ScratchGo: An Integrated Approach to Playing Computer Go

Woodward H. Folsom, IV

1 Introduction

The game of Go is a conundrum in the field of artificial in- telligence. The number of rules and types of pieces are smaller than those of chess or even checkers, yet computer agents for Go are currently unable to regularly defeat professional players on even the smallest board. Several aspects of Go, including the percentage of potential moves which are legal and very large branching factor of the game space, give rise to this problem.

The focus of this project was on small board Computer Go, an unsolved problem since the first agent was devised by Albert Zobrist in 1970 [1]. The initial objective was to combine tree search and machine learning techniques to develop a platform capable of online play against humans and other computer agents. A secondary goal was to minimize the use of expert knowledge in creating this program.

1.1 Go Basics

The game of Go, also known as Wei-qi in China and Baduk in Korea, is one of the oldest board games currently played. It is also one of the simplest games to learn. According the the American Go Association, chess great Edward Lasker famously noted, "the rules of go are so elegant, organic, and rigorously logical that if intelligent life forms exist elsewhere in the universe, they almost certainly play go" [2].

Go is typically played on a wooden board imprinted with a 19 by 19 grid of lines. Two players, Black and White, alternate placing stones on the intesections of these lines. Adjacent stones of the same color are considered to be part of a *group*. If any of the stones in a group is adjacent to an empty intersection, that group is said to be *alive*. If a stone or group should be completed surrounded by enemy stones at every adjacent intersection, that group *dies* and is removed from the board. It is typically illegal for a stone to be played in a position where it was no *liberties* – adjacent empty intersections – unless doing so captures an enemy group. Play continues until one player resigns or both players pass in succession.

There are many systems of scoring Go, of which the most widely used are Japanese and Chinese. For simplicity, Chinese scoring is used here. In this system, a player receives one point for every stone on the board[1], plus one point for every intersection of *territory* surrounded by that player's stones. Intersections adjacent to both players' stones are not counted as territory. In addition, White receives a bonus known as *komi* in return for moving second. Komi is typically 5.5 – 7.5 for a 9x9 go board (the fractional portion prevents a tie game).

2 Related Work

Despite the challenges faced by classical approaches, the field of computer Go has seen dramatic advances since the use of Monte Carlo Tree Search (MCTS) techniques to evaluate the best next move was first proposed in 2006 [3]. MCTS is a randomized algorithm which uses a large number of random action selections followed by heuristically guided play to arrive at an estimate of a potential move's value. The most straightforward MCTS algorithm typically referred to as *vanilla* MCTS, is exceedingly simple:

Beginning wth the root node (current state), perform the following four steps until some resource, typically time, is exhausted:

Selection Select a node to grow, starting at the root

Growth Apply a random new action to selected node

Rollout Random play until game ends

Update Update applicable node win, visit statistics

Vanilla MCTS suffers from two major drawbacks: it is not efficient, and is not consistent. In other words, it is quite slow to converge on a solution, and is not guaranteed to converge on the optimal solution.

Some researchers have partially mitigated these weaknesses with hybrid algorithms, such as UCT-RAVE, which forms an online generalization between related positions, using the average outcome of each movecombining this rapid but biased estimate of a moves value with the slower but unbiased Monte Carlo estimate [4]. However, even UCT-RAVE, like the basic Monte Carlo method it is based on, can overlook an optimal move if the random sampling is poor. An additional criticism of Monte Carlo methods in the arena

[1] See Section 4.4 for a slight clarification

of computer Go is that they tend to play poorly when the ideal sequence of moves, or *tesuji*, depends on the order of stone placement.

ScratchGo uses a hybrid design which incorporates learning from self-play using neural networks and TD(λ) reinforcement learning. This technique has been successfully applied to other games, most notably backgammon [5]. Gerald Tesauro at IBM Research used TD learning to train the 1995 game TD-GAMMON to play backgammon at a level on par with the best human players in the world. This approach was particularly innovative in that heuristics were *bootstrapped* through self-play with no a priori knowledge of game strategy. It was this feature which made TD-learning appealing for ScratchGo, since it was anticipated that time constraints would prevent compiling an elaborate library of hand-crafted heuristics.

3 Approach

3.1 Design

This project was formulated to significantly expand on Tree Search vs. Monte Carlo in Go for Small Boards, a 2-week project developed during the Spring 2012 AI course taught by professors Frank Dellaert and Thad Starner at the Georgia Institute of Technology. In this limited scope endeavor, a classical tree-search algorithm was evaluated against Monte Carlo simulation on a small (6x6 board).

While these techniques as presented in AI: A Modern Approach Chapter 5 [4] produced the expected results in this earlier study (alpha-beta outperformed simple Monte Carlo in this constrained scenario), several planned features were not implemented and are considered a suitable starting point for this project. These features included transposition tables (Zobrist hashing, from [1]), a parallel implementation of Monte Carlo search and a true implementation of UCT- RAVE using weights to be determined experimentally. These combined techniques could potentially allow deeper search into the game tree than any one approach, making this Go agent competitive on small boards (9x9 and smaller).

A secondary objective which was considered but ultimately unsuccessful in the earlier study was to be be tackled using the hybrid approach outlined above: solving Go on the 6x6 board. While this may seem trivial compared to the usual 9x9 small board, the largest solved go board evident in the literature is in fact 5x5 [6] . This is unsurprising as there are over 400 billion legal states for a board of this size [7]. Even so, this feat was only possible because the size of the 5x5 board eliminated many confusing life or death configurations, where it is unclear (even to human players) which player controls the territory until the game is fully played out. Consequently, many search branches could be efficiently pruned for the 5x5 case.

3.2 Implementation: Objectives

Because this MS project was compressed into a single semester, completion depended on methodical progress and accurate appraisal of project milestones. This project was expected to entail approximately 30 hours per week of research, development and testing. In the industry, agile software development proposes that project schedules are best kept using an iterative sequence of rapid design, development and delivery known as *sprints*. Consequently, the following 1-2 week sprints were proposed:

Sprint 1 Apply for a Ranked Robot account on KGS Go Server. Fix major deficiencies in existing game code- base: Zobrist hashing, super-ko rule, etc. Research the UCT-RAVE algorithm, in particular how weights are determined.

Sprint 2 Implement a graphical user interface to facilitate testing. Research methods for accelerating move selection and placement using commodity 3D graphics hardware.

Sprint 3 Implement offline referee functionality to greatly in- crease the speed of testing and training. This will also allow the application to function without a connection to the KGS Go Server on the Internet.

Sprint 4 Implement UCT-RAVE and use ML techniques to train the search parameters.

Sprint 5 Parallelize the Monte Carlo simulation code.

Sprint 6 Tune the tree-search algorithms for speed using [8], attempt to solve 6x6 board.

Sprint 7 Research techniques (e.g. neural networks) to identify solved life-death problems. Integrate these detection algorithms, attempt to solve 6x6 (or larger) board again.

Sprint 8 Implement rudimentary fuseki (openings). The system should be capable of learning new openings by observing opponents rather than relying on a set library of hand-crafted rules.

Sprint 9 Implement rudimentary joseki (corner-play). Again, the system should discover the solutions by evaluating the game state rather than responding to set triggers.

Sprint 10 Evaluate the performance of the agent during online ranked play. Analyze the capability of the system to solve Go-like games and problems on a variety of board sizes.

3.3 Implementation: Results

Sprint 1 The KGS Go Server allows participation by AI agents known as *Robots* as well as human players using web-based and mobile clients. Unranked players are allowed to suggest their own approximate skill level using the 30 kyu (novice) – 1 dan (master) ranking system common to Go tournaments. Ranked players are tracked over time and assigned accurate ratings based on performance against other players, either robot or human. A Ranked Robot account was requested from the KGS administrators at the onset of Sprint 1. Unfortunately, this status was not granted by the completion of Sprint 10. Consequently, no long-term study of ScratchGo's strength could be undertaken.

In addition to creating several unranked accounts to allow ScratchGo to play against other programs on KGS, Sprint 1 consisted of fixing bugs and improving the performance of the existing vanilla MCTS method. Zobrist hashing was implemented to support super-ko (repetition of a whole board pattern) detection. Research into UCT-RAVE revealed that the constant balancing exploration and exploitation in the tree search (see Sprint 8 Outcome) was tuned experimentally to the customary value 0.4.

Finally, several days were allocated to reading *Mathematical Go: Chilling Gets the Last Point* [9] and considering the application of surreal combinatorics to pruning the search space. However, it seems clear that the chilling technique, which simplifies combinatorial analysis of loop-free games, cannot be used in most actual Go games due to repetition of local patterns. Since this requirement would rarely be met until near the end of a game in any case, this interesting approach was abandoned for ScratchGo.

Sprint 2 Implementation of a 2D user interface for ScratchGo was easily accomplished using Java's Swing framework and textures freely available from Wordpress. This feature payed immediate dividends as it made testing against opponents incapable of connecting to KGS, such as GoFree for the Android tablet, much faster. Since dynamic profiling revealed that very little time was consumed by placement of stones and comparison of board states (hashing accomplished this), the notion of using graphics acceleration for move selection was abandoned.

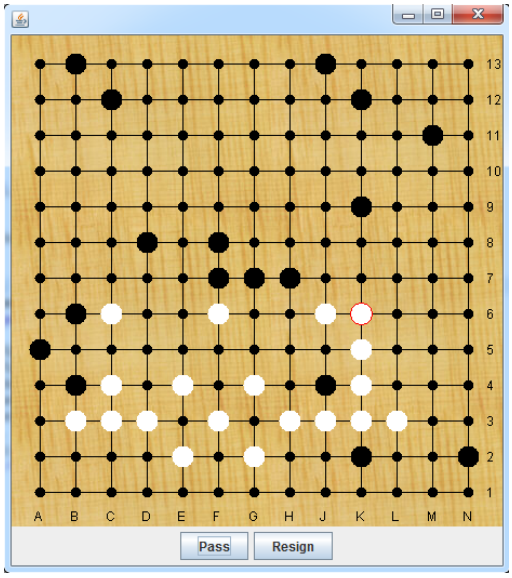

Figure 1. A snapshot of a game in progress against a human opponent using ScratchGo's Java graphical user interface.

Sprint 3 As with the creation of a GUI during the previous sprint, development of an offline game mode greatly increased the speed of testing ScratchGo.

Sprint 4 Sprint 4 produced an implementation of the UCT-RAVE algorithm due to Kocsis and Szepesvari [3]. With extensions and parallelization (see Sprint 5, Sprint 6), this algorithm consitutes the core tree search used by ScratchGo. This method is related an optimal solution to the Multi-Armed Bandit problem as formulated by Auer, Cesa-Bianchi and Fischer [10].

Multi-Armed Bandit Problem Given a multitude of slot machines (one-armed bandits) with varying payoffs, determine the optimum order of play such that the regret is minimized:

$$\rho = T\mu^* - \sum_{t=1}^{T} \hat{r}_t \tag{1}$$

$$\tag{2}$$

The regret ρ is the difference between the actual reward received and the reward which would have been received by playing the slot machine with the best payoff every turn.

UCB1 The UCB1 algorithm due to Auer et al. [10] is optimal in that the regret grows logarithmically in the number of trials:

$$\bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}} \tag{3}$$

Unlike 'vanilla' Monte Carlo, UCB1 balances exploration and exploitation by adding to $\bar{X}_j$, the average reward received from slot machine j, a factor based on the number of times machine j has been tried as well as the total number of trials.

UCT Kocsis and Szepesvari adapted UCB1 to tree search as follows to yield UCT :

$$Q^{\oplus}_{UCT} = Q_{UCT}(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}} \tag{4}$$

The coefficient c influences the tendency of UCT toward exploration (high c) or exploitation (low c). It is generally tuned experimentally to a value between 0.4 and 0.6.

UCT-RAVE Modifying the update portion of UCT to incorporate the All-Moves-as-First (AMAF) heuristic yields UCT-RAVE (Rapid Action Value Estimation).

Sprint 5 One objective of ScratchGo was to leverage modern multi-core architectures. Monte Carlo UCT is difficult to parallelize in a simple way because any new leaf node could impact the next iteration. Ensemble MCTS is an elegant solution. This approach, also known as Root Parallelization (RP), is due to Chaslot, Winand and van der Herik [11]. After a number of UCT searches are run in parallel, the values $Q^{(i)}(s,a)$ for each simluation i are simply averaged:

$$Q_{RP}(s,a) = \frac{\sum_i Q^{(i)}(s,a) \cdot n^{(i)}(s,a)}{\sum_i n^{(i)}(s,a)} \tag{5}$$

While some researchers (see Fern and Lewis [**?**]) claim better results using RP than with a single application of UCT using an equal number of roll-outs, this result could not be replicated in ScratchGo.

Sprint 6 During this Sprint, incremental enhancements were made to the UCT-RAVE algorithm. Unfortunately, experiments with hand-tuning a SMAF (some-moves as first) heuristic did not result in a noticeable improvement in game outcome. Additional approaches researched included α-AMAF, which interpolates between UCT and AMAF estimates of action values [12].

At this point, time constraints prompted reconsideration of the goal of solving 6x6 Gol. According to Tromp [7], a 6x6 go-like game would allow more than 62,567,386,502,084,877 board configurations. Even accounting for reflection and rotation transpositions, and the fact that less than half (but more than a quarter) of these configurations would be legal under normal rules, it would be impractical to store a look–up table of these configurations for use in search optimization. Since this exercise would otherwise contribute little to a functional Go game, this idea was abandoned. It should be noted that the largest Go-like game solved to date is 5x5 (5 orders of magnitude smaller), and this size allows early analysis of many configurations due to the inability for a surrounded group to 'make life' in such a small space.

Sprint 7 Sprint 7 initially focused on adapting an existing neural network library to the task of temporal difference learning. Both the Neuroph [**?**] and Encog [**?**] frameworks were extensively investigated for this use. However, these Java toolkits contained many optimizations that hampered easy modification and some necessary classes were closed to extension. Consequently, an entirely new neural network package was developed for this sprint. Emphasis was on simplicity and ease of modiciation rather than performance, and the typical algorithms used for training a feed-forward neural network were drawn directly from chapters 1–5 of *Neural Smithing* [13].

The initial test case for this work was to learn the XOR function using a simple Multi-Layer Perceptron with a single hidden layer. Hidden layers used the Simoid activation function:

$$S(t) = \frac{1}{1 + e^{-t}} \tag{6}$$

$$\tag{7}$$

The output neuron used the tanh activation function:

$$\tanh(t) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{8}$$

The essential goal of neural network training is to minimize the difference between the learned function and the ideal function using some error metric, such as the normalized root-mean-squared error of the training pattern output values :

$$E_{NRMS} = \sqrt{\frac{\sum_{i=1}^{p} \|o_i - t_i\|^2}{p}} \tag{9}$$

The back–propagation method [13] was used to learn the XOR function by iterative gradient descent, where

$$\nabla E = (\frac{\delta E}{\delta w_1}, \cdots, \frac{\delta E}{\delta w_l}) \tag{10}$$

The rate of change in the weight of each connection, δw_i, is determined by the gradient of the error and the *learning rate* γ:

$$\delta w_i = -\gamma \frac{\nabla E}{\delta w_i} \text{for i } = 1, \ldots, l \tag{11}$$

After the basic feed-forward network and backpropagation training code was successfully tested on the obligatory XOR problem (see Table 1), it was applied to the problem of learning when it is necessary to avoid passing in Go. The network used consisted of 3 input neurons $[Score, Pass_t, Pass_{t-1}]$, a hidden layer, and one output neurons [Value]. After 10,000 games of self-play, the network correctly learned to output a low Value (representing the utility of passing) when losing, and to only rarely pass when winning (see [**?**]).

x	y	XOR (ideal)	XOR (output)
0	0	0	0.028
0	1	1	0.964
1	0	1	0.963
1	1	0	0.028

Table 1. Multi-layer perceptron training results.

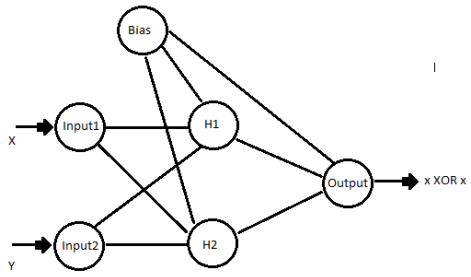

Figure 2. The XOR network consists of 2 input neurons, 2 hidden neurons and 1 ouput. A bias node transmits a constant output signal to both hidden neurons and the output.

Sprint 8 Training of the PassFilter using backpropagation was a relatively simple exercise due to the fact that a bad choice of whether to pass leads immediately to a loss on the following term. In order the develop a more general evaluation function, the credit assignment problem had to be confronted. This is a particularly difficult issue in computer Go because most stones remain on the board for the duration of the game, and can influence win or loss dozens of turns after being played.

The approach implemented in ScratchGo is due to Tesauro[5]. As the name implies, TD(λ) uses a coefficient to control the rate at which learning occurs due to a reward received at a future state. By varying λ in the range [0..1], training takes on the characteristics of backpropagation or Monte Carlo simulation, respectively. This technique has been used to great effect in other games. It can be applied to reinforcement learning in neural networks, as was done in Tesauro's world-class backgammon program TD-GAMMON.

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k \tag{12}$$

This is known as the *forward view* of temporal difference learning. An alternative formulation to (12) is the *backward view* in which an *eligibility trace* is maintained until a reward is eventually received. As the name implies, the eligibility trace is defined relative to the eligibility trace of the preceding state. Unlike the backward view, it allows some learning to occur online, i.e. as new game states are observed:

$$\overrightarrow{e}_t = \lambda \overrightarrow{e}_{t-1} + \nabla_{\overrightarrow{W}} f(\overrightarrow{W}, s_t) \quad (13)$$

As with Sprint 7, a simple test case was needed to evaluate the quality of the TD(λ) trainer. Rather than begin with go, a test case was created for a simple 2 player game with a known optimal policy: Tic-Tac-Toe. As expected, the MLP was able to learn a seemingly optimal policy through 50,000 games of self-play beginning with randomized connection weights.

Game	Initial	Trained–Trained	Trained–Random
1	O Wins	Tie	Tie
2	O Wins	Tie	Win
3	X Wins	Tie	Win
4	O Wins	Tie	Win
5	Tie	Tie	Win
6	Tie	Tie	Win
7	X Wins	Tie	Win
8	X Wins	Tie	Win
9	X Wins	Tie	Win
10	O Wins	Tie	Win

Table 2. Training a Tic-Tac-Toe network.

This update function was used in a TD(λ) trainer to supplement the batch-mode back-propagation trainer developed during Sprint 8. As with the previous neural network trainer, learning was performed using self-play with an ϵ-greedy trainer, that is, random action selection with probability ϵ and action selection using the neural network policy with probability $(1-\epsilon)$. Because the goal was to develop a network specialized for Fuseki (openings), the game was played to completion after ply 10 and the actual game result (1.0 for a win by black, 0.0 for a win by white) was assigned to Y_k.

Sprint 9

Sprint 10

4 Evaluation

4.1 Deliverables

During the course of this 4-month project, 8 of the 10 sprints were fully or partially completed. Table 3 summarizes the success of ScratchGo in meeting each objective.

In all, roughly 78% of planned features were implemented. However, the latest version of ScratchGo also includes several useful features which were not initially planned. For example, significant time was spent implmemented an SGF game file parser to record games

Deliverable	Features Completed
Sprint 1	100%
Sprint 2	100%
Sprint 3	100%
Sprint 4	75%
Sprint 5	100%
Sprint 6	25%
Sprint 7	100%
Sprint 8	100%
Sprint 9	0%
Sprint 10	n/a%
Total	78%

Table 3. Summary of ScratchGo features delivered.

for debugging. This module also allows import of game databases for batch learning (used for the PassFilter). This was used to learn the PassFilter, although the TD(λ) trainer uses online learning to learn the FusekiFilter network. Figure 3 shows an actual tournament game parsed from SGF data by ScratchGo and exported to LaTeXformat.

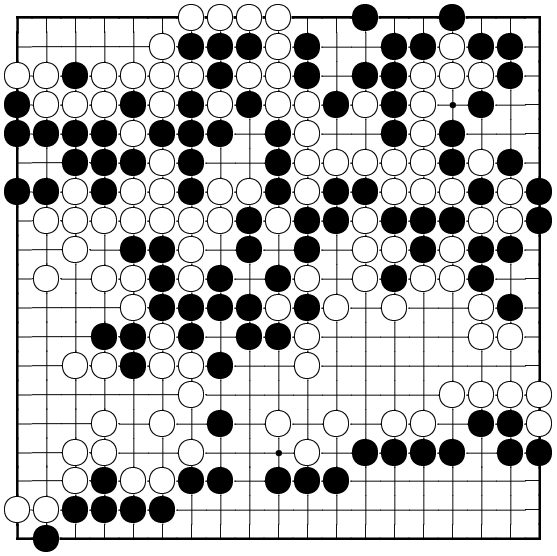

Figure 3. A snapshot of a game in progress.

Furthermore, as noted in Section 3.3, the necessity of writing all new neural network modules caused sprint 8 to consume much of the schedule allocated for Sprint 9 as well. However, the end result was applied to learning when to pass from self-play. ScratchGo is thus capable in principal of learning 'killer move' heuristics from self-play. Because ScratchGo 'boot-straps' in this manner, the platform could easily be adapted to other games non-game search tasks.

Finally, Sprint 10 was assigned a result of "n/a" be-

cause the author was unable to obtain a "ranked robot" account on the KGS Go Server. The site administrators had not yet responded to requests for this account by the time ScratchGo was completed.

4.2 Search Algorithm Evaluation

The core tree search functionality was implemented beginning with bug fixes to existing code (Policies 1–3) and proceeding through iterative enhancements to Monte Carlo Tree Search (Policies 4–6), resulting in a hybrid solution which combined parallel UCT-RAVE with rollout guided by policies trained using TD(λ) reinforcement learning. Table 4 lists the results of a round-robin tournament in which each Policy was pitted itself and every other algorithm in a 10-game match (5 as Black and 5 as White).

Policy	Wins vs.						
(As Black) ↓	1	2	3	4	5	6	7
1) Random	40%	100%	0%	0%	%	0%	0%
2) Alpha-Beta	20%	100%	0%	0%	0%	0%	0%
3) UCT	100%	80%	20%	0%	0%	0%	0%
4) UCT-RAVE	100%	60%	80%	80%	40%	100%	100%
5) UCT-SMAF	100%	80%	80%	80%	40%	80%	100%
6) RootParallel	100%	100%	100%	100%	100%	60%	100%
7) RP+Nnet	100%	60%	80%	0%	0%	20%	40%

Table 4. Results of round-robin tournament

4.3 ScratchGo vs. GoFree

GoFree is a popular Android version of Go by commercial developer AI Factory. It was picked for comparison to ScratchGo because of the low cost and relative weakness as compared to GnuGo. As shown in Table 5, it was easily beaten by ScratchGo on the easiest difficulty settings (1-2 out of 10) but quickly began to encounter the same scoring and passing issues which proved apparent against GnuGo, as discussed in Section 4.4. It should also be noted that only a handful of games were played at each difficulty level since these tests could not be automated - ScratchGo's moves were played manually against GoFree running on an Android tablet.

Level	Wins vs. GoFree (Android)
1	100%
2	100%
3	50%
4+	??%

Table 5. Summary of performance vs. GoFree.

4.4 ScratchGo vs. GnuGo

Despite the solid performance of the Root-Parallel UCT-RAVE implementation against simpler tree search algorithms and novice opponents, ScratchGo was completely unable to beat GnuGo on any difficulty level (GnuGo has many settings which impact performance, the most obvious of which governs search depth). This is relatively unsurprising since GnuGo incorporates a large large number of heuristics including fuseki, joseki, life-and-death problems, false eye shapes, snap-backs and so on. As the performance of ScratchGo improved, it became apparent that a major issue lies in the way ScratchGo scores the terminal game states. While GnuGo considers 'dead' stones as territory owned by the opponent, ScratchGo has no concept of dead shapes as such, and thus these stones count as points unless actually captured by the other player. This permits the possibility that ScratchGo will mistakenly score an outcome as a win when in fact it is a loss due to dead stones in enemy territory, as shown in Figure 4.

Figure 4. GnuGo (Black) wins by 5.5, but ScratchGo calculates a 14.5-point win by White.

Future versions of ScratchGo will incorporate Benson's algorithm [14] for determining whether any stones should be removed from the board prior to scoring a terminal states. This correction will be aided by additional changes to board state data structures as detailed in Section 5.

4.5 ScratchGo vs. Humans

Due to the lack of access to a ranked robot account on the KGS go server, it it impossible to objectively assess the performance of ScratchGo against human players. However, humans often entered the KGS go server game room when ScratchGo was run against various computer agents and occasionally managed to connect to the game before the intended opponent could be launched. Subjectively, ScratchGo often fared well again human novices. For example, the following outcome resulted when a human player connected to ScratchGo running in RootParallel configuration (see Figure 5).

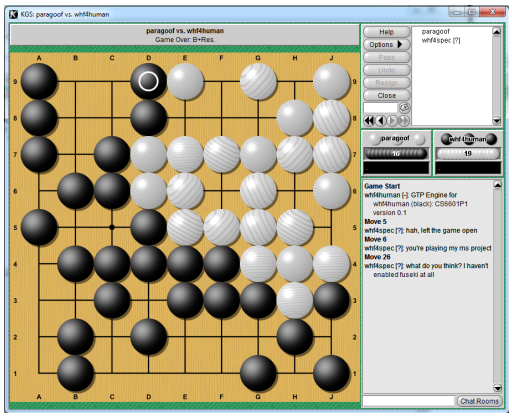

Figure 5. The human player (White) resigned when the capture of a large group of white stones at the bottom effectively ended the game.

5 Discussion

The latest iteration of ScratchGo a persistent online agent capable of playing Computer Go against human and AI opponents on the KGS Go Server. As the name implies, it is capable of learning to play go 'from scratch', i.e. by discovering some heuristics through self play and learning others through experience.

However, it is clear that substantial work remains for ScratchGo in order to reach parity with established computer Go agents. This is hardly surprising, since as Martin Müller noted in 2002, many competitive Go programs are the product of 5-10 person-years of effort and include dozens of modules governing heuristics for strong play [15].

Future Work

Work in at least three areas would help to overcome the notable deficiences of ScratchGo versus experienced opponents.

Game state representation As noted in Section 4, rollout efficiency is greatly hampered by simplistic recursive evaluation of group liberties. Dynamic profiling using the EJ-Technologies JProfiler tool revealed that 40% of all time consumed by the action evaluation function was used for recursive liberty calculation. Replacing the current board representation and liberty calculation method with a more efficient model which directly represents groups of stones and keeps a running total of liberties as new stones are added would greatly increase the effective tree search depth. This approach is used by GnuGo [16] and numerous other programs based on GnuGo.

Neural Network Encoding While the PassFilter successfully uses backpropagation to learn when passing is an immediate killer (or losing) move, the relatively poor performance of the JosekiFilter indicates that a strong evaluation function was not learned using a raw board encoding. As with Tic-Tac-Toe, it may be the case that a better choice of input features would lead to better convergence using a less complex network.

Heuristics As shown by online play against GnuGo, ScratchGo suffers greatly from a lack of expert knowledge in determining moves which build strong shapes. Human players can accomplish similarly strong play by drawing on a repertoire of strong as weak shapes such as the bamboo joint (Fig. 6) and the empty triangle (Fig. 7). It would be straightforward to add an additional NeuralNetworkFilter to ScratchGo with evaluates the immediate neighboard of a potential action and generates an output corresponding to the strength of the shape. A library such shapres suitable for verifying the learned evaluation function can be found in any introductory Go text, such as *So You Wan to Play Go?* by Johnathan Hop [17].

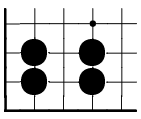

Figure 6. The bamboo joint is easily defended.

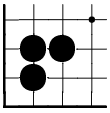

Figure 7. The empty triangle is inefficient.

Acknowledgements I would like to acknowledge the assistance of my project advisor, Dr. Frank Dellaert, for his encouragement and constructive criticism over the course of this project.

References

[1] A. Zobrist. A new hashing method with application for game playing. *Computer Science Department Technical Report*, 88, 1970.

[2] J. Simmons. Notable quotes about go. http://www.usgo.org/notable-quotes-about-go, 2012. [Online; accessed 6-December-2012].

[3] L. Kocsis and C. Szepesvari. A new hashing method with application for game playing. *Computer and Automation Research Institute of the Hungarian Academy of Sciences*, 2006.

[4] S. Russel and P. Norvig. *Artificial Intelligence; A Modern Approach*. Prentice Hall, third edition, 2010.

[5] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38:58–68, 1995.

[6] Erik C. D. van der Werf, H. Jaap Van Den Herik, and Jos W. H. M. Uiterwijk. Solving go on small boards. *International Computer Games Association Journal*, 26:10–7, 2003.

[7] Tromp and Farnebäck. Counting legal positions in go. http://homepages.cwi.nl/˜tromp/go/gostate.ps/, 2009. [Online; accessed 30-January-2012].

[8] M. Littman. A new way to search game trees. *Communications of the ACM*, 2012.

[9] E. Berlekamp and D. Wolfe. *Mathematical Go; Chilling Gets the Last Point*. A K Peters Ltd., 1994.

[10] Cesa-Bianchi N. Freund Y. Auer, P. and R. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Scientific Computing*, 2002.

[11] Winands-M. Chaslot, G. and J. van den Herik. Parallel monte-carlo tree search. *Sixth International Conference on Computers and Games*, 2008.

[12] D. Helmbold and A. Parker-Wood. All–moves–as–first heuristics in monte–carlo go. http://users.soe.ucsc.edu/ dph/mypubs/ AMAF-paperWithRef.pdf, 2012. [Online; accessed 6-December-2012].

[13] R. Reed and R. Marks. *Neural Smithing*. The MIT Press, 1999.

[14] D. Benson. Life in the game of go. *Information Sciences*, 10:17–29, 1976.

[15] M. Müller. Computer go. *Artificial Intelligence*, 134:145–179, 2002.

[16] D. Bump. Gnu go documentation. http://homepages.cwi.nl/˜tromp/go/gostate.ps/, 2009. [Online; accessed 30-January-2012].

[17] J. Hop. *So You Want to Play Go?*, volume 1. Sunday Go Publications, 2008.