Knowledge and Search in Computer Go – an extended abstract

Ken Chen
Department of Computer Science
University of North Carolina
Charlotte, NC 28223, USA

Keywords: computer Go, knowledge, search, move decision strategy

I. Introduction

Go[1] is a most challenging game to program. The size of the Go board and the nature of the Go game prohibit the use of classic full-board game-tree search paradigm used successfully with chess and many other two-person perfect information games. Go has a very high branching factor, about 250 on the average. It generates a huge game tree of order in the neighborhood of 10^{600}. Classical full board game-tree search can only scratch the surface of it. NxN Go has been proved to be P-space hard [Lichtenstein and Sipser 1980] and exponential-time complete [Robson 1983].

Go programmers have found that understanding Go game positions is extremely hard for the machine. Static evaluation on Go board configurations is essentially impossible to achieve any reasonably high degree of accuracy on regular basis (except for near end games and very calm positions).

Despite the intrinsic combinatorial explosion and positional understanding problems, computer Go has made encouraging progress in the past 30 years. The strength of Go programs has improved from total novice level to intermediate amateur level.
In this talk, we will discuss the following main topics:
a. The essential Go knowledge that a program must have to play reasonable games and its representation and organization.
b. Various search methods that can be used to obtain knowledge and to help make move decision.
c. Move decision strategies that current Go programs use
d. Suggestions for a new move decision strategy and an improvement to current paradigms

II. Knowledge

Go is a territorial game. A Go program looks at black and white stones scattered on a 19 by 19 grid then it needs to figure out each side's territory and potential territory. There is a huge gap between the two ends. It is logical to use a hierarchical model creating intermediate steps and knowledge structures to bridge the gap. A typical hierarchical model is as follows:
Stones – Blocks – Chains – Groups – Territory

[1] The game Go is played on a 19X19 grid using black and white stones. There are two players. One uses black stones and the other uses white stones. They alternately place their stones one at a time onto some empty board intersection points. Unlike chessman, a stone never moves, but it may disappear from the board, called captured, when it looses all its liberties i.e. completely surrounded by opponent's stones. With the exception of Ko points, every empty grid point is a legal position for next move. The objective of the game is to secure more grid points, called territory, than the opponent does. Go, just like chess, is a two-person perfect information game.

We shall discuss each level in order, in the following five subsections.

II.1 Stones and input

A program needs to know the current stone distribution on the board and history move sequence (so it knows who is to play next, any ko or triple ko situations, ...). Logically, the board can be viewed as a two-dimensional, 19 by 19, arrays of Black, White and Empty. But for convenience of pattern matching, we usually add several layers of boarder around the board so that pattern-match routines do not have to constantly worry about invalid array indices. So it becomes a 2x by 2x two-dimensional array of Black, White, Empty and Boarder. In order to avoid implicit multiplications associated with two-dimensional array access, the board is declared as a one-dimensional array in many Go programs for efficiency reason. The game history can be stored in a stack to allow fast move execution and undo (during search look-ahead).

II.2 Blocks and capturing

A bock, also called string, is a set of adjacent stones of one color. Its stones are captured in unison when they loose their liberties (empty adjacent points). If we view a board configuration as a graph, with stones as nodes and any two vertically or horizontally adjacent stones of same color determining an edge, then a block is a component of this graph. Depth-first-search based graph-component algorithm can be used to identify blocks efficiently.

The capturing status of blocks can be classified into three categories:
No danger – the other side won't be able to capture it.
Critical – if the other side plays first, it can be captured; if the bock side to play first it can escape.
Dead – there is no way to escape even the block side plays first.

A Go program usually use capturing tactic search routines, see III. Search, to determine the status of each block. To save time, blocks with many liberties, say 5 or more, are automatically classified as "no danger". Some programs use heuristics to help the classification job. Ladder routine can find a continuous Atari way to capture a block with two liberties. It has an average branching factor close to 1 and is extremely fast. Ladder should be an essential module of any Go program.

II.3 Chains and connectivity

A chain is a collection of inseparable blocks of same color. There are three ways to recognize the connectivity of two blocks:
Heuristic - if two blocks have two or more common liberties or share one protected liberty, they are in the same chain.
Pattern match – connection patterns can be used to recognize connectivity.
Search – goal oriented local connection search can be used.

II.4 Groups and safety

A group is a strategic unit of an army of stones. It consists of one or more chains of same color plus dead opponent blocks, called the prisoners, connected through empty points[2] with influence above a threshold. For details, see [Chen 1989].

[2] They are called the spaces of the group.

Every stone not in a dead block radiates influences across the board. This influence is at maximum at its immediate neighboring spaces and decays as distance increases. Many programs use influence method. The influence propagation is programmed differently though. For examples, Go Intellect uses exponential decay with a distance reduction factor ½, Many Faces uses linear decay 1/distance.

Once groups are identified, the numbers of eyes they have can be decided via heuristics, patterns, or life/death search.

a. **Heuristics** – [Chen and Chen 1999] presents a powerful classification of eye-point types based on diagonal index and heuristic down grading rules. Each eye region's eye number can be estimated heuristically. For example, the number of eyes in an eye-region, with no deficiency and prisoners, can be determined by the following heuristic table, where Ext(R) is the # of surrounding points of region R. Mark Boon was the first person drew my attention to the relationship between Ext(R) and the number of eyes that R has.

Length of Ext(R)	No. of eyes
<= 6	1
7	1.5
8(Square Four)	1
8(Curved Four)	2
8(any other shape)	1.5
9(containing a Square Four)	1.5
9(not containing a Square Four)	2
>= 10	2

b. **Patterns** – many programs use eye pattern libraries to determine eye numbers.

c. **Life/death search** – it is more reliable than heuristics and patterns but it is also more time consuming. Usually, it works better when the group is nearly completely surrounded and not as accurate when the group is more open. [Wolf 1991 & 2000]

The safety of a group is determined by its eye number, its ability to expand (empty neighbors), to run (freedom), to connect to friendly groups, and the safeties of adjacent opponent groups.

II.5 Territory and potential territory

Territory can be estimated by measuring interior spaces (spaces surrounded by grid points belonging to same group) and prisoners (dead opponent stones), adjust by the safety of the group. A group with low safety should be counted as opponent's territory. The numbers of captured stones from each side are easy to keep track and will be included in the scoring under Japanese rules. If Ing Chinese rules are used, we can start with group area (including spaces, stones, and prisoners) and safety. A major source of evaluation error comes from the safety estimate when a group is unsettled, especially in the middle of a fighting. For that reason, Goliath counts a grid point as black, white, or unsettled – crisp 1, -1, or 0. Most programs use fractional numbers in attempts to capture uncertainty. For examples, Go Intellect uses 1/64 point as basic measuring unit & Many Faces uses 1/50 point.

An even more difficult part of evaluation is on potential territory – how to evaluate the values of outside influence, thickness, and moyos? The influence function can only play a limited role here. The number of additional moves needed to surround a

moyo into secure territory is a useful knowledge. [Z. Chen 2000] suggests count each extension of outside stone as 3 points of potential territory at opening. For a nearly completely surrounded weak group, other than almost dead and very light ones, he discounts potential territory by (16/number of additional opponent moves needed to surround the group)*(2 - #eyes). Much additional Go knowledge is still needed to get reasonably good estimate of potential territory. Static evaluation functions in Go programs today are far from accurate. At a computer Go tournament, one can frequently observe that two competing (top) programs have a difference on game evaluation of more than 30 points on a same board configuration. Of course, the exact accurate evaluation function has the same complexity as the whole game of Go, which is beyond reach. An evaluation function that can produce good approximation most of the time seems still extremely difficult to create.

III. Search

Search is a most powerful tool for computer games, with Go included. We can use search to discover knowledge of capturing, connecting, eye-making, territory-surrounding, ...etc. We can roughly divide it into local goal oriented search and global move-selection search. We shall discuss them in the next two subsections.

III.1 Local goal oriented search

Local searches generate and test local moves for achieving some specific goal that can be measured locally. The following is a list of common local goals and their natural basic evaluations.

Capturing:
Goal - capture one block
Evaluation – number of liberties of the block
Semaei:
Goal - capture opponent's block before opponent captures our block for two adjacent blocks
Evaluation – number of relative liberties of the two blocks
Multi-blocks capturing:
Goal - capture one of a set of related blocks of same color
Evaluation – number of total weighted liberties (with blocks with fewer liberties carry more weights)
Connection:
Goal - connecting two chains into one chain
Evaluation – distance between two chains
Life and death:
Goal - making two eyes for a group
Evaluation – number of eyes of the group
Territory:
Goal - surround more territory
Evaluation – local territory estimate

The candidate move generation should be highly selective using heuristic domain knowledge. For example, in capturing search, if one tries all liberties, all secondary

liberties (empty points adjacent to liberties), and all chain connection points, then the resulting local search will be way too slow to be practically useful. Heuristic knowledge needed to reduce the branching factor of the search. One such heuristic is that a liberty with higher number secondary new liberties should have higher priority. Useful search algorithms for local search include:

Selective alpha-beta and its variations – Forward pruning is generally used to speed-up the search. Iterative deepening is useful for time control. Genetic algorithms can be used to tune-up search parameters.

Proof-number search [Allis & al. 1994] – Combining with heuristic candidate move selection, it is effective for local Boolean valued goal oriented search.

Lambda-search [Thomsen 2000] – It guarantees the correctness of the search result, but it won't be fast enough to solve complex tactic problems in real time.

III.2 Global move-selection search

A Go game has about 250 legal moves available at each turn on the average. For any global search algorithm to work, it has to be selective on move candidates. Modified Alpha-Beta searches are usually used for this purpose. Go Intellect uses global selective search to make move decision [Chen 1990, 1998, & 2000]. It has about 20 goal oriented move generators, each generates 0 or more moves with associated move values. A linear combination of each move's move-values from all move generators determines the priority of the move. Only about a dozen or so top candidate moves, those with highest priorities, are actually tried on the board. After two plies, any stable node is evaluated without further node expansion in the global search tree. An unstable node must expand unless it reaches a predetermined depth limit. The mini-max back up of the evaluations of the terminal nodes combined with the urgency values of candidate moves determine the move selection.

B* [Berliner 1979] and probability based B* [Palay 1982, Berliner & McConnell 1994] are best-first search procedures which may handle high branching factor well. For B* to work, a tight optimistic bound and pessimistic bound needs to obtain from node evaluation. Otherwise, the algorithm won't converge in real time. Program Jimmy uses a B*-like global search. The probability distribution required for probability based B* is difficult to construct in Go.

IV. Move decision strategies

In this section, we'll discuss move decision strategies used by current Go programs. They can be roughly divided into four categories: static analysis, try and evaluate, global selective search, and incentive/temperature approximation. There are some other approaches to move decision making in use today, such as neuron networks and pattern matching from huge library of professional games, but they failed to produce strong programs.

IV.1 Static analysis

Programs in this category do not do global search, but perform various goal oriented local searches. Since these local searches do not need to perform for each node of a global search tree, they tend to be more thorough. Each program has its own set of move generators to suggest candidate choices. Programs Dragon, Explorer, & FunGo are

in this category. Dragon uses priority scheme. It divides possible moves into 13 priority levels and selects the move in the highest non-empty priority level with the biggest move value provided by the related move generator(s). Explorer adds the values from all move generators for each point then select the point with maximum sum to play. FunGo uses the maximum value over all move generators for each point and more sophisticated and time consuming tasks will be done on the 18 points with highest values to make the final selection.

IV. 2 Try and evaluate

Candidate moves are generated then a thorough evaluation is performed with each candidate move added to the board in turn. The one with highest evaluation will be chosen. GnuGo, Go4++, Many Faces use this strategy. GnuGo's move generators do not assign valuations but rather move reasons. The actual valuation of the moves is done by a separate single module. GnuGo's source code is in the public domain:
http://www.gnu.org/software/gnugo/devel.html
Go4++ uses pattern matching to generate large number of candidate moves with ranking. A thorough evaluation based on a connectivity probability map is done on each terminal node of this 1-ply global search. Many Faces performs a quiescence search for the evaluation of each candidate move. The search result is modified by the estimate of opponent's gain if playing locally first. Move generators in Many Faces generate (reason, value) pairs for each candidate move. Maximum value is taken in a reason category for each point and values are added over different categories for a same move point.

IV.3 Global selective search

Programs prerform some variation of alpha-beta look-ahead with heuristically selected small set of move candidates at each non-terminal node. The mini-max back up determines the move and scoring estimate. Programs using this strategy include Go Intellect, SmartGo, Indigo, Go Master, Jimmy. Go Intellect uses quiescence search modified by urgency. SmartGo uses iterative deepening and widening. Indigo performs global search with urgent moves and calm moves done separately. GoMaster converts everything to points in evaluation including influence, thickness, sente, gote,...etc. Jimmy performs global selective search using B* type upper and lower bounds associated with each move candidate.

IV.4 Incentive/temperature approximation

Programs in this category consider consequences of either side playing first on a local situation. They include HandTalk, GoMate, Wulu, Haruka, KCC Igo, Goliath, GoStar, and Golois. Programs HandTalk, GoMate, and Wulu use the sum of move values of both sides modified by "move efficiency" to decide the move to play. Local look-aheads are performed but no global look-aheads. Haruka performs 1~4 general local searches, called main searches, each with about 10*10 scope and search depth 3~5, width 6~9. KCC Igo identifies critical areas then performs local searches with candidate moves mostly from pattern matching. Goliath performs two local searches for each critical area, one with either side playing first. The biggest difference on the results of the two searches determines the move selection. Candidate moves are from pattern libraries with patterns represented by bit strings. [Boon 1989]

V. Conclusion

There is no single method that really dominates the rest methods. Performance of a program has a lot to do with the implementation effort – especially on how much useful Go knowledge the programmer(s) has programmed into. It is important to follow good programming practice so that the program can be modified/improved conveniently over time.

Decomposition search [Mueller 1999] has sound theoretical foundation in combinatorial game theory [Berlekamp & al. 1982]. Unfortunately, except during late stage end game, a mutually independent decomposition of the board is impossible. We propose a modified decomposition search as follows: we loosely identify active areas of the board then perform various local searches as suggested by the combinatorial game theory for each area, allowing the area/scope of the search grow dynamically and evaluating the terminal nodes globally on the full board.

Most of the global search performed by programs today based on evaluation of points. But the reward function of Go game is really binary – win or lose. Winning a game by 100 points is the same as winning a game by 1 point – a win. So it is more useful for the static evaluation function to estimate the probability of winning in stead of the expected number of winning points.

Acknowledgement

A couple months ago, I conducted a survey on move decision strategies with well-known Go programs in the world and I received enthusiastic responses from the program authors. These responses made section IV of the talk possible. I would like to thank the following computer Go researchers and programmers for their survey responses and their contribution to computer Go: Zhixing Chen (HandTalk, GoMate, Wulu), Michael Reiss (Go4++), Ryuichi Kawa (Haruka), David Fotland (ManyFaces), Naritatsu Yamamoto (KCC Igo), Daniel Bump (Gnu Go), Shun-Chin Hsu (Dragon), Anders Kierulf (SmartGo), Martin Mueller (Explorer), Bruno Bouzy (Indigo), Tristan Cazenave (GoLois), Jimmy Lu (GoStar), Yong-Goo Park (FunGo), Jee Won Ho (GoMaster), and Jimmy Yen (Jimmy).

References

[Allis & al. 1994] L. V. Allis, M. van der Meulen, H. J. van den Herik, Proof-number search, Artificial Intelligence, 66, 1994, pp. 91-124.

[Berlekamp & al. 1982] E.R. Berlekamp, J.H. Conway, and R.K. Guy, Winning Ways. Academic Press, New York, 1982.

[Berliner 1979] H. Berliner, The B* tree search algorithm, a best-first proof procedure, Artificial Intelligence 12, 1979, pp. 23-40.

[Berliner & McConnell 1994] H. Berliner and C. McConnell, B* Probability Based Search, CMU-CS-94-168.

[Boon 1989] M. Boon, A Pattern Matcher for Goliath, pp. 12~23, Computer Go, No. 13, 1989.

[Burmeister and Wiles 1997] J. Burmeister and J. Wiles, AI Techniques Used in Computer Go, Proceedings of the Fourth Conference of the Australasian Cognitive Science Society, Newcastle, 1997.

[Chen 1989] K. Chen, Group Identification in Computer Go, Go chapter in the book "Heuristic Programming in Artificial Intelligence" edited by D. Levy & D. Beal, pp. 195-210, Ellis Horwood, Fall 1989.
[Chen 1990] Chen, K., Move Decision Process of Go Intellect, pp. 9-17, Computer Go, No.14, 1990.
[Chen 1998] K. Chen, Heuristic Search in Go Game, Proceedings of Joint Conference on Information Sciences '98, vol. II, pp. 274-278, 1998.
[Chen and Chen 1999] K. Chen and Z. Chen, Static Analysis of Life and Death in the game of Go, Information Sciences, Vol. 121, Nos. 1-2, pp. 113~134, 1999.
[Chen 2000] K. Chen, Some Practical Techniques for Global Search in Go, ICGA Journal, Vol. 23, No. 2, 67-74, June 2000.
[Chen 2001] K. Chen, A study of decision error in selective game tree search, Information Sciences, Vol. 135, No.3-4, 177-186, July 2001.
[Z. Chen 2000] Z. Chen, The small world of computer Go, book in Chinese, Zhongshan University Publisher, 2000.
[Kao 1997] K. Kao, Sums of Hot and Tepid Combinatorial Games, Ph.D. thesis, University of North Carolina at Charlotte, 1997.
[Kao 2000] K. Kao, Mean and temperature search for Go endgames, Information Sciences, 122, 2000, pp. 77-90.
[Kierulf & al. 1990] A. Kierulf, K. Chen, and J. Nievergelt, Smart Game Board and Go Explorer: A study in software and knowledge engineering, Communications of ACM, pp. 152-166, February 1990.
[Lichtenstein and Sipser 1980] D. Lichtenstein and M. Sipser, Go is polynomial-space hard, Journal of ACM, vol. 27, no.2, pp. 393-401, 1980.
[Mueller 1995] M. Mueller, Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory, Ph.D. Dissertation, Swiss Federal Institute of Technology Zurich, 1995.
[Mueller 1999] M. Muller, Decomposition Search: A combinatorial games approach to game tree search, with applications to solving Go Endgames, In IJCAI-99, pp. 578-583, 1999.
[Palay 1982] A. Palay, The B* Tree Search Algorithm – New Results, Artificial Intelligence 19(2), 1982.
[Robson 1983] J.M. Robson, The Complexity of Go, Proc. IFIP, pp. 413-417, 1983.
[Thomsen 2000] T. Thomsen, Lambda-Search in Game Trees – with Application to Go, ICGA Jpurnal, Vol. 23, No. 4, pp. 203~217, 2000.
[Wolf 1991] T. Wolf, Investigating Tsumego Problems with "RisiKo", Heuristic Programming in Artificial Intelligence 2, edited by D. Levy & D. Beal, pp. 153-160, 1991.
[Wolf 2000] T. Wolf, Forward pruning and other heuristic search techniques in tsume Go, Information Sciences, 122, 2000, pp. 59-76.