Measuring the Performance of Go Programs

Martin Müller

Informatik, ETH Zürich
December 12, 1990

Abstract

Human Go players measure their performance by a widely accepted rating scale. But it is hard to assign such a rating to computer programs. We explain the reasons behind this problem and suggest other ways to measure a program's performance.

A Survey of Possible Performance Measures

In this paper we look at four ways of testing a program's level of performance:

- Play between two programs
- Play between human and computer
- Computer analysis of a set of test positions
- 'Reverse analysis' of master games

Each of these tests has its specific strengths and weaknesses. To get an adequate overall picture of a program, it is necessary to combine the tests, each of which will give a different perspective of a program. We discuss each test in turn and show how to use it for testing and improving a program. We debate possible automation of tests and their use for finding errors and locating missing knowledge in a program.

Some Definitions of Performance Measures

To measure the quality of a move played by a Go program, one must define a standard against which to compare the move. In theory, the exact value of a move in a given position can be computed by minimax evaluation of the game tree. This computation is not possible in realistic Go positions and must be replaced by the expertise of a player stronger than the program. Given such a player's opinion about the value of moves, a program's performance can be measured in various ways:

- A binary measure assigns the value 1 to a move judged to be correct and the value 0 to other moves. Summing up this measure for a set of positions yields the percentage of correctly recognized moves.
- An error measure counts the estimated difference in points between a proposed optimal move and the program's move. This method of counting gives less importance to small inaccuracies and more importance to really bad blunders.

Most of the time, the player used as an expert will be human. In Go, this expert knowledge is on a very high level compared with current program strength. For automation purposes, it is desirable to replace the human expert by another program. A more powerful program can in principle act as an expert for a weaker program. This approach has been tried in computer chess [Thompson 82, Berliner / Goetsch / Campbell / Ebeling 89].

Test 1: Play between two Computer Programs

This is the most obvious and also the most popular measure. Computer Go tournaments determine the relative ranking of all Go programs. However, there are some features of a program that do not show up in a game against another computer:

- The ranking obtained by playing other computers can not be compared with human ranking. Humans learn to exploit a program's specific weaknesses very quickly. The Kyu ranking sometimes claimed for a program is always a first game ranking. Experiments show that after several games against the same human opponent, program performance deteriorates by orders of magnitude.
- Current Go programs have little knowledge, so the program is tested only in the restricted set of positions that other programs can generate. For example, most computers cannot play double threats well, while humans do search for them. It also seems that computers are less able to play for influence and other concepts with a fuzzy definition.
- A game is only one variation in the tree of possible moves. A program may play correctly in a particular game, while it would collapse if the opponent tried a different variation that is worse in theory. Playing only one sequence does not guarantee stability of the program's reading.
- The 'bug of the day', i.e. the worst move of a game, often determines the eventual winner, even if the losing program plays better Go on average.
- Playing the program against an earlier version of itself gives some indication of the progress made. It is not a reliable measure of strength because the new version will exploit all the recently removed weaknesses of the old version, while the old version still cannot exploit the remaining weaknesses of the new version.

Computer vs. computer play can be automated by modem connections and appropriate protocols like TeleTsuke or the new protocol proposed by Kierulf and Christensen [Christensen / Kierulf 90] . It seems that so far no protocol supports playing a series of games automatically.

An unusual application of computer vs. computer play is to match a program against a very stupid opponent. The most primitive program that is able to complete a game can be defined as follows:

- Choose randomly from all legal moves, observing the following
- Constraint: Do not fill one point eyes.
- If no such move exists, pass.

Playing against such weak opposition might seem silly at first, since the stronger program can win giving handicaps of 80 stones and more. But it is still a good test to observe a program's behavior in strange positions, and to see its reaction against unexpected moves.

Test 2: Play between Human and Computer

Matching a program against humans is a very common test. One advantage of playing against humans is that the program gets to play opponents of widely varying styles and strengths. The games contain a much wider range of positions than seen in computer vs. computer play.

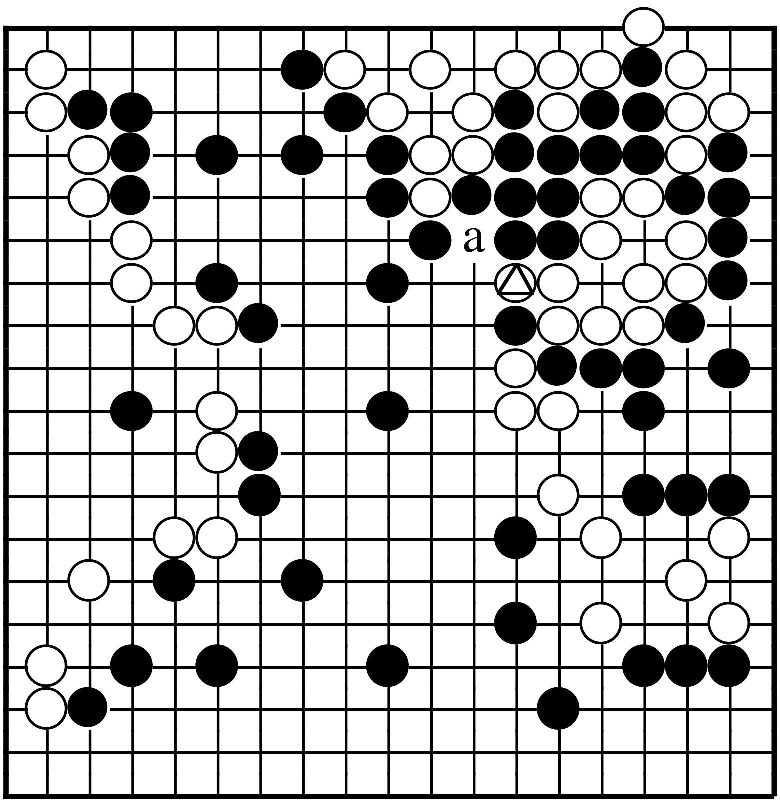
Another important advantage is that humans often try different variations, until they finally find a gap in the program's defenses. This is ideal for testing and debugging purposes. However, for testing the response to really strange moves an opponent playing random moves seems best.

Playing against human opponents obviously cannot be automated. It is possible to automatically record and collect all games played by a program.

Test 3: Test Positions

There are many ways to set up test positions:

- Specialized problem sets for life & death, capturing stones, the endgame etc.
- Find-the-best-move problems: Positions where a definite best move is defined by expert players.
- Decision problems: Positions where the program has to find the correct move out of two possibilities given.
- Bug collections: Positions in which the program has gone wrong before.
- Traps: Positions where a move looks superficially good, but is bad in reality. A famous example of a trap [Kato 78, p. 2] is given in the figure below (Black to move):



To quote Kato:

"Black thus has thirteen stones in atari above and one stone in atari below. Most people would think straightaway of connecting at 'a', but you must consider the board as a whole before deciding whether this is wise or not."

In this position it is better to rescue the single stone and attack the big white group below it.

Because the designer of a test suite has complete freedom on the set of positions included, this testing method is very powerful. With carefully selected test sets one can get independent measures of performance in different stages of a game, leading to a detailed profile of program behavior. While setting up a test suite certainly takes a lot of work, it is then very easy to run it over and over again automatically.

We think that developing a widely accepted set of test positions is an urgent task for the near future. The popularity of the Bratko-Kopec set of test positions in computer chess [Kopec / Bratko 82] supports this thesis.

Test 4: Reverse Analysis of Master Games

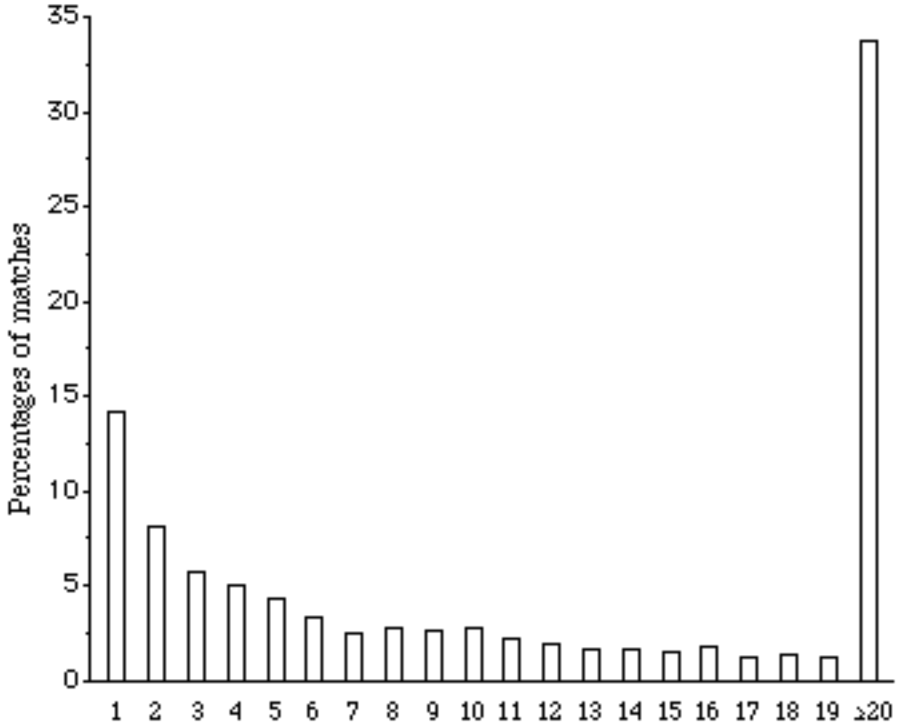
Computer moves can be analyzed by human Go players who know the program's inner workings. This process takes much effort and cannot be automated. We can make use of good player's knowledge in another form: In a game between master players, most moves will be good. Seeing a program's evaluation of these moves gives interesting information about the program. I call this procedure 'reverse analysis', because a weaker player (the program) is 'analyzing' the moves of a stronger player.

The Experiment

Our program 'Swiss Explorer' assigns a numerical value to each possible move in a given Go position. The moves are ordered by evaluation value, leading to a ranking of the moves. For classifying master moves, 20 categories were chosen: Categories 1 to 19 contain the master moves that were ranked in first to 19th position by the program, while category 20 contains all moves with rank 20 or worse. About 3000 moves from randomly selected master games were assigned to these categories.

Results

The results are summarized in the figure below. Explorer would play about 14% of the master moves, rank 8% of the moves as second best, 6% in third position and so on. On the low end of the table, we see that almost 34% of the master moves were ranked twentieth or worse.



Ranks of master moves according to Swiss Explorer

From the statistics, we can distinguish three types of master moves:

- About one third of all moves get ranked in first, second or third place. The program correctly recognizes that these moves are very important.
- One third of all moves is ranked from 4th to 19th place. It seems that the program 'knows' about these moves, but it does not recognize their full value.
- The other 34% of moves are completely missed by our program. This may be either due to missing knowledge or due to errors in the evaluation of the move.

Reverse analysis has proved very helpful for finding the following types of program errors:

- Explorer moves that are evaluated much higher than master moves indicate bugs in the evaluation.
- Master moves with low evaluation show gaps in the program's knowledge.

Some Remarks about the Statistics

The evaluation method used causes some peculiarities. One factor decreasing the performance of our program in the test is that it sometimes gets addicted to a particular local situation and wants to play there repeatedly during the analysis of a game, while the master players consistently ignore the area. The same mistake in evaluation affects many moves. This problem could be avoided by sampling only single positions from a larger games database instead of analyzing all positions that occur in a game, thereby removing the high degree of correlation between positions.

Another factor concerns moves with equal evaluation. Even if the program's proposed move has the same value as the master move, it can be counted as wrong. Moves which get the same evaluation are ordered randomly, and only one of them can get to first place. This does not matter much in the opening and middle game, since the evaluation is sufficiently fine-grained there, but in the late endgame many small moves will get the same evaluation.

Some Performance Figures

For comparison, we have tried a similar test on humans. Several test persons were presented with professional games unknown to them. Their task was to predict the next move. In contrast to the detailed statistics obtained from the computer, humans are not willing to try out their 20 best moves, so we restricted the experiment to 3 tries in each position. The figure below shows the results:

Strength	Rank 1	Rank 2	Rank 3	Rest (≥4)	# Moves
5 Dan	42%	14%	7%	36%	249
5 Dan	45%	14%	7%	33%	263
5 Dan	49%	13%	5%	31%	229
2 Dan	39%	9%	6%	45%	182
2 Dan	44%	9%	6%	38%	192
1 Kyu	47%	15%	7%	29%	268
1 Kyu	38%	9%	3%	48%	239
12 Kyu	34%	7%	4%	53%	239
16 Kyu	27%	8%	4%	59%	239
16 Kyu	23%	15%	7%	53%	92
16 Kyu	26%	7%	5%	61%	176
Explorer	14%	11%	8%	67%	2952

The Relation to Playing Strength

The statistics gained by reverse analysis show some relation to playing strength, with stronger players getting a greater percentage of moves ranked first. Still, the correlation between the percentage of master moves guessed correctly and playing strength does not seem to be simple. A primitive pattern matcher specializing on very few tactical moves like capture, escaping from atari and the like can get similar ratings (more than 10%) as a good program. The number of master moves not considered at all (those in category 20 in the example above) is probably a better measure of program strength.

Using Reverse Analysis to improve a Program

The results of reverse analysis can be used directly to debug and improve a program. One method is to work specifically on the moves ranked second best, trying to adjust parameters to get them to first position. Another way of improving the program is to analyze the moves it misses completely and write new code to generate such moves.

Testing the Similarity of Programs

Reverse analysis is not confined to analysis of master games. Trying to predict the moves of another program or an old version of the same program gives a measure of similarity between programs. For example, our program Swiss Explorer and Ken Chen's program Go Intellect share a common ancestor. Still, Explorer's analysis of a game played by Go Intellect against itself showed that Explorer's play duplicates Intellect's in only 10% of the moves played. The two programs have developed in very different directions.

Future Research Problems

Performance measures for Go programs are a challenging research topic, and this paper is only a small first step towards developing good measures. Several applications of reverse analysis need to be explored in depth.

The Role of Lookahead

Current Go programs can only do a very selective kind of lookahead. Just a small number of all possible moves are included in the further search. The effect of lookahead is to reorder these moves according to their minimax values instead of their static evaluation. How does this reordering affect the statistics collected through reverse analysis?

Consider the case where at most 5 candidate moves will be selected for lookahead. This will not affect the categories 6 and higher, but the moves that fell into the categories 1 to 5 will be reshuffled. Under the assumption that lookahead usually has a beneficial effect, we expect more master moves in category 1 or 2 now, while the number of moves in the lower categories 4 and 5 should decrease correspondingly.

Such an experiment could show the effects of lookahead directly. It could also help to decide the fundamental question: Is it better to have a simple evaluation function and much lookahead, or a very elaborate evaluation and less lookahead?

Move Types

We would like to assess a program's strength in different classes of positions requiring different types of moves. A simple way to achieve this is to perform reverse analysis on selected sets of positions, where each set contains only positions of a specific class. For example, a program's performance in a collection of Joseki or life & death problems could be measured.

An automatic classification of input positions is another challenging research topic.

Related Work

We are not aware of any previous work on measuring the performance of Go programs. There are several well established rating lists for human Go players, and some, like the list of the American Go Association, give ratings for computer programs as well. Concerning test positions, there are many problem books designed for human players. Still, there seems to be no test suite that specifically addresses computer programs.

In chess, the situation is much better. Several well established test suites exist. The best known of these is the Bratko-Kopec test mentioned before. There is a rating list specifically for computer chess programs, maintained by the Swedish Computer Chess Association and published in the ICCA Journal.

Conclusion

Measuring the performance of a Go program naturally leads to new tools for testing and improving the program. The notion of performance is fuzzy in itself: a ranking number alone cannot describe the program very well. A variety of complementary tests are necessary to get an adequate picture of a complex system like a Go program.

References

[Beal 89] Advances in Computer Chess 5. BEAL, D.F. (Ed.), North Holland, Amsterdam, 1989.

[Berliner / Goetsch / Campbell / Ebeling 89] Measuring the Performance Potential of Chess Programs. BERLINER, H., GOETSCH, G., CAMPBELL, M. and EBELING, C. *[Beal 89]*.

[Clarke 82] Advances in Computer Chess 3. CLARKE, M.R.B. (Ed.), Pergamon Press, Oxford, 1982.

[Kato 78] Kato's Attack and Kill. KATO, M., Ishi Press, Tokio, 1978.

[Christensen / Kierulf 90] A Communications Protocol for Cooperative Game Analysis over Modem. To be published in: Computer Go Journal, 1991.

[Kopec / Bratko 82] The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. KOPEC, D., and BRATKO, I. *[Clarke 82]*

[Thompson 82] Computer Chess Strength, THOMPSON, K.*[Clarke 82].*