Artificial Intelligence for Go

CIS 499 Senior Design Project Final Report

Kristen Ying
Advisors: Dr. Badler and Dr. Likhachev
University of Pennsylvania

PROJECT ABSTRACT

Go is an ancient board game, originating in China – where it is known as weiqi - before 600 B.C. It spread to Japan (where it is also known as Go), to Korea as baduk, and much more recently to Europe and America [American Go Association]. This game, in which players take turn placing stones on a 19x19 board in an attempt to capture territory, is polynomial-space hard [Reyzin]. Checkers is also polynomial-space hard, and is a solved problem [Schaeffer]. However, the sheer magnitude of options (and for intelligent algorithms, sheer number of possible strategies) makes it infeasible to create even passable novice AI with an exhaustive pure brute-force approach on modern hardware. The board is 19 x19, which, barring illegal moves, allows for about 10^{171} possible ways to execute a game. This number is about 10^{81} times greater than the believed number of elementary particles contained in the known universe [Keim].

Previously investigated approaches to creating AI for Go include human-like models such as pattern recognizers and machine learning [Burmeister and Wiles]. There are a number of challenges to these approaches, however. Machine learning algorithms can be very sensitive to the data and style of play with which they are trained, and approaches modeling human thought can be very sensitive to the designer's perception of how one thinks during a game. Professional Go players seem to have a very intuitive sense of how to play, which is difficult to model. However, currently it seems the best algorithms are those from the Monte Carlo family, which attempt to develop order (in the form of a strategy) from the chaos of random game simulations. Programs such as Crazy Stone and MoGo have demonstrated the promise of these algorithms. It is expected that computer programs may outperform professional Go players within the next decade [Keim].

This project is an investigation into designing AI for the board game Go, and implementing a Monte-Carlo based artificial intelligence, as well as implementing some other approaches. The final deliverable product is a program that allows AI

"death matches", pitting any two implemented AIs against each other. It uses XNA for graphics and user input, and also has a two-human-player version and a human-versus-computer version.

Project blog:
http://kaising.wordpress.com

1. INTROUDUCTION

Computers have beaten pro human chess players through their superior computational ability. With the ancient board game Go, however, the sheer number of possibilities, as well as the subjective nature of what a more desirable board state is make it a more challenging problem. On February 7th, 2008, a computer finally beat a pro Go player – but with a 9-stone handicap on the human player, and with processing power over 1000 times that of Deep Blue [Guillame].

1.1. Significance of Problem or Production/Development Need

The purpose of this project is to investigate the field of AI, to determine if it is an area of personal interest for future work as a Master's student. The secondary purpose is to work with XNA and implement a game, to gain personal experience in preparation for entry into the video game industry.

1.2. Technology

This project uses C# and XNA to produce code. The main implemented paper is the technical paper written by the creators of MoGo, However, researching many approaches from various sources is an important aspect of this project as well.

1.3. Design Goals

.

1.3.1 Target Audience.

The target audience of the final product is mainly myself and colleagues; also people who are interested in computer Go. The aim is not to be the best Go AI, but to investigate approaches to AI as a player new to Go and the field of AI, and what can be done on the limited computational resources of a laptop.

1.3.2 User goals and objectives

The user is able to play a game of Go against any of the AIs, or against another human, as promised in the project proposal. Here the goal of the user(s) is to win the game, using knowledge of the game Go. The user may also pit two AIs against each other, and watch the “AI deathmatch”.

1.3.3 Project features and functionality

The main features of the game are 1. The ability to pit AIs against each other, 2. the ability to play Go against an AI, 3. supporting human vs. human matches, and 4. a visual interface for the game on the XBox 360.

2. Prior Work

One of the first, if not the very first implementations of artificial intelligence for Go was by PhD student Albert Zobrist in 1970. It used two influence functions to assign numeric values to the possible move locations. One influence function was based on which colors occupied what locations; it gave +50 to a location with a black stone, and – 50 to a location with a white stone. Then, for four iterations, positions received -1 for each adjacent location with a negative value, and received +1 for each adjacent location with a positive value. The other influence function was based on which locations were occupied. Based on available information, then, the program pattern matched against a database. Via scanning the board searching for various rotations of each pattern, it would decide what the next move should be. In order to make the program’s decision more sound, the game was divided into stages (e.g. beginning, endgame, etc.), and only patterns appropriate for the given stage were searched for. Some lookahead (3 moves) was added to incorporate particular aspects of the game that require some planning (e.g. saving/capturing strings, connecting/cutting strings, ladders, making eyes). This program was able to defeat novices. Some other earlier programs were also based on variations of influence functions and pattern matching, though a number tried to account for more aspects of the game, such as attempting to build models analogous to the way Go players structure their perception of the game [Burmeister & Wiles].

Pattern matching programs still have a place. Bill Newman’s open-source Computer Go program Wally, for example, can be a useful beginning training partner for machine learning techniques. Its tunability and straightforwardness but decent number of pattern recognizers make it playable, but not too hard. It relies on pattern matching, with no lookahead [Newman].

In a slightly different approach, other subsequent algorithms investigate the concept of life and death, detecting stones that are “live” – i.e. remaining on the board. BensonsAlgorithm, for example, finds stones that cannot be captured by a given player’s opponent, even if that given player does nothing to aid these “live” stones. These algorithms can be very fast [House].

There are still many approaches that make at least some use of pattern matching, such as observing connections between stones, or even analyzing patterns to make a finite state automaton as the program GNU Go does. There are also programs that make use of heuristics based on board configuration, and center-of-the-board vs. edge-of-the-board placement, and using techniques such as the physics center of gravity of a group to find its center. For example, it may be advantageous to place stones along the edge of an area that is heavily dominated by the other player. Yet other algorithms focus more strongly on these areas of dominance. The term “influence” of a piece is used to describe its significance on the board – e.g. a nearly captured stone has close to zero influence, whereas one in the center of the board with few neighbors may have high influence. The exact interpretation of “influence” depends on the implementation, but it is generally a long-term effect, and stronger groups have greater influence on the surrounding area. An area where a player has a lot of influence, i.e. an area that is likely to become a player’s owned territory, is called a *moyo* or framework. A variety of algorithms are based on these concepts, such as GNU Go’s influence function that computes how likely an area is to become fully captured territory for a given player. Bouzy’s 5/21 algorithm, which borrows from computer vision techniques, also seeks to calculate influence [House].

Furthermore, Alistair Turnbull pointed out that if a given player tends to claim more of a certain region of the board when stones are placed randomly, this suggests that this player has influence in that region. Thus this concept of influence/moyo is relevant to some random methods as well, such as Monte-Carlo approaches [House].

Crazy Stone by Rémi Coulom uses Monte-Carlo Tree Search and in 2006 began the current trend of using algorithms from this family. The Monte-Carlo method creates playouts, or played games with random (light playout) or heuristic-based (heavy playout) moves. Applied to game trees, each node keeps a win rate, remembering the number of playouts that were won from this position. As is often desirable in Go, such analysis favors the potential of winning, regardless of the potential margin by which the game may be won. Thus such algorithms may often result in winning by a small margin. Crazy Stone also utilizes pattern learning [Burmeister & Wiles].

The program MoGo, which received some early inspiration from Crazy Stone, made headlines when it defeated a pro player in a full-scale 19x19 game of Go on February 7th, 2008. This program uses the algorithm UTC (Upper Confidence bounds applied to Trees) for Monte-Carlo [Gelly]. This algorithm is useful for balancing the effort expended in finding new moves to explore, and exploring the promising-looking moves that have already been discovered. This is possible because each potential move, corresponding to a branch in the decision tree, is considered a one-armed bandit machine (i.e. like a slot machine with a reward rate) [Bentley]. The problem of exploring options at any given turn can then be considered as evaluating a multi-armed bandit situation. Like a gambler with a multi-armed slot machine, the algorithm must balance between pulling a lever/arm that it has determined to have a good payoff rate (exploring a promising move), vs. trying other arms in an attempt to find a better payoff rate (exploring largely unexplored moves).

There is also a commercial computer program called “Many Faces of Go” that uses a variant of Monte-Carlo Tree Search, indicating that this can be at least playable without the great computing power that MoGo had access to. However, this project has been many years in development, so its generation may be beyond the time constraints of this project.

The main challenges of Go in comparison to other board games seem to be the sheer number of possible moves for each player, as well as the issue of evaluating how “good” a given board configuration is for a player. For example of the latter, even a beginning player may be able to recognize the potential for a stone pattern called the ‘ladder’ and look 40+ turns ahead to see how much it could benefit them (a very deep search for a naive branching algorithm). Thus encoding such evaluation in a program is not trivial.

There are many approaches to reduce the size of move searches, while still retaining strong predictive powers. In addition to the fairly well-known alpha-beta search, there are algorithms that make use of a technique known as Zobrist Hashing. This is a way to implement hash tables indexed by board position, known as transposition tables [Wikipedia]. Some have considered more unusual techniques, such as making use of Markov Chains [House]. A Markov chain is a stochastic process having the Markov property, which in this context means that the current state captures all information that could influence future states – which are determined by probabilistic means. This could certainly apply to the current state of a Go board. But so far UCT seems to be empirically the most successful at reducing move search scope.

The machine learning technique of genetic programming has been investigated as well. Genetic programming involves generating programs that are hopefully successively more “fit” at solving the problem, through mutating the code in fixed ways (that allow it to be a program) and evaluating the fitness of individuals at each generation via a fitness function. John Koza has popularized a method of ensuring that mutated programs are still syntactically correct. In one attempt has been made to apply genetic programming to Go, where the program consists of functions of the form IfPointAt(x,y,z). x, y, and z are each a triple, representing board position, color at that position (empty, white, or black), and an action (pass, resign, move). The function proceeds to check the condition x; if it is true, it executes y, otherwise it executes z. The fitness function is responsible for not only evolving good strategies, but also teaching the rules of Go (e.g. not playing where there is already a stone in place). Mutations included mutating x, y, z parameters between constants and nested IfPointAt expressions, mutating constant values in expressions, and moving around portions of the expression tree. However, this experiment showed that genetic programming may have a very difficult time producing good Go players (those that worked often resigned after only several moves). The challenges to this approach include the enormous number of possible expression trees [Greenberg]. Further efforts have been made, but it seems unlikely that genetic programming will become as competitive an approach as UCT in the near future.

Neural networks may be more successful. Although backpropagation is difficult due to the problem of credit assignment (i.e. which moves receive credit for a win, and which moves receive blame for a loss), Richards et. al. have created a scheme which sidesteps this issue. Their algorithm has a population of neurons, units of strategy that store connections to subsequent neurons, and also a population of neuron blueprints, which use neurons to form larger strategies. Neurons receive a fitness score based on the success of the multiple blueprints in which they are used. “Elite” or the most successful units within each population (neurons and blueprints) are bred by selecting pairs of them, and having their two offspring replace less successful ones. Various operations are performed to maintain variety as well. This particular solution is called SANE (Symbiotic Adaptive Neuro-Evolution) and has successfully been used in areas such as robot control and the simpler Go-like game Othello. SANE made good progress against Newman’s Wally program; it learned to defeat the pattern-matching program in 20 generations for a 5x5 board, and 260 generations (taking 5 days on their hardware) on a 9x9 board. It seems that neural network approaches such as this can learn much faster with a more deterministic opponent; however this may not be desirable for creating a program for humans to play against. Furthermore, the authors estimated that training it to a 19 x 19 board would have taken over a year on the available technology (with the paper being dated 1997) [Richards et. al.].

Using genetic algorithms to create an evaluation function is another approach. An evaluation function reports an estimate of how advantageous a given board configuration is for a given player. Per Rutquist’s paper, “Evolving an Evaluation Function to Play Go,” describes a way of applying Genetic Algorithms [1997]. It uses a vector of patterns and machine learning to evaluate the fitness of a set of weights for the pattern set. Weights indicate the importance of the pattern. Then A genetic algorithm is used to evolve the pattern set. This program learned to do well with training/test sets, but reportedly did not play well overall (against GnuGo). It had scores of -80 with no training, then -20 with training. Even this improvement, though, is noticeable, and it is interesting to see that it is possible.

However, genetic approaches do not seem to be at the forefront of competitive research, as of the creation of this report. Because of the extreme complexity of calculations, and somewhat subjective, intuition-based skill employed by skilled human players, Go programs that seek to closely emulate natural or human thought processes have received far less attention recently – at least within the realm of competitive Go programs. The Monte-Carlo technique, applied to many games to generate statistical information, seems to be the most promising for defeating top human Go professionals. It has been estimated that programs in this family will become greater Go players than humans within the next decade [Keim].

3. PROJECT DEVELOPMENT APPROACH

3.1. Algorithm Details

3.1.1. Alpha Version

An outline of the old plans for the AI algorithm:

1. When it is the computer's turn, take the state of the board, and play out the move tree to the end of the game* many times using random placement**.
2. Use A* Search*** to pick a move in the tree with the empirically best win rate from step 1. Win rate = (number of games won in that branch) / (number of games total in that branch).
3. Keep the subtree for that move, update it based on the opponent's move****, and then elaborate on the updated sub-tree for the computer's next move.

Notes:

* For now, the end of the game will be defined as when no more legal moves exist.

** This will eventually use pattern recognizers, to make the chosen simulated moves of each player more intelligent. An example is: when it is possible to capture a large group of stones, do so.

*** This will eventually be an intelligently pruned search, where it stops investigating a branch once it appears to be loosing terribly. Given the nature of Go, in which large areas may be captured in a single move, however, it may be difficult to determine what makes a board state unfavorable enough to stop searching. The pruning, therefore, may be very simple.

**** This step, keeping the tree, makes the search incremental. One area I am not certain about, however, is what to do if the opponent takes a move that was not investigated during the previous turn, and is not included in the tree. I do not see any way to update the existing tree yet in that case.

An additional aspect that would be desirable is to have some persistent accumulated statistics. These are not necessary, but could be used to select more intelligent random moves when simulating games and adding to the decision tree.

3.1.2. Final Version

Rather than investing more time in this one simple AI, effort has been spent (with approval) investigating various AIs and how they perform against each other. Please see Section 5, “Results”, for a description of the algorithms and how they fared.

3.2. Target Platforms

3.2.1 Hardware

XBox 360; though the program can be developed and run on my laptop.

3.2.2 Software

C# and XNA

3.3. Project Versions

3.3.1. Project Milestone Report (Alpha Version)

Literature Review / Research: I researched decision trees and previously explored approaches to Go AI to gain knowledge in this area. The most useful resources in designing the general approach of the program thus far have been my advisors. Burmeister and Wiles' "Computer Go Tech Report" was useful in providing overviews of various approaches to Go AI, past and present. Andrew Moore's slides, "Decision Trees and Information Theory" are very useful for understanding decision trees in more detail. The site "Sensei's Library" has also been a great resource; in particular, I recently discovered that there is at least one Go novice who is creating a Go AI program and documenting his progress. I intend to look further into the posts on this site as I seek to improve my program. Also, I haven't read the XNA resources yet.

Two-Human-Player Go: I wrote a program that enforces the rules of Go and allows two human players to complete a game of Go. The display and input are through the command line and an ascii representation of the board.

Computer Go: I added a one-player mode to the game, where a human player may compete with artificial intelligence. The artificial intelligence is very basic, using randomization and a simple analysis of the resulting win rate of possible moves. It builds on a decision tree with each turn. However, if the human player takes a move that was not explored previously in the tree, the computer begins calculating data from scratch again. ***This could be avoided by requiring the computer to evaluate every legal move for its turn, and subsequently every legal move for the human player's next turn. However, this would still only save the data generated for the one branch that is actually taken (determined by the computer's next chosen move and then the human player's (unpredictable) next move).

3.3.2. Project Milestone Report (Final Version)

Code Overhaul: This program was completely restructured, salvaging some data structures and functions from the Alpha Review version. This new design allows different AIs to easily be added in, and also switched between. It also now follows a structure that meshes well with XNA.

XNA: The visuals are now in XNA, and the program takes input from the XBox360 controller. There is a 2D view that gives a nice, clear view of the board for all game versions, and a "prettier" 3D version that can be switched to.

New AIs: There are now a variety of AIs to change between. Most of these are simple AIs, but they still gave insight into the nature of the problem. See Section 5, "Results" for a description of the AIs and their performance. The Computer Go from the Alpha Review has been transformed into "Naïve Monte-Carlo AI", and is also mentioned in that section.

AI Deathmatches: The game now supports pitting AIs against each other, and watching the game play out between them.

3.3.3. Project Final Deliverable Changes

This section is similar to that of my project proposal. The only differences are that, instead of focusing on gameplay testing on the XBox360, it has the feature of "AI deathmatches". The reduced focus on XNA was approved at the Alpha Review, and the addition of AI deathmatches is an improvement.

4. WORK PLAN

4.1.1. Project Milestone Report

01.21.2009	Meeting with Dr. Likhachev to discuss project concept
01.26.2009	Acquisition of basic knowledge of Go
02.06.2009	Reading through background papers and those recommended by Dr. Likhachev; final evaluation on which algorithmic approach to take *Evaluation of whether Go is doable, or if a different game should be selected
02.11.2009	Investigation of XNA's capabilities, bare bones C# data structures / rules begun (should be able to play a 2 person command line version of Go).
02.16.2009	Begin coding AI
02.23.2009	Pre-alpha evaluation of what is reasonable to expect by alpha version
03.16.2009	Finished alpha version (playable)

Solidify what will be expected of final version
More coding and evaluation

(This is the same as in the project proposal, except that the last item has been combined with the first two items that were under "Project Final Deliverables" and all have been put under the same date.)

4.1.2. Project Final Deliverables

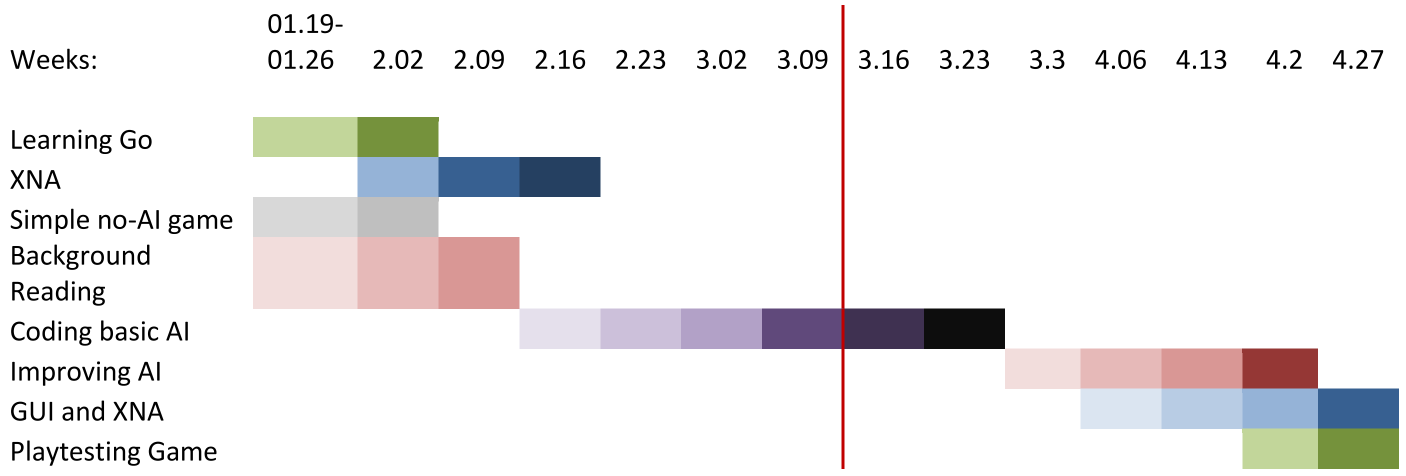
03.23.2009	Clean up code – combine two-human-player code with one-player code cleanly.
03.30.2009	further work on improving AI based on advisor suggestions and further investigation into resources such as Sensei's Library
04.06.2009	XNA experimentation
04.13.2009	Testable 'finished' product
04.20.2009	Evaluation of product, more AI implementation
04.27.2009	Incorporation of findings from evaluation

(Altered since project proposal to reflect greater focus on AI and less on interface / XNA)

4.1.3 Project timeline.

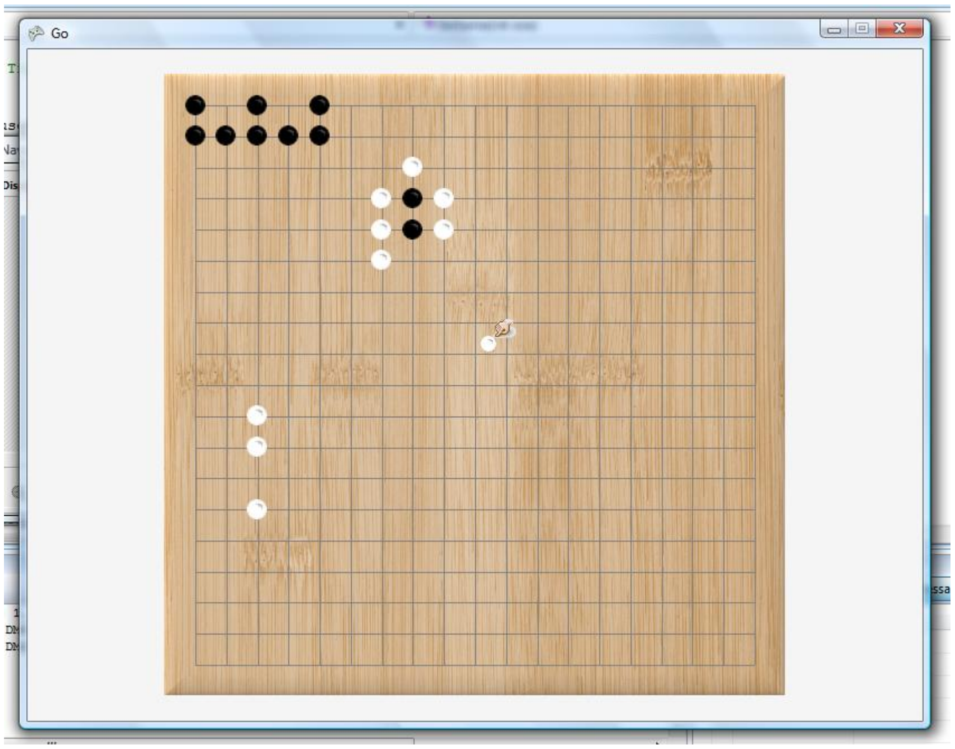
Please see 4.1.1 and 4.1.2 above.

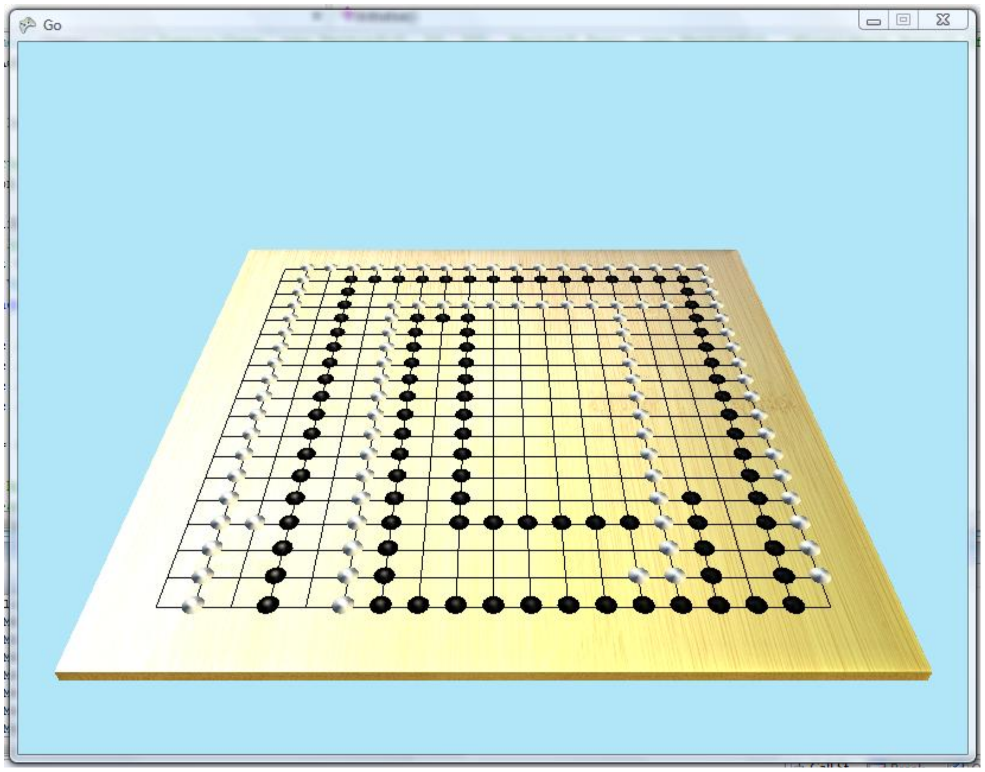
4.1.4 Gant Chart



5. Results

Visuals: XNA views in 2D an 3D are functional, taking input from the XBox360 controller. XNA was found to be very easy to learn, and for the purposes of this project a good visualizer tool.





Modes: The game supports human vs. human games, human vs. AI games, and AI vs. AI "deathmatches." AIs can also be pitted against themselves in a "deathmatch." The user can swap out AIs for human vs. AI games, and AI "deathmatches" at any point, and also swap out human vs. AI players at any point. For example, they may play part of a game against a human, switch out to an AI as their component for a few turns, then switch out themselves to have AIs finish the game, etc. It is trivial to hook in new AIs; it just must guarantee that it attempts a legal move. The framework guarantees that there is still a legal move when it passes control to the AI.

AIs: I implemented a number of AIs. Genetic AI (based on the paper by Per Rutquist) has been cut from the game, as it is very much unfinished, but the rest of the AIs have been left in. Some descriptions here and results are mostly from my blog, so those sections have been cut from Section 7, "Work Log."

Please see the PowerPoint slides and blog for more media and data.

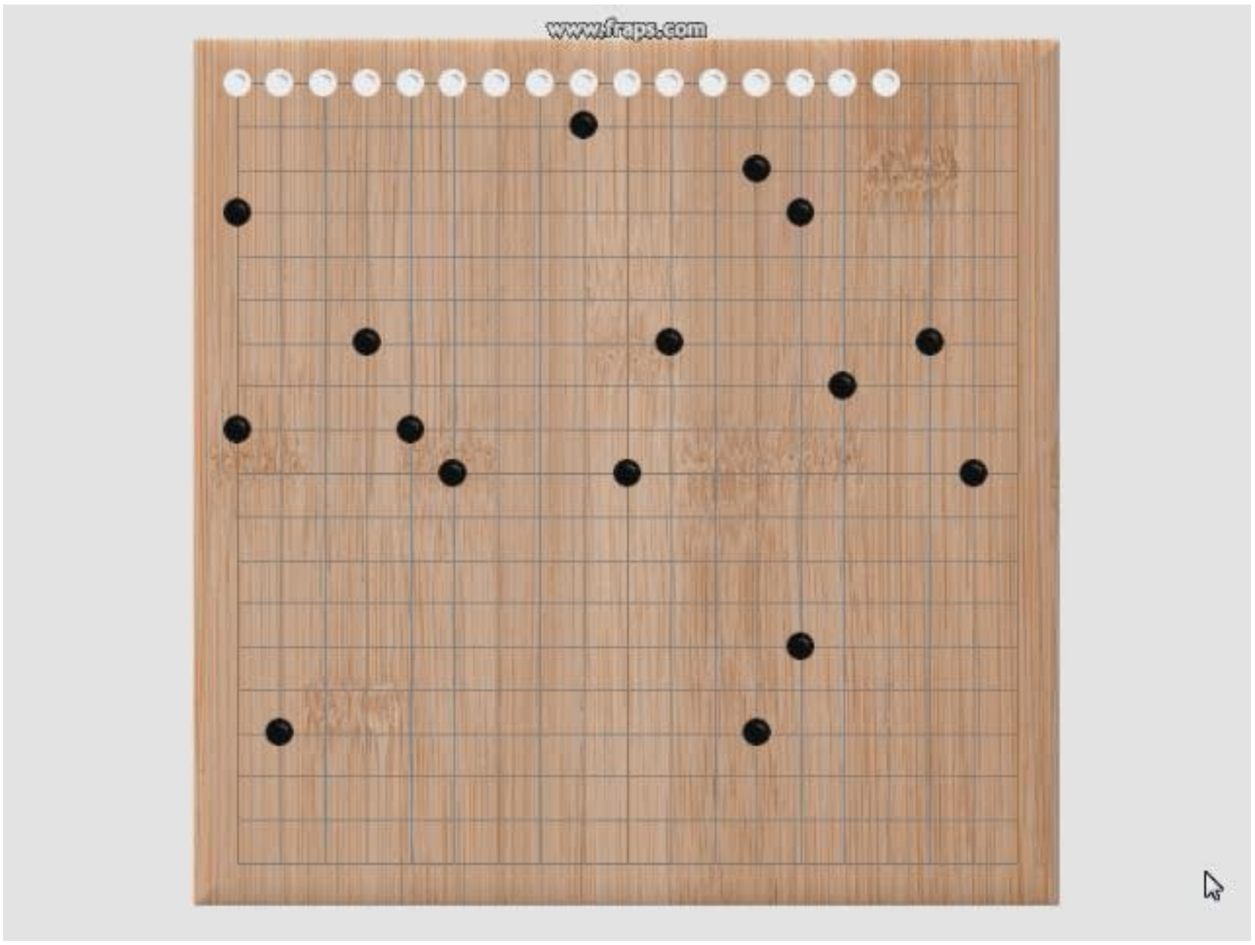
Random AI:

This was the first toy AI. It randomly places legal moves on the board. This is an ideal "control" tester for the other AIs. Intuitively, a standard for whether AIs show any intelligence at all could be, "do their decisions result in outperforming random moves?"

Crawler AI:

This AI places stones left to right, in successive rows, avoiding any stones in its way. It tends to win against Random, showing that even a simple strategy can accomplish some objectives. More concretely, the strength of this weak approach is that it can create "eyes" as it swallows up lone stones place by Random or other simple AIs. Crawler vs. Crawler matches are deterministic, and not that exciting. This AI was very useful in testing out the AI deathmatch framework against Random, however.

Testing with Toy AIs (See PowerPoint slides for movie):



Naïve Monte-Carlo AI:

This is the AI from my Alpha Review, reformatted to mesh well with the new code. The only non-obvious aspect of integrating this AI was how to deal with updates to the root of the move tree that persists between turns. Originally, in the alpha demo, updates were made to the existing tree based on which move was taken by either player. If the taken move was already explored, the tree was updated to have that node as the new root. This update is still trivial to implement when the computer takes a move, as it simply compares its computed move to the children of the current tree root (which is stored in the object). However, additional steps would need to be made for an update to be called in the controller, passing the move that the opponent took, to the NaiveMonteCarlo AI(s) currently playing. The GoAI base class could be updated to have an update(Move m) function, but for now it seemed better to handle this entirely within NaiveMonteCarlo, since it is just one of the simple test AIs. To handle this issue, NaiveMonteCarlo's takeTurn override function currently infers the opponent's move by comparing the current configuration of the board to the previous configuration of the board (stored at the end of its most recent turn).

NaiveMonteCarlo is fairly slow at this point, when combined with the graphics updates in XNA. However, it is still watchable, and more so on a small board (e.g. 10×10). In terms of comparison with Random and Crawler, NaiveMonteCarlo has so far outperformed Random, and usually outperforms Crawler. NaiveMonteCarlo seems to show the beginnings of 'intelligent' move choices toward the end, when there are fewer choices left, and it is more likely to find differences in win rate for given moves. Crawler has probably generated large contiguous blocks of stones by this point, which NaiveMonteCarlo considers – instead of assuming the opponent is random and simulating the moves in that manner. Therefore, it is easier for it to determine what moves are likely to result in capturing large blocks, and thus is better at acquiring points.

It makes sense that NaiveMonteCarlo does well against Random, since the moves it simulates for its opponent are random. It was curious to see how NaiveMonteCarlo does against Crawler, etc. if it simulates the opponent's moves (and possibly even its own) using an algorithm similar to Crawler's (or whichever AI it is being tested against). However, this was just for the sake of experiment, as the AI usually shouldn't have the advantage of knowing the opponent's strategy (especially when it is deterministic, like Crawler's). See "Heavy Monte-Carlo AI" for the results of such "heavy" playouts.

Greedy AI and Greedy Randomized AI:

I added Greedy AI into my AI deathmatch program, partly inspired by some of the ideas from the user "Holigor" at Sensei's Library. It doesn't look ahead at all, and merely follows a simple greedy heuristic for each turn it takes. Currently it uses the following criteria:

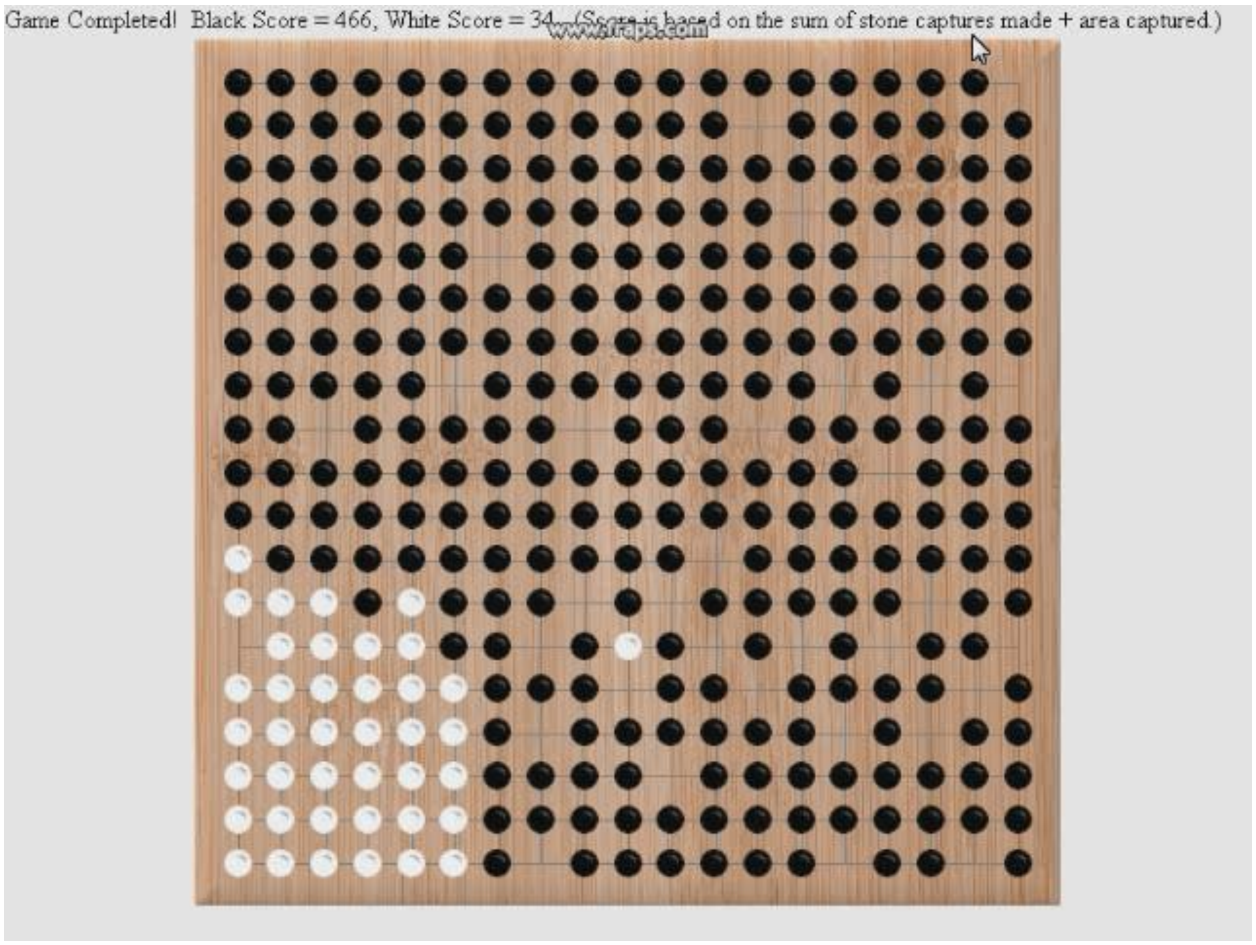
1. If an opponent's stone(s) can be captured, do so; capture as many as possible with one move.

2. Otherwise, find the largest group of friendly stones and increase the group's liberties as much as possible.

During testing, this approach won against Random AI, Crawler AI, and even Naive Monte-Carlo AI. Similar sucess for greedy algorithms seems to have been found by other novice players coding AI for Go as well, on Sensei's Library. I think that Naive Monte-Carlo AI might do better against Greedy AI if it were allowed to play more game simulations per legal move. Although this would make watching playouts between Naive Monte-Carlo and Greedy tedious, even competitive Monte-Carlo-based approaches like MoGo may give the computer 12 seconds or so to determine a move.

Greedy AI is deterministic; to make it more interesting I added in some randomization. Greedy Randomized AI builds a list of the "best so far" capture moves and also a list of "best so far" defensive liberty-increasing moves. The values associated with each (number of captures, etc.) are stored in lists as well, where indexes match those of the move lists. Once it has checked every legal move, it first looks for the best capture value (which is at the end of the list). If this is >= 1 (i.e. capture(s) can be made) it looks toward the beginning of the list, until it finds a move with less-than-maximum captures (they are contiguous, the way moves are added on). Then a move is randomly selected from this "best captures possible" list. If no captures can be made, a similar process is performed for finding a best liberties-increasing move. This algorithm is a little thrown together, but it's very fast and works so far.

This algorithm is fairly successful at boxing off areas of the board, and not placing stones on spots that are already have all their liberties taken by a friendly stone. Greedy avoids this because such a move would not increase its liberties (and could decrease them). "Eyes" like this are highly desirable (since two of them in a group will make it impossible for the group to be captured); perhaps the algorithm could be tweaked to specifically attempt to form them, while still remaining a greedy algorithm without lookahead. This algorithm is also useful for testing Monte-Carlo AIs.


Game Completed! Black Score = 466, White Score = 34 (Score is based on the sum of stone captures made + area captured.)

Greedy Score AI:

One of the AI variants I added prior to my presentation was Greedy Score AI. This approach was similar to Greedy Randomized AI, except it made its greedy choice based solely on the score of the board that would result if a given move were taken. It simply simulates each legal move it could choose from, scores the resulting board, and finds the move(s) with the highest values. If there is more than one tied, it selects one of them randomly. This AI tended to build clusters, but was far weaker than Greedy AI / Greedy Randomized AI. This was because it lacked the "boxing in" behavior that arose (somewhat indirectly) from the simple, novice-like heuristics that are in the other greedy AIs. So it seems that simple, reasonable heuristics are more effective that attempting to determine the "goodness" of a board state, such as by scoring. This may be sound reasoning for why Distance AI is struggling so much.

Heavy Monte-Carlo AI:

Another AI variant I added prior to my presentation is Heavy Monte-Carlo AI, which is similar to Naive Monte-Carlo AI, but has heavy playouts. Heavy playouts refers to the usage of some heuristics/patterns/"AI" to select the stone placements, instead of having completely random playouts. I used the Greedy Randomized AI as the selector. However, this approach made the AI very very slow; I would argue that Naive Monte-Carlo is better because of its speed (which allows it to compute statistics from a greater number of simulations performed in the same amount of time). Also, I think one must be very careful in choosing how to bias heavy playouts. Biasing them in a way that is far from the way moves will actually be made may confine the program to a more narrow strategy, whereas randomized playouts allow patterns to emerge on their own. The MoGo paper does describe making use of patterns in the random playouts, which improves them by making each playout more representative of reality. However, without carefully selected and tweaked patterns, I think AIs in this project are better off with fast, randomized playouts when playouts are needed.

Novice AI:

Novice AI was an experiment similar to Greedy AI, that uses a novice's heuristics in placing the next stone – such as trying to maximize its total number of liberties. However, this is an example of what a difference is cause by variations in the particular huristics used. Novice AI did not even create clusters, like Greedy Score AI; its priority for creating liberties was strong enough that it just placed stones on the board at spaced out intervals. Of course these stones were left undefended, and this heuristic, which sounds useful, actually was extremely detrimental and prevented it from even forming groups. I just included this toy AI as an illustrative example to use in my presentation.

Distance AI:

This AI is based on an evaluation function, and works as discussed in my blog. It uses a library of boards that have a precomputed “win rate”. For a given potential move, Distance AI evaluates how “good” that move will be by simulating it, and then finding boards in the library similar to the resulting board.

The library is generate if needed, serializing to save the resulting object to a file. In the event that the library is not generated, it just deserialize the saved file. If generating a library, the program reads in all the files currently in a specified directory. It parses each read file using Phil Garcia’s parser from GoTraxx. Then it saves records of type LibraryBoard, storing a board configuration in the format used by the AI, and an empirically calculated win rate.

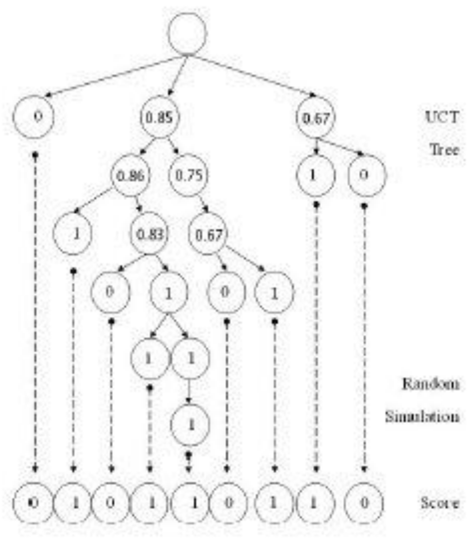
This AI is not very talented still; I think it is mainly because it is difficult to make a working Distance function, and also difficult to create a library that will give good results for any board configuration. The library, however, could be very useful if I choose to finish Genetic AI, as it needs boards and results for training the weights of its weighted pattern-search vector.

UCT / Monte-Carlo AI:

The idea is to view the problem of deciding which move to take next as a multi-armed bandit situation. The analogy here is that of a gambler with many machine arms to choose from – each with different playout rates. His problem is to balance exploiting promising-looking arm(s) he has found to have the best empirical playout so far (exploitation) vs. exploring new arms to try to find one with a better empirical payout (exploration). This exploration vs. exploitation issue is seen in deciding the next move to take in a Go game as well, when using Monte-Carlo methods. Namely, the AI should find a good balance between finding out more about the moves that look best thus far (exploitation) vs. exploring new moves to try and find a better one.

The UCT/Monte-Carlo technique described in a MoGo paper by Gelly et. al. in escence seeks to balance these aspects in the way it creates a random graph. It is based on calling as many simulations as it has time to complete; the process can be stopped and the graph (tree) evaluated at any point. A single simulation consists of going down the existing tree depth wise, selecting transitions and therefore nodes (which represents board states / moves) . Selection is made via upper confidence bounds (UCB), to favor both unexplored moves, and existing ones that look promising based on previous simulations. When the UCB move-selector picks a move that hasn’t been explored yet (i.e. doesn’t have a node yet), a new leaf is created for it. Then one random Monte-Carlo simulation is played out from this node to the end of the game. The wins and visits counts of the new leaf node and all its friendly ancestors (all those that represent moves executed by this AI, so every-other node) are updated based on whether the one Monte-Carlo evaluation is a win or a loss. There is a possibility of reaching the end of the game while in the process of using UCB to find already-explored nodes (e.g. the move within the one leaf we add ends the game). In this case, I simply update the leaf and friendly ancestors with whether it won or not (even if the leaf was an enemy move). Some aspects of this algorithm may be slightly different from the approach taken in the MoGo paper, but is very closely based on the UCT pseudocode found within it.

Digram of the random tree built by this UCT / Monte-Carlo process, from the paper "Modification of UCT with Patterns in Monte-Carlo Go" by Gelly, Wang, Munos, and Teytaud:



This approach performs well; it seems to have many advantages of the Naive Monte-Carlo AI, but is much much faster in evaluating the board.

6. REFERENCES

Learning Go:

American Go Association. "A Very Brief History of Go." *American Go Association*. URL: http://www.usgo.org/resources/gohistory.html. March 13, 2009.

British Go Association. "Introduction to the Game of Go." *British Go Association*. URL: http://www.britgo.org/intro/intro1.html. April 16, 2008.

Mori, Hiroki. "The Interactive Way To Go." *PlayGO.to.* URL: http://www.playgo.to/interactive/. 1997-2001.

Reiss, Michael. "Go in Five Minutes Flat." *Mick's Computer Go Page*. URL: http://www.reiss.demon.co.uk/webgo/rules.htm. 2004.

Existing Programs and Algorithms:

"AITopics/Go." *Association for the Advancement of Artificial Intelligence.* URL: http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/Go. January 22, 2009.

Has useful summaries and links for topics such as Crazy Stone and MoGo.

Bentley, Peter. "Letting Stones Go Unturned." *PASCAL - Pattern Analysis, Statistical Modelling, and Computational Learning*. 06 Feb 2008. 25 Apr 2009 <http://pascallin.ecs.soton.ac.uk/article4.pdf>.

Burmeister, Jay, and Wiles, Janet. "CS-TR-339 Computer Go Tech Report." *The University of Queensland Australia*. URL: http://www.itee.uq.edu.au/~janetw/Computer%20Go/CS-TR-339.html#3.0. 1995.

This describes an overview of various academic implementations of Computer Go, as well as more recent competitive ones.

Chaslot, Guillaume. "Supercomputer with innovative software beats Go professional." URL: http://www.cs.unimaas.nl/g.chaslot/muyungwan-mogo/. January 22, 2009.

Information on MoGo.

CiteULike. "Group: computer-go." *CiteULike.* URL: http://www.citeulike.org/group/5884/library. January 22, 2009.

There are 420 articles here, many of which refer to Monte-Carlo – based techniques, and a few of which refer to genetic algorithms. Once it has been decided which will be used as resources, those chosen articles will be listed as individual sources.

Coulom, Rémi. "Efficient Selectivity and Back-up Operators in Monte-Carlo Tree Search." *Rémi Coulom's Home Page.* URL: http://remi.coulom.free.fr/CG2006/. January 22, 2009.

This is the page for Crazy Stone, by programmer Rémi Coulom. It includes a technical paper describing the program.

Garcia, Phil. “GoTraxx.” www.ThinkEdge.com. http://www.thinkedge.com/blogengine/page/GoTraxx.aspx. April 30, 2009.

Phil Garcia’s Computer Go program, GoTraxx. He was very helpful in the process of building a library for my program, which uses his SGF parser.

Garcia, Phil. "[computer-go] SGF parsing." 09 Jul 2007. computer-go.org. 28 Apr 2009 http://computer-go.org/pipermail/computer-go/2007-July/010542.html .

Provides links to SGF parsers, including a C# one the author wrote.

Gelly, Sylvain. “MoGo.” *Sylvain Gelly’s Home Page*. URL: http://www.lri.fr/~gelly/MoGo.htm. January 22, 2009.

Home Page of one of the programmers behind MoGo; contains links to a technical report and other information.

Gelly, Sylvain, and Wang, Yizao. “Exploration and Exploitation in Go: UCT for Monte-Carlo Go.” *University College London.* URL: http://www.homepages.ucl.ac.uk/~ucabzhu/workshop_talks/mogoNIPSWorkshop.pdf. December 9, 2006.

Has high-level Monte-Carlo algorithms and mentions some issues.

Gelly, Sylvain; Wang, Yizao; Munos, Rémi; and Teytaud, Olivier. “Modification of UCT with Patterns in Monte-Carlo Go.” *INRIA.* URL: http://hal.inria.fr/docs/00/12/15/16/PDF/RR-6062.pdf. April 28, 2009.

Contains the algorithm / pseudocode use for UCT MC AI.

Gorobei. “A Novice Tries To Write A Go Program.” *Sensei’s Library*. URL: http://senseis.xmp.net/?ANoviceTriesToWriteAGoProgram. March 10, 2009.

Some musings by the user Gorobei, and replies by other users. These notes are useful as a starting point to investigate various approaches to writing AI for Go – for example, he lists some common approaches and a few of their issues in the beginning.

Greenberg, Jeffrey. "Breeding Software to Play the Game of Go." *Inventivity*. 26 Apr 2009 http://www.inventivity.com/OpenGo/Papers/jeffg/breed.html .

Useful overview of genetic algorithms and how they may be applied to Go. Does not have an actual working solution, but highlights many of the challenges and assumptions that may be needed for further work.

“Holigor.” Holigor’s Domain. http://holigor.fcpages.com/index.html . April 30, 2009.

This is the web page of the "Sensei's Library" user who inspired my Greedy algorithm, with his "crawler."

House, Jason. "Computer Go Algorithms." *Sensei's Library*. 18 Mar 2009. 25 Apr 2009 http://en.wikipedia.org/wiki/Zobrist_hashing .

A very useful overview of many algorithms used in Computer Go, with very brief explanations and links.

Keim, Brandon. "Humans No Match for *Go* Bot Overlords." *Wired Science*. URL: http://blog.wired.com/wiredscience/2009/03/gobrain.html. March 11, 2009.

Interesting article forwarded from Dr. Badler, discussing how Monte-Carlo techniques create order from chaos to play Go well – but don't really model the human mind. Makes a prediction that programs will be capable of defeating professional players fairly consistently within the next decade.

MoGo project. "MoGo: a software for the Game of Go." URL: http://www.lri.fr/~teytaud/mogo.html. January 22, 2009.

Home page of MoGo, which is considered to be the first program to beat a pro player at Go.

"Monte Carlo Tree Search." *Sensei's Library*. URL: http://senseis.xmp.net/?MonteCarloTreeSearch. September 11, 2008.

This describes the family of algorithms used by competitive Go programs such as Crazy Stone and MoGo.

Moore, Andrew. "Decision Trees and Information Theory." Slides, credited as taken from http://www.cs.cmu.edu/~awm/tutorials.

Notes on decision trees and concepts such as entropy.

Newman, Bill. "another simple-minded PD go-playing program." 1988. 27 Apr 2009 <http://www.fzort.org/mpr/projects/wally-ce/wally-orig.c.txt>.

Source code for a simple Go-playing program that relies on pattern-matching, with no lookahead. This has been used as a training partner for some machine learning approaches.

Reiss, Mick. "Go++, the world's strongest Go playing program." *Go++.* URL: http://www.goplusplus.com/. January 22, 2009.

This program features board sizes of 19x19, 13x13, and 9x9. It has features such as handicaps and load/save, and has won several tournaments. It seems that at least Reiss' Go4++ program would require too great computational resources for this senior project.

Reyzin, Lev. "Go is PSPACE hard." Yale University Clique Talk. URL: http://www.cs.yale.edu/homes/lr288/presentations/Go.pdf. March 13, 2009.

Slides describing reductions to prove that Go is polynomial-space hard.

Richards, N., Moriarty, D., and Miikkulainlen R., "Evolving Neural Networks to Play Go." 1997. The University of Texas at Austin. 27 Apr 2009 ftp://ftp.cs.utexas.edu/pub/neural-nets/papers/richards.apin97.ps .

A paper on Neural Networks and Go. Read PostScript file online using http://view.samurajdata.se/.

Rutquist, Per. "Evolving an Evaluation Function to Play Go." *Ecole Polytechnique*. 1997. The University of Texas at Austin. 27 Apr 2009 http://www.lri.fr/~marc/EEAAX/papers/theses/per.ps.gz .

Schaeffer, Jonathan et. al. "Checkers Is Solved." *Science Express*. July 19, 2007. URL: http://www.sciencemag.org/cgi/content/abstract/1144079. March 16, 2009.

Shiba, Kojiro, and Mori, Kunihiko. "Detection of Go-board Contour in Real Image using Genetic Algorithm." *IEEE Xplore.* URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01491921. August 4, 2004.

This is a report available from IEEE Xplore, originally from the 2004 SICE Annual Conference in Sapporo.

Schraudolph,Nicol, Dayan,Peter, and Sejnowski,Terrence , "Nici Schraudolph's go networks." Computational Neurobiology Lab at the Salk Institute. 28 Apr 2009 <http://satirist.org/learn-game/systems/go-net.html>.

Evolving Go players; describes which opponents are useful to train against as well (a random player, Wally with varying levels of randomization, and then Many Faces of Go).

Verma, Vandi, Thrun, Sebastian, edited by Lyle Ungar. "Combining Classifiers: Boosting." Slides based on Rob Schapire's IJCAI'99 talk.

Formalized boosting.

"Zobrist Hashing." *Wikipedia*. Wikimedia Foundation, Inc.. 25 Apr 2009 http://en.wikipedia.org/wiki/Zobrist_hashing.

Explanation of Zobrist Hashing as it pertains to games.

Online Go Servers:

IGS the Internet Go Server. URL: http://www.pandanet.co.jp/English/.

Online Go Server. URL: http://www.online-go.com/.

The KGS Go Server. URL: http://www.gokgs.com/.

Learning XNA and C#:

Dolan, Mike. "XNA: Make your own XBOX games in 10 steps." *Fierce Developer.* URL: http://www.fiercedeveloper.com/story/xbox-make-your-own-games-xna-10-steps-diy. December 16, 2006.

Liberty, Jesse, and Xie, Donald. "Programming C# 3.0, 5th Edition." *ProQuest.* URL: http://proquest.safaribooksonline.com/9780596527433. December 20, 2007.

Microsoft Corporation. "Visual C#." *Microsoft.* URL: http://msdn.microsoft.com/en-us/library/kx37x362.aspx. 2009.

Microsoft Corporation. "Learn XNA Game Studio Express." *Microsoft*. URL: http://msdn.microsoft.com/en-us/xna/bb219593.aspx. 2009.

Omark, Jöran. "XNA Tutorial." *XNAtutorial.com.* URL: http://www.xnatutorial.com/?page_id=46. 2006.

7. Work Log

This material is taken almost directly from my blog, as was suggested. Effort was made to remove entries that were used directly in the "Results" section.

Week 1 (Jan 21, 2009 – Jan 25, 2009)

This week I have been solidifying the aim of my project. I have also been investigating prior implementations of Go programs, after recently acquiring a working understanding of the rules of Go.

The two additional reasources I've used to learn Go are: Introduction to the Game of Go, from the British Go Association, and Go in Five Minutes Flat, a breif overview by Michael Reiss.

The most useful summary of prior Go implementations I've found so far is this one: Computer Go Tech Report by Jay Burmeister and Janet Wiles. Although there is no pseudocode, the general ideas of a number of earlier algorithms are conveyed.

I met with my advisor, Dr. Maxim Likhachev, today for guidance and have begun looking into the algorithms he suggested.

Week 2 (Jan 26, 2009 – Feb 01, 2009) –

I have been looking further into existing work on this topic, and based on this reevaluating options for my own project. So far the most relevant work has been that of students (such as the early projects mentioned in this history) and the commercial game by David Fotland, The Many Faces of Go. I will be meeting with advisers this week to make a final decision on what the expectations of this project will be. My goal is to begin coding by the end of next weekend.

Also, I created my project proposal.

Week 3 (Feb 02, 2009 – Feb 08, 2009) –

After reading a number of explanations of the rules of Go and obtaining a Go board set to practice on, I have compiled a list of the rules that will be considered in this program. I'm including them here also in case anyone reading this is not familiar with Go:

1. The board is a 2D grid consisting of 19 x 19 lines.

2. One player has black stones, one player has white stones. Black takes the first turn, and the players take alternating turns placing one stone on an intersection on the board grid.
3. A player may choose to pass, and not place any stones on the board during their turn.
4. The game terminates when both players pass consecutively.
5. An opponent's stone or connected group of stones is captured (and removed from the board) when that stone / group is completely surrounded by enemy stones. A stone/group is surrounded when all of its liberties (adjacent free spaces) are taken and occupied by enemy stones.
6. At the end of the game, the game can be scored in one of two ways: [1. Area scoring] - each player's score is the number of his stones on the board, plus the number of empty intersections surrounded by his stones. [2. Territory scoring] - stones that are unable to avoid capture (dead stones) are removed and added to the opponent's prisoners (stones capture during the game). Each player's score is the sum of the prisoners he has caught, plus the number of empty intersections surrounded by his stones.
7. A stone may not be placed in a position that will cause it to immediately be captured, such as on an intersection which has all four liberties occupied by opponent stones ("no suicide" rule). The exception to this is when doing so will capture the opponent stones, and thus allow the stone placed this turn to remain.
8. Ko/Eternity - If, on turn n, placing a stone down on a given intersection will result in the same board configuration as turn (n-2) (i.e. the configuration at the end of the current player's previous turn), that move is illegal. Thus two players may not infinitely alternate between the same two board configurations.
9. Seki/Mutual Life - This is when opposing groups share liberties that neither group can fill without leading to the capture of the group. Area left open are draw points (called "domi").

Some aspects of the game are not yet included here. For example, there are designated ways of giving one player a 'handicap', by giving thier opponent a head start of up to 9 stones in predetermined locations on the board. Additionally, players may use a smaller grid, such as 13 x 13 instead of 19 x 19. Such options are not essential to gameplay, however, and furthermore would have very little affect if any on the design of the AI in these early stages.

Assuming I keep this topic after meeting with my advisers this week, I intend to implement a simple command-line game that can take input from two human players and enforce the above rules (with perhaps the exception of #9) by February 11th.

I've been looking into several approaches, including A* search, machine learning with recognizers, and boosting at the advice of my advisers and Joe Kider. I spoke with Dr. Maxim Likhachev on February 3rd and Dr. Norm Badler on February 4th for advice. We went over the decisions I made regarding which rules to implement, and they assisted me in understanding some of the details and potential application to Go of the AI algorithms I've been investigating. We also discussed some implementation details and concrete expectations for the alpha version.

The AI will begin with a Monte-Carlo based approach, calculating the resulting win rate of a given move in a decision tree by playing out random games. Additionally, more intelligent move choices may be explored by using situation recognizers (which can be as simple as recognizing when a stone may be captured in one move). Basing the framework more on Monte-Carlo techniques seems to be preferable to a purely machine-learning based framework, as I am not an experienced Go player myself. I would have difficulty determining what advanced recognizers should be looking for, and what actions should be taken.

Additionally, there are a huge number of states that a Go board can be in. Each of the 19×19 intersections can be occupied by a white stone, a black stone, or neither, creating a huge number of possibilities (minus those states created by illegal moves, or other violations of the game rules). Having a complete stored "library" of full board states, and their precalculated best moves to choose from, might have some advantages; but constructing this library doesn't seem feasible. Therefore, each move taken by the computer will depend on its empirical findings from randomly played out games during that session, guided by some (at least initially) very simple recognizers that can later be expanded on.

Some of the implementation details we discussed included having an extra row or column on each edge of the board, to simplify edge cases. For example, a stone on the corner only has two adjacent spaces that must be enemy-occupied for it to be captured. So when searching for which opponent stones that can be captured, the extra border locations can count as occupied by one's own stones. Then capture on the edges of the actual board will not need to be handled specially. Additionally, the option to have smaller boards should be included, as this will faciliate seeing the evaluations the AI makes throughout a (shorter) game. Being able to load a board state from an ascii representation in a file would also be useful.

I still plan to have a two-player version of Go that inforces rules and has a bare-bones interface by the 11th, and expect to begin coding the AI the following weekend.

An outline of the AI algorithm thus far:

1. When it is the computer's turn, take the state of the board, and play out the move tree to the end of the game* many times using random placement**.
2. Use A* Search*** to pick a move in the tree with the empirically best win rate from step 1. Win rate = (number of games won in that branch) / (number of games total in that branch).
3. Keep the subtree for that move, update it based on the opponent's move****, and then elaborate on the updated sub-tree for the computer's next move.

Notes:

*   For now, the end of the game will be defined as when no more legal moves exist.

**   This will eventually use pattern recognizers, to make the chosen simulated moves of each player more intelligent.

*** This could be an intelligently pruned search, where it stops investigating a branch once it appears to be loosing terribly. Given the nature of Go, in which large areas may be captured in a single move, however, it may be difficult to determine what makes a board state unfavorable enough to stop searching.

**** This step, keeping the tree, makes the search incremental. One area I am not certain about, however, is what to do if the opponent takes a move that was not investigated during the previous turn, and is not included in the tree. I do not see any way to update the existing tree yet in that case.

Week 4 (Feb 09, 2009 – Feb 15, 2009) –

Dealt with design issues such as the one described in the following post:

I finished my other work sooner than expected today, so I began setting up some of the basic functionality of the simple two-human-player Go program. As discussed, it is going to take command-line input from players, enforce the rules of Go, and print a simple ascii representation of the board. One less straightforward aspect is in determining suicide moves / capture. This is not just a local check; for example, a player may attempt to place a white stone down on an intersection that is surrounded by black stones above and to the right, the edge on the left, and a white stone below:

B

|

W? ——— B

|

W

Locally it seems like this would be a legal move. However, the white stone already on the board (W) may be part of a group of white stones whose only remaining liberty (open adjacent location) is the spot the white-stone player is now attempting to fill (W?). In this case the move would actually be illegal, since it would be a ’suicide move’ for white.

Due to issues such as these, I’m wondering if it would be more efficient to maintain some state representing the current groups of stones on the board, in addition to the int array I currently have.

I now have a simple program that enforces basic rules of Go, as outlined in earlier posts. I opted not to keep track of ‘groups’ of 4-connected stones because it seemed to add more code complexity than convenience or speed. Instead, my algorithm to tell if a stone is captured checks the stone’s four adjacent neighbors. Since the internal representation of the board actually has a one-space-deep border all around, this can be done for every playable spot on the board (i.e. edges and corners do not need to be treated specially here; they are just filled with a defined BORDR value instead of a WHITE, BLACK, or EMPTY value). If all four neighbors are “captured” then the stone is also captured. Conceptually, a neighbor is “captured” if it meets one of the following conditions:

1. It contains a BORDR value

2. It contains an opponent color value

3. It contains a same-color value, but this same-color stone has no liberties and is not part of any group with any liberties

An outline of the recursive portion of the algorithm:

```
bool isStoneCapturedRec(row, column, friendlyColor)
{

//Evaluation of local (r,c) data to determine true or false if possible:
If the value at (r,c) is the opponent color, or the value at (r,c) is the border value, count this point
as captured (i.e. return true)
If the value at (r,c) is the empty value, count this point as not captured (i.e. return false)

//Otherwise, we know (r,c) contains the friendly color value:
Set a boolean isCaptured to true;
&& this isCaptured boolean with each of the values obtained from performing
isStoneCapturedRec on its four neighbors (i.e. if one of its neighbors is an empty spot, or a
friendly colored stone that is part of a group with a liberty remaining, return true.)

//Updates that ensure termination:
If isCaptured ends up to be true, set (r,c) to the border value (so that if this spot is checked again,
it returns false immediately).
Else set (r,c) to the free value (so that if this spot is checked again, it returns true immediately).

Return isCaptured.

}
```

One drawback of this is that it requires a copy of the board to work with, since it changes values to ensure termination of the recursion. I intentionally implemented this on my own first to ensure it would be my own work, but perhaps now it would be beneficial to search for how other programs handle this aspect as well. It seems like this algorithm will need to be very efficient when being used by the AI.

Week 5 (Feb 16, 2009 – Feb 22, 2009) –

I recently worked on a simple example of machine learning with my C# class partner, Rebecca Fletcher. I thought I would include my thoughts on this approach to AI after applying it to a simple case, as it is related to one of the suggested approaches I considered for my senior design project.

As instructed by our teacher, PhD student Jeff Vaughan, we began implementing a simple perceptron to recognize handwritten images as representing a number "3″ or a number "5″. In this particular interpretation, the black and white images are translated to a feature vector, with each pixel mapping directly to one of the ordered elements in the vector. In this case, white pixels are represented as 1 and black pixels are represented as -1 in the feature vector. There is also a single mutable weight vector, which determines how influential each element (pixel position) in the the feature vector is. The function H assigns a guess to a given example by taking the dot product of a given feature vector, and mapping this value to a label. Specifically, negative values are mapped to the label 0, and positive values are mapped to the label 1. Then the weight vector may be updated if the guess is incorrect. If the guess was too low (i.e. the dot product was negative and mapped to 0, when it should have been positive and mapped to 1), the weight vector is increased by the particular feature vector, scaled by a "learning rate" *alpha*. Similarly, if the guess is too high, the weight vector is decreased by the feature vector scaled by *alpha*. Therefore, the provided update function can be summarized as follows:

new_weight_vector = current_weight_vector + (correct_label - predicted_label) * *alpha* * feature_vector

As mentioned by Dr. Badler earlier and seen elsewhere, more complicated systems can have a more specifically designed length-n feature vector. This feature vector may then be weighted with a weight vector if desired, but with keeping the elements distinct. The result may then be mapped to an n-dimensional space. This n-dimensional space is then divided into judgements on the situation. The handwriting perceptron here is a simpler version of the same idea, mapping the weighted data to 1-dimensional space, with the judgment division at zero.

After working on the handwriting perceptron, it is even more clear that using solely this form of AI would not be ideal for my Go program. As an inexperienced Go player, I would not know how to design the feature vector, weighting system, or n-dimensional space so as to encourage useful learning. Additionally, even if I were much more familliar with Go, it seems that the design could be strongly biased by my own playing style and the strategies I would consider. However, machine learning in a style somewhat like this could perhaps be used to enhance the moves chosen by the Go program in playing out randomized games.

I have been working on the move tree and win rate calculations for the AI. I'm using a very straightforward implementation to figure out exactly how the recursion should be structured. This version uses too much memory to thoroughly investigate a full 19×19 board, but it can be applied to very small boards for these testing purposes. I expect to meet my goal of working AI on a 19×19 board by the alpha review.

After the alpha review, I am tentatively planning to get most of the XNA-related functionality working over break. Then I will be able to spend the remainder of the course fine-tuning visuals and investigating how the AI may be improved.

Week 6 (Feb 23, 2009 – Mar 01, 2009) -

I explored alternative ways of representing and manipulating data, such as in the description below:

Tonight I will be working on improving how the decision tree is constructed by the AI. After test the recursion by using small board sizes, and a debugging-friendly algorithm that simply makes copies of the board at various points, I'm considering methods of using a single board. I figured I might as well type out some notes as I think.

In addition to storing the most recent move taken to reach it, a move node in the decision tree could also store the two moves previous to it (or it's parent, but storing these two moves is all that is needed and is more direct). If the recursive function that expands a node into possible subsequent moves also returns the update to the number of captured stones for the next move, moves could be "undone" after the recursive call. This way the same board would be passed around. This makes the algorithm much less parallel, but would resolve issues that arise from having too many copies of the board.

For example, let (r, c) be the move (taken by white) stored at node n, i.e. the move taken to reach the state of the board implicitly stored in the path to this node. We return to this node after exploring a move (i,j) where a white stone surrounded by three black stones was finally captured by black's placement at (i,j). This "undoing" step is done at each level of the recursion, so we know that the only difference between the current state of the board (after returning from investigating (i, j)) and what the state of the board initially was at node n (immediately after move (r, c)) is the placement of stone (i, j) and any captures that it caused that turn.

It is easy to take away stone (i,j). We could then see that there has been 1 captured stone, and notice the spot adjacent to (i,j) that is both empty and surrounded by black stones. A recursive algorithm similar to the isCaptured check could find surrounded groups of empty spaces adjacent to (i,j). However, this does not resolve the case where placing (i,j) capures a group of x stones, but also inadvertently creates a surrounded group of x empty spaces. So the recursive algorithm could return the exact places of the stones captured with move (i,j). This way we are undoing all placements of stones (namely (i,j)) and all (if any) removals of stones. We can also use the size of the array of captured stones to decrement the record of captures for each player. In this way we have restored the state to what it was immediately after move (r,c), and can now investigate another chosen branch.

The remaining question is, is all this really worth the potential speed increase over just playing games out to the end, then starting again from the beginning (i.e. current real state of the board)? It does save the time of calculating captures and legal moves more than necessary, though, so I'll try it out for the sake of experimentation and see what happens.

Week 7 (Mar 02, 2009 – Mar 08, 2009) -

I read the material suggested and provided by Joe – slides on decision trees by Andrew Moore, and slides on Boosting by Verma & Thrun.

Week 8 (Mar 09, 2009 – Mar 15, 2009) -

I switched over to a more simplistic way of handling the data, so as not to involve the "undoing" of moves – which was workable but unnecessarily complicated; it seemed that it would be ridiculous to debug once more complex move selections / pruning are in place.

Week 9 (Mar 16, 2009 – Mar 22, 2009) –

My schedule has been a little ridiculous lately, but I have made progress on my project, and will be writing posts describing it throughout the week.

For my alpha review, I used the two-player Go program and my experiments thus far to code up a prototype for one simple approach to the AI. As a simple outline, I sketched out a bare-bones Monte-Carlo-esque approach. To determine it's next move, the program simulates a fixed number of games for each possible legal move, building up a tree of move results. Legal moves are randomly chosen to simulate both its moves and the opponent's moves, and the tree is updated as soon as a simulated game is completed – based on whether it is a win or a loss. It's fairly fast to do many simulations for each possible legal next move the computer can take. So the first level of the tree is exhaustive, but subsequent levels of simulated player- and computer-moves are not. Once the simulation process is completed for a given computer turn, the program chooses the legal move with the best empirical win rate. Also, the tree is persistent, so long as the opponent's next move is one that was explored by the random simulations. However, if the opponent happens to take a move that was not considered in the existing random simulation data, the tree needs to be reconstructed from scratch. Overall, this approach seems slightly more effective toward the end of the game, but of course is not a very intelligent solution – it is intended to just be an outline of how the program will work.

My alpha review document is in .pdf format here.

Week 10 (Mar 23, 2009 – Mar 29, 2009) –

I met with Joe Kider, Dr. Badler, and Dr. Likhachev for my alpha review to present my progress and ask questions regarding further work on the AI. Since this project is for my personal education, rather than an attempt to outperform commercial programs, we discussed options beyond the currently popular Monte-Carlo approach as well. One of these is having an n-dimensional vector space based on characteristics of the board state, as mentioned in an earlier post. K-Mean clustering could then be used to evaluate moves. Also, limiting the search to be only a fixed number of n moves deep could allow the move tree to be more dense (i.e. more possible moves two moves after this one, three moves after this one, etc. are explored, even though they are not explored to some 'end' of the game.) Such a process could make it more likely for the tree to include the move that the opponent happens to make next, thus making it more likely that the data may be reused in subsequent turns (keeping the algorithm incremental more often).

Additionally, another option could be to define a distance function – DIST(a,b) – providing some value for how "different" any one given board arrangement (a) is from any other given board arrangement (b). Detailed statistical data could be generated for a subset of possible board configurations, stating which moves are best based on many simulations. This data could be stored in a library of board configurations. Then, during gameplay, the computer could use DIST(a,b) to compare the current board to boards stored in its library, and find the closest one. The pre-computed statistical data for this closest match could then be used to choose a next move.

For either approach, it could be useful to make move decisions in different ways based on the current state of the game. For example, this strategy design pattern may choose to use an exhaustive tree search, exploring all possible moves, when it is determined that it is close to the end of the game. 'Close to the end of the game' could be defined with a simple criteria such as there being n or less legal moves currently – although it would have to be careful that the exhaustive tree search would not be able to become too big due to captures opening large areas of the board (creating many more legal moves), etc. The beginning of the game could also be improved, such as by using one of the traditional opening move sequences for Go.

Time permitting, it could be interesting to code up a number of approaches, and allow the player to select one to play against – or even select two AI approaches to play against each other.

Although this would of course not definitively prove which approach is best, it could help illuminate the strengths and weaknesses of my particular implementations of each.

Week 11 (Mar 30, 2009 – Apr 5, 2009) –

I found a page of Sensei's Library ("A Novice Tries To Write A Go Program") where the user Gorobei describes attempts, as a novice, at writing computer Go. This user's comments, as well as the responses from other users, seem like they more helpful than some of the descriptions of more competitive cutting-edge programs, which can be vauge in some areas of their implementation. Also, it is mentioned that a very simple algorithm, which effectively fills out sequential positions on the board, simply placing stones in rows, can defeat some slightly more complicated algorithms. It seems like this algorithm could be an interesting implementation to compare my Go AI against.

A related page from Sensei's Library ("Yet Another Novice Tries To Wrtie A Go Program") is also of interest. This outlines one particular approach, and some implemented improvements to it, in a fairly detailed manner. I intend to learn more about the strategies mentioned on this page, and possibly incorporate some into an AI implementation of my own.

The second page references the book EZ-GO: Oriental Strategy in a Nutshell, by Bruce and Sue Wilcox, as a possible source of further inspiration (reviewed here). Although it could be helpful in some areas, it seems that reading through this entire book may not be necessary, and may not be the best use of time at this point. It could, however, provide some direction for future improvement for personal interest after this project is completed for senior design.

I met with Dr. Likhachev on Tuesday to discuss my progress thus far, and clarify some aspects of the AI approach that makes use of a distance function. I thought I'd explain the approach in more detail here.

The distance function is based on selected features. The selection and evaluation of these features is somewhat arbitrary, and will probably require some trial and error. Some potential evaluations of features could include counting the current captured area and captured pieces (representative of what the player's score would be if the game ended then), as well as a count of how many uncapturable "eye" structures the player has on the board. The precomputed library contains sample boards, with a score S_n for each of the n boards. This score represents the likelihood of winning given the board configuration, and is based on statistical data (based on simulated games) computed offline. When finding a score for a given board b not in the library, the program searches for the k boards that are closest to b (i.e. have minimum distance from b based on the distance function). To avoid a linear search to find these k nearest neighbors, boards can be put in a hash table. It would make sense for k to be less than or equal to than the number of boards in each bin, so finding multiple bins for a board is never needed.

Once the k nearest boards are found, a score for board b can be interpolated from them. For example, let k=2, and with nearest neighbor boards 1 and 2, with scores S_1 and S_2 respectively, withdistances d_1 and d_2 from board b respectively, and let distances be represented by a number between 0.0 and 1.0. Then I think it would be reasonable for the score for b could be:

S_b = (1-d_1)/((1-d_1)+(1-d_2)) * S_1 + (1 -d_2)/((1-d_1)+(1-d_2)) * S_2

During gameplay, when the computer is to decide its next move, it may create the next level of the tree – i.e. create a set of hypothetical boards that are the result of executing a single move on the current board configuration. For the hypothetical boards that are not already fully evaluated from previous turns, the program may compute the score of each using the library lookup and weighting processes described above. Then the program may take the move that results in the board with the higher score. To make this more complex and hopefully more successful, the program could explore the tree to depth n, assign scores, and pick a next move based on the branch with the best calculated score. Additionally, the "goodness" (score) of the board for the opponent could also be evaluated in a similar fashion, and the program could aim for minimizing the "goodness" of the board for the opponent as well.

Week 12 (Apr 6, 2009 – Apr 12, 2009) –

Last week I began learning XNA and building a GUI/visualizer for my Go AI project. My goal was to learn enough so that over the weekend I could create a visual component that would 1. visualize the current state of the board, 2. take user input (placing a stone) in an intuitive way, and 3. enforce the rules of Go (whose turn it it, what constitutes a legal move, perform captures when approppriate, etc.).

I purchased the O'Reilly book Learning XNA 3.0 by Aaron Reed in eBook format. This book provides a very clear walkthrough of XNA; I feel like this could have been the type of book that some experienced programmers dislike for being too hand-holding (it also explains well-known game concepts like framerate), but it's actually well laid-out in my opinion, and it was very easy to get a quick grasp of how XNA is set up and build an intuition of how to write games in it from this book. I went through most of the 2D tutorials in detail fairly quickly. Overall, XNA is organized in a very obvious way, e.g. classes inheriting from Game can override an Update function and a Draw function, which do exactly what you'd expect them to do. In terms of future usefulness of learning XNA, I can definitely see myself prototyping games in XNA if not making finished products for future school assignments.

The only thing so far that I disliked about XNA is that one of the first methods I saw of drawing primitive lines seemed somewhat unintuitive. Perhaps I am just too used to OpenGL. However, I found a class called PrimitiveBatch.cs (downloaded here) that allows primitives to drawn in a much more OpenGL-like way (i.e. with a call to a Begin() function, adding verticies one at a time, and then a call to an End() function). The XNA Community Forums have been a quick way to find solutions to common issues such as this.

I was soon able to build a 2D representation of my board, purchasing a stock photo of bamboo wood from iStockphoto and scaling / drawing lines over it according to the current dimensions of the board (which must be in the form nxn, with $0 < n < 20$). Stones are represented by simple .png textures I made in GIMP. I will probably only allow the user to pick between some set board sizes, such as 9×9, 14×14, and 19×19, ultimately, but the very small board sizes can be useful for testing. For now the display is only 2D, but the program's Model-View-Controller design allows for the view to be swapped out fairly easily if I have time to experiment with a 3D representation and a camera for fun.

For user input, I originally thought it would be nice to leave the Windows mouse cursor hidden (this is the default) and draw a custom icon over the cursor position that indicated whose turn it was. I stole the hand icon from GIMP as a temporary stand-in and added a black or white stone (depending on whose turn it is) to the image. However, upon showing both the Windows cursor and the custom cursor, it became obvious that the custom cursor image lags a bit. Especially given that stronger AI methods may cause a decent amount of lag between calls to Draw, this is bad behavior even though it is still playable. For now I show both the custom cursor, to indicate whose turn it is, and also the Windows cursor for accurate and responsive clicking.

I completely restructured my code to allow for different AI classes / a human player to be swapped out easily, and also to accommodate using XNA. I met my weekend goal by Saturday evening/Sunday morning, and had a working two-human-player Go program in XNA.

As mentioned previously, I completely restructured my code to better accomdate XNA and swapping out AI algorithms. Rather than the fast-to-implement but hacky single class that initially handled manipulating the board and displaying text to the console, it is now based on the Model-View-Controller design pattern, with the following classes:

GoBoard (Model):

- Holds the internal representation of the board configuration and performs various operations on it.
- Responsible for information such as whose turn it is, how many stones each side has captured, etc.
- Has no XNA dependencies.

GoView2D (View):

- 2D board visualization & user input using XNA; draws the current state of the board, which is passed to it by the controller.
- Only needs to know whose turn it is for drawing the cursor texture.
- Only handles input when it is a human's turn according to the GoGame controller.
- Inherits from XNA's DrawableGameComponent, and is automatically updated when the controller is updated, and redrawn when the controller is redrawn.

GoGame (Controller):

- Coordinates view, model, and AI classes, and who gets to make a move on the board.
- Holds an enum for each player type, including HUMAN and various AIs.
- Has an array of an AI player class of each type (indexed by player type enum) for black, and also such an array for white.

The AI arrays stored by the controller are of type AiGo[], as all AI classes inherit from AiGo. Each class created to populate these arrays is told which color player it is (black or white). The controller also stores an enum value _playerBlack and an enum value _playerWhite. When it is an AI player's turn, e.g. player white, it uses that player's array and looks up the value at index [_playerWhite], and calls the takeTurn (board) function that all AI classes override. It would be easy to set up an AI, then, that uses one AI strategy for the beginning of the game, another AI strategy mid-game, and a third AI for endgame play.

Human input is handled as a special case. Since I wanted to experiment with a polling-based design, using a default XNA setup, I opted not to implement events for now. When it is the human's turn, the program does not allow any gameplay updates to occur (i.e. does not let the AI do anything), until the human has taken its turn. All four permutations of human or AI players as black or white are easily switched between by changing _playerBlack and _playerWhite enum values. This may not be the best design, but it was very fast to set up and works smoothly for both debugging the code and play.

For now I am considering the game over when the player whose turn it is has no more legal moves. This is because, for now, passing one's turn and not putting down a piece is not an option. Therefore, it would be possible for games to never terminate if the game-over requirement were that neither player has any legal moves. (e.g. what happens if white has more available moves, but black does not and it is black's turn? The game never terminates.)

It seems that the Ko rule is actually that "A play is illegal if it would have the effect (after all steps of the play have been completed) of creating a position that has occurred previously in the game" (from the Wikipedia entry for Rules of Go). Although in practice all the examples I came across involved creating the same board configuration as two turns ago, and this simpler rule is the one I originally learned, I suppose it might be possible to violate this stricter rule with moves taken many turns apart. I need to think about this further, taking all the rules into account, but even if it is possible to silently violate this rule, I might not enforce it so strictly – and just ensure that all games terminate despite not enforcing this rule so strictly. It seems impractical to keep track of all previous board configurations instead of just the most recent two, especially for when AIs are simulating many full Go games.

I had my first AI comparison test this past Sunday, after restructuring my code and integrating it fully with XNA. Its purpose was to ensure the controller class was properly set up to accommodate AI vs. AI games, and dealt with only two AIs: Random AI and Crawler AI. Random AI collects a list of legal moves, and randomly selects one with equal probability. It is of course trivial to defeat in a Human vs. Random AI game; a standart 19×19 board is huge, and it is easy for the human to set up pieces without much hindrance from the Random AI. Random AI vs Random AI games give the AI that goes first (black) an advantage, it seems, but it is possible for the AI that goes second to win. The Crawler AI simply fills legal moves in left to right order, row by row. Crawler AI vs. Crawler AI is entirely deterministic; the pieces alternate in a checker-board-like pattern, so the AI that goes second (white) captures all of the opponent pieces and leaves a board full of its pieces and gaps. The gaps are spaces where opponent (black) pieces were captured and can no longer be placed on (due to the no-suicide rule).

The Crawler AI vs Random AI match was very straightforward and simple, but demonstrates that the system for pitting AIs against each other and displaying the match is working. Regardless of which side started first (i.e. for both Crawler AI (black) vs. Random AI (white) and for Random AI (black) vs. Crawler AI (white)), Crawler AI seems to win more often. However, sometimes the Random AI wins (unscientifically estimated as 1 in 4 to 1 in 5 times). As Crawler AI crawls along, it surrounds and captures any of Random's pieces in its path. However, this leaves gaps in Crawler's captured side of the board. If Random can by chance make a wall against the advancing edge of Crawler's pieces, and also fill in gaps in Crawler's advancing area, the entire area can be captured. If this happens, Crawler usually loses by a large margin.

On Monday I added in the naive Monte-Carlo AI (NaiveMonteCarlo) from my alpha demo, to compare it to the existing AIs and ensure that adding in more complex AIs (that require more information about the state of the game, retention of information between turns, etc.) can also be done easily with this design.

The only non-obvious aspect of integrating this AI was how to deal with updates to the root of the move tree that persists between turns. Originally, in the alpha demo, updates were made to the existing tree based on which move was taken by either player. If the taken move was already explored, the tree was updated to have that node as the new root. This update is still trivial to implement when the computer takes a move, as it simply compares its computed move to the children of the current tree root (which is stored in the object). However, additional steps would need to be made for an update to be called in the controller, passing the move that the opponent took, to the NaiveMonteCarlo AI(s) currently playing. The GoAI base class could be updated to have an update(Move m) function, but for now it seemed better to handle this entirely within NaiveMonteCarlo, since it is just one of the simple test AIs. To handle this issue, NaiveMonteCarlo's takeTurn override function currently infers the opponent's move by comparing the current configuration of the board to the previous configuration of the board (stored at the end of its most recent turn).

NaiveMonteCarlo is fairly slow at this point, when combined with the graphics updates in XNA. However, it is still watchable, and more so on a small board (e.g. 10×10). In terms of comparison with Random and Crawler, NaiveMonteCarlo has so far outperformed Random, and usually outperforms Crawler. NaiveMonteCarlo seems to show the beginnings of ‘intelligent’ move choices toward the end, when there are fewer choices left, and it is more likely to find differences in win rate for given moves. Crawler has probably generated large contiguous blocks of stones by this point, which NaiveMonteCarlo considers – instead of assuming the opponent is random and simulating the moves in that manner. Therefore, it is easier for it to determine what moves are likely to result in capturing large blocks, and thus is better at acquiring points.

It makes sense that NaiveMonteCarlo does well against Random, since the moves it simulates for its opponent are random. It would be interesting to see how NaiveMonteCarlo does against Crawler if it simulates the opponent’s moves (and possibly even its own) using an algorithm similar to Crawler’s. However, this would just be for the sake of experiment if it is attempted, as the AI usually shouldn’t have the advantage of knowing the opponent’s strategy (especially when it is deterministic, like Crawler’s).

Week 13 (Apr 13, 2009 – Apr 19, 2009) –
Week 14 (Apr 20, 2009 – Apr 26, 2009) –

I found a tutorial for serializing in C# (to use for Distance AI) here, and have been cleaning up my code a bit. I have also recently made a 3D visualizer for a game of Snake in XNA. XNA surprisingly does not seem to have any way of generating primitive 3D objects, i.e. no functions analogous to glutSolidSphere(), etc. I borrowed some simple models – namely a low-poly sphere and a cube – from the XNA Creators Club Online Shader Series 4 tutorial. Drawing is fairly straghtforward, and simply involves iterating through effects and bones as described in the tutorial on 3D models from the O’Reilly book I purchased (which provides source code as well).

I added a camera class and rough version of a 3D visualizer class to my Computer Go program based on what I learned from the Snake game. Although the benefit of allowing the users to play in this view, and adding details like fancy camera movement might be minimal, if I have time it might be nice to have a pretty 3D visualizer to display AI battles for my presentation (time permitting).

I have also begun making my presentation slides, and will be adding funtionality to allow the user to select AIs during runtime. There could be UI elements such as drop-down menus to select AIs for each player (which could be swapped out mid-game), or even something as simple as a pop-up menu describing what key to press to select a given AI for a given player.

Week 15 (Apr 27, 2009 – May 3, 2009) –

Analyzed Greedy AI and Greedy Randomized AI; see Section 5, "Results."

I've been working on deciding how to create a better library of Go games (board configurations and results), rather than just generating them using my AIs and calculating win rates emperically as I've been trying thus far. Sensei's Library has a page describing and linking to various game databases, and it seems that most of them are in SGF (Smart Game Format). red-bean has some helpful explanations of this format, with format rules specific to Go and more detailed explanations of the "Add Stone" and "Move" notations. I've also found a Python script to translate .SGF to CSV, and a way to translate .SGF to a printable format. Perhaps even more useful, after looking around a bit, I found some SGF parsers – including one in C#. This parser, written and provided by computer-go.org user Phil G, looks promising. Getting it to build just took a little bit of tweaking; it needs NUnit to build (Project>Add Reference>Browse>nunit.framework.dll), and I needed to change it to embed a manifest using default settings (it was trying to use an app.manifest file that didn't seem to exist).

For personal interest, I've been researching some machine learning techniques that have been applied to Go.

One paper I read was Greenberg's "Breeding Software to Play the Game of Go". This example of Genetic programing I read about seems very interesting, but not necessarily competitive, as the program needed to learn how to play Go properly first. Early trials apparently resulted in the programs that worked choosing to resign early on.

Another paper, "Evolving Neural Networks to Play Go" by Richards, Moriarty, and Miikkulainlen, investigates neural networks applied to Go. This implementation uses SANE (Symbiotic Adaptive Neuro-Evolution), which has successfully been used in areas such as robot control and the simpler Go-like Game Othello/Reversi. To avoid the issue of credit/blame assignement to individual neurons (i.e. which moves are responsible for winning/losing the game), they do not use any straightforward back propagation. Instead, they maintain a population of neurons, and also a population of "blueprints" that indicate how neurons should be used in a larger strategy. Neurons, then, can be assigned fitness values based on the performance of the blueprints they are used in. This process, which also involves subsequent breeding of the "elite" members within each population, has had success against Newman's pattern-based Wally program (which apparently can be a good trainer, since it can play but isn't overpowering).

I emailed Phil Garcia (website: http://www.thinkedge.com/blogengine/page/GoTraxx.aspx), the author of GoTraxx for help with using his SGF parser. He was very helpful with the process of figuring out how to use his code to make my library, so I just wanted to mention him here in addition to in the credits of my paper.

Week 16 (May 4, 2009 – May 8, 2009) –

Cleaning up AIs, most of which were finished up in the previous week prior to the May 1st presentation, as well as preparing final non-code deliverables.
