A Problem Library for Computer Go

Tristan Cazenave
Labo IA, Université Paris 8
cazenave@ai.univ-paris8.fr

Abstract

We propose to renew the interest for problem libraries in computer Go. The field lacks a free and open library of problems. The level and the comparison of programs may well benefit of such a library, and the scientific papers published about computer Go could also benefit from an open and publicly recognized problem library. We analyze the problem related to empirical methods in computer Go, and we propose some improvements to the current ways of dealing with evaluation of programs.

1 Introduction

We advocate for an open problem library in computer Go, its use for giving information on programs performance and validate experiments, as well as the joint elaboration of such a library by computer Go researchers.

We do not claim to have a complete and perfect benchmark for computer Go. Instead we try to establish a viable one, and we try to support the use of such a benchmark in computer Go.

The second section introduces computer Go and describes the work related to problem libraries for computer Go. The third section supports the claim that it is a good idea to try to establish a problem library for computer Go. The fourth section deals with what should be tested in such a library. The fifth section gives hints on practical problems related to the use of a problem library in computer Go.

2 Related work

2.1 Computer Go

The game of Go is an old Chinese game, very popular in Korea, Japan and China. Humans have acquired a lot of knowledge on Go during its long history (to give and idea of the amount of human experience accumulated over time, we can cite Chinese historians who often write that Yao, a semi-mythical emperor of the 23rd century BC invented Go in order to instruct his son Dan Zhu [Fairbain 1995]). Go is a perfect information game between two players: Black and White. A Go board consists of a 19x19 grid, empty at the beginning of the game. The average number of possible moves on a Go board is 250, and games easily last for 200 moves. Therefore a brute force combinatorial approach is not relevant for computerizing the game of Go. Instead, strategic knowledge and goal directed selective search are best suited to the game. Go programs are still very weak compared to humans. The best Go program is officially ranked 9 Korean kyu, the best Go player is 9 dan. It means that the best Go players can let the program play the first 17 moves in a row and still win. The main reason for this weakness is that there is no simple algorithm that can handle all the intricacies and the combinatorial explosion of the game. Building a Go program demands some work on quite different aspects of the game. It is often said that a Go program is as weak as its weakest part. The problem is that there are many difficult parts that have to be done well to have an average Go program. Different specialized searches are usually used to compute the different sub-goals of the game. Such specialized sub-goals are for example: capture (removing the opponent stones of the board), connection (linking two stones by a path of neighboring stones of the same color), life and death (living is ensuring that a set of stones can never be captured), and semeais (a race to capture between two neighboring sets of stones, the issue is often that one set ends alive and the other captured).

The world of computer Go is dominated by tournaments where programs play against other programs. Some international competitions between programs are organized every year: the computer Goe Ing cup, the Korean Garosu cup, the American 21st century cup, or the European computer Go tournament. A possible test methodology that could be established for programs could be for instance to play many games against a fixed version of gnugo [Gnugo 2001], for example gnugo2.6. This is quite easy using the standard Go Modem Protocol. However, it may not be the best test for a program. It often happens in competitions that program A beats program B, program B beats program C,

Tristan Cazenave, Dept Informatique, 2 rue de la Liberté, 93526 St Denis Cedex, France.
http://www.ai.univ-paris8.fr/~cazenave/
Tel: 33 1 49 40 64 04 Fax: 33 1 49 40 64 10

and program C beats program A because each program has its specific weaknesses and strong points, and the outcome of a game depends of a large numbers of factors. So establishing a freely available program such as gnugo as a source of experiments is not the best way to have a good performance measure for programs (even if it still gives valuable information). For example gnugo2.6 is quite weak at life and death of groups, so some programs that are also weak at life and death but better in another domain could win over gnugo. However, when opposed to a program strong at life and death, the results would be different. Whereas a well balanced problem library could find the strong points and weaknesses of a program. Moreover, a well balanced test library is easier to build than a well balanced and strong Go program. Competitions on the tests suites could be organized as side events of the program to program competitions. There are many recognized tournament for programs, but there is not yet a well recognized test base : this is an important point to improve program performance, and also the scientific study of computer Go…

2.2 Computer Go Problem Libraries

The first proposal for a test base for computer Go I am aware of (and the only one) is a paper by M. Müller [Müller 1991]. All the arguments in favor of such a base given in this paper are still valid today. It has been followed by the online test base associated to M. Müller's PhD. Thesis [Müller 1995], and the format has been extended the same year to take other kind of problems into account [Cazenave & Müller 1995]. This work is a good base to build on, and I will advocate for a more frequent use of problem libraries in computer Go, and for an extension of the problems covered by such libraries.

Gnugo has a weak form of problem library, it uses regression testing on some games by giving three possible values to a move : Good, OK or Bad. It would be fine to involve the Gnugo people in the construction of a more elaborate open problem library.

Thomas Wolf proposes 40.000 problems automatically generated by GoTools [Wolf 2000], but many of them are quite artificial and do not represent well the typical problems a program has to face within a real game. Another obstacle to their wide use, is that the problems cannot be distributed freely. In contrast, D. Fotland recently gave in a computer format the problems in Wolf's paper and asked Go programmers to test their programs on them. We would like to generalize such good manners.

Creating a problem library could improve the evaluation of the myriad of algorithms developed for computer Go. The hard work made by some Go programmers is sometime not well deserved by the experiments they choose. Recently some programmers reported interesting work, but test their programs using what we think to be bad benchmarks [Ricaud 1997, Tajima & Sanechika 2000]. Problem libraries where the program has to order three, four or five pre-selected moves are not very convincing tests. A very simple program choosing a random order between the moves can be ranked 9 kyus or even better according to such methods. Of course, the Go programs tested on such test suites have a real knowledge of Go, but their evaluation is biased by the inadequacy of the test.

Other attempts to create test suites where tried in Chess [Kopec & Bratko 1982]. But test suites have to be carefully built. According to computer Chess experts, there are many examples from Chess where programs perform well on test suites, but quite differently in practice. For example it is possible for the test suite to be biased, having many examples that illustrate the same fundamental point. This issue of the balance of the test suite has also to be taken into account in a Go related test suite. The problems by Kano [Kano 1985a,b, 1987] are a well balanced test suite, unfortunately, they are copyrighted, which prevents from distributing them freely.

3 Why?

In this section, we give arguments in favor of a publicly available Go problem library.

The everyday work of the full-time programmers of the best go programs is roughly:

1. Make the program play against another one,
2. find the worst move,
3. correct it,
4. goto 1.

One argument in favor of a standard test suite is to save the time lost analyzing games, but this is not the main point of a test suite as it is always beneficial to analyze one's program games. More important is the problem of finding many of the relevant positions where the program fails to play the right and obvious move. Some of these positions might not appear when testing the program against weak opponents or against other programs, however it is quite important to handle them well as it gives a large advantage to the programs that consistently handles many of them. Much of the time of the Go programmers is used to find such positions where programs fail to see a relatively simple but important move. Archiving them in a problem library will help programmers improve the level of programs at a faster pace.

The usual way to test a program is to make it play against a fixed version of the same or of another program. However, there are so many things to improve in a Go program that it can be discouraging to test against another computer opponent, as the changes made, even when they are positive, may not have an immediate impact on the results. From a psychological point of view, it is important for a programmer to see that his program is improving when he works on it. Having a meaningful and relatively fast measure of strength would help some programmers.

A good test bench for Go programs are the books by Kano [Kano 1985a,b,1987] because they provide a quite extensive set of examples of frequently occurring tactical Go problems. However, these examples are copyrighted, so they cannot be used as an open benchmark for Go programs. Each time a Go programmer wants to use this benchmark, he has to buy the book and then enter by hand in his computer the hundreds of problems. This is clearly a large waste of time.

Another point is that it is sometime too complicated to describe all the parts of a Go program. Giving the detailed result of a program on a benchmark may give more information on the strength and weaknesses of a program than a general written overview of the program. Even the author can gain some knowledge out of this kind of test and realize that other programs handle well some cases where his program is bad. The potential problem with strong Go programs is that they are commercial programs, and that their authors are not willing to give information on how they work. They acquired the hard way the information on the sub-games of Go that can be analyzed and programmed. However, they do not give vital information on their program when giving their performances on a problem library, much less information than when they uncover a well hidden secret. This information on problem solving performance is still useful for them to publicly claim that their program is the best on some points, while the other programmers are happy to know the state of the art on the same point and to measure the distance between their program and the best one.

Other researchers dealing with various search algorithms have similar problems [Gent & al. 1997] [Kaindl & Kainz 1999] evaluating their algorithms. The good side of the game of Go is that programs can play against each other to find the best one. The bad side is that due to the competition, knowledge is often not shared among programmers. A problem library is a middle way, where programmers might accept to share some information, and could also check their algorithms on more or less standard pitfalls.

From a research point of view, an interesting research issue is the Meta-Knowledge used when developing a large software project based on a lot of knowledge [Menzies 1999]. This is typically the case when developing a Go program. Such concerns as initial acquisition of knowledge, better testing and better maintenance [Menzies 1998] naturally arise. Analyzing the development and the usefulness of a Go problem library for the development of a Go program with a knowledge engineering approach might give interesting results. A related study can be found in [Kierulf & al. 1990].

4 What?

In this section, we focus on the problems related to the selection of experiments.

Recently a nice approach to static analysis of life and death has been published [Chen & Chen 1999]. The authors detailed a number of cases where the life and death of a group can be determined statically. Every cases should be covered by at least one problem in a life and death test base. A life and death test base should also contain all the problem mentioned by T. Wolf in his paper [Wolf 2000]. Other simple problems should also be included to check that programs avoid obviously bad moves, and find obviously good ones.

The same kind of case by case analysis would be worthy for other sub-games of the game of Go, such as semeais, capture, connection etc… Here again, each case should be covered by at least one problem.

Some of the problems are mandatory because they are the typical example of an important class of problems. Some other are non-mandatory ones as they cover situations that do not often appears in real games…

Another important issue in the development of a Go program is the creation of a regression test : when the author makes a change in a program in order to play a good move in a position, this change should not make the program play a bad move in another already debugged position. So the good way to make incremental changes to a program is to memorize all the positions where a bug or a wrong move has been corrected and run the test for all the positions once a new bug has been corrected. Given the complexity of Go, it is very likely that a programmer that does not use this methodology performs circles in the space of Go programs: it often happens that fixing a bad behavior in a position unfixes a previous change. One counter measure is to use theorem proving so as to ensure that once a thing is added, it cannot give false results anymore. The good news is that by proving theorems at the tactical level, programs can also get better and smarter [Cazenave 2000,2001a]. However, it is not always possible to rely on theorem proving, especially at the strategic level or when tactical positions are inter-related. Even when using theorem proving, test libraries can be used to improve it by detecting the missing knowledge in some situations. It would be nice that author of Go programs exchange their regression test libraries (some of them are composed of more than 10.000 positions…).

There are different types of Go problems, some programs excel in some parts and have practically no knowledge of some other parts:

- Escape/Capture problems
- Connect/Cut problems
- Eye/Remove Eye problems
- Life/Death problems
- Semeai problems
- Endgame problems
- Fuzeki problems (openings)
- Obvious moves problems
- Bad moves problems
- Professional moves problems

- Influence and Moyos problems
- Double Threat problems
- Problems illustrating strategic concepts such as miai, aji, kikashi, yosu-miru…
- Small board problems
- and some more…

We also propose to extend the format to take into account all these goals and possibly others. For example, we build on the accepted internet standard for Go programs: the SGF standard. However, there is no standard notation for defining a problem. We have defined one, as an extension of the previous attempt [Cazenave & Müller 1995]. It includes new statements to take into account moves that do not reach directly any given goal, but that threatens two goals, and will reach at least one of them as the opponent will protect the other. The double atari move is the most simple example of such moves (atari means only one empty neighboring intersection left, it is a threat to capture the stone). We introduce the predicate move(db):captureOneString(cb,dc) in our problems to denote that the move in cc capture one of the two strings in cb or dc (see figure 1). A more complete definition of the new predicate we introduce can be found on our web site [Cazenave 2001b].

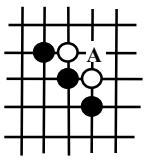

Fig. 1: Black at A is a double atari

Another issue is the creation of a web interface to enable people to enter problems and test themselves the problems by playing online against a program, such as for the GoTools applet [Vesinet & Wolf 2001]. An advantage of a web based Go program is debugging: allowing people to submit error reports, and therefore saving the programmer's time. Another possibility for an online program, is to automatically detect important missing knowledge by selecting games that were lost by large margins. Playing games against humans is important in computer Go because programs never refutes some moves as they are too weak to do so. Such an interface would also facilitate the comparison with human problem solving abilities, which are still better than that of computers in Go [Macfadyen 2000]. However, on some relatively well defined and localized problems, humans can be out performed [Wolf 2000]. Following the published results of Wolf, some Go programmers tested the 64 published problems in their programs, it would be nice to know the relative programs strengths for other kinds of problems.

We propose a basis for a problem library [Cazenave 2001b], which will be extended regularly. We hope enough Go programmers will be interested to enhance it and make it alive proposing new problems, revising the current ones, testing their programs with them and sharing the results.

5 Practical Problems

There are many practical problems related to the effective organization of such a competition. First, as we said before, all the programs should be able to understand the format of the problems. Second, some choices have to be made to fix the rules of the competition: do we set a limited time for all the tests, a limited time for each test, or a penalty associated to the time taken to solve the problems? How to choose the problems in the test suite, and how to weight the relative importance of the problems in the final evaluation of the program? Maybe it is better to separate the competition into well defined sub-competitions such as a life and death competition, a semeai competition and so on, letting programmers enter the competitions they choose? All these questions and many others should be debated between Go programmers, and advice from people used to organize similar competitions is particularly welcome.

Reports on other experiences in organizing such competitions based on search problems such as SAT [Hoos & Stützle 2000], TPTP [Sutcliffe & Suttner 2001] and CSPLib [Gent, Walsh & Selman 2001] can help us. The knowledge of the usual pitfalls and remedies of such competitions may be re-used to avoid them in a new one of a similar style.

Competitions on the problem library could be organized during some of the international programs competitions. Of course, some people could be tempted to add very specific knowledge in their program to perform well at the standard test set, and still have very weak Go programs. A possible remedy would be to create a new small but well balanced test suite for each competition. However, it is more important to keep in mind that the problem library do not replace the program to program confrontation, it should only be considered as a tool to have a reliable and fast evaluation of a program strength, as well as an help to debug Go programs. It is a facility for the correct evaluation of Go programs rather than a goal in itself.

6 Conclusion

Another idea that I did not developed is to make Go programs automatically play against each other on an internet based server. Many programs are already present on Go servers, but there is not yet an automatic program that make them play each other on a regular and frequent basis.

It would be a good idea for computer Go tournament organizers to reserve a little prize for the programs that perform best on a carefully chosen test suite. Provided that the format of the problems are well recognized and that some programs are able to read it. We propose guidelines to

establish such a format, and already provide a first computer Go problem library following this format [Cazenave 2001b]. All the good programs come to the high prized tournaments, and we think that it will contribute to improve the general level of Go programs to create such a problem-based competition.

References

[Cazenave & Müller 1995] Cazenave T., Müller M.: Extending the sgf format to handle automatic program testing. http://web.cs.ualberta.ca/~mmueller/cgo/readmenew.html.

[Cazenave 2000] Cazenave T.: Abstract Proof Search. Proceedings of CG2000. To be published in LNCS 2001.

[Cazenave 2001a] Cazenave T.: Iterative Widening. Proceedings of IJCAI-01, Seattle 2001.

[Cazenave 2001b] Cazenave T.: Golib. http://www.ai.univ-paris8.fr/~cazenave/Golib.html.

[Chen & Chen 1999] K. Chen, Z. Chen, Static analysis of life and death in the game of Go, Information Sciences, 121, (1-2), (1999), pp. 113-134.

[Fairbain 1995] Fairbain J.: Go in ancient China. London, 1995.

[Gent & al. 1997] Gent I.P., Grant S. A., MacIntyre E., Prosser P., Shaw P., Smith B. M., Walsh T.: How Not To Do It. Report 97.27, University of Leeds.

[Gent, Walsh & Selman 2001] Gent I.P., Walsh T., Selman B.: CSPLib: a problem library for constraints. http://www-users.cs.york.ac.uk/~tw/csplib/

[GnuGo 2001] http://www.gnu.org/software/gnugo/

[Hoos & Stützle 2000] Hoos H. H. , Stützle T.: SATLIB - The Satisfiability Library. http://www.satlib.org/.

[Kaindl & Kainz 1999] Kaindl H., Kainz G.: Guidelines for the Experimental Comparison of Search Algorithms. IJCAI-99 Workshop on Empirical AI, Stockholm, 1999.

[Kano 1985a] Kano Y.: Graded Go Problems For Beginners. Volume One. The Nihon Ki-in. ISBN 4-8182-0228-2 C2376. 1985.

[Kano 1985b] Kano Y.: Graded Go Problems For Beginners. Volume Two. The Nihon Ki-in. ISBN 4-906574-47-5. 1985.

[Kano 1987] Kano Y.: Graded Go Problems For Beginners. Volume Three. The Nihon Ki-in. ISBN 4-8182-0230-4. 1987.

[Kierulf & al. 1990] Kierulf, A., Chen, K., and Nievergelt, J.: Smart game board and Go Explorer: a study in software and knowledge engineering. Communications of the ACM, 33(2):152 - 167, February 1990.

[Kopec & Bratko 1982] Kopec D., Bratko I.: The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. In Advances in Computer Chess 3. Clarke M. R. B. (ed.). Pergamon Press Oxford, 1982.

[Macfadyen 2000] Macfadyen M.: What Grade are Problems for? http://www.jklmn.demon.co.uk/gradprob.html

[Menzies 1998] Menzies T.: Evaluation Issues for Problem Solving Methods, T.J. Menzies, *Banff Knowledge Acquisition workshop, 1998*.

[Menzies 1999] Menzies T.: Knowledge Maintenance: The State of the Art, , *The Knowledge Engineering Review*, 14, 1, pages 1-46

[Müller 1991] Müller M.: Measuring the performance of Go programs. In International Go Congress, Beijing. 1991.

[Müller 1995] Müller M.: Computer Go Test Collection. http://web.cs.ualberta.ca/~mmueller/cgo/cgtc.html.

[Ricaud 1997] P. Ricaud, A Model of Strategy for the Game of Go Using Abstraction Mechanisms, in: Proceedings IJCAI'97, 1997, pp. 678-683.

[Sutcliffe & Suttner 2001] Sutcliffe G., Suttner C.: The TPTP Problem Library for Automated Theorem Proving. http://www.cs.jcu.edu.au/~tptp/

[Tajima & Sanechika 2000] Tajima, M. and Sanechika, N.: An Improvement of the Method on the Strategic Placing of Stones Based on the Possible Omission Number, IPSJ SIG Notes, Vol.2000, No.98, pp.85-94 (2000).

[Vesinet & Wolf 2001] Vesinet J.P., Wolf T.: http://lie.maths.qmw.ac.uk/GoToolsApplet.html

[Wolf 2000] Wolf T.: Forward pruning and other heuristic search techniques in tsume go, Information Sciences, 122, (2000), pp. 59-76.