SOME PRACTICAL TECHNIQUES FOR GLOBAL SEARCH IN GO

Keh-Hsun Chen[1]

Charlotte, USA

ABSTRACT

A position evaluation and a candidate-move-generation strategy for global selective search in Go are described. Moreover, some Go-specific enhancements to the basic global selective alpha-beta game-tree search procedure are discussed. Finally, empirical results on the performance of the enhancements are presented.

1. INTRODUCTION

Go is a board game invented in China four thousand years ago. It is very popular in Japan, China and Korea, and has gradually attracted players in the western world in recent years. Current estimations amount to 30 million players worldwide. The standard Go game is played on a 19×19 grid using black and white stones. There are two players. One uses the black stones and the other uses the 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 loses all its liberties, i.e., it is completely surrounded by the opponent's stones. With the exception of moves causing a full-board repetition (a situation referred to as ko), or causing own-stones suicide, every empty grid point is a legal position for play. The objective of the game is to secure more grid points, called territory, than the opponent. Go, just like chess, is a two-person perfect-information game.

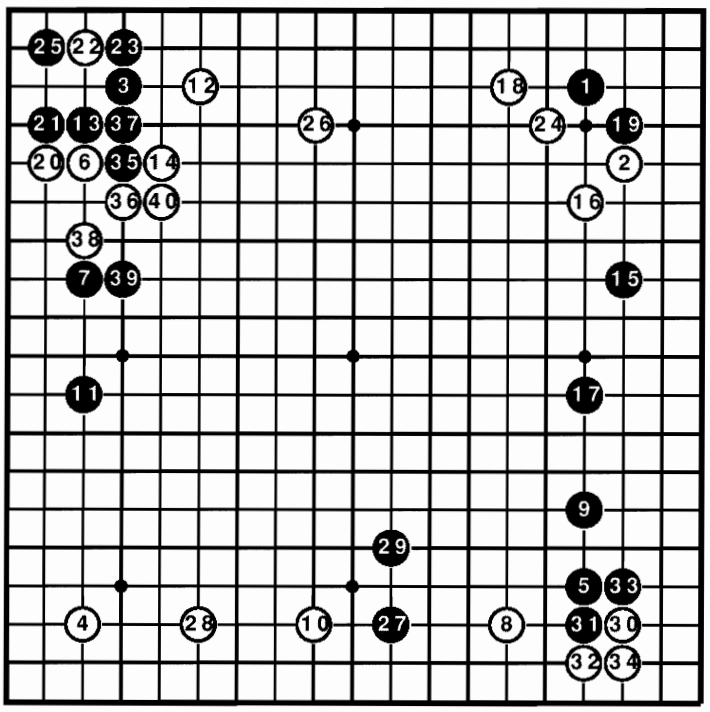

Figure 1: A partial Go game of moves 1 to 40 from a computer-Go tournament, moves are numbered in order.

Figure 1 shows the first 40 moves of a Go game taken from a computer-Go tournament match. For a detailed description of the Go game and its rules, readers may access the following web sites:

http://www.usgo.org/resources/whatisgo.html

and

http://www.britgo.org/intro/intro1.html.

[1] Department of Computer Science, University of North Carolina at Charlotte, Charlotte, NC 28223, USA.
E-mail: chen@uncc.edu

The branching factor is over 200 on average. The high number prevents any exhaustive global search algorithm from looking ahead sufficiently deep. Furthermore, understanding and evaluating Go game positions are extremely hard for the machine. The playing strength of Go programs today is far below the human expert level. This is true not only in regular 19×19 Go but also in 9×9 Go. The latter has an average branching factor of about 40, which is comparable to that of chess. So, the positional understanding problem per se posts already a seemly insurmountable challenge to Go programming. It is not uncommon to see two opposing Go programs with each of them indicating that it is over 20 points ahead of the other in the middle of a tournament game, at the same time on the same board configuration. The conclusion is obvious, Go is widely regarded as a most difficult game for the machine.

Many different general approaches have been used in building Go programs, including pattern matching (Zobrist, 1970), global selective search (Chen, 1990), decomposition search (Müller, 1999), static analysis (Chen and Chen, 1999), neuron networks (Enzenberger, 1996), etc. This paper focuses on the global selective-search approach as implemented in the author's program GO INTELLECT. We describe a basic position evaluation and a candidate-move-generation strategy for the global selective search in Sections 2 and 3, and discuss Go-specific enhancements to the basic global selective alpha-beta game-tree search algorithm in Sections 4 to 7. Empirical results on the performance of the enhanced global search procedure are presented in Section 8. Finally, Section 9 concludes the paper and suggests future research topics.

2. UNDERSTANDING AND EVALUATION

As we can see from Figure 1, a game configuration of Go is a scattered collection of black and white stones on the Go board. In order for the machine to find a good move or even just to decide whether a move is legal, some basic *understanding* of the current Go configuration is essential.

A multi-level hierarchy of arrays of records is used for knowledge representation in GO INTELLECT. The idea is adopted from Friedenbach (1980). At the lowest level, the current board configuration is just a two-dimensional array of black and white stones plus empty points. (For reasons of efficiency, many programs, including GO INTELLECT, use a one-dimensional array to represent the board avoiding implicit multiplication during array access.) The level above it is the level of *blocks*. A block is a directly connected set of stones of the same colour, also known as *strings* (see Figure 2). The empty points immediately adjacent to a stone in the block are called *liberties*. When a block loses all its liberties, the stones in the block will be removed from the board.

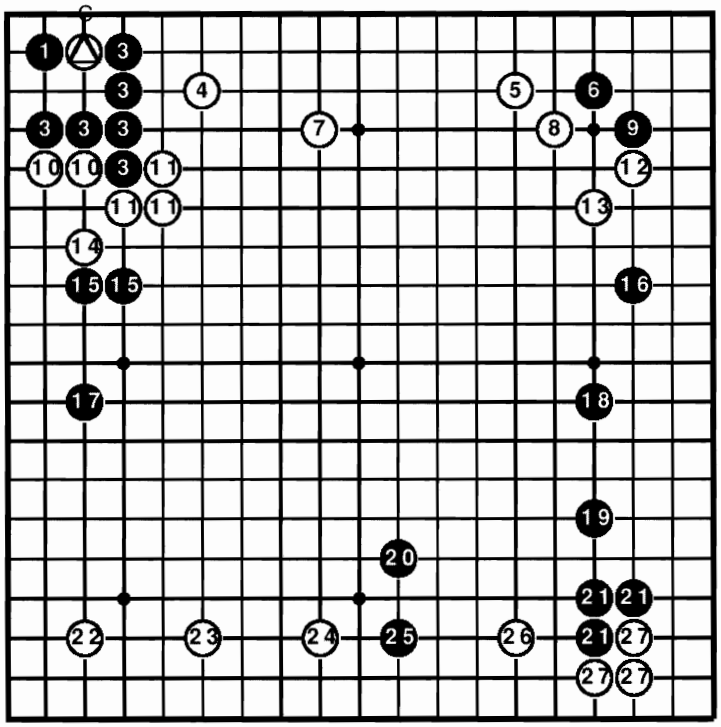

Figure 2: *Blocks* - stones of same block are marked by the same number.

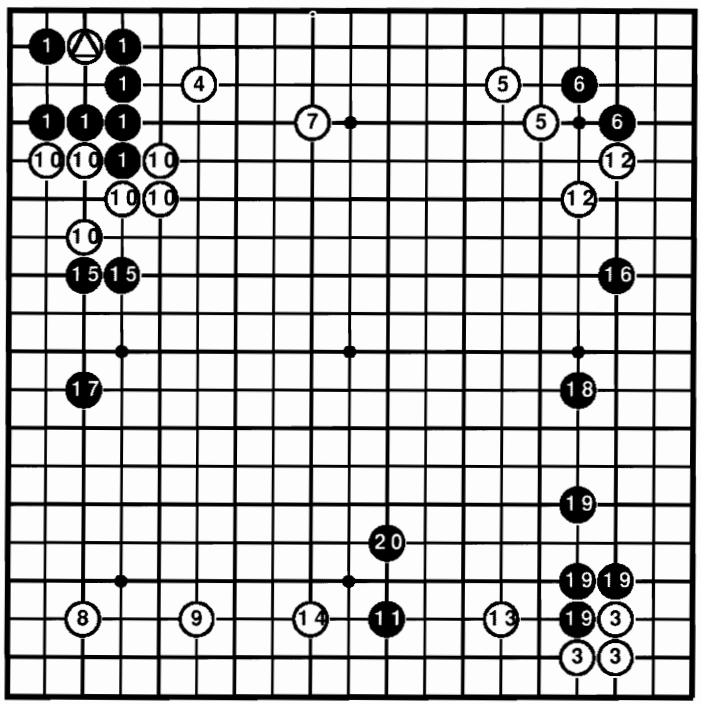

Figure 3: *Chains* - stones of same chain are marked by the same number.

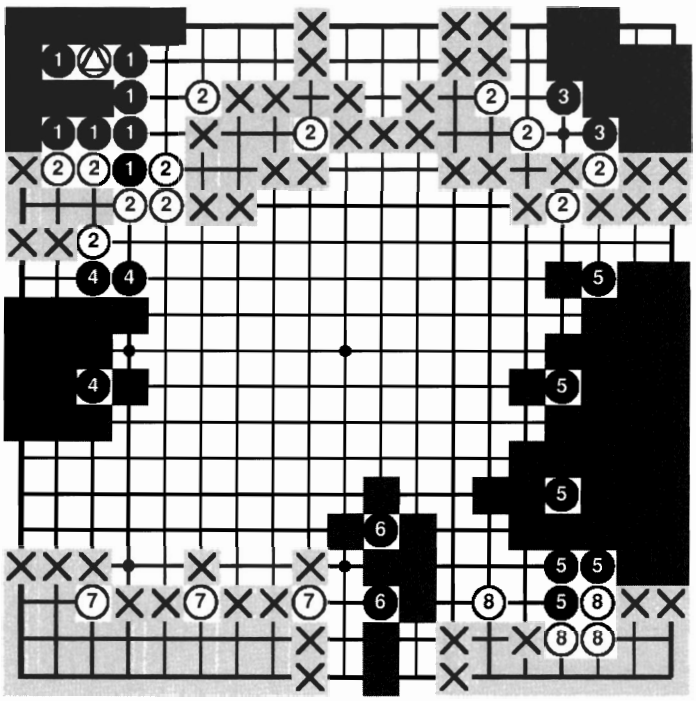

Figure 4: *Groups* - stones of same group are marked by the same number.

The next higher level is the level of *chains*. A chain is a collection of inseparable blocks of the same colour, (see Figure 3). Chain 5 is recognized by the multiple common liberties of its two blocks. Chain 10 is recognized by one protected common liberty of its blocks. Chain 19 is recognized by matching the stones with a linkage pattern.

The highest level is the level of *groups*. A group is a family of related chains plus the enclosed spaces and the possibly-dead opponent stones, called prisoners. A group forms a strategic unit in a Go game (see Figure 4). The spaces of a group are shaded, the prisoners are marked by triangles and the frontier spaces are marked by X. Groups have many important attributes, such as *area* and *safety*. For a detailed description of how the groups are identified and how associated properties are computed, we refer to Chen (1989). Go players know that a group requires two eyes to live. A set of heuristic rules for evaluating the number of eyes for a group can be found in Chen and Chen (1999).

For each new board configuration and for each node in the global search tree, the program typically performs several dozens of goal-oriented local searches.

These local searches include:

* *Ladder* - check whether each 2-liberty block can be captured by consecutive ataries and whether each 1-liberty block can escape from the opponent's consecutive ataries. (An atari is a move that reduces the liberty of an opponent block to 1.)
* *Capture* - check whether a block can be captured.
* *Multi-block capture* - check whether one of a set of target blocks can be captured.
* *Life/Death* - check whether a group can make 2 eyes, this search is done only when the heuristic rules fail to make a definite conclusion.
* *Linkage* - check whether 2 nearby chains can be connected when the heuristic rules and the linkage pattern library do not provide the answer.

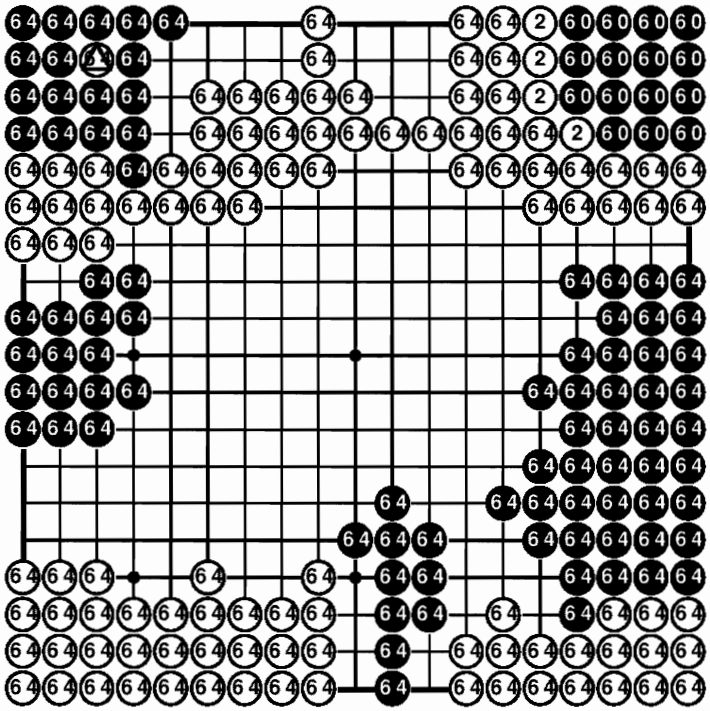

Figure 5: Territorial Evaluation.

The *safety* of a group is determined by the group's eye number and eye-making potential. The number of eyes and the eye-making potentials of adjacent opponent groups are also important factors in calculating the group safety. For simplicity, let us assume that the safety is a real number between -1 and 1. Safety value 1 means that the group is completely safe. Safety value -1 means that the group is dead for sure. For each point p on the Go board, we define `Territory[p]` to be the safety value of the group to which p belongs if the group belongs to the player who is to move, and (- safety value) if the group belongs to the opponent. If point p does not belong to any group then `Territory[p]` will be determined by the *influence* value (cf. Chen, 1989), and safeties of the groups in the nearby neighbourhood. Figure 5 shows the *territory* values in the range of -64 to 64 (dividing by 64 gives the normalized value between -1 and 1).

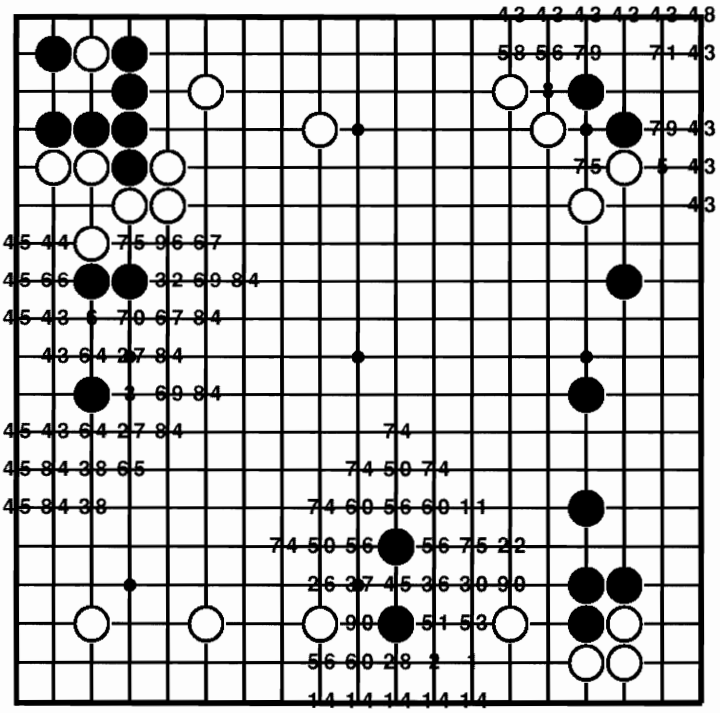

Figure 6: Moves and the associated weights generated by the move generator *ProtectGroup*.

The summation of `Territory[p]` for all points p on the board is a reasonable evaluation of a current board configuration or a current node in a global search tree. A major evaluation error frequently occurs when two or more groups are in a fight and each of them, with safety < 1, does not clearly have two eyes. An accurate evaluation depends on an accurate deep look-ahead on the development of the battle, which has a complexity equivalent to the original move decision problem of Go. Some of these fighting positions cause evaluation problems even to human experts.

3. GENERATING CANDIDATE MOVES

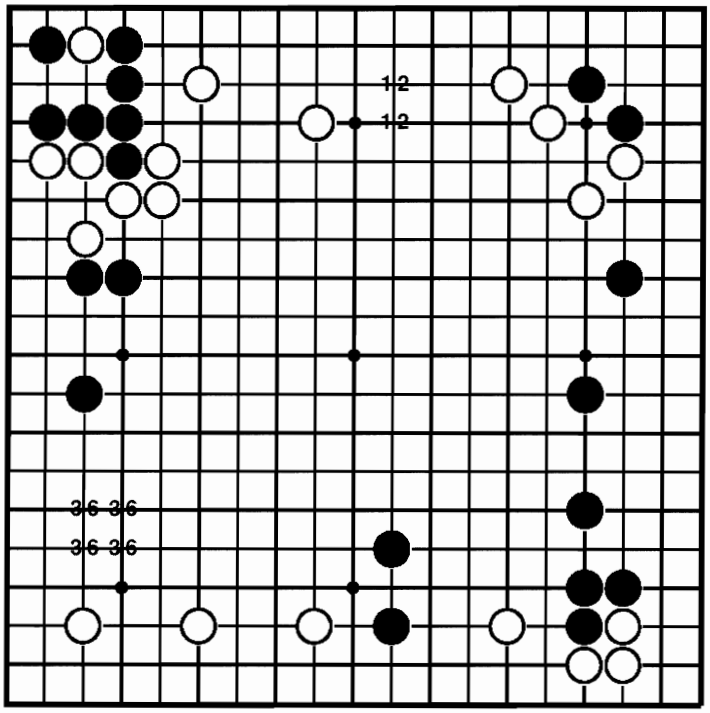

Figure 7: Moves and the associated weights generated by the move generator *EdgeExtension*.

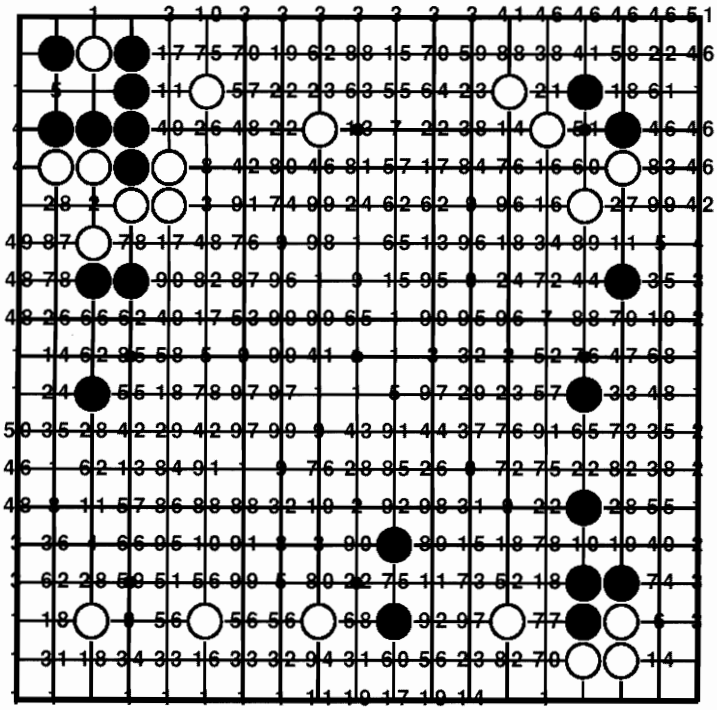

Figure 8: The combined move values, which rank the candidate moves for selection.

Since Go is a territorial game, it is natural to design special-purpose move generators to achieve one or more of the following basic goals:

* expand the player's territory,
* reduce the opponent's territory,
* increase the safeties of the player's groups,
* reduce the safeties of the opponent's groups.

GO INTELLECT has 20 move generators. For example, the move generator *ProtectGroup* produces the surrounding good-shape points of an unsafe group to increase the safety of the group (see Figure 6). The move generator *Edge Extension* suggests moves for edge extension and for preventing the opponent's edge extension (see Figure 7). Each move generator generates zero or more moves, each move has an associate weight. Pattern libraries are commonly used by Go programs to suggest plausible candidate moves. A pattern move generator and a YosePattern move generator (each with its own library to match the stones on the board) produce candidate moves recommended by the libraries with associate weights. Some move generators require special-purpose local searches. The sum of all weights of each point is computed (see Figure 8). The top 10 or so candidate moves with the highest combined weights are selected as potential moves to try in the global selective search. The half of the highest move value is used as a threshold. Only moves with a value reaching the threshold will be selected. So, frequently the number of candidate moves is fewer than the maximum allowed number of 10. (If no other move reaches the lower bound, the move with the highest move value is immediately selected for play.) The practical techniques discussed below frequently produce early cut-offs. Hence, usually only less than half of all selected candidate moves are actually tried in the search look-ahead.

4. CUT OFF THE SEARCH AT QUIESENCE

When the board configuration is stable, the evaluation method of Section 2 is quite accurate. But when it is not stable, the evaluation function may produce a value with a large error. Therefore, it is better that a global selective search procedure in Go does not use a fixed predetermined search depth. Instead, a pair of depth bounds *MinDepth* and *MaxDepth* can be used. Each node in the global search tree is computationally intense and may involve dozens of local searches. So, the depth bounds cannot be set too high in the present-day computational environment. GO INTELLECT currently uses 5 for *MaxDepth* and 1 for *MinDepth* in its global search. (Of course, goal-oriented local searches usually look more deeply.) During the global look-ahead, all nodes generated within the depth bounds are evaluated. If a node is stable, its territorial evaluation is passed up without further node expansion, because further look-ahead may be forced to terminate at unstable nodes producing unreliable results. It is not an easy task to decide whether a current node is stable. The following heuristic rule seems to work well at practice.

* If the evaluation of the current node and the evaluation of the parent node differ by an amount less than twice the *RegularMoveValue* then the current node is likely to be stable.

RegularMoveValue is an estimate of how many points an average move is worth at the current stage of the game. It is about 16 at the opening and it gradually decreases to 1 near the end of the game.

5. CUT OFF THE SEARCH WHEN THE TARGET VALUE IS REACHED

Quite often, a suicidal move or another kind of bad move can produce an unrealistically "good" (backup) evaluation. For example, a risky invasion might appear temporarily to make the opponent's territory smaller, or an atari move with a bad side effect might force the opponent to respond and push a bad problem off the search horizon. So looking at too many moves, the search engine will inevitably select some rather bad moves. An effective way to reduce such kind of bad-move decision and yet still to obtain a great deal of candidate moves to execute when needed is as follows.

* Set a target value before the search starts and stop examining further candidate moves as soon as the target value is reached.

  Target value = Max (parent node's evaluation value, current node's evaluation value) + 1

The above formula gives a good setting of the initial target value. We decrease this target value gradually, say by subtracting one for each new move tried. So, we have a moving-down target as we go down the selected candidate move list. If we have reached the target, do not look at any more moves from the candidate list, just use the best move so far to play or to back up its value. When the candidate moves examined so far fail to handle the opponent's threat properly, the current minimax back-up will fall short of the target value. In this case, more selected candidate moves can be tried in turn until the target is reached or candidate moves are exhausted. Hopefully, an adequate response can be found in the process. In order for the *target value technique* to produce good results, selected candidate moves needs to be roughly strongly-ordered (Chen, 1998).

6. ADJUSTMENT BY URGENCY VALUE OF THE CANDIDATE MOVE

Using minimax back-up of territorial evaluations to decide which candidate move to play has a big flaw (Chen, 2000). It may select those moves that look "big" and ignore solid/urgent defensive moves, for which the bad consequences of omitting (or the good consequences of playing) are beyond the search horizon. For example, a move protecting a crucial cut of our group may be wrongly evaluated by the territorial evaluation function as smaller than an edge-extension move elsewhere on the board. So, the bad consequences of failing to protect the cut are not detected within the current search horizon and will be recognized by territorial evaluation function only many moves later.

My solution to this problem is to introduce an *urgency value* for each move. The urgency value is in the range of 0 to about 20. The value originates from move generators and/or pattern-recognition routines.

* The urgency value of a move is added to the minimax back-up of the move when competing for the best choice at each node in the search tree, but the urgency value itself is not backed up.

A move leading to a position, in which the opponent has an urgent response, should not be penalized by the urgency value of the opponent's move.

7. AN ENHANCED GLOBAL SELECTIVE GAME-TREE SEARCH PROCEDURE

If we incorporate all the enhancements, mentioned in the previous sections, into the classical neg-max version of the alpha-beta procedure, we obtain the following outline of a global selective search routine for Go. It is presented in a Pascal/Modula2-like pseudo code. We reiterate that the urgency value plays an important role in selecting the best move but the urgency value itself is not passed back up.

```
AlphaBetaX(p: POSITION; alpha, beta, depthToGo, parentVal: INTEGER; VAR bestMove: POINT):
INTEGER;

VAR             eval, max, i, v, w: INTEGER;
                        pBest: POINT;
                        u, u0: INTEGER;
                        target: INTEGER;

BEGIN
                eval := StaticEvaluation(p);
                bestMove := Empty;
                IF (abs(parentVal + eval) < 2*RegularMoveValue) (*the node is stable*)
                THEN
                        RETURN eval;
                ELSE IF (depthToGo = 0) THEN
                        RETURN (eval-parentVal) DIV 2;
                        (*this usually gives better evaluation than eval when the node is not stable*)
                END(*IF*);
                generate top few legal moves m1, m2,...,mw;
                max := alpha;
                u0 := 0;
                target := Maxi(eval, -parentVal) + 1;
                FOR i := 1 TO w DO
                        u := Urgency(mi);
                        execute move mi on p to get position pi;
                        v := - AlphaBetaUT(pi, -Beta, -max-u0+u, depthToGo - 1, eval, pBest);
                        IF (v + u > max + u0) THEN
                                max := v; u0 := u; bestMove := mi;
                        END(*IF*);
                        IF (max >= beta) OR (max >= target-i) THEN
                                RETURN max;
                        END(*IF*);
                END(*FOR*);
                RETURN max;
END AlphaBetaUX;
```

This procedure can be called from the top level via

```
OldScore := Score;
Score := AlphaBetaX(current position, -999, 999, MaxDepth, OldScore, BestPoint);
```

8. EFFECTIVENESS OF THE ENHANCED GLOBAL SEARCH PROCEDURE

Fifty games were played between GO INTELLECT with the enhanced global selective search procedure outlined above and the same version of GO INTELLECT with a normal fixed depth (i.e., to *MaxDepth*) global selective search. Each version played Black, i.e., moving first, in 25 games, and played White, i.e., moving second, in the other 25 games. Both versions used level 8, GO INTELLECT's normal tournament level, with time limit one hour each. Moreover, 6 points komi was used (i.e., Black's territory was deducted by 6 points at the end to even out the advantage of moving first). The enhanced version won 17 games, 68% of 25, taking Black, and won 14 games, 56% of 25, taking White. Over all, it won 62% of the games. Furthermore, the enhanced version played 15.5% faster than the standard version on an average of fifty games.

In a separate double round-robin tournament I experimented with 8 variations of the same GO INTELLECT equipped with a global selective search with forward pruning, moving target, and urgency adjustment at different on/off settings. Each version played 14 games. The version with all three enhancement settings was the clear winner with 10 wins. The other versions, beating one another, had 6 to 8 wins each.

9. CONCLUSION AND FUTURE WORK

In contrast to the global selective search described above, there is decomposition search (Mü ller, 1999). This combinatorial game approach works well in endgame stages of Go, when the board can be divided into independent regions. In early and mid-game stages, it is essentially impossible to have mutually independent decompositions of the board. The author has tried a semi-decomposition search approach to computer Go, dividing the board into roughly "independent" regions without much success. The interactions of the developments of the decomposed regions are simply too great to ignore. Global selective search does not have the interaction/coordination problems of the semi-decomposition search. Two main questions remaining are: (1) How to improve the quality of the board evaluation function? and (2) How to ensure that a good move is included in a small set of selected candidate moves? The answers are necessary conditions for improving the decision quality of global selective-search-based Go programs.

The main weakness of the global selective-search approach to computer Go is that the same moves may appear time and again in different order and at many different places in the global search tree. This makes the search rather inefficient (it is exactly what decomposition search can avoid). Avoiding/reducing such inefficiency can make the search procedure several orders faster and allow the search to go more deeply. It will result in a much stronger program. A balanced combination of global selective search and decomposition search taking advantage of the merits of both methods may well be the best approach to computer Go.

10. REFERENCES

Chen, K. (1989). Group Identification in Computer Go, *Heuristic Programming in Artificial Intelligence*, (eds. D.N.L. Levy and D.F. Beal), pp. 195-210. Ellis Horwood Ltd., Chichester, England. ISBN 0-7458-0778-X.

Chen, K. (1990). Move Decision Process of Go Intellect, *Computer Go*, No.14, pp. 9-17.

Chen, K. (1998). Heuristic Search in Go Game Tree, *Proceedings of Joint Conference on Information Sciences '98*, Vol. II, pp. 274-278. The Association for Intelligent Machinery, Inc. ISBN 0-9643456-7-6.

Chen, K. and Chen, Z. (1999). Static Analysis of Life and Death in the game of Go, *Information Sciences*, Vol. 121, Nos. 1-2, pp. 113-134. ISSN 0020-0255.

Chen, K. (2000). Decision Error in Selective Game Tree Search, *Proceedings of Joint Conference on Information Sciences 2000*, Vol. I, pp. 978-981. The Association for Intelligent Machinery, Inc. ISBN 0-9643456-9-2.

Enzenberger, M. (1996). *The Integration of A Priori Knowledge into a Go Playing Neural Network*. http://home.t-online.de/home/markus.enzenberger/neurogo.html.

Friedenbach, K.J. (1980). *Abstraction Hierarchies: A Model of Perception and Cognition in the Game of Go*, Ph. D. Thesis, University of California, Santa Cruz.

Müller, M. (1999). Decomposition Search: A Combinatorial Games Approach to Game Tree Search, with Applications to Solving Go Endgames, *IJCAI-99*, pp. 578-583. Morgan Kaufmann, San Mateo, CA. ISBN 1045-0823.

Zobrist, A.L. (1970). *Feature Extraction and Representation for Pattern Recognition and the Game of Go,* Ph.D. Thesis (152 pp.), University of Wisconsin.