CSC 581a
December 20, 2002

Las Vegas Go

Piotr Kaminski
98-10403

Table of Contents

1. Introduction

"Go: an hour to learn; a lifetime to master." This maxim seems to apply not only to humans, but to computers as well. While Othello and checkers have already been tamed, with computer players beating human champions on a regular basis, and chess and backgammon are due to succumb any day now, Go has so far defied the best efforts of artificial intelligence (AI) programmers.

The classical approach to teaching computers to play games could be called the "what-if" approach: pretend to make a move, and then try to imagine what the opponent would do in response, repeating the procedure to "see" as far ahead as possible. This approach is intuitive, directly reflecting how most people would start playing a new game. Unfortunately, the combinatorial explosion of possible move choices usually prohibits a thorough exploration of the entire game tree; it is even unlikely that an evaluation will probe to a terminal game position. Thus, AI algorithms use all kinds of tricks to "prune" supposedly-irrelevant branches of the tree, and employ evaluation functions to guess at the value of intermediate positions. The development of these techniques, along with the steadily rising power of computer hardware, has proven sufficient to tackle most games played by humans.

Go is different. The game tree for Go is so wide and so deep that the usual approaches prove powerless against it: the AI flounders at the shallowest levels of the tree or investigates the wrong branches. It doesn't help that moves in Go have incredibly far-ranging effects; a stone placed at the beginning of the game could decide the outcome hundreds of moves later. These, and other issues, also make it very difficult to accurately evaluate the value of a position. [BC01] offers an extensive survey of the most promising AI techniques found so far, but the outlook is bleak. Today, even the best computer player can be beaten by a seasoned amateur, and many people believe that the chances of a machine becoming Go champion by 2020 are at best 50/50.

While the classical approach may yet succeed, it is clearly worthwhile to investigate other options. In 1993, Brügmann (a physicist by day, a Go hobbyist by night) made a daring proposal [Brü93]: eschewing tradition, he developed Gobble, a Go AI that relied purely on a statistical method called simulated annealing. The AI didn't play very well, but it did show some smarts—all without any knowledge of Go except the rules of the game. However, his ideas failed to attract much interest and this avenue of research was apparently never pursued further.[1] Brügmann himself seems to have exited the Go AI scene, and my efforts to contact him were unsuccessful.

This project tries to recreate the AI originally implemented by Brügmann and explore its performance on modern hardware. Since the source code to Gobble is not available, and [Brü93] leaves out many important details of the algorithms used[2], this task proved to be surprisingly challenging but ultimately successful.

The rest of this paper is divided as follows. Section 2 introduces simulated annealing and explains how it might be applied to the game of Go; it is mostly a summary of [Brü93]. I assume that the reader is already familiar with the basic rules of the game. If this is not the case, a quick glance ahead at Section 3.1 should be most helpful. Section 3 describes my implementation of this technique and details some of the obstacles encountered and the solutions I adopted. Section 4 discusses the results of a few experiments run to evaluate the AI and some algorithm variations. Section 5 concludes and presents directions for future research.

Throughout the paper, formal references to refereed papers are called out as usual, while informal pointers to potentially unsubstantiated information are given as in-line links.

2. Monte Carlo Go

This section first introduces the process of simulated annealing by relating it to actual physical annealing. It then shows how simulated annealing could be applied to Go, and describes some of Brügmann's experimental results.

[1] Apart from [BC01], the only other reference I found to Brügmann's original article is in [SDS00], which talks about training a neural network in evaluating board positions.

[2] "Anyone with some experience in programming will agree that an implementation of the algorithm should be straightforward." Hah!

2.1. *Annealing*

Before being able to simulate the physical process of annealing, we must first understand it. Imagine a metal heated up to near its liquid state: the atoms in the metal are moving rapidly and almost randomly. As the metal cools into its solid state, the atoms slow down until they are effectively locked into place by their mutual electromagnetic forces. The total amount of energy remaining in the material depends on the configuration of the atoms. The minimum energy is attained when the atoms form a regular crystalline structure.

The really interesting part, though, is that if the cooling process is slow enough, the billions of atoms somehow (almost) arrange themselves into this regular structure—even though individually each one moves quite randomly. This property was discovered very early (perhaps as early as 7000 BC), and annealing—repeated heating and slow cooling—has been used ever since to make metal objects stronger. If the metal is cooled too quickly, however, the atoms "freeze" in an unfavourable configuration far from the minimum energy levels.

2.2. *Simulated Annealing*

Annealing makes a tempting target as an optimization algorithm, since it does not rely on solving complex equations but rather on repeated small random changes to a system. It is especially appealing if the actual optimization equations are not known, or are too difficult to solve analytically. While annealing rarely results in a global optimum, it can often give an answer that's "close enough"; for example, simulated annealing has solved the Traveling Salesman problem for all practical purposes.

To simulate the physical annealing process, we need five components:

1. A model that can represent all possible configurations of the system of interest.
2. An energy function that measures the energy of a particular model configuration.
3. A set of elementary local moves between model configurations. The set of moves must be ergodic, connecting any pair of model configurations with a finite sequence of moves. (Moves between configurations will later be called "shifts" to prevent confusion with Go moves.)
4. A probability function that determines whether a move that raises the energy of the system is accepted or not. The function must depend on the current temperature of the simulated system, and may also depend on the energy delta of the proposed move.
5. A function that determines temperature based on time, usually called the cooling or annealing schedule. The function is only defined from the origin to some finite positive time, thus determining the length of the simulation.

The simulation begins by choosing a model configuration at random and setting the temperature to its initial (high) value. The simulation then iterates, advancing time and progressively lowering the temperature according to the cooling schedule. At each step, a random move to a neighbouring configuration is proposed. If the new configuration has a lower energy level, the move is always performed; otherwise, the move is performed with a probability defined by the probability function. The simulation ends when the cooling schedule reaches the end.

If the model was accurate and we picked a good probability function and cooling schedule, the final configuration's energy should be close to the global minimum. Finding the right functions seems to be more a matter of art than science. Also, it is clearly very important that the set of moves be ergodic, otherwise the initial (random) configuration could limit the range of the simulated model and prevent it from reaching a good minimum.

2.3. Annealing Go

It's not immediately obvious how to apply simulated annealing to Go. One might first try to make a direct analogy between a Go board and the system's configuration, and Go moves and shifts (system moves). Upon further reflection, this proves worthless. What we want is a winning sequence of moves; what simulated annealing gives us is the winning board—the configuration with the lowest energy. However, a hint for a better approach is embedded in this early failure.

Let us instead consider the *entire sequence of moves* in a game of Go as the system's model. The energy of a configuration is then trivial: it is merely the final score of the game, possibly inverted depending on which side we want to win. So far, this is very similar to the Traveling Salesman problem, where we wish to optimize the path and can easily compute the value of any given itinerary. There is one crucial difference, though: we are really trying to optimize *two* competing sequences of moves, one for Black and one for White. This makes it impossible to find well-behaved local shifts in the configuration space, since in general the slightest change in the order of moves played will unpredictably affect the final outcome of the game. This makes the energy function discontinuous and unsuitable for simulate annealing.

Not all is lost, however. It turns out that in Go many moves are good *no matter when they are played*. For example, a move by Black might be needed to secure an eye. If Black plays that move (as long as the local situation remains unchanged), the group lives; if White prevents Black from playing that move, the group dies. The final scores of the games differ, on average, by the value of the group.

This time-invariance heuristic is of course not perfect, and its applicability depends very much on the specific situation. Nonetheless, it is a useful guideline, and provides a basis for our annealing algorithm.

2.4. Gobble Algorithm

The basic idea is two-pronged:

1. Identify (time-invariant) good moves.
2. Play them as early as possible.

The idea is in good company, since it seems related to the so-called "killer heuristic" for α-β search as well as the suggestion oft given to novices to "look for the biggest move on the board".

The actual simulation works as follows. Initially, each simulated player chooses a random sequence of moves, taking into account that it's possible to play on a field more than once due to captures. A game between the two players is simulated, starting from the "real" game's current state and playing consecutive moves from each sequence; if a move happens to be illegal, the next legal one is played instead. At the end of the game, the score is calculated and added to the average value of each move played during the game.

The move sequences are then approximately sorted by move value. Moves are allowed to be out of the perfect order with a probability proportional to the temperature, accounting for configuration shifts that temporarily lead away from the minimum. The temperature is lowered according to the cooling schedule, and another game is simulated with the newly re-ordered sequences. When the temperature reaches zero, the "best" move for each player will be at the front of the respective move sequence.

A few details are missing from the above description: When does a game end? Exactly how is a sequence "almost" sorted? What is the cooling schedule? Answers to these questions will be provided in Section 3, in the context of exploring possible alternatives.

As motivation for the long discussion ahead, consider this: Gobble, running on a 16MHz PC and playing one move per minute on a 9x9 board using the algorithm above, managed to achieve a rating of about 25 kyu in informal experiments, with almost no knowledge beyond the rules of the game.

3. Las Vegas Go

Based on the information above, and with little knowledge of Go and no experience whatsoever with simulated annealing, I started to implement Vegos, the Las Vegas Go[3] AI. Naturally, I quickly ran into problems.

3.1. Go Rules

"Go: an hour to learn; a lifetime to master." is a lie. It takes far more than an hour to understand all the subtleties of the rules. It doesn't help that there's many incompatible rule sets in widespread use, and that nearly all rely on heuristics for determining the final score of a game. I finally found the Tromp/Taylor rules (a restatement of the New Zealand rules) that give a straightforward, almost mathematical definition of the game:

[3] Las Vegas being a bigger, shinier version of Monte Carlo.

1. Go is played on a 19x19 square grid of points (though the size can be changed by prior agreement), by two players called Black and White.
2. Each point on the grid may be colored black, white or empty.
3. A point P, not colored C, is said to *reach* C, if there is a path of (vertically or horizontally) adjacent points of P's color from P to a point of color C.
4. *Clearing* a color is the process of emptying all points of that color that don't reach empty.
5. Starting with an empty grid, the players alternate turns, starting with Black.
6. A turn is either a pass or a move that doesn't repeat an earlier grid coloring (positional superko).
7. A move consists of coloring an empty point one's own color; then clearing the opponent color, and then clearing one's own color. (Suicides are implicitly allowed.)
8. The game ends after two consecutive passes.
9. A player's score is the number of points of her color, plus the number of empty points that reach only her color. (Area scoring; it doesn't cost anything for a player to make a move within their own territory.)
10. The player with the higher score at the end of the game is the winner. Equal scores result in a tie.

It is also possible to add rules for komi (a set point bonus for White) and handicap (a set number or sequence of moves that Black may play with no reply from White). While the rules are simple, they are quite close to the more traditional versions, and should result in similar game outcomes.

Implementing this clear set of rules was fairly easy. While I wasn't overly concerned with efficiency, it was fun to implement the positional superko rule as a hash table lookup, so that the move validation time does not increase with the length of the game. Strangely, neither Gobble nor most of the other basic Go programs implement the superko rule.

3.2. To Pass or Not To Pass

Armed with a mechanism for enforcing the rules, I set out to implement Gobble's simulated annealing algorithm, when a conceptual difficulty reared its ugly head. According to the rules above, the game ends after two consecutive passes; it follows that a player (even a simulated one) must decide when to pass if the game is to end.[4] However, passing is anything but time-invariant! If passing were a "normal" move, subject

[4] Technically speaking, the game might end thanks to a lack of legal moves due to the superko rule, but this eventuality seems rather unlikely.

to the simulated annealing algorithm, then passing at the end of a successful simulated game would raise the move's value and tend to make the player pass sooner in the next one. Furthermore, a pass ending the simulated game at the wrong moment would assign an incorrect weight to the moves played (e.g. because the opponent's dead stones had not yet been eliminated from the player's territory).

Though I spent a long time looking for an elegant solution to this problem, I eventually followed Gobble's lead and redefined the rules of the game. The new rules eliminate passing by changing two of the points above as follows:

6. A turn is a move that doesn't fill an eye or repeat an earlier grid coloring.
8. The game ends when a player has no legal moves left on his turn.

The problem now is to define what an "eye" is. Since the full definition can become quite complex, I once again use Gobble's simplification: an eye is an empty point whose direct neighbours are all of the player's color, and whose diagonal neighbours contain no more than one stone of the opposite color (except in corners and border fields, where no diagonal neighbours of the opposite color are allowed). This directly forbids filling one's own eyes, and the superko rule indirectly forbids committing suicide in the opponent's eyes, since this would immediately repeat the previous position.

These "semi-primitive" game rules dispose of the passing problem, but have two undesirable side-effects. First, the distance the game further from commonly accepted rule sets, though I believe that a good strategy for a semi-primitive game will also be a good strategy for a conventional game. Second, they introduce an element of advanced Go knowledge (the eye) into the program, partially invalidating the claim that the AI knows only the basic game rules. This was the only argument against Brügmann's results with Gobble; Vegos dodges this criticism somewhat since the eye rules are part of the game, not the AI, and thus bestow an equal benefit on all players.

With adequate rules finally in place, it was time to tackle the annealing algorithm itself. Many niggling details remained to be settled: strategies for counting and weighing the moves, mixing the sorted order, and the cooling schedule itself. The next two sections describe the algorithms adopted in depth; the impatient reader is encouraged to skip ahead to Section 4, where I explore the consequences of these strategies on the quality of play.

3.3. Counting and Weighing

Gobble's description doesn't make clear exactly what counts as a "move". If a stone is played twice on the same field during a game, are those two different moves? Does the answer change depending on whether the opponent plays on the field in-between the two plays? This decision will affect how many distinct moves there are for each player, and thus indirectly how a game's score influences their weights. I have identified three distinct move-counting strategies:

- Global count: A count is kept for each field. Whenever a stone of either color is played on a field, its count is incremented. A move is identified by its color, the field it's played on, and the field's count at the time the play is made. Thus Black-A5-1 is different from Black-A5-2, and playing a White-A5-2 would bump the Black's last move to Black-A5-3.
- Local count: A count is kept for each field separately for each color. Whenever a stone is played on a field, the field's count for that color is incremented. A move is identified as above. Thus Black-A5-1 is different from Black-A5-2, but having a White-A5-1 (or White-A5-2) would not change Black's moves.
- Null count: No counts are kept at all. A move is identified only by its color and the field it's played on, so the same move can be played many times in one game. However, the game's score is only added to the move's average weight once, no matter how many times it was played.

Between cooling iterations, the move set is sorted by weight. For global and local counts, each move is unique; for the null count, duplicates are eliminated since all copies of a move would have the same weight and cluster together when the sequence is sorted.

A careful reading of [Brü93] reveals that, at least for some experimental setups, Gobble used the global count strategy.

3.4. Mixing and Cooling

Unlike move counting and weighing, Brügmann is fairly specific about Gobble's mixing and cooling strategy. After each cooling iteration, each simulated player's move sequence is sorted by weight, with the "best" move at the front. The program then makes a single pass from front to back, swapping two adjacent moves with probability p_{swap}, and independently of the moves' weights. The cooling schedule reduces p_{swap} linearly with a few extra "settling" iterations at the end where p_{swap} equals zero. Unfortunately, Brügmann omits the actual starting value he used for p_{swap}.

The effect of this mixing strategy is to shift a move n steps down the list with probability[5] $(p_{swap})^n(1-p_{swap})$, and shift a move 1 step up the list with probability p_{swap}. Moves can never rise more than 1 step per iteration. To investigate the importance of the mixing strategy, I added two more options:

- Push Down mixing: Gobble's standard strategy, as described above.
- Far Swap mixing: Sweep down a list once, swapping each move with one n steps further down the list with probability $(p_{swap})^n(1-p_{swap})$. The difference with Push Down is that moves between the pair being swapped are not disturbed, and moves have a better chance to travel upwards. On the other hand, the exact probability that a move will travel n steps down the list is more difficult to de-

[5] Brügmann states the probability as just $(p_{swap})^n$, which seems to be incorrect.

termine, since each move might participate in multiple downwards and one upward swap.

- Null mixing: No mixing at all. The perfectly sorted order is used directly.

Both the Push Down and Far Swap mixers use a linear cooling schedule, but p_0 and the settling length can be varied.

In Vegos, to ensure that each simulated player will always have enough prepared moves to complete a game, the move sequences are conceptually infinite. When sorting, all played (weighted) moves are put in front of all unplayed ones. Should a player run out of weighted moves, it will start generating random legal moves to extend the sequence. Move swapping is allowed to extend into the "virtual" tail of the sequence, swapping in random moves.

4. Vegos Experiments

Enough prevaricating about algorithmic details: does it work? This section attempts to answer the question by giving the results of a number of experiments that were run on Vegos.

Different AI algorithms are used in the experiments. They are given short, common names to make them easier to remember. The first computer player is Randy, a rather absent-minded fellow that makes all his moves at random. The second AI is Stanley, who uses the simulated annealing algorithm described above. Stanley is heavily parametrized, so a typical description looks like this:

`Stanley (3×1000 steps; count: Global; mix: Push Down, 0.95, 12)`

This means that Stanley is using the Global move counting strategy and a Push Down mixer with a linear cooling strategy with p_0 set to 0.95 and 12 settling steps. The first two parameters need further explanation. In the example above, 1000 is the familiar number of cooling steps that determines the number of simulated games that will be played during each cooling cycle. The p_{swap} value will decrease linearly over this number of steps, minus the number of settling steps; in this example, p_{swap} would start at 0.95 and decrease by 0.95/(1000-12) each step. The other parameter (3 above) is the number of cycles to be run. Each cycle will be run for the given number of steps and the weights of like moves that come out of each cycle averaged to reveal the best move.

The third AI, Ruby, will be introduced later on.

All boards shown are screen captures from GoGui, a Go front-end program that was connected to Vegos by means of the Go Text Protocol (GTP). While a far cry from being aesthetically pleasing, GoGui has the distinct advantages of being open-source and providing some extra analysis display capabilities. In theory, Vegos could be hooked up to any front-end that supports the GTP (e.g. gGo, Sen:te Goban, or any other program that can interface with Gnu Go), but I didn't have time to try this.

4.1. First Move

The natural place to start an investigation is at the beginning, and so I first spied on Stanley's view of an empty board. Given the first move with Black on a 9×9 board, Stanley generally prefers to play in the center. Figure 1 shows a typical analysis of the board after 10000 steps. The numbers indicate the expected score of a game if a move is made onto that field, while the blob on each field is proportional in size to the number of simulated games in which the move was played.

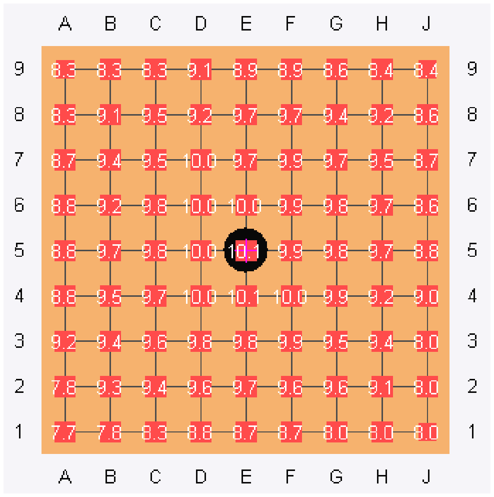

Figure 1. Stanley (1×10000 steps; count: Null; mix: Far Swap, 0.99, 10)

The exact numbers (and the extrema) vary each time the game is restarted, but the general proportions are fairly stable. The difference between the lowest and highest weights usually ends up around 2 to 2.5 points, and the largest difference between values on symmetric fields around 1 point (giving an error estimate of ±0.5 points). Interestingly, when the number of steps is lowered to 1000 in Figure 2, the weights are better differentiated with a Δ of 4 points. It's possible that increasing the number of steps past a certain point does not bring any increased accuracy—more on this later.

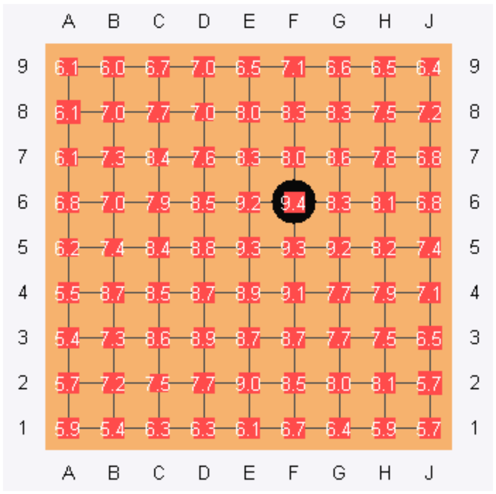

Figure 2. Stanley (1×1000 steps; count: Null; mix: Far Swap, 0.99, 10)

Another interesting feature to note is that all moves have been played approximately the same number of times. This is surprising, since the simulated annealing algorithm should shift the best moves to the front of the sequence and they thus ought to be played more often.

Occasionally things go drastically wrong: Figure 3 shows an atypical analysis result. Though the mixer was changed to Push Down, this effect can occur with either mixer. Lowering p_0 seems more likely to evoke it, but it is difficult to reproduce consistently. Even setting the mixer to Null doesn't always cause it.

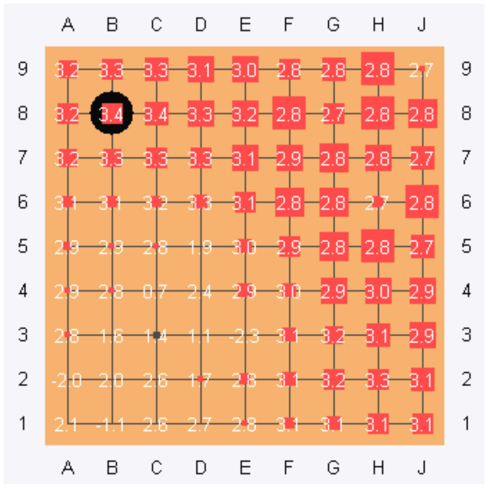

Figure 3. Stanley (1×10000 steps; count: Null; mix: Push Down, 0.99, 10)

My hypothesis is that this is the result of a capture, where the model configuration gets stuck in a local minimum early on. It might indicate a lack of ergodicity in the move set, or some other more subtle problem with the simulation algorithm or parameters.

Changing the counter to Local or Global gives even worse—and very consistent—results, as shown in Figure 4. The configuration once again seems to be captured, but this time the weights are completely random!

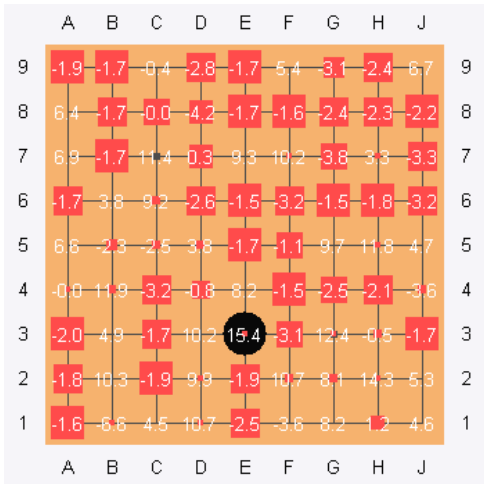

Figure 4. Stanley (1×10000 steps; count: Global; mix: Far Swap, 0.99, 10)

These first experiments seem to indicate that the Global and Local counters are worthless, and that the Null counter is not very sensitive to the mixer strategy or the number of steps. The Null counter also produces results that are closest to those reported for Gobble, which is strange since Gobble apparently uses a Global counter. Gobble also gets a much bigger Δ of 4.4 points on 10000 steps, further confirming that its algorithm is different from Vegos/Stanley.

I tried a few experiments on boards of size 19×19, but all the results were similar to those in Figure 3. I was unable find a combination of parameters that wouldn't result in capture.

4.2. Tournaments

After the micro-level experiments, it only seemed right to try some macro-level statistical studies of the program's performance. To this effect, I ran a series of tournaments between different AI setups to try to determine the relative strengths of the algorithms under different parameter values. This section summarizes the results of these competitions and makes a few hypotheses about their meaning.

The first small tournament I ran pitched Stanleys with varying counting techniques against each other. All Stanleys were set for 1×5000 steps, with a Far Swap mixer at 0.99, 10. All three counters were used. While the tournament was only over 12 games, the Null counter won all the games it participated in hands-down. The Local and Global counters *drew* all four of their games against each other. This seems like an amazing coincidence, and is likely a symptom of some bug in the counter algorithms. Nonetheless, combining this new information with the results observed in the previous section, I decided that the Null counter was completely superior to the other two, and used it exclusively for the rest of the matches.

In the second tournament, I demonstrated that Stanley can win against Randy (the random player) even with very few refinement steps. Three Stanleys participated, all using the Null counter and Far Swap mixer with 0.99, 10 (standard equipment from now on, unless otherwise mentioned). The Stanleys were set at 1×100, 1×300 and 1×500 steps. All played every game perfectly against Randy, demonstrating that the Stanley algorithm has a kyu rating above 50 (the generally accepted rating of a random player). Between themselves, the 100 step Stanley was fairly consistently beaten, but the scores were nearly even between 300 and 500 steps. This may indicate a tapering off of the performance curve, but it may also just be a statistically insignificant coincidence. More testing is necessary.

In an attempt to determine whether Stanley's reuse of previously successful moves was an effective strategy, I introduced a new AI player: Ruby. Ruby counts and weighs his moves in the simulation in the same way as Stanley, but always picks completely random moves when playing, independent of any learned weights. I first pitted Ruby against Randy, to see if it was at all effective. Rubys with step counts of 1×100, 1×300 and 1×500 played perfect games against Randy, and placed similarly to the three

Stanleys from the previous tournament amongst themselves, demonstrating that removing the learned move sequence and mixer preserved many properties of the algorithm. An all-Ruby tournament provided further confirmation that quality of play generally increases with the number of steps, with the 1×1000 Ruby losing more often than winning against the 1×2500 and 1×5000 Rubys.

The next stage was to pit Stanley against Ruby. With steps set at 1×1000, 1×2500 and 1×5000 on both sides, the results were clearly in favour of Stanley. The worst Stanley (1×1000) got more points than the best Ruby (1×2500, strangely enough), demonstrating that reusing the weighted move sequence when simulating gives a definite advantage. However, the results were not a total sweep for Stanley: the Rubys won a fair amount of games as well, showing that the move sequence is not a critical component.

The final tournament delved into the use of multiple cycles with Stanley. Six Stanleys participated, set at 1×1000, 2×500, 4×250, 1×2500, 2×1250, and 4×625. The most interesting result is that the 1×1000 player performed just as well as the 1×2500 one. The multi-cycle players all consistently performed worse.

These are only preliminary results, and far more studies are necessary to accurately characterize the various AIs.

5. Conclusions

I consider Vegos to be a guarded success. Stanley always beats Randy, even at low refinement counts, showing that it has "learned" something about playing Go after only being told the rules. Stanley is also superior to Ruby, showing that the simulated annealing algorithm has some positive effect on the quality of play. I was disappointed that I wasn't able to reproduce Gobble's results precisely; I'll need to further investigate the capture scenarios to determine the flaws in my algorithm or parameters. I was also disappointed with the multi-cycle results, since I was hoping to distribute cycles among multiple machines to improve quality of play. Instead, it seems that running a single (even if shorter) cycle is better.

Finally, here's a quick list of ideas of future work:

- Try Gobble's two-stage strategy, where the simulated annealing algorithm picks the top 3 moves, then evaluates them further using the same approach.
- Try higher-order counting strategies, where the identity (and thus weight) of a move depends on some of the moves that were played before.
- Replace simulated annealing with other algorithms, e.g. Noah's Flood. Simulated annealing is notoriously inefficient, and many replacements have been proposed in other domains.
- Look for other ways to distribute the computation; I still think it'll be easier to do this for statistical algorithms than for an α-β search.

- Try to teach Stanley to play using different rules. It would be interesting to try true primitive rules, but it's more urgent to figure out a decision algorithm for passing, so Stanley can start playing other opponents using the "normal" rules.
- Finally, try to raise the board size progressively to 19×19. This will require more computational power and should be seen as a long-term goal.

6. References

[BC01] Bruno Bouzy, Tristan Cazenave. *Computer Go : an AI Oriented Survey*. Artificial Intelligence, Vol. 132(1), pp. 39-103, October 2001.

[Brü93] Bernd Brügmann. *Monte Carlo Go*. Computer Go Magazine, 1993.

[SDS00] Nicol N. Schraudolph, Peter Dayan, Terrence J. Sejnowski. *Learning To Evaluate Go Positions Via Temporal Difference Methods*. IDSIA-05-00, 2000.