1 Why Computer Go Is Hard

In contrast to chess programs, the best go programs are far weaker than typical club players. While more effort has gone into computer chess, enough effort has gone into programming go that it is safe to say that computer go is a considerably harder problem. This is the same as saying that people are better at go than at chess, compared to machines.

The standard computer chess method, a brute force alpha-beta pruned search to an arbitrary depth (possibly with quiescence search extensions), has proved very effective not only for chess but also for checkers, othello, and many other games. It has been said that such methods cannot work for go because of the high branching factor. Assuming a branching factor of 250 for go versus 36 for chess, a seven-ply search with good move ordering should have about 900 times as many nodes. The combinatorics have led some researchers to give up on search and to try rule-based systems instead [Reitman & Wilcox 80]. But the history of computer chess should teach us not to discard brute force search too quickly. The high branching factor makes go more difficult for people as well; by itself it is not a reason to expect computer go to be hard. However it does mean that techniques to reduce search may be especially powerful.

Experiments with brute force search in go are impeded by the fact that no good fast evaluation functions have been found. One's first idea might be to evaluate as if all the stones are equally strong, and hope that the search will see far enough that any dead stones will be captured. This is roughly what is done for chess; the evaluation function doesn't worry about whether a piece is trapped. This can lead to horizon effects in chess programs, but deeper search is the usual remedy. In chess a trapped piece usually will be captured within a few moves (with best play), so search resolves the problem nicely. But in go a dead group may require dozens of moves to actually capture, and moreover those moves are often poor and should be left to the very end of the game. So a go program's search would have to be very deep indeed to be able to get away with assuming that all the stones on the board are alive. A better idea is for the evaluation function to analyze the strength of each group using search and other methods, and then estimate territory as if the dead stones were not there.

Brute force search may be a poor approximation to the look-ahead procedure used by human chess players, but it is a far worse approximation to the analysis used by go players. Even a novice can "read out" a forty move ladder, or imagine filling liberties in a *semeai* (capturing race) that takes twenty moves to resolve. The player knows that he needn't consider alternative moves in the search because these situations have a well-understood structure. Deciding whether a group is alive or dead can involve complicated reasoning processes. In Figure 1 the white group on the lower right is mostly surrounded and has no real eyes, but one must also read out some breaking-out sequences to see if it might live by capturing an attacker, making eyes, and/or escaping in some direction. Search is needed, but full-width search is much too slow. The real reason that computer go is hard is that there are many powerful reasoning methods people use to analyze go positions.

2 Golem's Evaluation Function

With my go program Golem I have tried to create simple algorithms to play the game based on fundamental principles of the game. Golem has no hand-coded knowledge of openings, connection patterns, or anything of that sort. Unlike most go programs it does not have rules that say "if you see this stone pattern play here." Instead it has an evaluation function which estimates territory, and uses a small top-level search to decide which move gains the most territory. Golem does use neural networks which have absorbed some knowledge of stone patterns by being shown professional go games, but I have tried to minimize its dependence on such knowledge in order to be sure that it understands (in some sense) the basic tactical concepts of go.

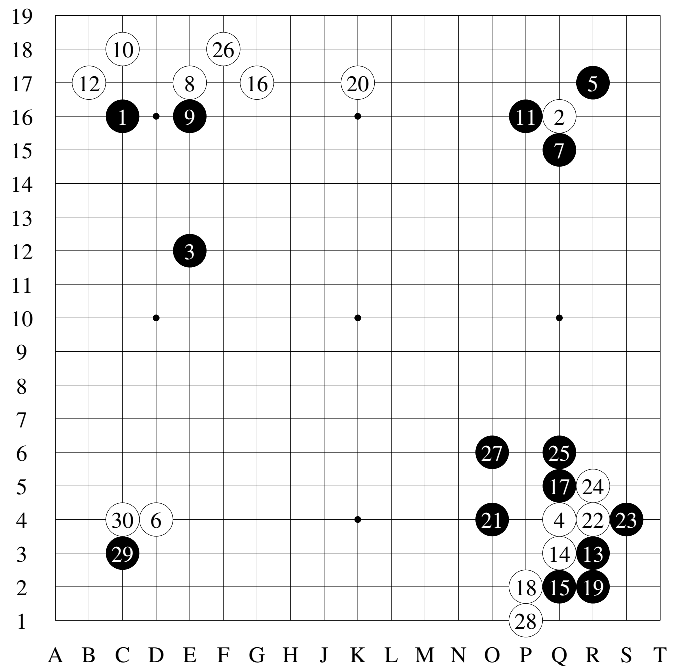

Figure 1: First 30 moves of a test game between Golem (Black) and the 1991 North American computer go champion The Many Faces of Go (version 3, level 16), written by David Fotland. Golem thinks the white group in the lower right is 50% alive, and that the black group there is 92% alive. But White has excellent resources in the moves S5 and O3, while the Black group is confined and unlikely to prosper, so this evaluation is optimistic. Golem plays the very unorthodox move Black 7 because it thinks this probably kills the White 2 stone. Most of the examples throughout this paper are taken from this game.

2.1 The Single-Target Search

The first thing Golem's evaluation function does is to examine each string of stones (the unit of capture) with three or fewer liberties to see if it can be captured in a ladder or easy *geta* (net). The search used for this purpose is highly selective; only two or three possibilities are typically considered at each node. By running this *single-target search* on each string of stones with each side to move, Golem can identify some strings as tactically dead or threatened. Golem uses the following simple rules to generate candidate moves:

1. If the defender has three liberties and is on move, he has escaped.
2. The attacker may try filling a liberty of the target string.
3. If the defender has only two liberties the attacker may make a move adjacent to both liberties.
4. If the defender has two liberties and an attacking string is in atari, the attacker may consider saving her string if it can escape from the ladder. (The program considers any sequence of ataris to be a ladder.)
5. If the defender is in atari he may only try moves which increase his liberties.
6. If the defender has two liberties he may atari an attacker, play adjacent to a liberty, or pass.
7. Beyond the specified depth limit the attacker is only permitted to fill in liberties, not to try the fancier moves allowed by rules 3 and 4.

Golem uses a neural network (described later) to forward-prune this search when the rules generate too many moves to consider. As an example consider the string of four black stones at B16 in Figure 2. White's possible moves are F16, D17, and C17, to be tried in that order. Black's possible responses to F16 are D17, G16, B14, C17, D18, A17, A15, and PASS, but it only considers the first three, and when these fail it assumes the other moves will fail also. The "patience" parameter determines how far the search will explore a sequence of play before assuming that the attack fails. In this problem Black can thrash about for nine moves before finally being put in atari.

White also threatens to capture the group in the lower right corner of Figure 2, and the search is able to see this because of the sequence White S2, Black T2, White S1. It rejects the more natural attack White R1, Black T2, White S1 because the S1 move isn't generated. (A new rule to generate "approach moves" would be helpful in this instance. If this group had outside liberties it would have to be treated as a life and death problem, but Golem does not currently have a life and death search routine.)

2.2 Tactical Status of Unoccupied Points

After determining the tactical status of each string of stones as above, Golem begins the process of estimating the strength of each group. First it finds the "protected liberties," i.e., the unoccupied points which one side or the other may not play on without being quickly captured. It just uses the ladder search routine for this, as the full single-target search is too time consuming. The strength of a group often depends on the weakness of its nearby enemies, so Golem initially guesses at the strength of each group based on its number of liberties, number of potential liberties (if that side plays first), and the tactical status of its neighbors. Then it makes a first pass through the territory-estimating procedure (described below) in order to classify some points as being under probable Black or White control. Each unoccupied point is put into one of five classes:

- A protected black liberty (White cannot play here without being captured).
- A protected white liberty.

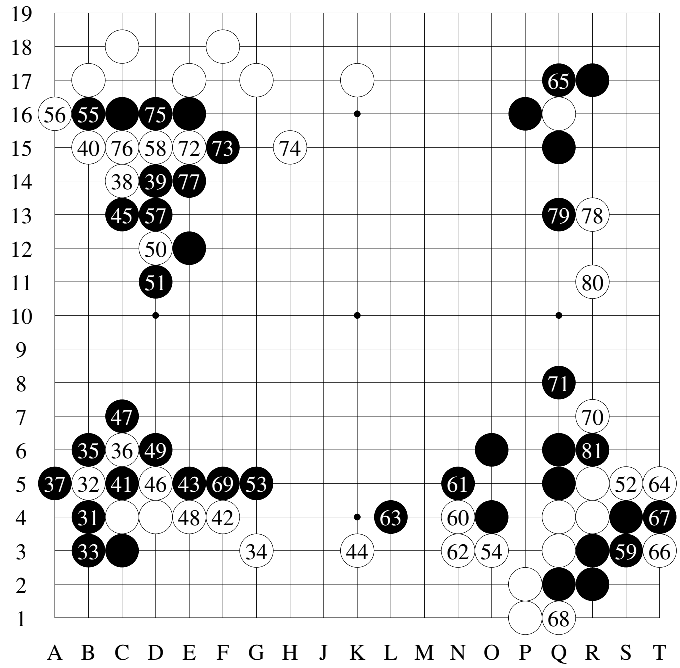

Figure 2: Moves 31-81. In the game, Golem's single-target search was too shallow to see that White threatened to capture four Black stones by cutting at F16.

- An "easy play" for Black. A stone here would clearly be connected to an apparently live black group (by a direct or diagonal connection). Also includes points that look like they are Black's territory.
- An easy play for White.
- A neutral point that is an easy play for both sides.

2.3 Eyes and Groups

With this information in hand, Golem assigns a fractional eye value for each side to each point, doing its best to recognize false eyes, possible eye space, half eyes, and true eyes. It looks only at the four neighbors and four diagonals of a point to calculate its eye value. Liberties of weak enemy stones can never become eyes, but they are given non-zero eye value to represent the chance that the group might live by attacking an enemy. Spaces in the open are given some eye value because if a group can run into the open it can usually live. Another routine makes sure that no two adjacent spaces or 2×2 block of spaces have total eye value greater than 1.0, and that no "star-5" pattern with the center empty has total eye value greater than 1.5. Some paranoia is introduced to reduce the effects of errors: Golem has more doubts about its own eyes than about its opponent's eyes.

Golem's concept of a group is quite primitive. Any two strings of the same color that can be solidly connected by two different moves, or by one protected liberty, are considered to be in the same group. It would certainly help to recognize some more subtle tactical connections between strings, and to recognize non-transitivity of connections in cases where the spaces involved overlap. Golem counts the number of *effective liberties* of each group. This is essentially the number of spaces that are liberties of the component strings minus the number of moves required to connect the strings.

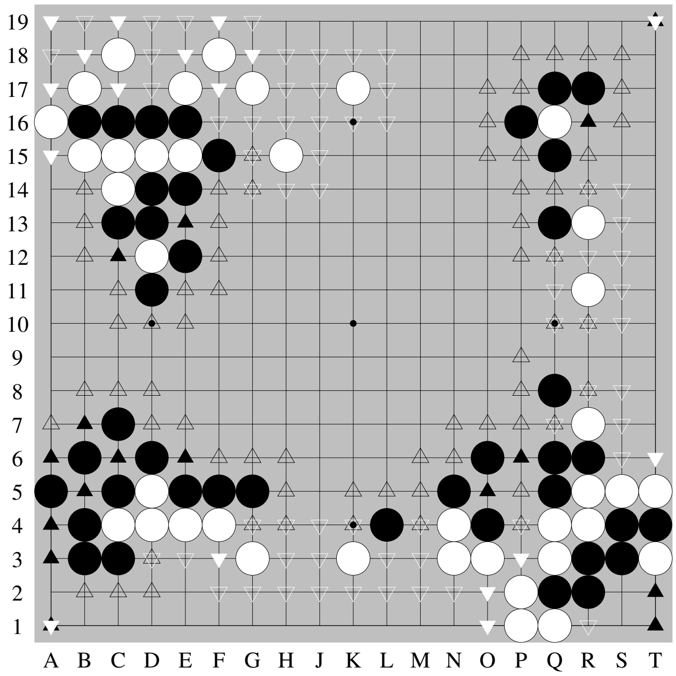

Figure 3: The spaces Golem identifies as protected liberties are marked with solid triangles. For example a white stone at E6 could be caught in a ladder, so this is a protected liberty for Black. The non-filled triangles indicate "easy plays," plays that Golem knows would be easily connected to existing strong groups. This information is important for identifying connections and eyes.

Golem then adds up the fractional eyes accessible to each group of stones, using a single-source shortest path algorithm to identify the accessible points, with a cost function based on the above classification of spaces. A group's *eyespace* value is a non-linear function of its total eye potential. The strength of each tactically viable string in a group is set to this eyespace value. Tactically unstable strings, i.e., those that the single-target search thinks will be captured or saved depending on who plays first, have a maximum strength of either 20% or 70%, depending on who is on move. In the final phase of strength estimation Golem increases the strength of any weak groups that look like they might win a *semeai* against a weak adjacent enemy group, basing this mainly on the effective liberty counts.

2.4 Counting Territory

Golem now uses a relaxation method to identify black and white territory. The method begins by removing any stones weaker than a certain threshold strength (adding a bias towards Black or White, see below), and then initializing values 0.0 for white stones, 1.0 for black stones, and 0.5 for empty points. It then iterates applying a transition function at each empty point. The transition function looks at the four neighbors of a point (and sometimes diagonals when there is the point would be a strong cutting play), and finds the minimum, maximum, and average of the values at those points. The new value becomes a weighted average of those three terms with 0.5. After twenty iterations over the whole board it counts up the black and white territory to produce an estimated score. The evaluation function calls this relaxation procedure with several different bias values, effectively ranging from "White will save all his weak stones and kill all of Black's weak stones" to the other way around.

This method of counting territory in the presence of weak groups is new. The usual method, such as that

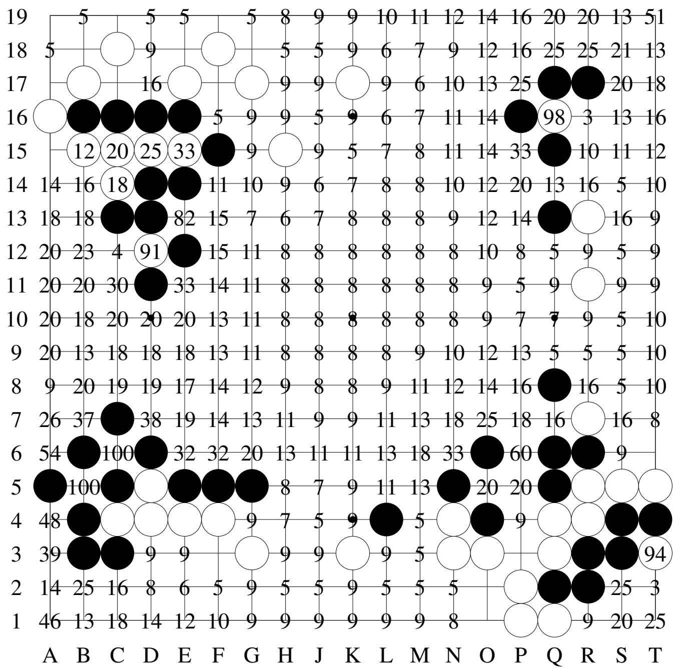

Figure 4: The numbers show Black's eye values as percentages of an eye. By summing the eye values near each group, Golem estimates that the black L4 stone is 77% alive, the Q5 group of six stones is 98% alive, and the R3 group would be 53% alive based on eyespace but is only 20% alive because it is tactical endangered.

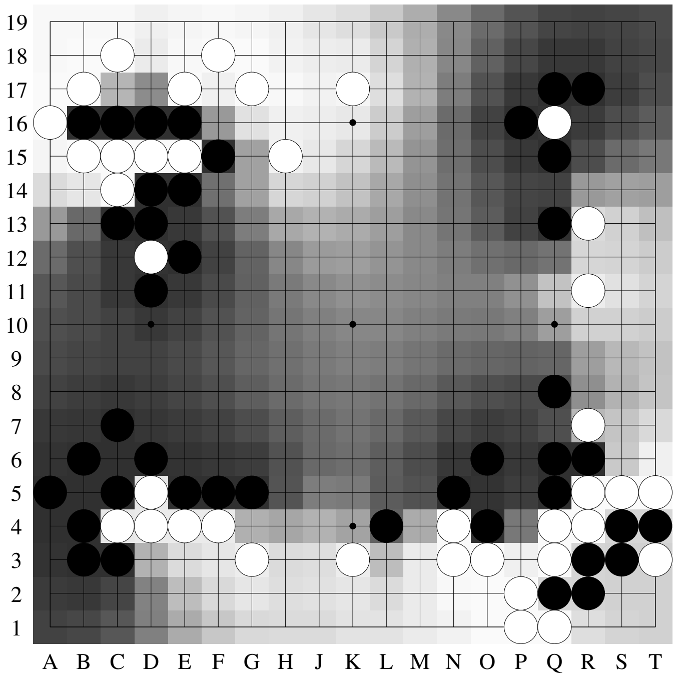

Figure 5: The shading indicates Golem's evaluation of who is likely to own each point of territory, averaged the 11 bias values. Golem considers itself to be 28.5 points ahead.

described by Fotland [Fotland 91] or Kierulf [Kierulf 90], is to radiate "influence" from the strong stones and perhaps negative influence from the dead stones, this influence being blocked or attenuated by other stones. One advantage to Golem's method is that a space adjacent to one strong white stone and three strong black stones will be counted as exactly neutral, and this generalizes to some extent to larger areas. It also deals correctly with the case of a dead black group with tactical dead white stones inside it. Golem effectively removes both the black and the white stones to count the territory; some methods would incorrectly credit the two dead white stones to Black.

3 Performance

Golem plays simply by doing a one-ply search, and usually prunes even that small search using the neural net values. The evaluation function is too slow to do more than this under typical tournament conditions. The current version of Golem has not played in any tournaments, so it is hard to know its playing strength. 14 kyu seems like a reasonable estimate. Golem plays aggressive, unorthodox moves in the opening, usually fights tenaciously in the middle game, and plays very poorly in the endgame. Some of the mistakes it typically makes:

1. It fails to prevent a double atari. Its single-target search is generally blind to double threats.
2. It often plays on the second line. By playing on the second line it can often create an eye and protected liberties, and does not realize how vulnerable it is to being surrounded.
3. It doesn't analyze connections between strings very well. If two strings are near each other it usually figures that they each can depend somewhat on the other's eyespace for life. This vague assessment leads to some disastrous mistakes.
4. It has a knack for making false eyes for itself which are just subtle enough that it can't recognize them as being false.
5. It plays foolish ataris, thinking that any group in atari is surely not entirely safe.
6. It underestimates the ability of an invading group to lean against its scattered enemies and escape. In general it pays more attention to taking away individual eyes than to surrounding an enemy with strong barriers.
7. It lacks any sense of *sente* (keeping the initiative).

But it usually makes reasonable moves, and appears to be approximately the same strength as the better commercially-available go programs. Its greed for territory and reckless disregard for safety make for an exciting style of play, and most of its games turn out lopsided one way or the other.

4 Neural Networks

Golem uses a neural network to provide a quick estimate of how good a move is, without evaluating group strength or territory. It currently uses two versions of this. The simpler one is designed for speed, and is used in the move ordering of the single-target tactical search. The features provided as inputs to this neural net are:

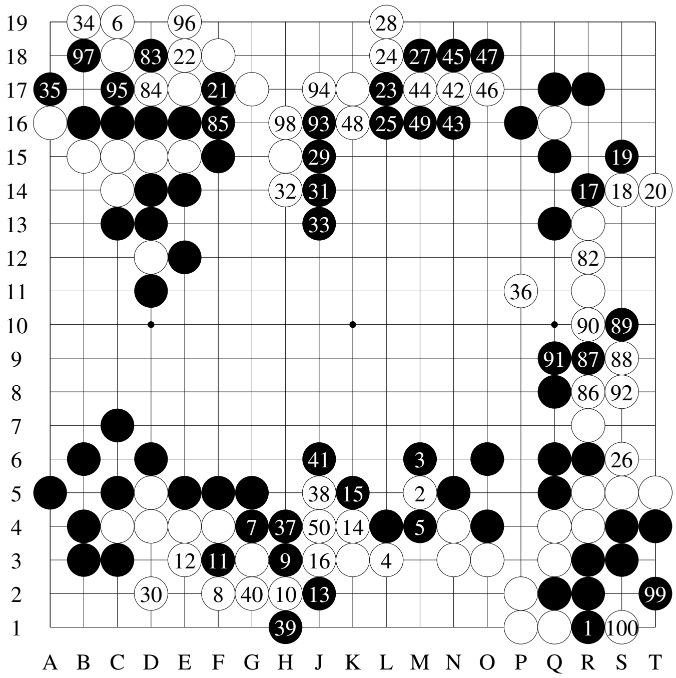

Figure 6: Moves 81-150, continuing the game from Figures 1 and 2. Golem estimates that Black is about 100 points ahead. It overestimates the value of Black's center *moyo*.

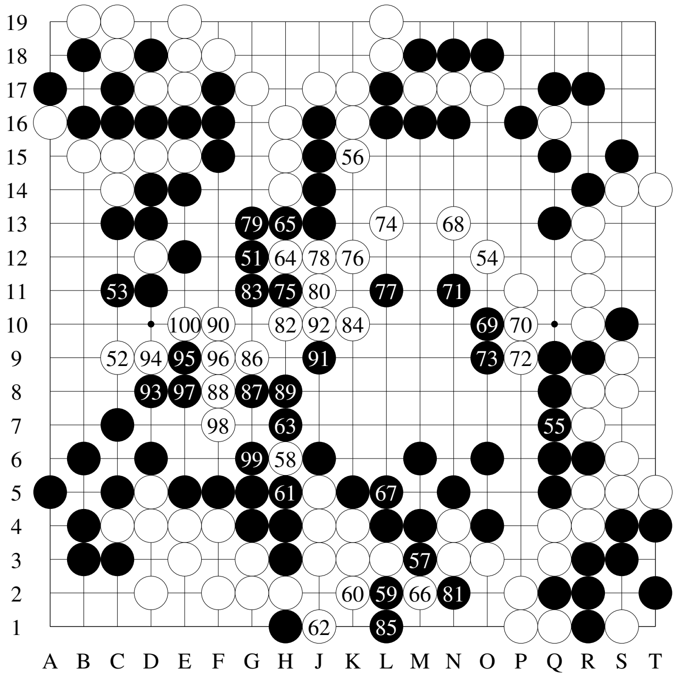

Figure 7: Moves 151-200. Part of the reason Golem failed to stop the White incursion in the center is that the version of Golem that played this game didn't understand diagonal connections, so it viewed the "tail" of the White group as vulnerable. This has since been fixed.

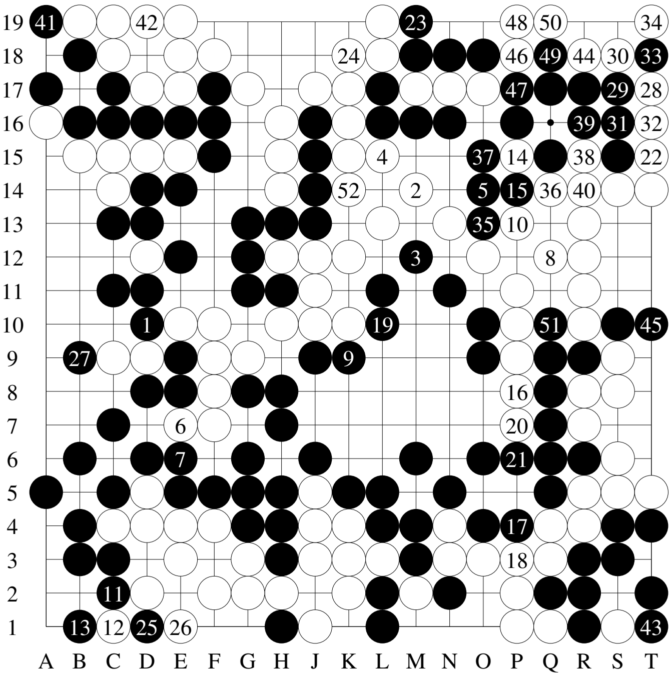

Figure 8: Moves 201-252. Golem's endgame play is quite atrocious, but it manages to win this game by 26 points. Moves after 252 omitted.

- For each of eight points surrounding the proposed move, what color is on that point. Possible values are side-to-move's stone, opponent's stone, empty, or off-the-board. There is a canonical order to the eight points so that, for example, the fourth input unit is associated with the point just "below" the move, i.e., in the direction of the nearest edge.
- How many liberties would the move have?
- If the opponent played there, how many liberties would she have?
- For each of the four neighbors, if it's a stone, how many liberties does it have?

This network has 29 input units, 10 hidden units, and 1 output unit, completely connected. It provides a value between -1.0 and 1.0 for any given move, and the hope was that it would give higher values for better moves. Obviously it doesn't have enough information to know which moves attack weak groups or stake out large territories, but it can and does give high values for moves that make good local shape. It was trained using 47,091 positions taken from games between professional go players. For each position it considered the professional's move and one random other move. The goal was to have it evaluate the professional's move at least 0.2 higher than the random move. For each case where it failed to do this, back propagation [Plaut, Nowlan & Hinton 86] was used to push the values in the desired direction. One reason for using this type of comparison training, rather than the traditional method in which the desired value for positive examples (good moves) is 1.0 and the desired value for negative examples -1.0, is that a move may be good in the endgame, but too "small" to play earlier. This way it is only comparing the professional's move to other moves available in the same board position. This method is similar to the comparison training method described by Tesauro [Tesauro 89].

After two thousand training epochs it was able to rate the professional move higher in 86.6% of the cases. In 10,993 test positions that weren't included in the training set, it rated the professional's move

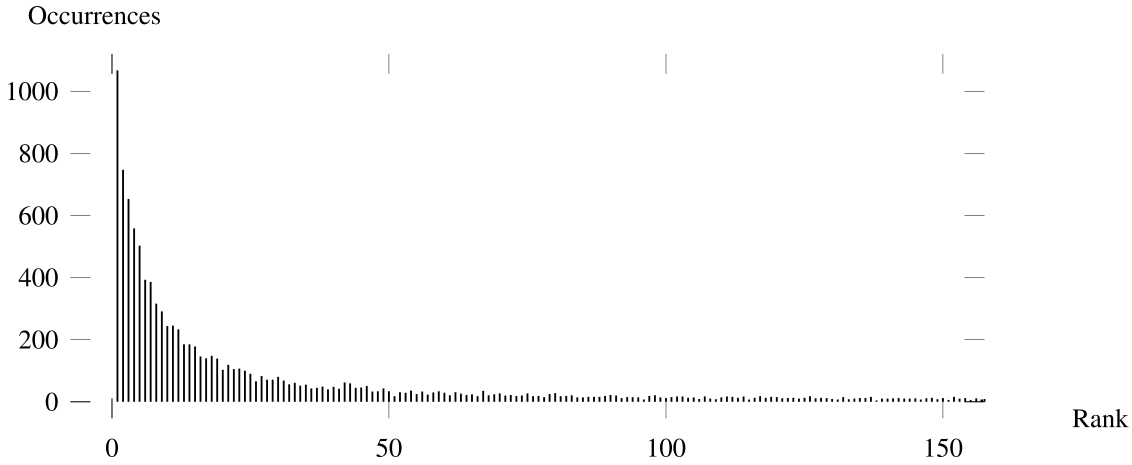

Figure 9: Histogram of the rank of the professional move among the values given to each legal move by the smaller neural net, over 10,993 positions.

higher in 87.0% of the cases. In other words when it rates each of the legal moves in a random position from a professional game, the correct move (or at least the professional's move) in that position is rated higher than 87% of the other legal moves, on average. By comparison, when Golem evaluates the territorial gain of each legal move in such a position, it on average considers the correct move to be only better than 74% of the other moves. However the move the neural net rates as best tends not to be as good as the move the evaluation function rates as best. In games between the two strategies, the evaluation function consistently wins.

Some larger networks with richer sets of features have been tried for use at the top level of the program, to prune the set of moves it has to consider. The one currently used has the following features:

- The value of the simple net above.
- The tactical status of each of the points in a 3×3 region about the candidate move, i.e., whether it is a protected liberty, a dead stone, etc. Note that this includes some tactical information about the candidate move.
- Has the move been identified as tactically saving or killing any stones?
- For each of eight directions, the color of the nearest stone, its distance, and its strength.
- Distance to the nearest edge.
- Distance to the second nearest edge.
- Is there a *ko*? (I.e., is there a move anywhere which is forbidden by the *ko* rule?)

These features are available once Golem evaluates the root position. The network using these features, trained the same way as the one above, prefers professional moves over random moves 90% of the time. Other features that have been tried include the color of larger neighborhoods of points, the size of nearby groups, and the overall density of stones on the board. All of these sets of features lead to about the same level of performance.

Golem's neural networks are extremely fond of contact plays, and tend to avoid starting new groups in empty regions. Part of the reason is that they have more information moves that are near other stones. But

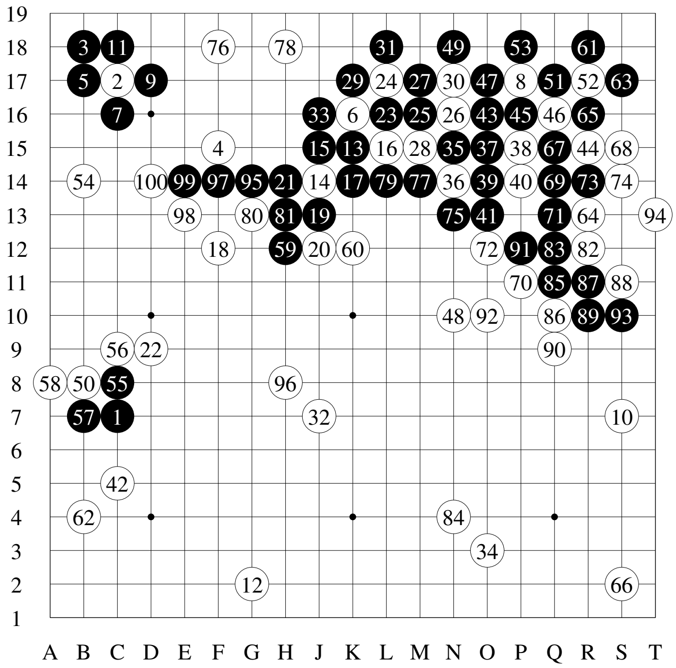

Figure 10: The first 100 moves of a game between the smaller neural net (Black) and a complete one-ply search (White). White won the game by 76 points.

the main reason is that in a typical position there are many points in large open regions that are all roughly equivalent. When the feature set is restricted to facts about the immediate 3×3 neighborhood these points look identical, but even when more features are included, there is not very much to distinguish one point from another in empty regions. The odds of any one such point being chosen by the professional go player are low, so these points show up much more often as negative examples. It is not clear what a satisfactory solution to this would be.

One feature whose inclusion actually seems to be harmful is the distance to the last move played. In theory this data is irrelevant; when considering what move to make in a given board position, it should not matter what order of moves produced that position. But in practice this is a very strong feature, because good players usually play moves that need to be answered locally. When this feature is added to the neural net it tends to override all the other features, and the highest rated move is almost always immediately adjacent to the last move played.

Another way to incorporate the neural net's knowledge of stone patterns into the program as a whole is simply to add bonuses to the values of moves that it likes. So if the one-ply search reports that the move E5 gains 15 points of territory, for example, one might add a few more points to that if the neural net rated E5 the best move. Such a combination actually plays better than either the straight one-ply search or the neural net by itself. But for developing and debugging the territorial evaluation function it's better not to use such bonuses, because they tend to gloss over deficiencies in Golem's fundamental understanding of the game. When the evaluation function (perhaps aided by search) deals properly with tactical properties of strong and weak stone formations, Golem will make "good shape" without recourse to neural networks or any other method of blindly imitating human play.

5 Future Work

Golem's evaluation function will have to be sped up before any elaborate search can use it. One way to do this is to have it cache some of its tactical search results. For example if it notes the points examined when reading out a ladder, it can avoid recomputation in a similar position if none of those points has changed. The more complicated a tactical search routine is, the harder it is to identify the conditions under which the result will remain valid. Golem's simple *geta* search may already be too complex to be amenable to this type of caching.

Golem's endgame play seems like the best place to focus further attention. In the endgame there is little room for vague heuristics; people can often calculate precisely how many points each play is worth. Some local search is necessary. A strong go program will need to be able to read out each possible territorial incursion it and its opponent might make. To do this will require a large body of knowledge to determine which moves are relevant follow-ups to a particular threat of incursion. Each border dispute on the go board is mostly independent from the other border disputes, so it doesn't make sense to have a full-board search that jumps around considering responses on one side of the board to moves made on the other side. We want it just to look at the responses that are relevant to determining the value of the first move in the sequence. It will also need to know when to stop a line of search, either because the sequence of play would naturally end there (switching elsewhere) or because further search would be too costly.

If we had such a search engine, there are at least two ways to conceive of using it. One is as a supplement to the top-level one-ply search. Thus, instead of just considering a single move and evaluating, the program would consider a sequence of moves and evaluate territory at the end of that sequence. The trouble is that this would not lead it to see the value of defending against its opponent's threats. This could perhaps be remedied by adding to the value of each candidate top-level move the number of points the opponent would gain by playing at the same spot. "The enemy's key point is your own," says the proverb, though that is not always true. Another way to use local search would be to build a second evaluation function, one which is sensitive to all the threats it can find for both sides. Thus if it sees that Black has a threat to destroy White's territory by pushing through on one side, then it should adjust its values for the probability of those points becoming White territory. If Black also has a threat to destroy the same territory from the other side, then the territory is probably doomed.

Another interesting approach to take would be to try to extend the relaxation method of territory analysis to take into account some knowledge about stone patterns, liberties, and the usual endgame sequences resulting from them. In some sense the evaluation function looks at the stones on the board and imagines where the territorial borders will be when the game is complete. If it knew that some stones could push in a certain direction in *sente*, its estimation should change.

One might ask why not try these methods on chess, which is in some sense an easier game and for which decent static evaluation functions and fast move generators exist? For one thing, computer chess is already at such a high level that further improvements are hard to measure. But the main reason is that these search decomposition methods will not work as well in chess. The pieces move about the board freely enough that most non-trivial tactical facts are not locally stable, in the sense that changes elsewhere on the board cannot affect them. Possible exceptions include facts about certain closed positions in chess (when the pawns are locked), some endgame considerations, and sometimes trapped pieces (e.g. a bishop trapped behind enemy pawns). These are cases where current chess programs fare poorly. But for the most part, full-width search, perhaps pruned by reasonableness criteria, seems close enough to what people do when they play chess that with some extra speed it suffices to play very well. In go there are much more powerful search methods people use, and go programs must approximately model these methods to win.

References

[Fotland 91] David Fotland, documentation for "The Many Faces of Go," posted to the electronic news group rec.games.go, November 1991.

[Kierulf 90] Anders Kierulf. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies*. Ph.D. dissertation, ETH Zürich, 1990.

[Plaut, Nowlan & Hinton 86] David Plaut, Steven Nowlan, and Geoffrey Hinton. *Experiments on Learning by Back Propagation*. Technical Report CMU-CS-86-126, Carnegie Mellon 1986.

[Reitman & Wilcox 80] Walter Reitman and Bruce Wilcox. "The Structure and Performance of the Interim.2 Go Program." In *Proceedings of the Sixth International Conference on Artificial Intelligence (Tokyo, Japan, August 20-23, 1979)*. Also in David Levy, editor, *Computer Games II*, Springer Verlag 1988.

[Tesauro 89] Gerald Tesauro. "Connectionist learning of expert preferences by comparison training." In David Touretzky, editor, *Advances in Neural Information Processing 1*, 99-106, Morgan Kauffman 1989.