the Monte Carlo agent, which uses minimax, we also see that the evaluation function of the minimax agent gives stronger results than simulation using minimax over random agents (which is the evaluation function of the Monte Carlo agent).

4. **Monte Carlo Agent:**

   The Monte Carlo agent against the regular minimax agent showed the advantage of going first in Go. Both algorithms were able to win ~85% of the time depending on who went first. Over the tests we only saw such behavior for small size boards and for algorithms of relatively even strength. This indicates that the advantage of going first lessens as the board size becomes larger or if the players are less evenly matched.

5. **Monte Carlo Agent with UCT:**

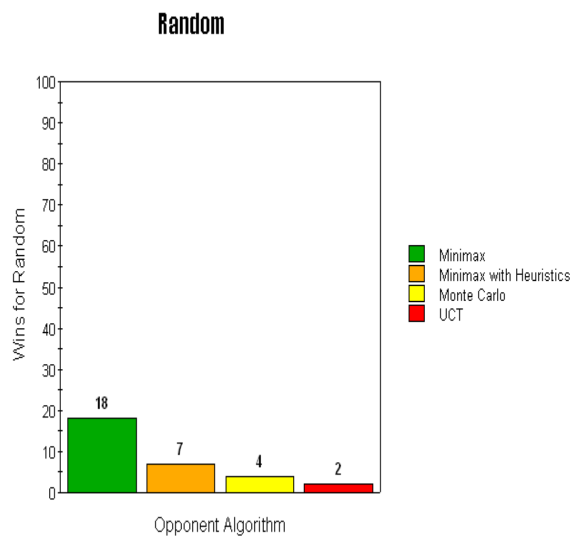
   Monte Carlo with UCT performed very well against each of the other algorithms, on both 5x5 and 9x9 boards. In both cases we gave the algorithm 30 seconds per turn in order to decide what play to make. This obviously worked out very well, as covered in the results section. The size of the board was by far the biggest obstacle to the performance of UCT. On a 19x19 board, 30 seconds was not nearly enough time for it to run enough simulations to actually make an informed decision on what play to make, and the algorithm performed horribly. The board was simply too large, there were too many possible moves to explore at each iteration, and running the actual depth charges themselves took a significant amount of time more than on smaller boards. In order to try to get some sort of game results for UCT on a 19x19 board, we started upping the amount of time allowed per turn. We tried upping it to 1 minute, then 2, then 3, followed by 5 and finally 10 minutes per turn. At 10 minutes per turn, we finally saw some promising results, as it was winning most of the time. Of course this is from a very small sample size, as we weren’t able to run very many of these simulations due to each game taking a bit more than 14 hours to complete.

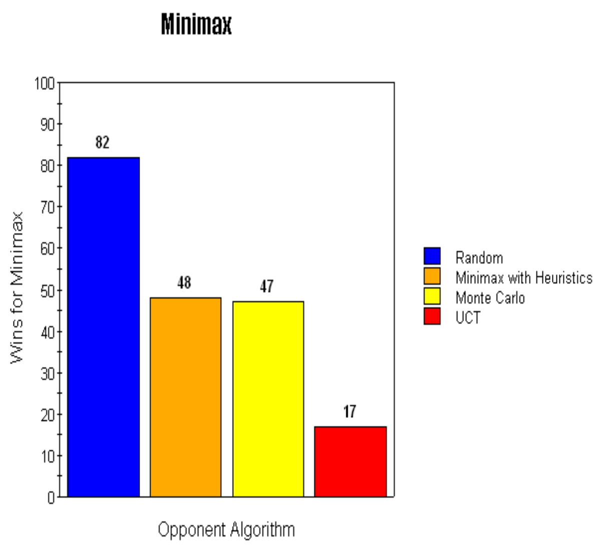
Conclusion:

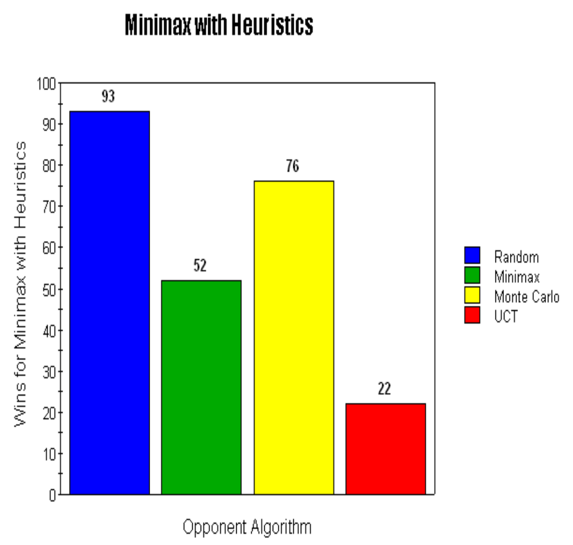
In conclusion, we see that there are two main effective approaches to computer Go. The first is the minimax approach that relies on heuristics to make it faster and more effective. The second approach is the Monte Carlo approach that relies on simulation and careful exploration of the state space in order to perform well. The first approach requires in-depth knowledge of the underlying aspects of the game in order to develop useful heuristics. Unlike the minimax approach, the Monte Carlo approach does not require any understanding of the game in order to perform well. Furthermore, the first approach is less dependent on the size of the board as well as the time allowed per move; which are the main weaknesses of the second approach. Thus we see a tradeoff between the speed and optimality in that the first approach is fast but suboptimal where as the second approach is slow but much stronger. Finally, we conclude that the second approach should be the one to focus on as any advancements to the second approach will be more useful since the second approach is more general and thus can be used to solve other problems, apart from Go.

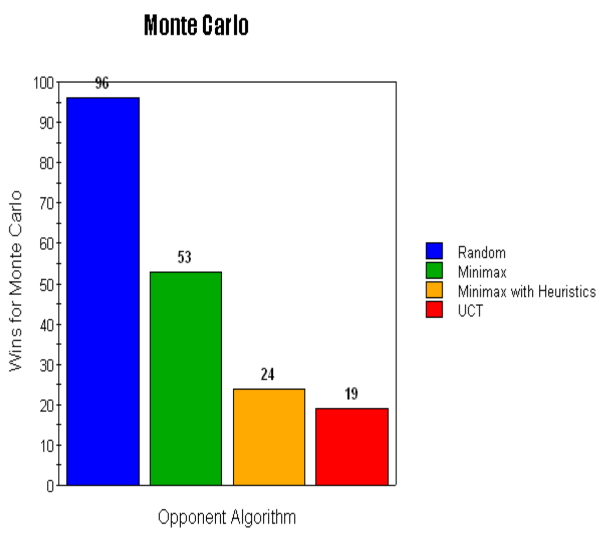
(See next page for result graphs)

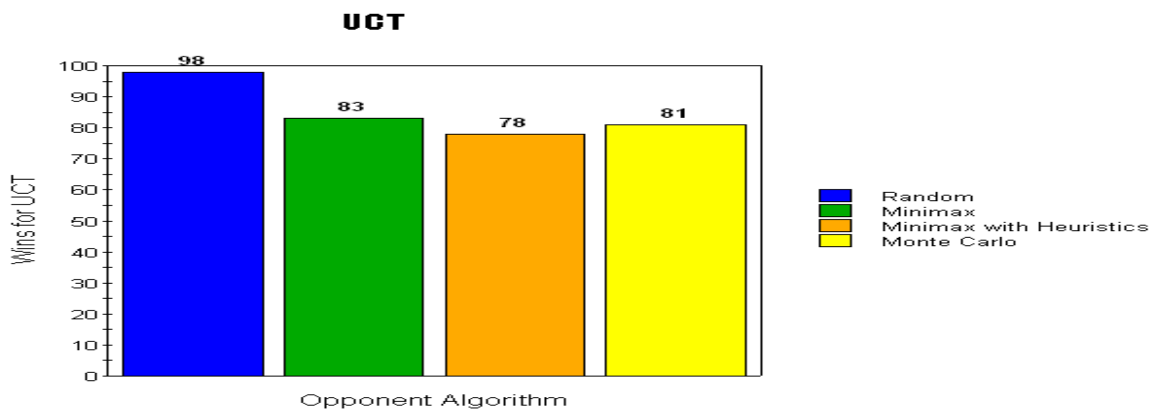
Test Results:


Random
Wins for Random
Opponent Algorithm
Minimax
Minimax with Heuristics
Monte Carlo
UCT


Minimax
Wins for Minimax
Opponent Algorithm
Random
Minimax with Heuristics
Monte Carlo
UCT


Minimax with Heuristics
Wins for Minimax with Heuristics
Opponent Algorithm
Random
Minimax
Monte Carlo
UCT


Monte Carlo
Wins for Monte Carlo
Opponent Algorithm
Random
Minimax
Minimax with Heuristics
UCT


UCT
Wins for UCT
Opponent Algorithm
Random
Minimax
Minimax with Heuristics
Monte Carlo