Artificial Intelligence for Go

Kristen Ying

Advisors:

Dr. Maxim Likhachev & Dr. Norm Badler

1. Introduction
2. Algorithms
3. Implementation
4. Results

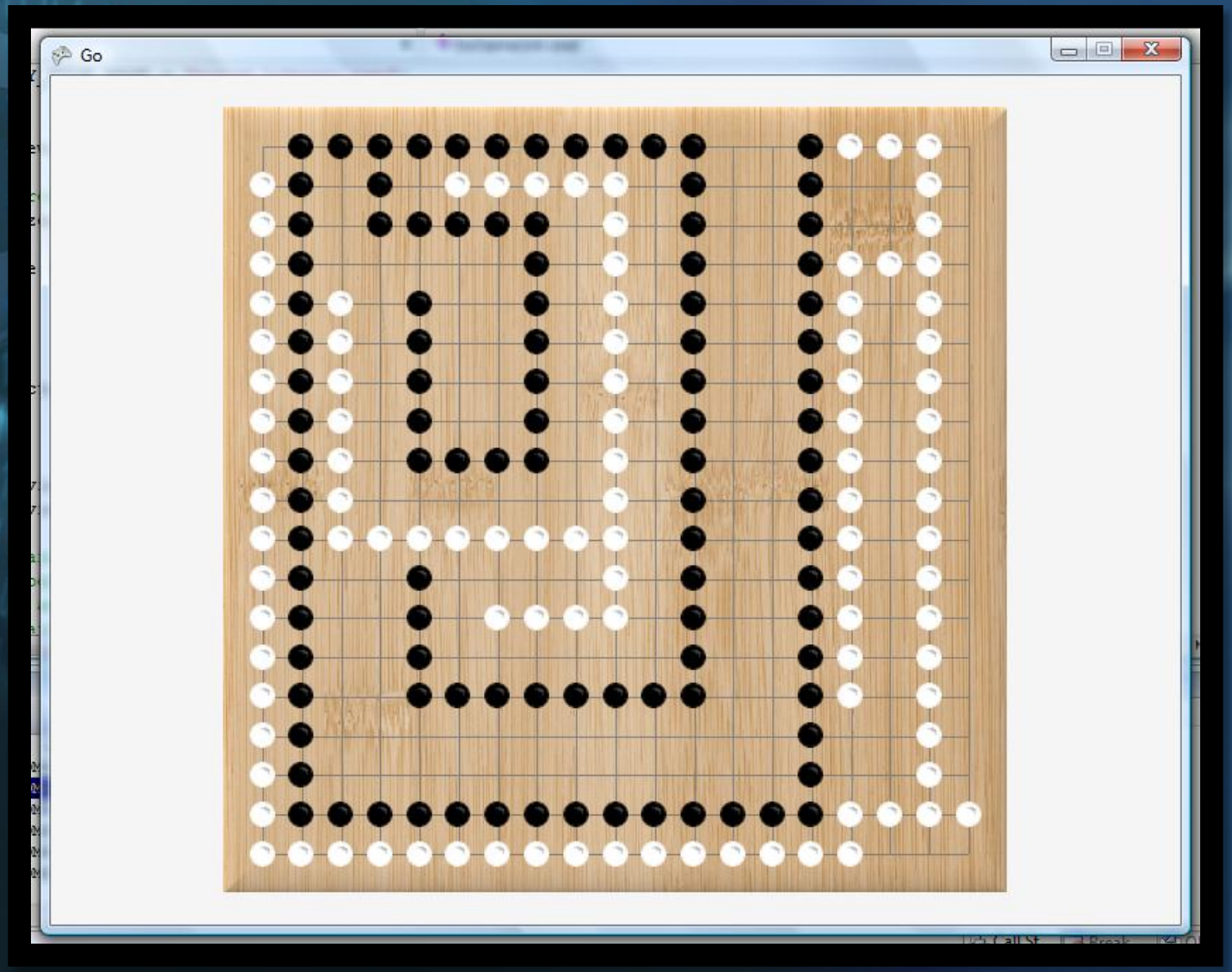
1. Introduction
2. Algorithms
3. Implementation
4. Results

What is Go?

- 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].

- The objective is to capture opponent stones and surround area on the board.

The Board



Rules of Go

1. The board is a square 2D grid; professional games are typically played on a board with 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.

Rules of Go (continued)

6. 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.
7. 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.
8. 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”).

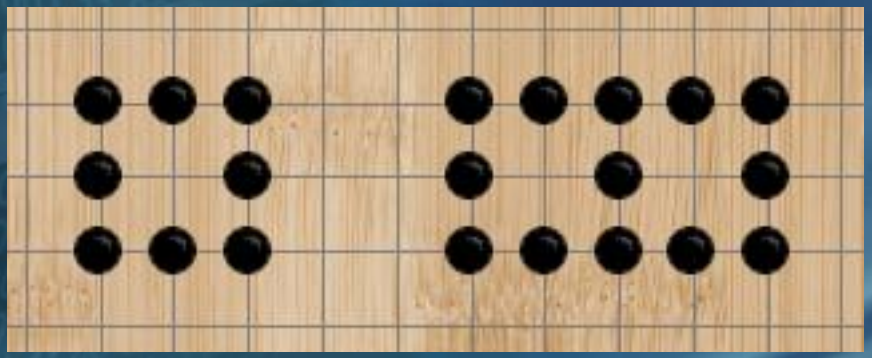
Rules of Go (continued)

9. 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.
10. Various alterations can be made, such as playing on a smaller board, or giving "handicaps" to a stronger player. To give a handicap, the board begins with a certain number of the weaker player's stones in predetermined strategic points.

Simple Rules, Complex Play

- Despite the very straightforward rules, many strategies and important patterns emerge.
- Example: “Eye” patterns
  - **If a group has a single internal open space or “eye”, it can be captured; the opponent may surround the group and then fill the space.**
  - **However, if a group has two (or more) “eyes”, the opponent can not play in either due to the no-suicide rule. Thus that group cannot be captured.**

Eyes - example



- The formation on the left could be captured if white takes all its external liberties, and then places a stone in the center.
- The formation on the right (two eyes) cannot be captured.

Why do people try to make Go AI?

- Because it's challenging, and requires coders to seek new AI board game techniques.
- Go is fun!
- Also it's a great way to begin investigating AI for games, as many approaches can be applied to it.
- For neuroscience: possibly, through machine learning techniques, some insight can be gained into how humans learn and think.

Go is P-Space Hard

- Given a position on an n by n Go board, determining the winner is polynomial-space hard. This can be proved by reducing TQBF (a P-Space complete set) to the "generalized geography" game, to a planar version, and then to Go [Lichtenstein & Sipser].
- 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.
- Chess is only 8x8, and a powerful minimax is feasible.

Some Numbers…

- The professional sized board is 19 x19.
- Barring illegal moves, this 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].

Major Challenges

1. The sheer size of the move tree.

2. Difficulty of determining what makes a “good” board. It is difficult to create an evaluation function that determines how advantageous a given board state is for a given player.

i.e. it’s difficult to prune the move tree in conventional ways

Recent Progress

- 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].
- This program, MoGo, beat a pro player with only a 7-stone handicap in 2009.
- It seems feasible for computers to outperform professional Go players within the next decade [Keim], but they certainly aren't there yet.

Goals of this project

- Investigate Artificial Intelligence, as a potential area of interest as a Master's student.
- Learn XNA and C# (and begin learning Go).
- Code a two-human player game using XNA.
- Write AI for Go.

1. Introduction and History
2. Algorithms
3. Implementation
4. Results

Overview

- There are a lot of ideas out there! Resources like Sensei's library have pages and pages of novice-written algorithms, ideas for further research, etc.

- There are many combinations of approaches. For example, MoGo uses UCT, Monte-Carlo, and pattern-based techniques.

Pattern Matching

- Albert Zobrist created one of the first (if not the first) implementations of Go AI in 1970.
  - **It used one influence function, which considered adjacent pieces, to assign values to the possible next moves**
  - **It also used another influence function that tried to match patterns in a library to the board configuration**
- Simple pattern-matchers can be good initial training programs for machine learning techniques (e.g. Bill Newman's Wally)

Life and Death

- A group of stones is considered “live” if it will remain on the board.
  - **A group with two eyes is “live” in the strongest sense; they can never be captured.**
- Benson’s Algorithm uses this concept.
- These algorithms can be very fast.

Center vs. Edge Heuristics

- Consider strategies such as center of the board vs. edge of the board placement.

- Also may use techniques such as calculating the physics center of gravity of a group, and then trying to place stones around the edge to enclose this center of gravity [House].

Influence / Moyo

- “influence” is a long-term predictive effect; e.g. a nearly captured stone has close to zero influence, whereas a stone in the center of the board with no neighbors may have high influence
- An area where a player has a lot of influence, i.e. is likely to become their territory, is called a framework, or *moyo*.
- Alistair Turnbull: if a given player tends to claim the majority of a certain region of the board when stones are placed randomly, this indicates that the player has influence in that region.

Monte-Carlo

- Use of random playouts (simulations of the game to the end) to generate an expected win rate for each move.
  - **Win rate can be thought of as an estimated likelihood of it leading to a win if that move is played.**

- CrazyStone by Rémi Coulom relies heavily on Monte-Carlo with little hardcoded knowledge of Go, and has performed well

Reducing the Scope of Move Search

- The challenge is to make the move search more manageable , while retaining prediction strength
- In addition to the fairly well-known alpha-beta search, there are algorithms that make use of a technique known as Zobrist Hashing for more efficient search.
  - **This is a way to implement hash tables indexed by board position, known as transposition tables [Wikipedia].**
- Some have considered more unusual techniques like 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.**

Upper Confidence Bounds for Trees

- Multi-Armed Bandit Problem
  - **Each legal move is an arm of a multi-armed bandit**
  - **find the most promising looking branches, and expend more resources exploring those**
- Keep selecting children according to upper confidence bounds, until a new 'leaf' is found (tree part); perform simulations to evaluate this new leaf (random part)
  - **Update ancestors based on winrate**
- MoGo: Use of patterns to have more meaningful Monte-Carlo sequences
  - **Not using patterns to prune the tree**
  - **Parallelization → 70k simulations per move on a 13x13 board**
  - **Also many hand-coded aspects to include various Go skills**

Genetic Programming

- Mutating code and using natural selection to evolve a program.
- John Koza has popularized a method of ensuring that mutated programs are still syntactically correct.
- Jeffrey Greenberg applied genetic programming to Go
  - **It also had to learn the basic rules of Go**
  - **Initial attempts did not result in strong Go players (but the process is very interesting)**

Genetic Algorithms

- Per Rutquist's paper, "Evolving an Evaluation Function to Play Go," describes a way of applying Genetic Algorithms to Go.
  - **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.)**
  - **A genetic algorithm is used to evolve the pattern set.**
- Learned to do well with training/test sets, but doesn't play well overall (against GnuGo).
  - **Scores of -80 with no training, then -20 with training.**

Neural Networks

- Use of Neurons (units of strategy) with connections to other Neurons
- Credit assignment problem in backpropagation:
  - **which moves get credited/blamed for a win/loss?**
- Richards et. al. designed a system that evolves two populations: one of Neurons and one of Blueprints
  - **Blueprints use neurons to form larger strategies.**
  - **Uses SANE (Symbiotic Adaptive Neuro-Evolution).**
  - **Neurons receive credit/blame based on the multiple blueprints they are used in.**
- Learned to defeat Wally on a 5x5 board in 20 generations, but doesn't scale to 19x19 reasonably.

Deciding what to implement…

- Main Goals:
  - **Investigate AI as a potential field of further study as a Master's student**
  - **learn XNA, C#, and Go to make a Go AI**
- Approach:
  - **Read about as many approaches as possible.**
  - **Make some toy AIs to build the core program around.**
  - **Implement UCT- and Monte-Carlo-based approach, because MoGo has had great success with these.**
  - **Combine with a distance function to predict the "advantageousness" (expected win rate) for boards.**

1. Introduction
2. Algorithms
3. Implementation
4. Results

Code Organization

- Coded up a framework to have AI deathmatches, with visuals using XNA (also can have two-human player games that merely enforce the rules).
- White-Player and Black-Player each have an array of the classes that inherit from GoAi.
  - **AI implementations override TakeTurn (board)**
- The active AIs for each player are indicated by an index value.
- Core framework guarantees that there is at least one legal move left when passing control to the AI.
- All AIs guarantee their chosen move is always legal.

Tools Used

- C#
- XNA
  - **XNA Creators Club tutorials and models**
  - **The O'Reilly book**
  - **PrimitiveBatch code**
- .SGF files from real Go games
- GoTraxx parser

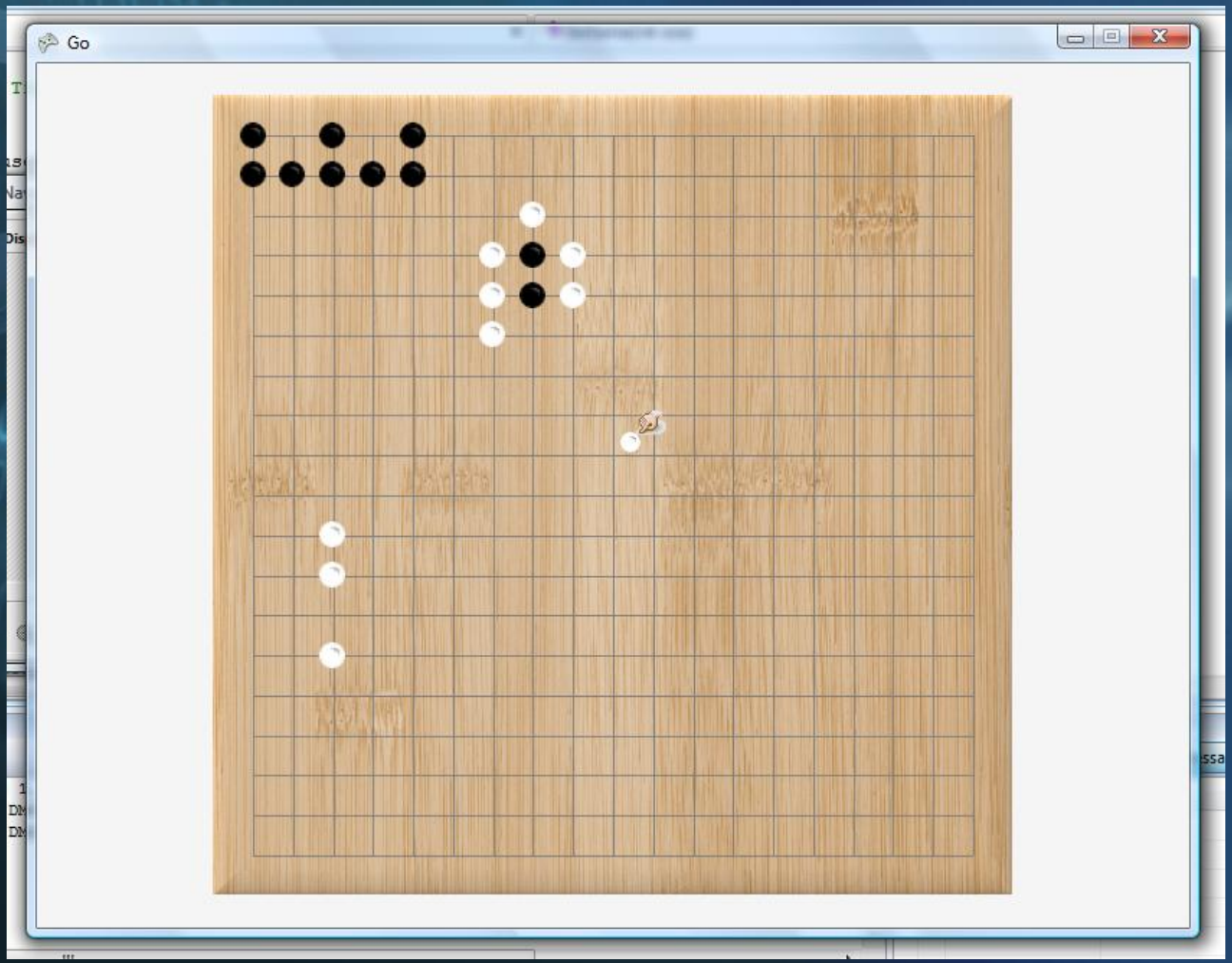
XNA

- XNA = XNA's Not an Acronym
- Framework for hooking into graphics, user input, etc. on Windows or XBox360
  - **Trivial to switch between platforms**
- Useful classes to inherit from
  - **Game**
  - **GameComponent**
- Many powerful systems for download, e.g. Synapse Gaming's "SunBurn"

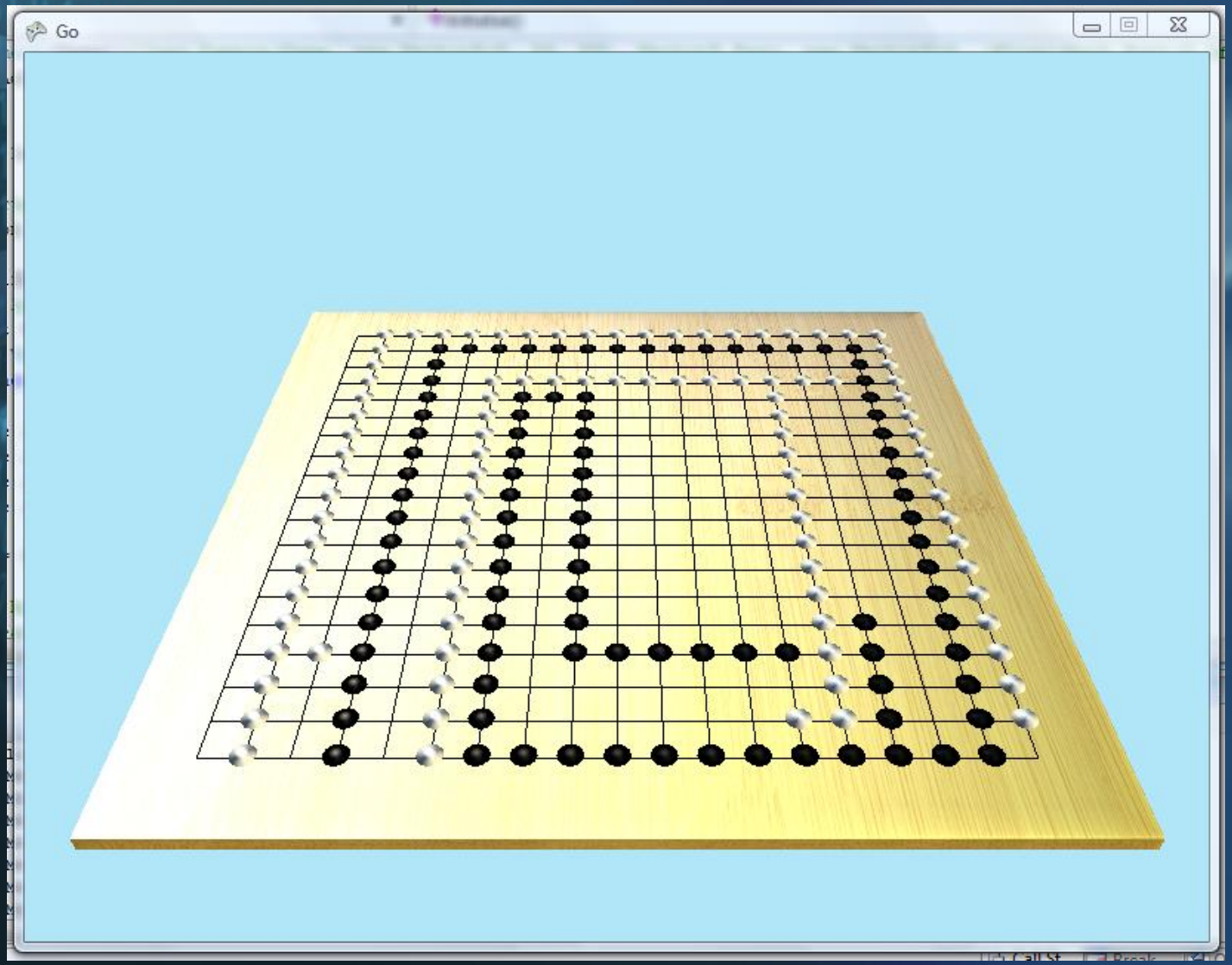
Visuals

- 2D Viewer
  - **Allows players to place stones on the board when there is a human player**
  - **Sprite-based elements for the board and stones**
  - **Uses PrimitiveBatch, downloaded from Microsoft to makes lines, etc. much more intuitive**
- 3D Viewer
  - **Just for fun**
  - **Intended for watching AI deathmatches**
  - **Camera viewing board and stone models**

2D Viewer



3D Viewer



SGF

- SGF = Smart Game Format
- Used Phil Garcia's SGF file parser, written for his program GoTraxx
  - http://www.thinkedge.com/blogengine/page/GoTraxx.aspx
- Translated the result into Library records
- Phil Garcia was very helpful via email ☺

Simple Game Format

- Excerpt:

(;W[dd]N[W d16]C[Variation C is better.](;B[pp]N[B q4]) (;B[dp]N[B d4]) (;B[pq]N[B q3]) (;B[oq]N[B p3])) (;W[dp]N[W d4]) (;W[pp]N[W q4]) (;W[cc]N[W c17]) (;W[cq]N[W c3]) (;W[qq]N[W r3])) (;B[qr]N[Time limits, captures & move numbers] BL[120.0]C[Black time left: 120 sec];W[rr] WL[300]C[White time left: 300 sec];B[rq] BL[105.6]OB[10]C[Black time left: 105.6 sec Black stones left (in this byo-yomi period): 10];W[qq] WL[200]OW[2]C[White time left: 200 sec White stones left: 2];B[sr] BL[87.00]OB[9]C[Black time left: 87 sec Black stones left: 9];W[qs] WL[13.20]OW[1]C[White time left: 13.2 sec White stones left: 1];B[rs] C[One white stone at s2 captured];W[ps];B[pr];W[or] MN[2]C[Set move number to 2];B[os] C[Two white stones captured (at q1 & r1)] ;MN[112]W[pq]C[Set move number to 112];B[sq];W[rp];B[ps] ;W[ns];B[ss];W[nr] ;B[rr];W[sp];B[qs]C[Suicide move (all B stones get captured)]))

- Powerful but at first glance complicated tree-based representation.

Board Library

- Generate if needed, serializing to save to a file
  - **In the event that the library is not generated, just deserialize the saved file**
- If generating, reads in all the files currently in a specified directory
- Parses each read file using Phil Garcia's parser from GoTraxx.
- Saves records of type LibraryBoard, storing a board configuration in the format used by the AI, and an empirically calculated win rate.

Toy AIs

- Random AI
  - **Exactly what it sounds like…**
- Crawler AI
  - **Proceeds left to right, down the board, filling in adjacent spaces**
- Novice AI
  - **tries to maximize the liberties of its next piece**
- Useful for testing the AI system (they're very fast on all permitted board sizes), and gaining an intuition as to what works

Greedy AI

- Decides based only on the predicted results of its current turn; no lookahead.
- If it is possible for it to capture an opponent piece, it does so – capturing as many stones as possible.
- Otherwise, it looks for the largest group of friendly stones, and increases its liberties as much as possible
- Heuristics inspired by user “Hologor” of Sensei’s Library, and his “crawler” AI.

Greedy Score AI

- Similar to Greedy AI.
- Makes a greedy choice, taking the move that will maximize its score on the immediately resulting board.

Naïve Monte-Carlo AI

- For each legal move, performs a constant number of random playouts.
  - **These game simulations run until the game is over**
- For each move, a win rate is calculated based on the playouts.
- The AI takes the move with the highest empirically calculated expected win rate.

Heavy Monte-Carlo AI

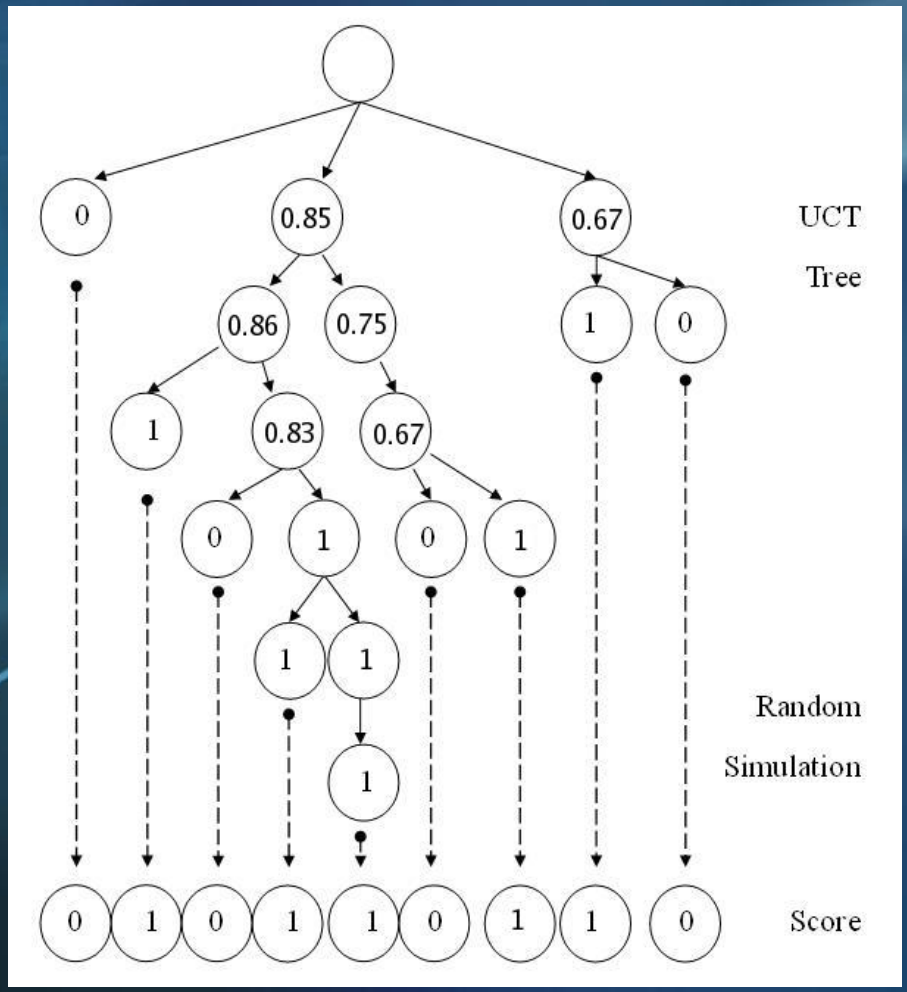
- Also uses Monte-Carlo technique without UCT.
- Playouts are made with heuristic guided moves instead of random ones.
  - **This is called a “heavy” playout.**
  - **The heuristics used were those that were most successful in the Greedy algorithms used.**

Distance AI

- Searches n (friendly) turns deep → move tree
- Uses a distance function to find similar boards in the library
- Uses a weighted average of the similar library boards' previously calculated win rates
  - **(weighted by distance from this board)**
- The move that resulted in the board with the highest score is taken.

Monte-Carlo / UCT AI

From MoGo paper:



Monte-Carlo / UCT AI

- MoGo also uses patterns, to improve the meaningfulness of its random playouts at the leaves
  - **MoGo seems to be applying stored patterns locally, to find moves that will create patterns close to the current area of the board, rather than applying pattern-matching to arbitrary areas**
  - **It only considers moves close to the most recently played move**

- The Monte-Carlo / UCT AI in this project does not use patterns, and in this sense is closer to CrazyStone or earlier versions of MoGo.
  - **It seems that applying patterns without great amounts of trial and error can do more harm than good.**

Genetic AI

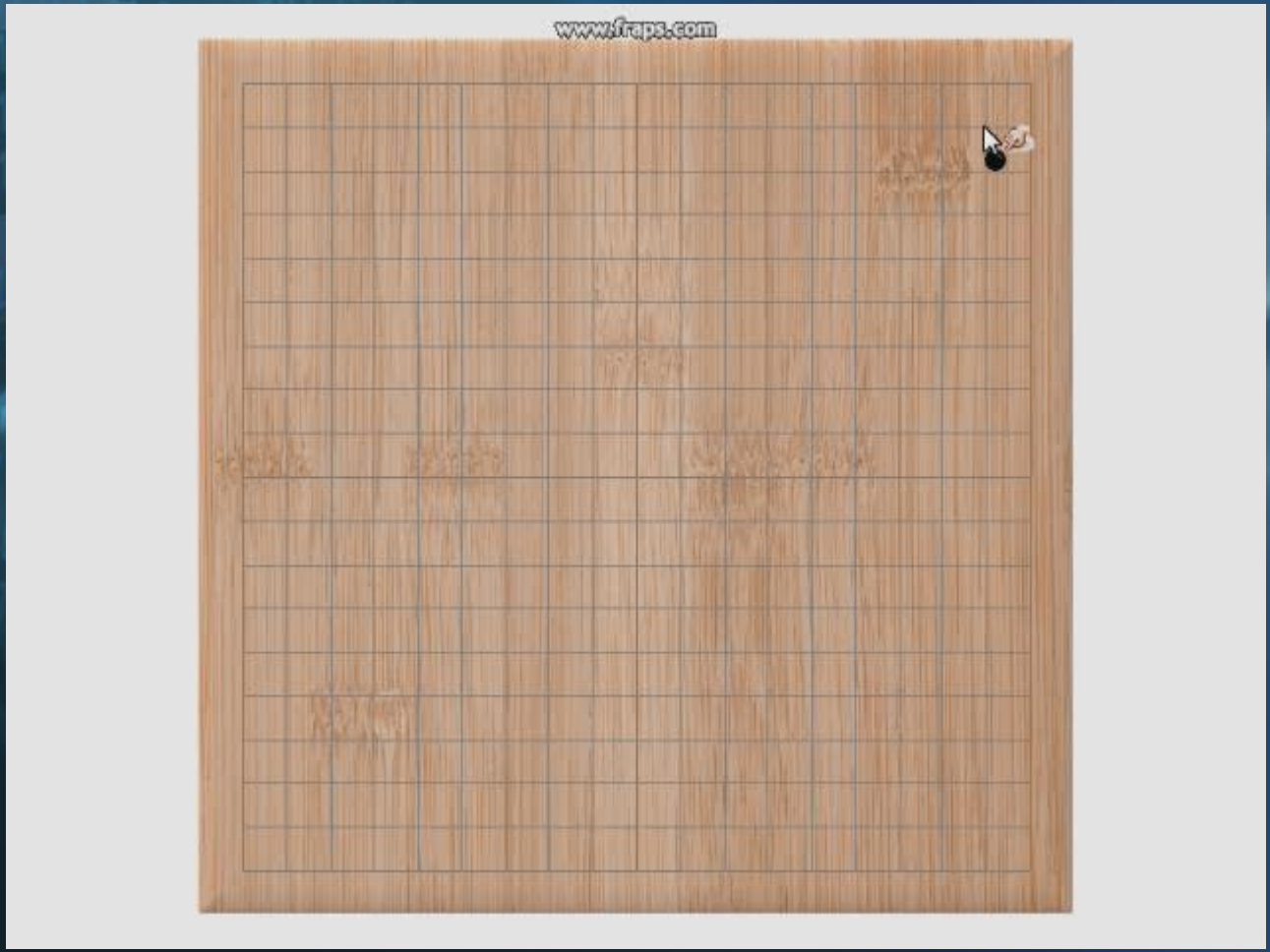
- Beyond the scope of this project, and just for personal entertainment. (Not finished…)

- Loosely based on the paper by Per Rutquist.
  - **Also borrows UCT techniques, but uses the evolved evaluation function to help decide which branches to investigate further.**
  - **A deathmatch between this and Distance UCT AI could be interesting.**

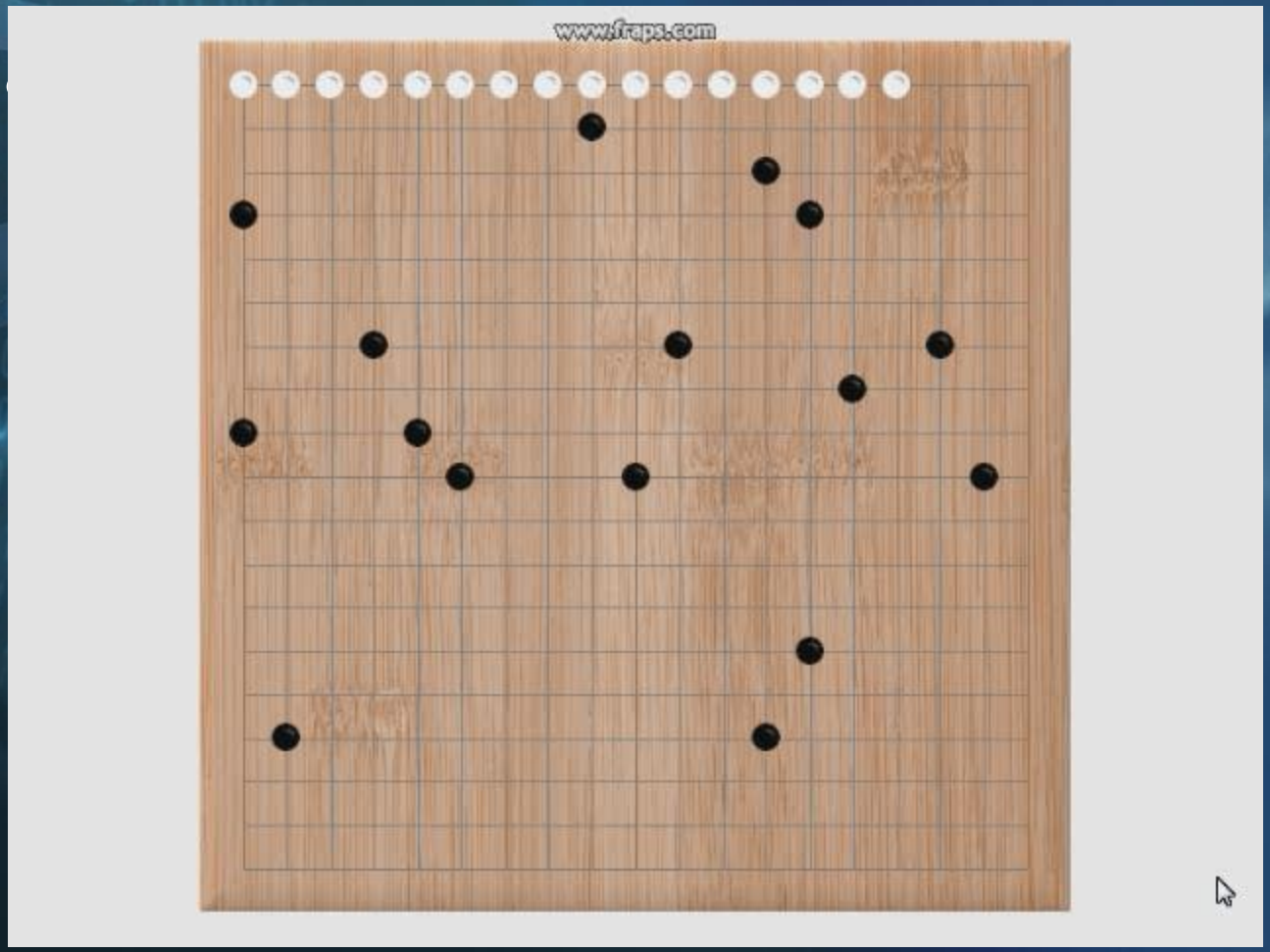
- Potential further investigation: using a UCT-inspired AI, but investigating the branches with the best scores from the Distance evaluation, and the branches with the best scores from the Genetic evaluation.

1. Introduction
2. Algorithms
3. Implementation
4. Results

Human Player



Example of Testing with Toy AIs



Greedy AI

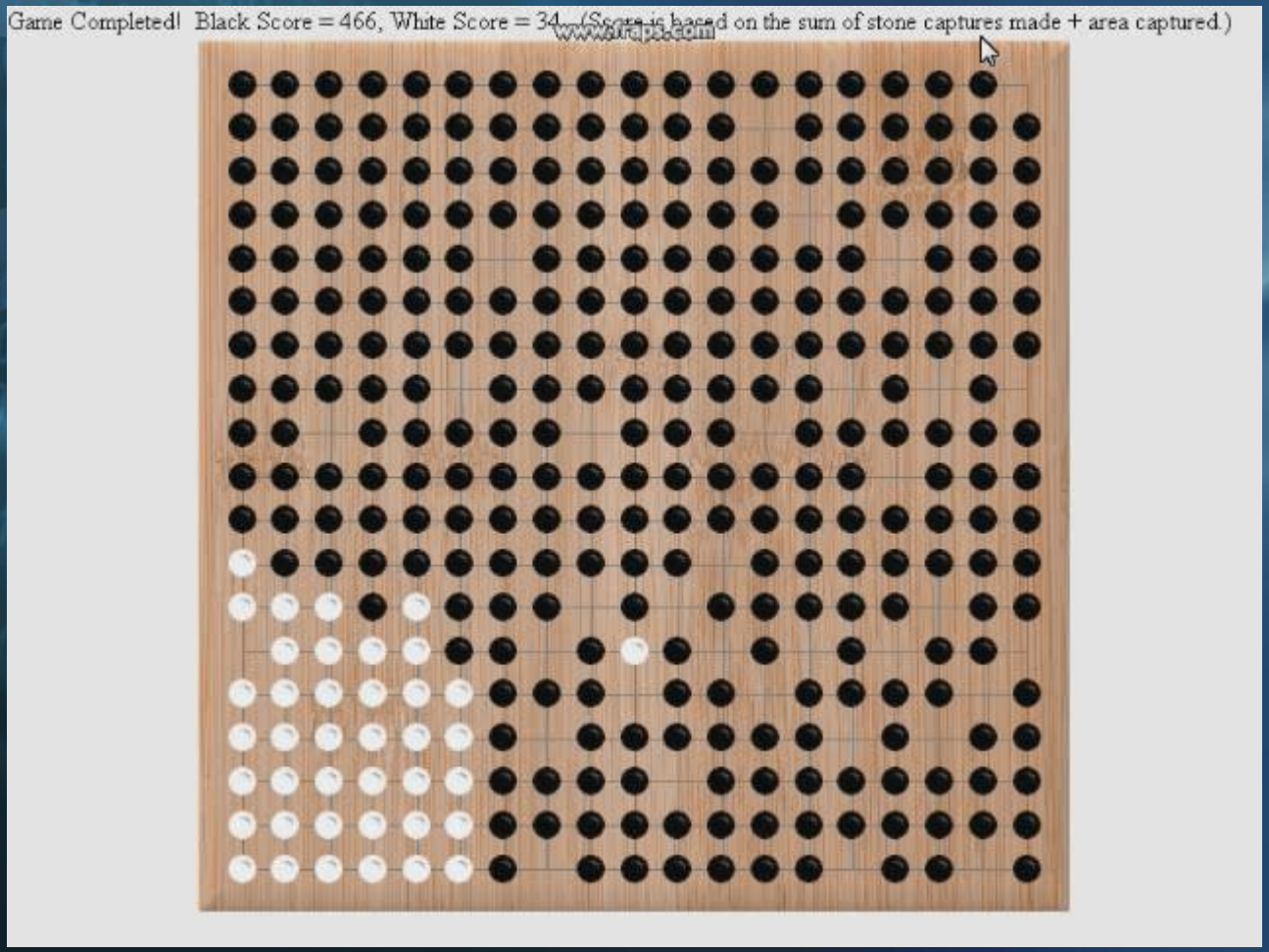
- Randomized version (but nonrandomized version had similar results).
  - **Usually pretty good at not removing its own eyes, as that would not be adding liberties.**
  - **This set of greedy heuristics leads to behavior that helps box in sections of the board.**

- Always very fast on the range of legal board sizes
  - **Strategy is not heavily dependent on board size**

Greedy AI

- Initially beat all the currently implemented AIs, even Naïve Monte-Carlo.
- When more complex AIs are given inadequate resources (number of game simulations per move, or library resources) this one wins

Greedy AI vs. Random AI



Greedy Score AI

- Did not do as well as Greedy AI
- Without lookahead, it lacked the effective “boxing in” behavior that Greedy AI achieved through its greedy novice-like heuristics.

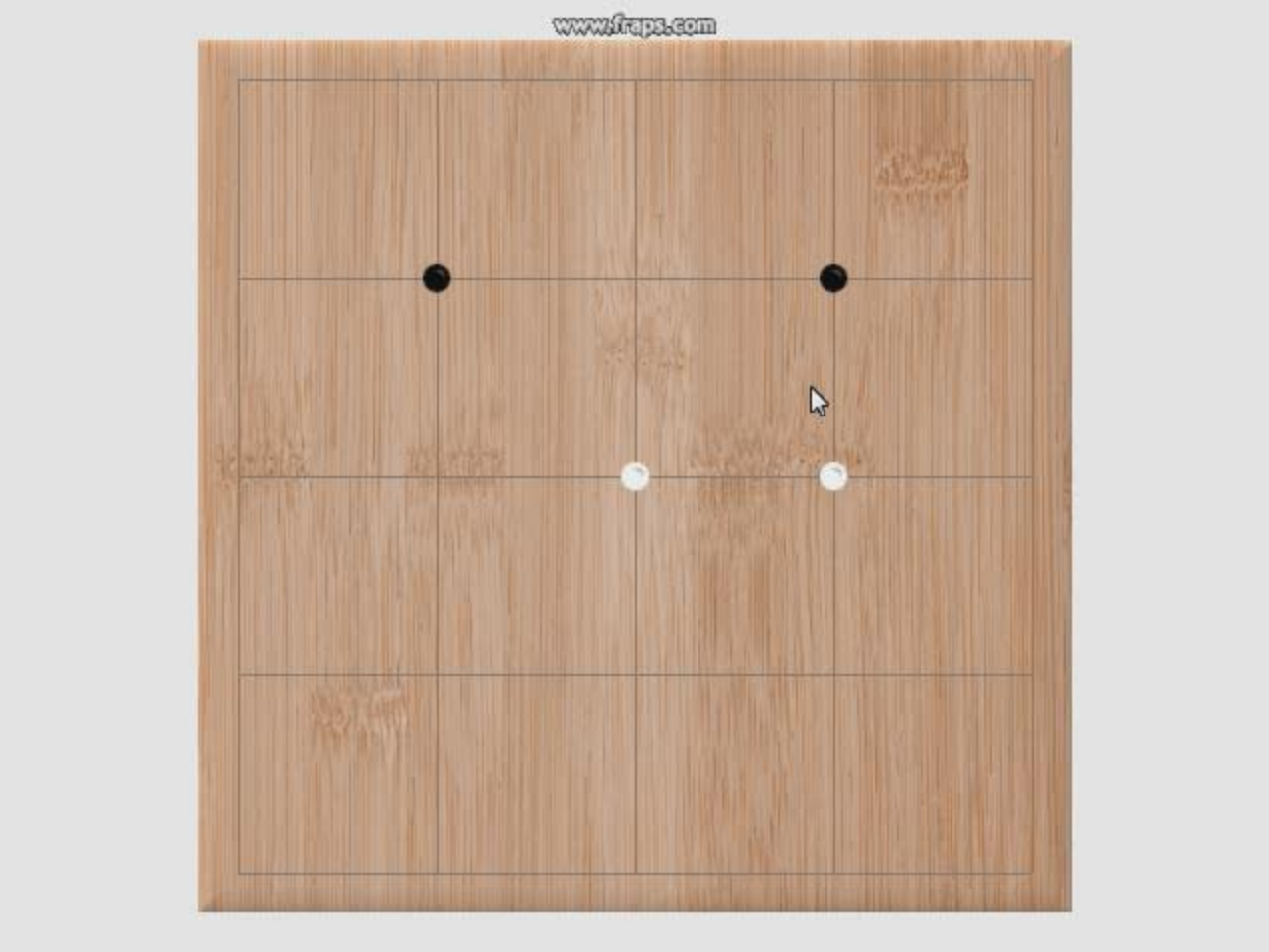
Greedy AI vs. Naïve Monte-Carlo

- During initial trials, Greedy AI / Greedy Random AI defeated Naïve Monte-Carlo AI consistently.
- Considered simulate moves with the same greedy algorithm, but that's cheating…
- Naïve Monte-Carlo AI did not have adequate resources to compete with Greedy AI on large boards.

Naïve Monte-Carlo AI

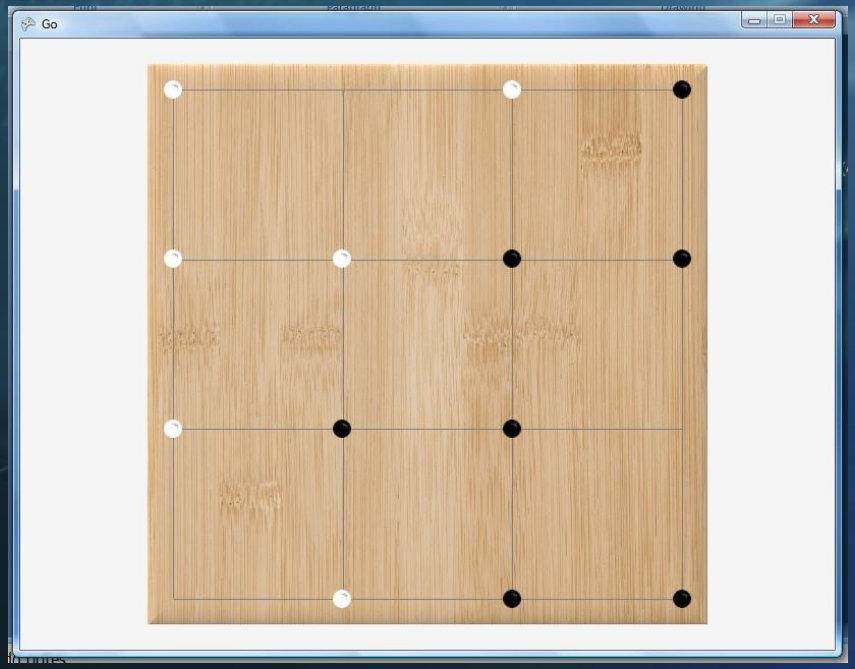
- Great on small boards with high number of sims per move .
  - **e.g. 100+ sims per legal move on a 5x5 board**
- Doesn't scale well.
- Random moves on a Go board actually work decently well to determine influence / *moyo*, especially since there is only one way to move – namely placing down a stone.

Naïve Monte-Carlo vs. Greedy



Heavy Monte-Carlo

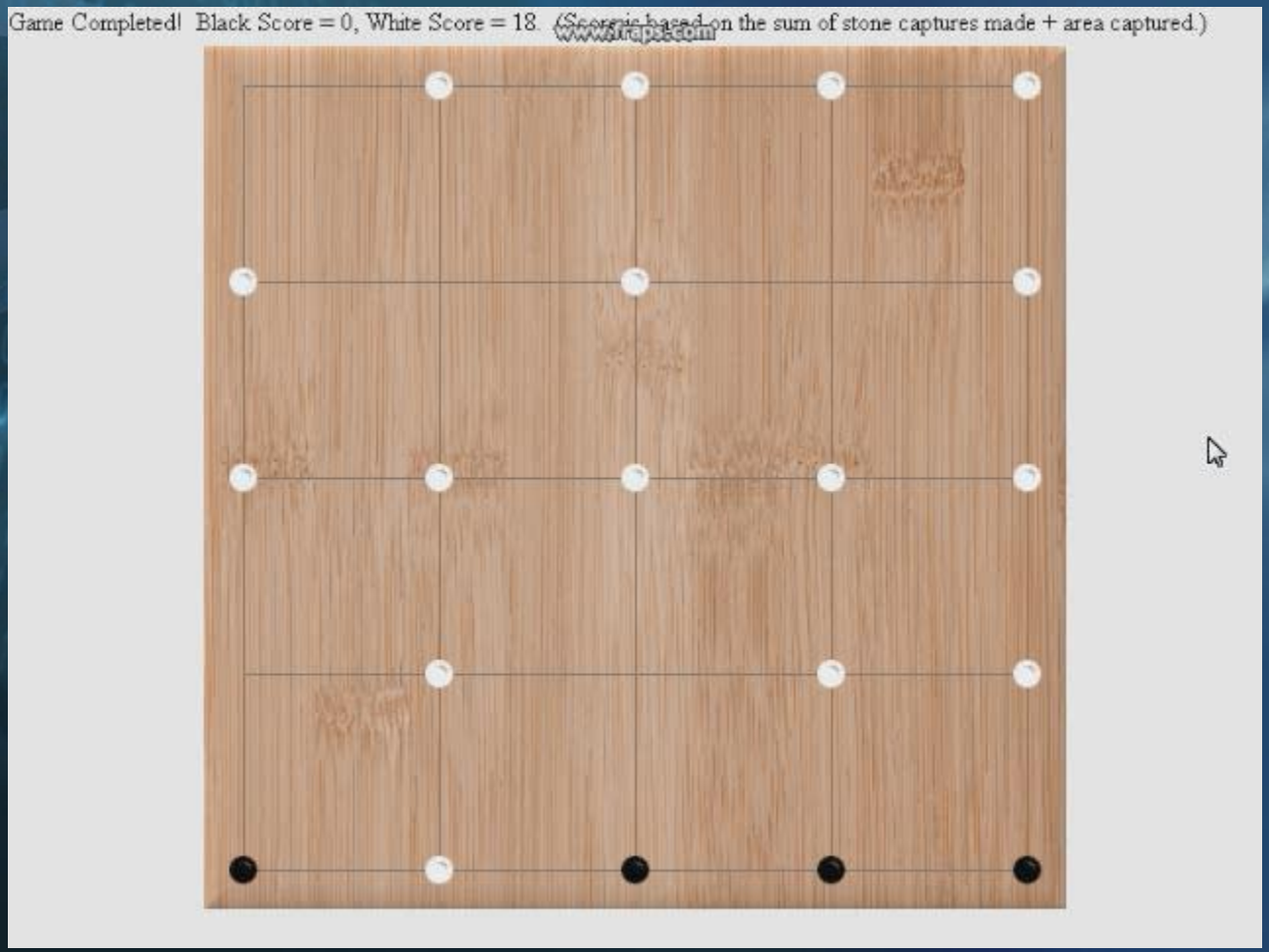
- Even 100 simulations per legal move on a 4x4 board was very slow
- Didn’t do noticeably better than Naïve Monte-Carlo against Random, at least on the 4x4 board.



Distance AI

- Very dependent on the quality of the library
  - **Issues of having enough such that there are always enough boards very similar to the current board**
- Distance function requires a lot of hand-editing
  - **Details such as how to deal with boards in the library that are of a different size.**
- With it's current library, it seems only slightly better than Random
  - **A bad library could feasibly make it play a strategy that is worse than random moves**

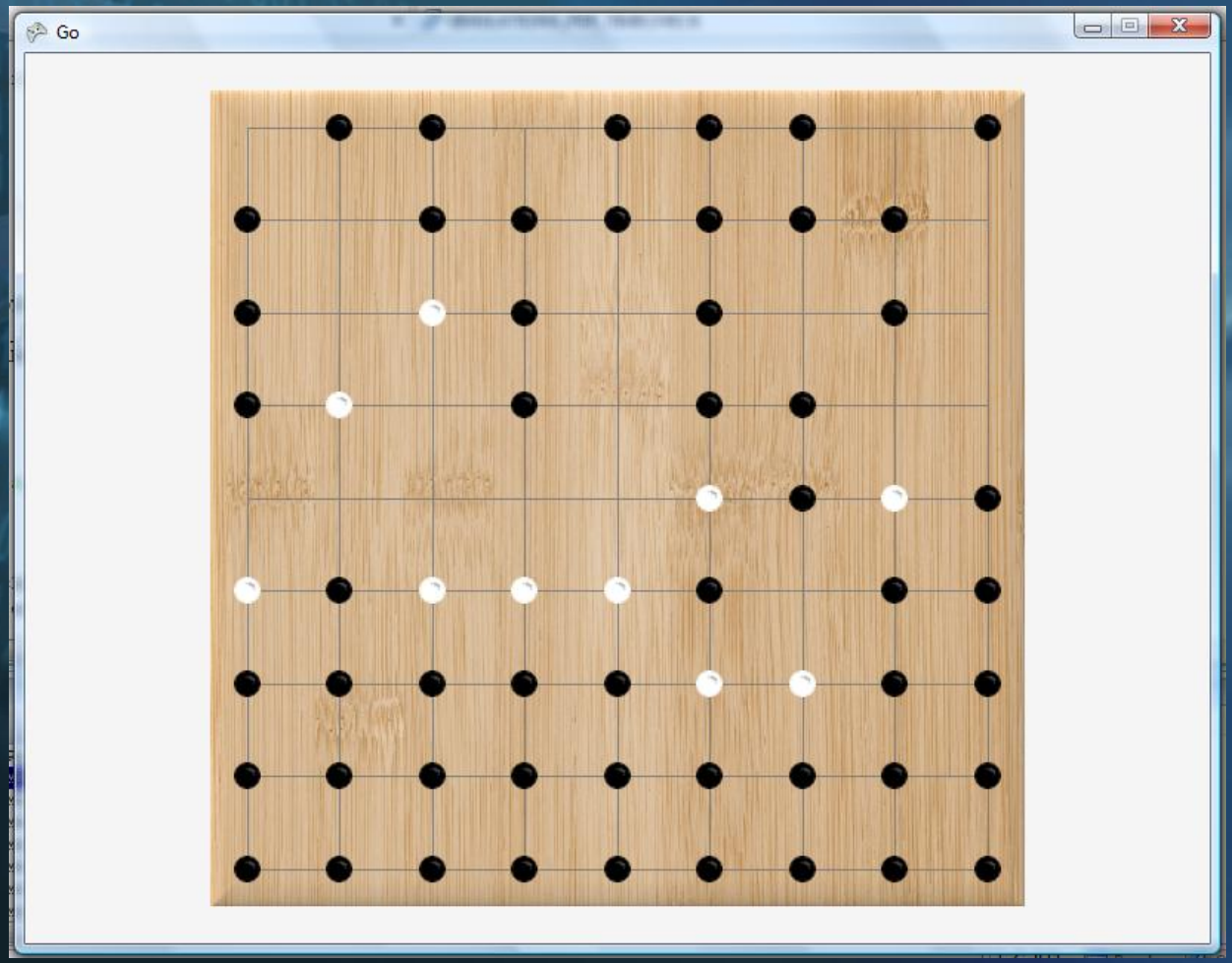
Distance AI


Game Completed! Black Score = 0, White Score = 18. (Score is based on the sum of stone captures made + area captured.)

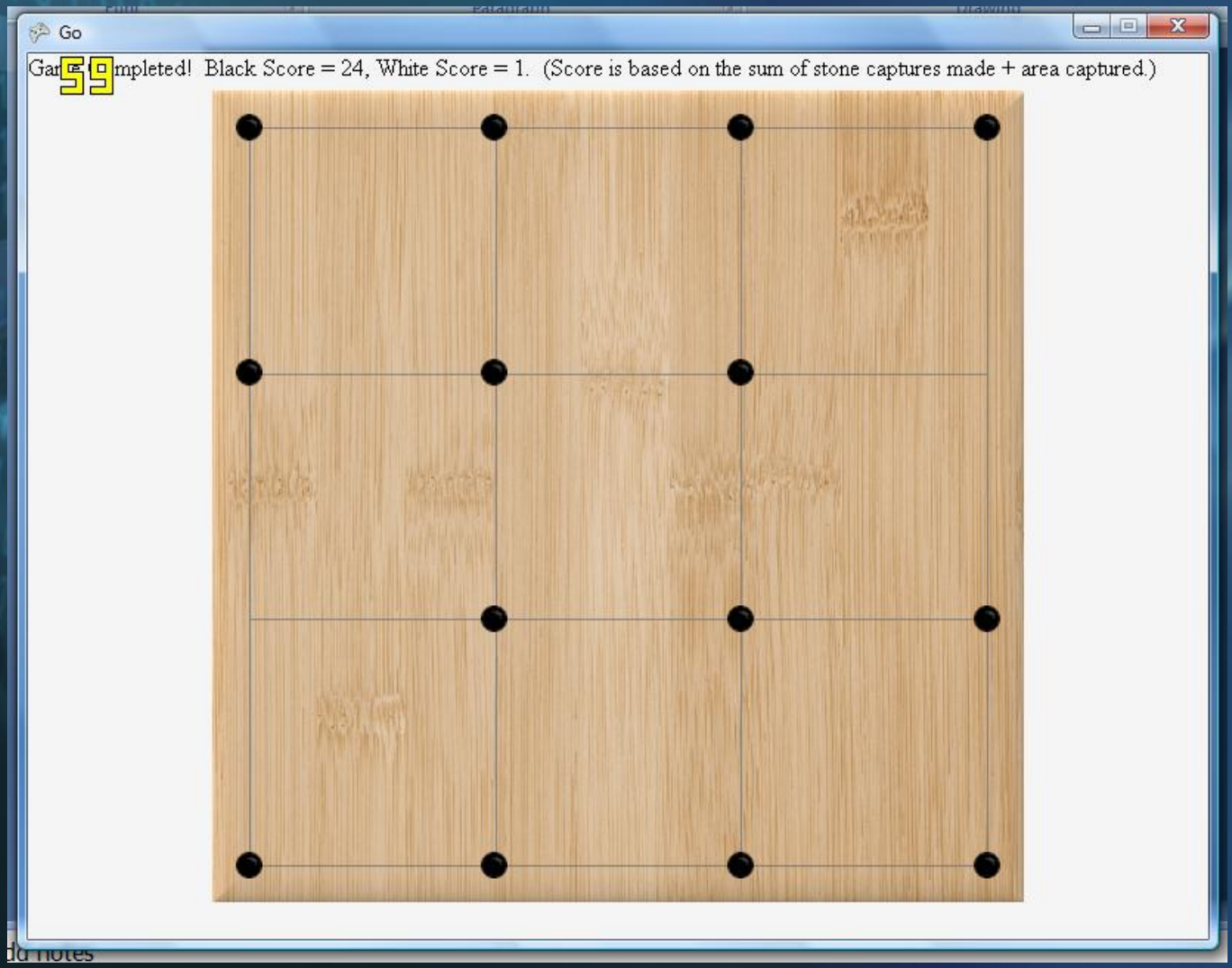
Monte-Carlo / UCT

- Still fairly slow when allowing high numbers of simulations
- Shows signs of strategic moves, as Naïve Monte-Carlo does – seems to have preserved predicitve power
- MoGo has been allowed 12 seconds per move decision, with huge computing resources

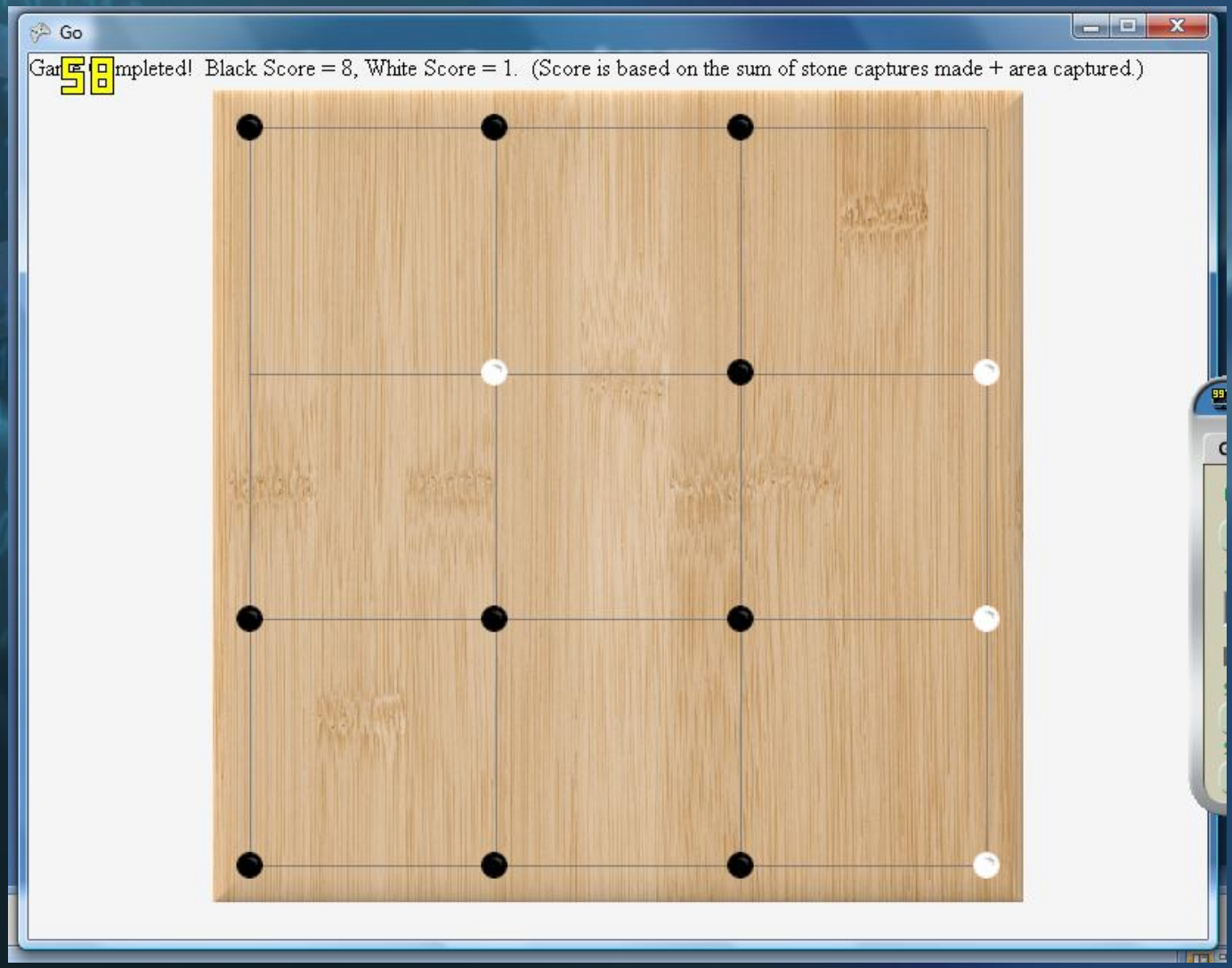
Monte-Carlo / UCT



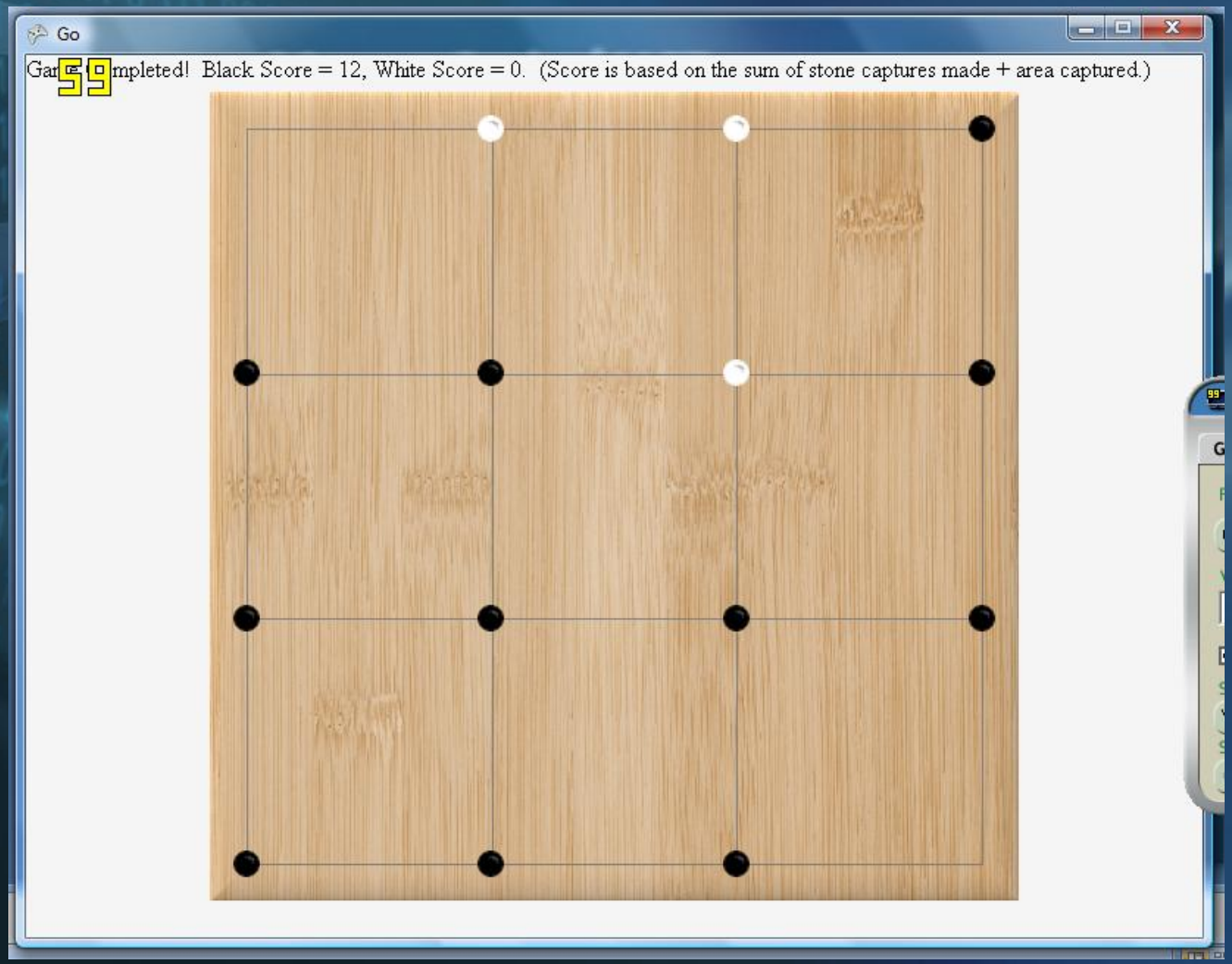
Monte-Carlo / UCT



Monte-Carlo / UCT



Monte-Carlo / UCT



Future Direction

- (Next week) Deciding how much control to give the user, adding a UI based on this, and putting it on the XBox360. (Lua?)
  - **Code has been designed for AIs to be swapped out by changing an int, etc.**
  - **Also tweaking AI, and adding some Monte-Carlo features to Distance AI.**
- Creating a more space efficient tree.
- More work with Genetic AI.
- Looking into applying patterns locally, as done in MoGo, and alternative ways of reducing the scope of the move search.

Summary

- Findings seemed similar to other novice Go players coding AI, found on Sensei's library
  - **Greedy AI was inspired by a poster there, saying how his simple heuristic-based AI was stronger than an AI with lookahead.**
- Monte-Carlo performs well, when given enough resources, as expected.
- UCT with MC from the MoGo paper provides some speedup/scalability, but would likely need the further optimization and hardcoded skill found in MoGo to be great.

Take-Home Points

- Monte-Carlo techniques are well suited to the nature of Go, as it allows complex patterns to emerge on their own
- Monte-Carlo alone doesn't scale well
- MC / UCT is the most competitive approach, as this is a good solution to the Multi-Armed Bandit problem, when carefully optimized

Accomplishments


Thank You!