# COMPUTER GO: KNOWLEDGE, SEARCH, AND MOVE DECISION

*Keh-Hsun Chen*[1]

Charlotte, NC, USA

ABSTRACT[2]

This paper intends to provide an analytical overview of the research performed in the domain of computer Go. Domain knowledge that is essential to Go-playing programs is identified. Various computation and search techniques that can be used effectively to obtain helpful domain knowledge are presented. Four different move-decision paradigms applied by today's leading Go programs are discussed. Conclusions are drawn and two proposals of improvements to current move-decision paradigms are presented.

## 1. INTRODUCTION

Go[3] is a most challenging board game to program. The size of the Go board and the nature of the game render the classic full-board game-tree search paradigm powerless, despite its very successful performances 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 the order of around $10^{600}$ nodes. Classical full-board game-tree search can only scratch the surface of it. The combinatorial-explosion problem is much more severe in Go than in other board games, such as chess. Theoretically, *N*-by-*N* 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 positions is extremely hard for a machine. Static evaluation of Go-board configurations is essentially impossible: it is hard to achieve any reasonably high degree of accuracy on regular basis (except for near endgames and very calm positions). Moreover, the dynamic evaluation and the positional understanding problem create even more difficult challenges for computer Go.

Despite the intrinsic combinatorial explosion and the positional understanding problems, computer Go has made encouraging progress in the past thirty years. The strengths of leading Go programs have improved from total novice level to intermediate amateur level. In this paper, we discuss four main topics:

a. the essential domain knowledge that a Go program must have to play reasonable games, in combination with its representations and organizations;
b. various types of search problems and search methods that can be used to obtain knowledge and to help make move decisions;
c. move-decision strategies that current Go programs use;
d. conclusions and suggestions of two proposals to the current move-decision paradigms.

## 2. KNOWLEDGE

Go is a territorial game. A Go program looks at black and white stones scattered on a 19-by-19 grid. It needs to figure out what each side's territories and potential territories are, so that it can make intelligent move decisions. There is a huge gap between the two ends: the board configuration and the move decision. It is

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

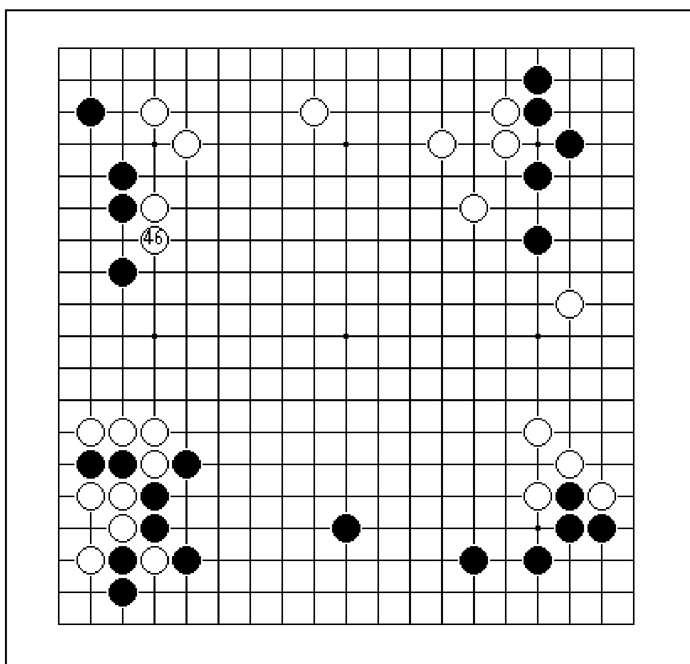
[2] The paper is an edited and extended version of the author's invited lecture "Knowledge and Search in Computer Go" presented at the Game Programming Workshop 2001 held in Hakone, Japan, October 26 to 28, 2001 (Chen, 2001b).

[3] The game of 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 intersection points of the Go board. Unlike a chessman, a stone never moves, but it may disappear from the board, called captured, when it has lost all its liberties, i.e., it is completely surrounded by the opponent's stones. With the exception of ko points, which create full-board repetitions and are prohibited, every empty grid point is a legal position for the 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.

logical to use a hierarchical model creating intermediate steps and knowledge structures to bridge the gap. A typical hierarchical model consists of something similar to the following five layers: Stones – Blocks – Chains – Groups – Territories. In the Subsections 2.1 to 2.5 each layer is discussed in the order given.

## 2.1 Stones and input

A program needs to know the current stone distribution on the board and the history of the move sequence leading to that position (so that it knows who is to play next, any ko or triple ko situations involved, etc.). Naturally, the board can be viewed as a two-dimensional, 19-by-19, array of Black, White and Empty.




But for the convenience of pattern matching, we usually add several border layers around the board so that pattern-match routines do not have to worry constantly about getting invalid array indices. Hence, the board becomes a $2n$ by $2n$ two-dimensional array of Black, White, Empty, and Border. In order to avoid implicit multiplications associated with two-dimensional array accesses, the board is often declared as a one-dimensional array for efficiency. The game history can be stored in a stack to allow fast move execution and move undo (especially during search look-ahead). Figure 1 presents a board configuration at the beginning of a mid-game stage. The configuration is meant to become familiar with the placement of stones. We use the same configuration in Figure 2 (for blocks), in Figure 3 (for chains) and in Figure 5 (for groups).

**Figure 1:** Stones: a board configuration at the beginning of a mid-game stage after 46 moves.

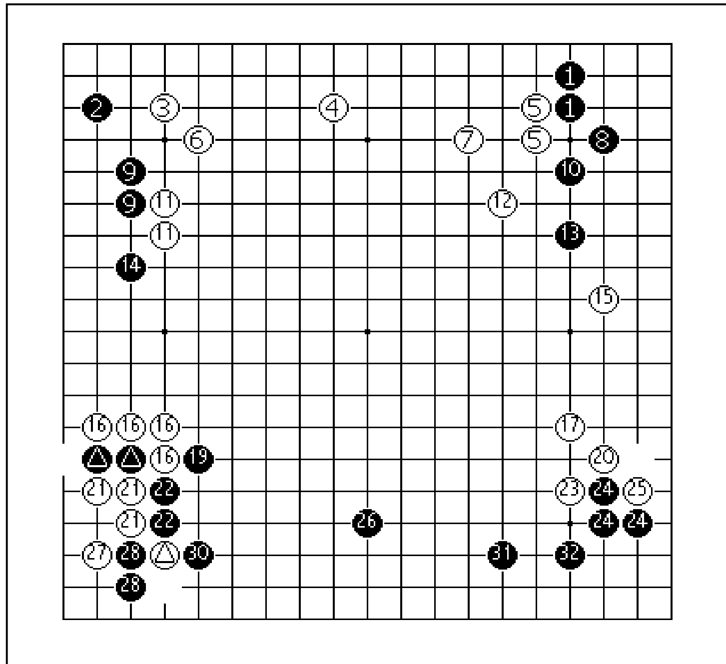
## 2.2 Blocks and capturing

A block, also called a string, is a set of adjacent stones of one colour. According to the rules of Go, stones of a block are captured in unison when the block loses all its liberties (empty adjacent points). If we view a board configuration as a graph, with stones as nodes and any two vertically-adjacent or horizontally-adjacent stones of the same colour determining an edge, then a block is a component of this graph. A depth-first search-based graph-component algorithm can be used to identify blocks efficiently.

The capturing status of a block can be classified into three categories.

- No danger – the other side will not be able to capture it.
- Critical – if the other side plays first, it can be captured; if the block side is to play first it can be secured.
- Dead – there is no way to escape from capturing even if the block side plays first.

A Go program typically uses capturing-tactic search routines to determine the status of each block (see Section 3). In order to save time, blocks with many liberties, say 5 or more, are automatically classified as “no danger” in most Go programs. Some programs use heuristics to help the block-status classification. A ladder routine can be used to find a consecutive-atari way to capture a block with two liberties. It has an average branching factor near 1, so it is extremely fast. A ladder routine should be an essential module of any Go program.



In Figure 2 the stones of each block are marked with a block identifier (a number ranging from 1 to 32). Each critical block has an associated capture point (c) and escape point (e), if they are the same the common point is marked with d. Block 25 is critical. Dead blocks are marked by triangle marks on their stones, each dead block has a capture point but no escape point.

**Figure 2:** Blocks and capturing statuses.

### 2.3 Chains and connectivity

A chain is a collection of inseparable blocks of the same colour. There are three ways to recognize the connectivity of two blocks: by heuristics, pattern matching, and search. A short description of each of them follows below.

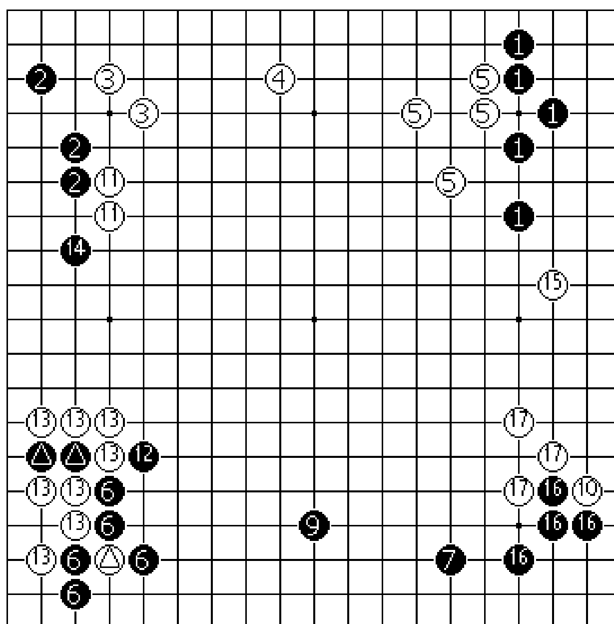


**Figure 3:** Chains: Chain 3 is recognized from two common liberties of its blocks. Chain 17 is recognized from a protected common liberty of its blocks. Chain 6 is recognized from the dead stone. Chain 5 matches a connection pattern. Chain 2 is decided by a connection search.

**Heuristics** - If two blocks have two or more common liberties or share one protected liberty, they are in the same chain. Blocks adjacent to a dead opponent block are in a chain too. See Figure 3.

**Pattern matching** - Connection patterns can be used to recognize connectivity. Figure 4 shows some common simple connection patterns.

**Search** - Goal-oriented local-connection search can be used to decide the connectivity (more in Section 3).

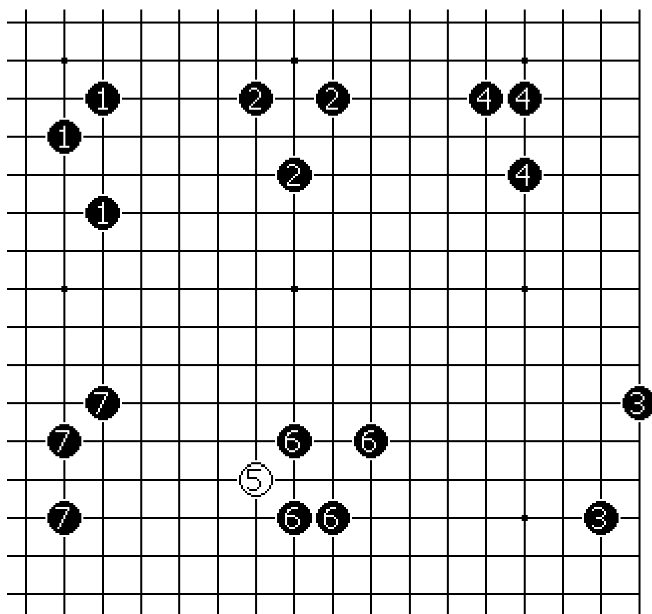


**Figure 4:** Some simple connection patterns - note that pattern 3 is only valid at the edge.

## 2.4 Groups and safety

A group is a strategic unit of an army of stones. It consists of one or more chains of the same colour plus the dead opponent blocks, called the prisoners. The chains and dead blocks are connected through empty points[4] that have an influence above a certain threshold. For details, we refer to Chen (1989). In Figure 5 six groups are identified (see their group id number).

Every stone not in a dead block radiates influences across the board. This influence is at a maximum in its immediate neighboring spaces and decays as the distance increases. Many programs use influence methods. The influence propagation is programmed differently though. For instance, GO INTELLECT uses exponential decay with a distance reduction factor of ½, MANY FACES uses a decay of 1/distance.

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

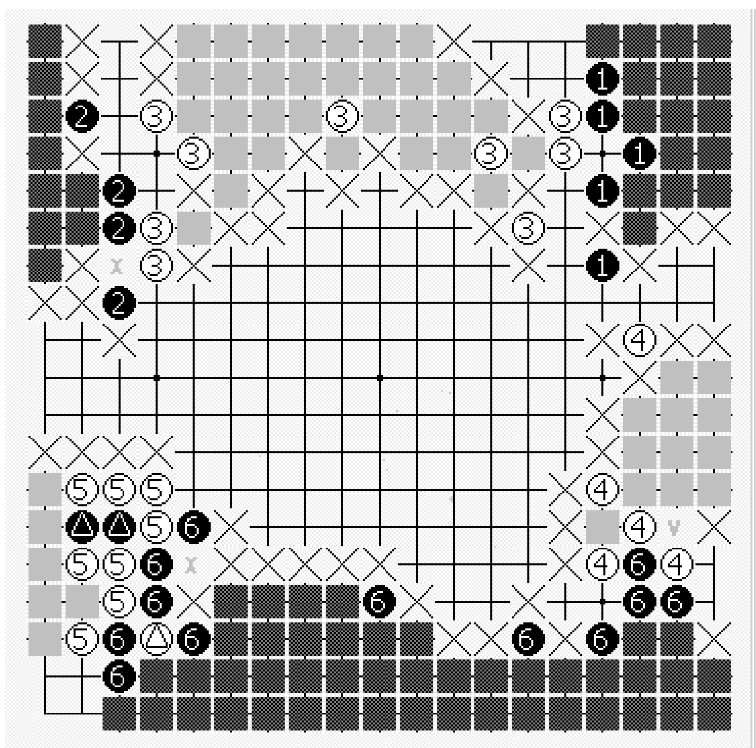


**Figure 5:** Groups: Stones are marked by a group id number. Prisoners are marked by triangles. Spaces are in shade. Frontier spaces are marked by X.

Once groups are identified, the numbers of eyes they have can be decided via (a) heuristics, (b) patterns, or (c) life-and-death search.

a. **Heuristics** – Chen and Chen (1999) present a classification of group space points into three types: "full eye-point", "partial eye-point", and "false eye-point". They are based on diagonal-index rules and heuristic-downgrading rules. Let us call a point on the Go board a controlled point, if one of the following four conditions applies: (1) it is occupied by one of our live stones, (2) it is occupied by an opponent's dead stone, (3) it is a border point, (4) it is one of our controlled empty points. An empty board point is said to be a non-controlled point if it is accessible by both sides. The diagonal index of a board point is the number of our controlled points, plus 1/2 times the number of non-controlled empty points of its four diagonal neighbors. We can make an initial classification of all space and prisoner points of a group as follows. If the point is in line 1 (edge), a full eye-point requires a diagonal index 4, a false eye-point has a diagonal index 3 or less, otherwise it is a partial eye-point. If the point is above line 1, a full eye-point requires a diagonal index 3 or more, a false eye-point has a diagonal index 2 or less, otherwise it is a partial eye-point. Heuristic eye-point downgrading rules can provide the required adjustments. For instance, the most common downgrading rule is as follows.

*Downgrading Rule 1:* For each false empty eye-point, the adjacent empty eye-point should be downgraded by one level.

An example is shown in Figure 6. The b at the bottom line is a false eye-point, so its neighbor empty point should be downgraded from a full-eye to a half-eye. For a full set of heuristic downgrading rules we refer to Chen and Chen (1999).

A set of adjacent eye-points is referred to as an eye-region. Each eye region's eye number can be estimated heuristically. For instance, the number of eyes in an eye-region of full eye-points, without deficiency and prisoners, can be determined by the heuristic values shown in Table 1, in which external boundary means the surrounding points of the region[5].

---

[5] Mark Boon was the first person who drew the author's attention to the relationship between the length of external boundary and the number of eyes.

| Length of external boundary | 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 |

**Table 1:** Number of eyes in a perfect eye region.

For a good understanding we provide some examples. Knife-Five has an external boundary of length 9 and it contains a Square-Four, hence it provides 1.5 eyes. Straight-Four has an external boundary of length 10, it has 2 eyes. Straight-Three has an external boundary of length 8, it has 1.5 eyes.

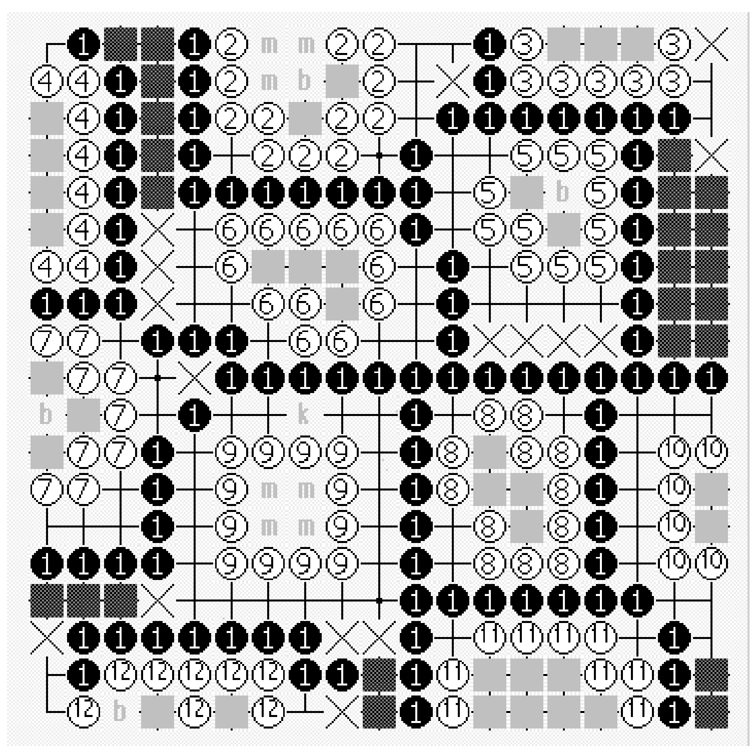


**Figure 6:** The number of eyes determined by heuristics.

Now we are well equipped to determine the numbers of eyes of groups on the board via heuristics. Let us consider Figure 6. The group 2's eye-region has an external boundary of length 9 and contains a Square-Four, so it has 1.5 eyes. Group 4's eye-region has an external boundary of length 10, it has 2 eyes. Group 5's eye-region has an external boundary of length 7, it has 1.5 eyes. Groups 7, 8, and 9 each has an eye-region with external boundary of length 8. The additional features listed in Table 1 determine the eye numbers. Group 9 is a Square-Four, hence 1 eye; group 8 is a Curved-Four, hence 2 eyes; group 7 is neither Square-Four nor Curved-Four, hence 1.5 eyes. The point beside b in group 12 is initially classified as a full eye-point, since it has a diagonal index 4. But b is a false-eye point, so it neighbor point is downgraded to a half eye-point. Group 12 has only 1.5 eyes. Readers can verify according to Table 1 and the downgrading rule that group 3 has 1.5+0.5 = 2 eyes. Group 10's eye-region has a rather short external boundary (length 6), so it has only one eye. Group 11's eye-region has an external boundary sufficiently long for 2 eyes.

b. **Patterns** – number of eyes of an eye-region can be revealed through pattern matching with eye-shape libraries. GNU Go and Golois use this approach.

c. **Life-and-death search** – search is usually more reliable than heuristics and/or patterns, but it is also more time consuming. Normally, search works better when the group is nearly completely surrounded; search is not so effective when the group is more open (Wolf, 1991, 2000).

The safety of a group can be determined by the following five factors: (1) its eye number, (2) its ability to expand (empty neighbors), (3) to run (freedom), (4) to connect to friendly groups, and (5) the safeties of adjacent opponent groups.

### 2.5 Territory and potential territory

Territory can be estimated by measuring interior spaces (spaces surrounded by grid points belonging to the same group) and prisoners (dead opponent stones), with an adjustment for the safety of the group. A group with a low safety should be counted as opponent's territory. The numbers of captured stones from each side are easy to keep track of and will be included in the scoring under Japanese rules. If Ing Chinese rules[6] are used, we can start with the 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 big fight. For simplicity, GOLIATH counts a grid point as Black, White, or Unsettled – a crisp 1, -1, or 0. Most Go programs use fractional numbers in an attempt to capture the uncertainty. For instance, GO INTELLECT uses 1/64 point as basic measuring unit and MANY FACES uses 1/50 point.

An even more difficult part of the evaluation is the 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 useful knowledge. Z. Chen (2000) suggests to count each extension of an outside stone as 3 points of a potential territory in the opening stage of a game. For a nearly completely surrounded weak group, other than almost dead or 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 obtain a reasonably good estimate of the potential territory. Today, static evaluation functions in Go programs are far from accurate. In a computer Go tournament, one can frequently observe that two competing (top) programs have game evaluations of the same board configuration with a difference of more than 30 points. Of course, a precise and accurate evaluation function is assumed to have the same complexity as the game of Go itself; therefore it is beyond any reach. Even creating an evaluation function that may be off mark occasionally and just produces a good approximation most of the time, seems still extremely difficult.

## 3. SEARCH

Search is a most powerful tool for computer games, including Go. We can use search to discover knowledge of capturing, connecting, eye-making, territory surrounding, etc. We can roughly divide search into local goal-oriented search and global move-selection search. We discuss them in the Subsections 3.1 and 3.2.

### 3.1 Local goal-oriented search

Local searches generate and test local moves for achieving a specific goal that can be measured locally. Below we provide a list of seven local characteristics with their specific local goals and their natural basic evaluations.

**Capturing**:
Goal - to capture one block
Evaluation – number of liberties of the block

**Ladder capturing:**
Goal - to capture one block by consecutive Ataris
Evaluation – Success (target block captured)
Failure (no more Atari moves)

**Semeai**:

[6] The rules are developed by the Chinese Go promotor Ing Chang-Ki. He offered a million-dollar prize for the first computer Go program that beat a human professional. He proposed the Ing rules which are more logical and scientific than the Japanese rules and the traditional Chinese rules. The Ing rules count both life stones on the board and secured empty spaces as points. (Japanese rules count only secured empty spaces minus own prisoners.)

Goal - to capture the opponent's block before the opponent captures our block for two adjacent blocks
Evaluation – number of relative liberties of the two blocks (when one block has an eye and the other block does not, the number of common liberties should only be credited to the side with an eye)

**Multi-blocks capturing**:
Goal - to capture one of a set of related blocks of the same colour
Evaluation – number of total weighted liberties (blocks with fewer liberties carry more weights)

**Connection**:
Goal - to connect two chains into one chain
Evaluation – distance between two chains

**Life-and-death (Tsume-Go)**:
Goal - to make two eyes for a group
Evaluation – number of eyes of the group

**Territory**:
Goal - to surround more territory
Evaluation – local territory estimate

The candidate move-generation should be highly selective using heuristic domain knowledge, among others based on the local characteristics. We discuss the strengths and weaknesses of the local search techniques implied above.

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 much too slow to be practically useful. Heuristic knowledge is needed to reduce the branching factor of the search. One such heuristic is that a liberty adjacent to a higher number of secondary new liberties should have higher priority.

In **ladder capturing search:** only Atari moves need to be considered. Usually one of the two Atari moves will give the target block three liberties after extension, hence there is no need to search further for that branch. It is essentially a linear time algorithm.

In **semeai search:** important questions are whether the two competing blocks of opposite colour have eyes[7] and what are the sizes of the eyes? An eye-space of size 4 to 6 and not of two-eye shapes (Flower-Six, Knife-Five, Square-Four, or T-Four) is called a big-eye. If one side has a big-eye, the other side has a small eye, size 1 to 3, or a smaller big-eye, or no eyes, the number of common liberties can only be credited to the side with a bigger eye in semeai. If the two sides both have an equally sized big-eye, or both have a small eye of any size 1 to 3 not necessarily of same size, then either side needs to discount its common liberties for winning the semeai, otherwise it is seki. If both sides have no eyes, either side can only count all common liberties (>= 1) as 1 for winning the semeai, otherwise it is seki. Flower-Six has 12 liberties; Knife-Five has 8 liberties, Square-Four or T-Four has 5 liberties. Small eye of size i, i = 1, 2, or 3, has i liberties. The number of prisoners in the eye-space should be subtracted from the liberty count. Müller (1999b) classifies all semeai problems into nine different classes.

For targets of **multi-blocks capturing search:** we can define an equivalence relation among blocks of the same colour with few liberties (say 3 or less) as follows. Two blocks are related if and only if they share a common liberty with two empty neighbors (or the common liberty is a valid cut). An equivalence class of this relation is a target set for multi-blocks capturing search. For instance, in Figure 7, blocks 3, 7 and 4 form a target set. A useful heuristic for multi-blocks capturing search is to give common liberties high priority.

In **connection search:** we count the length of a path joining two grid points as follows: if there is already a chain of our colour on the board, going through the chain is free. Moving to an adjacent empty point counts 1. Moving to a diagonal empty point with two connection options also counts 1. Moving along the path, the total count is the path length. The shortest path can be found using breadth-first search or priority-first search algorithms. The length of the shortest path joining two points is the distance of the two points. We can use empty points along the shortest path(s) as candidate moves for both sides in the connection search.

[7] Assume neither has two eyes, otherwise there is no semeai race.

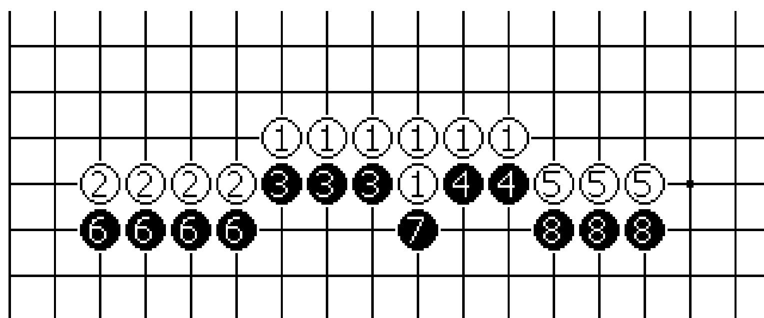


**Figure 7:** Blocks 3, 7 and 4 form a multi-blocks capturing target set through the transitive closure of sharing open common liberties.

In **life-and-death search:** there are four types of moves that deserve high precedence: capturing moves, separation moves, blocking moves, and diagonal moves. If an opponent surrounding block can be captured, capturing the opponent block may be the easiest way to live. If an empty grid point is adjacent to two or more full eye-points and those full eye-points do not have other common empty neighbors, playing at the separation point makes two eyes! Both blocking and diagonal moves can upgrade the status of eye-points.

In **territory-surrounding search:** knight moves, single jumps, big knight moves, double jumps are usually good candidate moves. Pattern libraries are useful in candidate-move generation.

The following algorithms are instances of useful search algorithms for local search in Go.

**Selective alpha-beta and its variations** – Forward pruning is generally used to speed-up the search. Iterative deepening is useful for time control. Partial depths[8] allows the search to look deeper in a certain key portion of the game tree during a fixed depth-bound search or iterative deepening. Genetic algorithms can be used to tune-up search parameters.

**Proof-number search** (Allis, Van der Meulen, and Van den Herik, 1994) – It is a best-first search procedure, which repeatedly selects and expands a "most-proving node" until the root node's game value is known. It is effective for a local Boolean-valued goal-oriented search when the branching factor is small. It needs to be combined with heuristic candidate-move selection to tackle more complicated tactical problems. Since it is a best-first search algorithm, the whole search tree needs to be kept in memory, which may cause memory problems.

**PN*-search algorithm** (Seo, Iida and, Uiterwijk, 2001) – It transforms a best-first Proof-number search into an iterative-deepening depth-first approach. It is a very promising search algorithm for life-and-death and other tactical problems in Go.

**Lambda search** (Thomsen, 2000) – Moves are classified into a monotone hierarchy $(\lambda^i)_{i = 0, 1, 2, \ldots}$, where $\lambda^0$ -moves are those moves that immediately achieve the goal, and $\lambda^{i+1}$-moves are those moves that, if ignored by the opponent, can achieve the goal by using $\lambda^i$ -moves only. It guarantees the correctness of the search result, but it will not be sufficiently fast to solve complex tactical problems in real time.

**Abstract Proof Search** (Cazenave, 2001) – This is a theorem-proving approach to games where a simple definition of the search goal can be given. Go tactic problems are natural candidates for this method if their abstract analyses are provided.

**Partial order bounding** (Müller, 2001) – It is a new way to evaluate terminal nodes in a game tree. This approach can be used with any game-tree search algorithm. It is effective for solving semeai problems involving "big eyes".

---

[8] Moves that are "active" may not be counted as a full ply, e. g., an Atari move may be counted as 1/8 ply in a capturing search.

### 3.2 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 needs to be selective on producing move candidates during the search process. Modified alpha-beta searches are normally used for this purpose. For instance, GO INTELLECT uses the global selective search to make its move decision (K.Chen, 1990, 1998, 2000). It has about 20 goal-oriented move generators, each of them 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 the candidate moves determines the move selection.

The evaluation functions of most Go programs are based on scoring. For each board point, we assign a value between –1 and 1; 1 means the point surely belongs us, -1 means it clearly is the opponent's territory. Fractional values are usually allowed. GO INTELLECT uses group relative-safety and influences to decide on the value assignment for each grid point. Adding the values of all 361 board points adjust by komi, we obtain a basic board evaluation.

B* (Berliner, 1979) and probability-based B* (Palay, 1982; Berliner and McConnell, 1994) are best-first search procedures which can handle a tree with a high branching factor well. For B* to work in Go, a tight optimistic bound and pessimistic bound needs to be obtained from node evaluation. Otherwise, the algorithm would not converge in real time. The Go program JIMMY uses a B*-like global search. The probability distribution required for probability-based B* is difficult to construct in Go.

## 4. MOVE-DECISION STRATEGIES

In this section, we discuss move-decision strategies as used by current Go programs. They can be roughly classified into four paradigms: 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 (Enzenberger, 1996) and pattern matching from a huge library of professional games (Zhang, 1990), but so far they failed to produce strong programs. We characterize the four paradigms in the Subsections 4.1 to 4.4.

### 4.1 Static analysis

Programs of the static-analysis paradigm do not perform any global search, but perform various goal-oriented local searches, such as capturing, connection, life and death. Since these local searches do not need to be performed for each node in a global search tree, they tend to be done more thoroughly. Each program has its own set of move generators to suggest candidate choices for global-move selection. The programs DRAGON, EXPLORER, and FUNGO are in this category. DRAGON uses a priority scheme. It divides all possible moves into 13 priority categories: capturing/escaping, urgent pattern, joseki, …, small yose. It selects the move in the highest non-empty priority category with the biggest move value provided by the related move generator(s). EXPLORER adds the values from all move generators for each point and then selects the point with a maximum sum to play. FUNGO uses the maximum value over all move generators for each point. It then selects 18 points with the highest values in order to perform more sophisticated and time-consuming tasks for the final move selection.

### 4.2 Try and evaluate

In the try-and-evaluate paradigm, first candidate moves are generated and then a thorough evaluation is performed with each candidate move tested in turn on the board. 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 single module. GNUGO is developed by many people jointly. Its source code is in the public domain: http://www.gnu.org/software/gnugo/devel.html.

GO4++ uses pattern matching to generate a large number, about 40 or so, of candidate moves together with a ranking. A thorough evaluation based on a connectivity probability map is done on each move candidate. MANY FACES performs a quiescence search for the evaluation of each candidate move. The search result is modified by the estimate of the opponent's gain if the opponent is playing locally first. Move generators in MANY FACES generate (reason, value) pairs for each candidate move. The maximum value is taken in a reason category for each point and values are added over different categories for a move point.

### 4.3 Global selective search

Many programs perform some variation of alpha-beta look-ahead with a 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 two separate global searches, one with urgent moves and the other with calm moves. GOMASTER converts everything to points in the evaluation including influence, thickness, sente, gote, … etc. JIMMY performs global selective search using B*-type upper and lower bounds associated with each move candidate.

### 4.4 Incentive/temperature approximation

Programs using the incentive/temperature approximation paradigm consider the consequences of each side playing first in a local situation. They include HANDTALK, GOEMATE, WULU, HARUKA, KCC IGO, GOLIATH, GOSTAR, GOLOIS, and STONE. The programs HANDTALK, GOEMATE, and WULU use the sum of the move values of both sides modified by "move efficiency" to decide on the move to play. Local searches are performed but no global look-ahead. HARUKA performs 1 to 4 general local searches, called main searches, each with about an 10*10 scope, a search depth of 3 to 5 plies, and a width of 6 to 9 moves. KCC IGO first identifies critical areas, then performs local searches with candidate moves mostly taken from pattern matching. GOLIATH performs two local searches for each critical area, one for each 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). STONE contains a set of tactic move generators and a set of position move generators. Each move generator has a temperature predictor and a move searcher. A move generator may find itself applicable to one or more (or none) sub-games (sub-regions). For each applicable sub-game, the temperature predictor heuristically produces a maximum temperature (upper bound) and a minimum temperature (lower bound). For the sub-game with highest maximum temperature, the associated move searcher will be invoked. The move searcher will find the best move for the sub-game and the temperature of the sub-game, which will replace the sub-game's old maximum and minimum temperatures. This process is repeated until a lower bound is greater than or equal to the rest of the upper bounds and the best move of that hottest sub-game will be chosen.

## 5. CONCLUSIONS

There is no single paradigm that really dominates the other paradigms. The performance of a program has a strong relation to the implementation effort – especially the question of how much useful Go knowledge the programmer has input into the program is relevant. To follow good programming practice is of utmost importance, so that the program can be modified/improved conveniently over time.

*Decomposition search* (Müller, 1999) involves many local searches, but the entire process is for global move selection. It has a sound theoretical foundation in combinatorial game theory (Berlekamp, Conway and Guy, 1982). It consists of roughly the following four steps: (1) game decomposition and subgame identification, (2) local combinatorial game searches, (3) simplification of combinatorial game expressions, and (4) sum game play. Unfortunately, except for late-stage endgames only, a mutually-independent decomposition of the board is impossible. We therefore propose a modification called *soft decomposition search* (Chen, 2002). The idea is 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. We do not perform a mutually-independent decomposition of the Go board, which is only possible in a very late endgame stage, instead we try an approximation of the decomposition. This hopefully can produce a better incentive/temperature approximation.

Most of the global search performed by Go programs today is based on the evaluation of the points. But the reward function of a 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 logical and more useful for the static evaluation function to estimate the probability of winning, instead of the expected number of winning points. For instance, a program is leading comfortably by 30 points and facing a choice whether to enter a fight with 50 points swing. Assume that the program has 90 percent chance to win the fight. A point-based evaluation function will lead the program into the fight, since it increases the expected value of winning from 30 points to 70 points. A winning-probability-based evaluation function will lead the program to avoid the fight, since entering the fight would reduce the winning probability from near 100 percent to 90 percent. The author is currently developing such an evaluation function and strategy.

## 6. ACKNOWLEDGEMENTS

In Fall 2001, the author conducted a survey on move-decision strategies of the well-known Go programs in the world and received enthusiastic responses from most program authors. These responses made Section 4 of this paper possible. The author would like to thank the following computer Go researchers and programmers for their survey responses: Zhixing Chen (HANDTALK, GOEMATE, 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 Müller (EXPLORER), Bruno Bouzy (INDIGO), Tristan Cazenave (GOLOIS), Jimmy Lu (GOSTAR), Yong-Goo Park (FUNGO), Jee Won Ho (GOMASTER), and Jimmy Yen (JIMMY), Kuo-Yuan Kao (STONE). The author would also like to thank H. J. van den Herik, Editor-in-Chief of ICGA Journal, for editing this paper.

## 7. REFERENCES

Allis, L. V., Meulen, M. van der, and Herik, H. J. van den (1994). Proof-number search, *Artificial Intelligence*, Vol. 66, No.1, pp. 91-124. ISSN 0004-3702.

Berlekamp, E.R., Conway, J.H., and Guy, R.K. (1982). *Winning Ways,* Academic Press Inc., London. ISBN 0-12-091101-9.

Berliner, H. (1979). The B* tree search algorithm, a best-first proof procedure, *Artificial Intelligence*, Vol 12, No.1, pp. 23-40. ISSN 0004-3702.

Berliner, H. and McConnell, C. (1994). B* Probability Based Search. Report CMU-CS-94-168.

Boon, M. (1989). A Pattern Matcher for Goliath, *Computer Go*, No. 13, pp. 12-23.

Bouzy, B. and Cazenave, T. (2001). Computer Go: An AI oriented survey, *Artificial Intelligence*, Vol. 132, No. 1, pp. 39-103. ISSN 0004-3702.

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

Cazenave T. (2001). Abstract Proof Search, *Computer and Games 2000*, Second International Conference, CG2000 (eds. T. A. Marsland and I. Frank). LNCS 2063, pp. 39-54. Springer-Verlag, Berlin. ISBN 3-540-43080-6.

Chen, K. (1989). Group Identification in Computer Go, *Heuristic Programming in Artificial Intelligence*, (eds. D.N.L. Levy and D. 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 *Proceedings of Joint Conference on Information Sciences* '98, Vol. II, pp. 274-278.

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). Some Practical Techniques for Global Search in Go, *ICGA Journal*, Vol. 23, No. 2, pp. 67-74.
Chen, K. (2001a). A study of decision error in selective game tree search, *Information Sciences*, Vol. 135, No.3-4, pp. 177-186. ISSN 0020-0255.

Chen, K (2001b). Knowledge and Search in Computer Go, Proceedings of the 6th Game Programming Workshop (GPW 2001) (ed. H. Iida), pp. 94-101. IPSJ Symposium Series, Vol. 2001, No. 14. ISSN 1344-06401.

Chen, K. (2002). Soft Decomposition Search in the Game of Go, to appear.

Chen, Z. (2000).*The small world of computer Go* (in Chinese). Zhongshan University Publisher.

Enzenberger, M. (1996) The integration of a priori knowledge into a Go playing neural network, http://home.t-online.de/markus.enzenberger/neurogo.html.

Kao, K.(1997). *Sums of Hot and Tepid Combinatorial Games*. Ph.D. thesis, University of North Carolina at Charlotte.

Kao, K. (2000). Mean and temperature search for Go endgames, *Information Sciences*, Vol. 122, No. 1, pp. 77-90. ISSN 0020-0255.

Kierulf, A, Chen, K., and Nievergelt, J. (1990). Smart Game Board and Go Explorer: A study in software and knowledge engineering, *Communications of ACM*, Vol. 33, No. 2, pp. 152-166. ISSN 0001-0782.

Lichtenstein, D. and Sipser, M. (1980). Go is polynomial-space hard, *Journal of ACM*, Vol. 27, No.2, pp. 393-401. ISSN ...

Müller, M. (1995). *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. Ph.D. Dissertation, Swiss Federal Institute of Technology Zurich.

Müller, M. (1999a). Decomposition Search: A combinatorial games approach to game tree search, with applications to solving Go Endgames, *IJCAI-99*, pp. 578-583. Morgan Kaufmann, Sun Meteo, Ca. ISBN 1045-0823.

Müller, M. (1999b). Race to capture: Analyzing semeai in Go. In Game Programming Workshop in Japan '99, volume 99(14) of IPSJ Symposium Series, pages 61-68, 1999.

Müller, M. (2001). Partial order bounding: A new approach to evaluate in game tree search, *Artificial Intelligence*, Vol. 129, Nos. 1-2, pp. 279-311. ISSN 0004-3702.

Palay, A. (1982). The B* Tree Search Algorithm – New Results, *Artificial Intelligence,* Vol. 19, No. 2, pp. 145-163. ISSN 00004-3702.

Robson, J.M. (1983). The Complexity of Go. *Proceedings IFIP*, pp. 413-417.

Seo, M., Iida, H., and Uiterwijk, J.W.H.M. (2001). The PN*-search algorithm: Applications to tsume-shogi, *Artificial Intelligence*, Vol. 129 Nos. 1-2, pp. 253-277. ISSN 0004-3702.

Thomsen, T. (2000). Lambda-Search in Game Trees – with Application to Go, *ICGA Journal*, Vol. 23, No. 4, pp. 203-217.

Wolf, T. (1991) Investigating Tsumego Problems with "RisiKo". *Heuristic Programming in Artificial Intelligence 2* (eds. D. Levy and D. Beal), pp. 153-160. Ellis Horwood, pp. 153-160, UK.

Wolf, T. (2000). Forward pruning and other heuristic search techniques in tsume Go, *Information Sciences*, Vol. 122, No. 1, pp. 59-76. ISSN 0020-0255.

Zhang, Y. (1990). Personal communication.