Computer Go

Martin Müller
NTT Communication Science Laboratories
Atsugi, Japan
mueller@rudolph.brl.ntt.co.jp

Abstract

Computer Go is maybe the biggest challenge faced by game programmers.

1 Introduction

The introduction briefly describes the rules of the game, and the history of computer Go. Section 2 gives an overview of the current state of the art, and discusses the many challenges posed by this research domain. Section 3 surveys the different kinds of knowledge built into a heuristic Go program. Applications of minimax game tree search in Go are described in Section 4, and Section 5 discusses subproblems of the game for which specific solution techniques have been developed. The final Section 6 poses challenge problems for further research in the field. A glossary contains brief definitions for technical terms that are marked by a star in the text, as in: *komi**. There is an extensive but by no means exhaustive bibliography.

1.1 The Game of Go

Go is played between two players Black and White, who alternatingly place a stone of their own color on an empty intersection on a Go board, with Black playing first. The standard board size is 19×19, but smaller sizes such as 9×9 and 13×13 are often used for teaching or fast games. The goal of the game is to control a larger area than the opponent. Figure 1 shows the opening phase of a typical game.

The capturing rule states that if stones of one color have been completely surrounded by the opponent, so that no adjacent empty point remains, they are removed from the board. Figure 2 shows two white stones with a single adjacent empty point (liberty) at 'a'. If Black plays there, the two white stones are captured and removed from the board. If White plays on the same point first, it will now require Black three moves at 'a', 'b' and 'c' to capture the three stones. Capturing and recapturing stones can potentially lead to the infinite repetition of positions. The *ko* rule* prevents that. A basic ko is shown

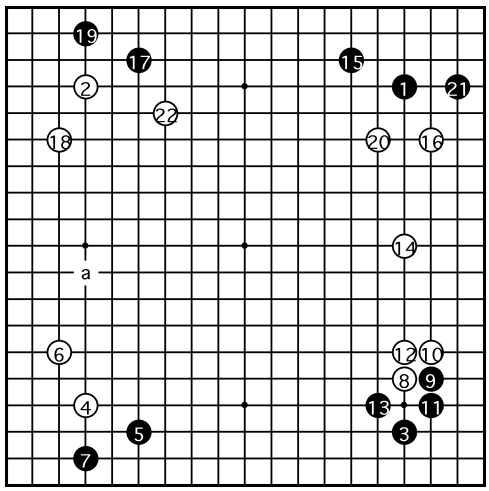

Figure 1: The game of Go

in Figure 3. Rules differ slightly in their treatment of rare, more complex cases of repetition. Players can pass at any time; two or three consecutive passes end the game. Differences in playing strength can be balanced by a handicap system, which allows Black to place several stones in a row at the start of the game.

2 State of Computer Go

Of all games of skill, Go is second only to chess in terms of research and programming efforts spent. Yet in playing strength, Go programs lag far behind their counterparts in almost any other game. While Go programs have advanced considerably in the last 10-15 years, they can still be beaten easily by human players of moderate skill. The Ing foundation's million dollar price for a professional level Go program, which expires this year, will of course go unclaimed. However, the prize has already served its purpose, by giving a great boost to the popularity of Go amongst games programmers and researchers.

At the time of this writing (summer 2000), the future of big, world championship caliber events is uncertain. The FOST cup has been cancelled for lack of funding this year, and the traditional Ing tournament will stop altogether. However, small-scale computer Go tournaments continue to be held in Asia, Europe, North America and on the internet [53]. While human Go players are still concentrated mainly in Asia, Computer Go has become a truly international activity, with serious programs being developed all over the world. After the world championship-level performance of programs in games such as chess, checkers or Othello, there is an increasing interest in Go as the final frontier of

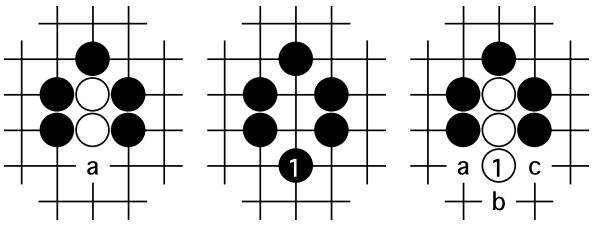

Figure 2: The capturing rule

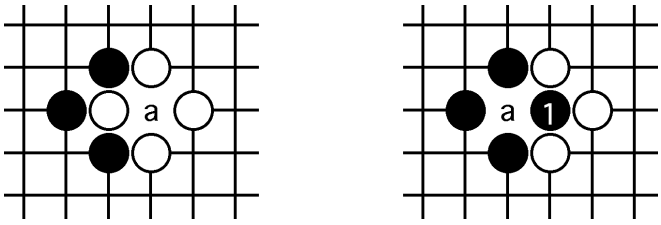

Figure 3: Ko

computer game research.

2.1 The Challenge of Computer Go

Computer Go poses many formidable conceptual and technical challenges. Most competitive programs have required 5-10 person-years of effort, and contain 50-100 modules dealing with different aspects of the game. Still, the overall performance of a program is limited by the weakest of all these components. The best programs *usually* play good, master level moves. However, their performance level over a *full* game can be much lower.

The following two games illustrate both sides of the story: on one hand, programs can look quite proficient in the right kind of games, but on the other hand they can lose games even with a ridiculous number of handicap stones against a human player who knows how to exploit their weak points.

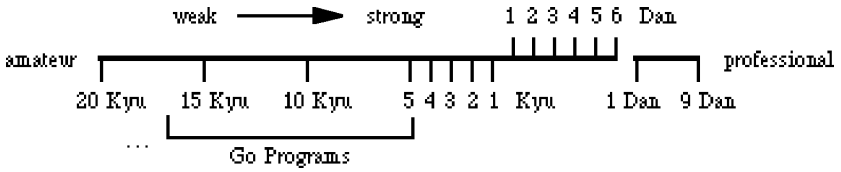

Figure 4: Go programs on the human ranking scale

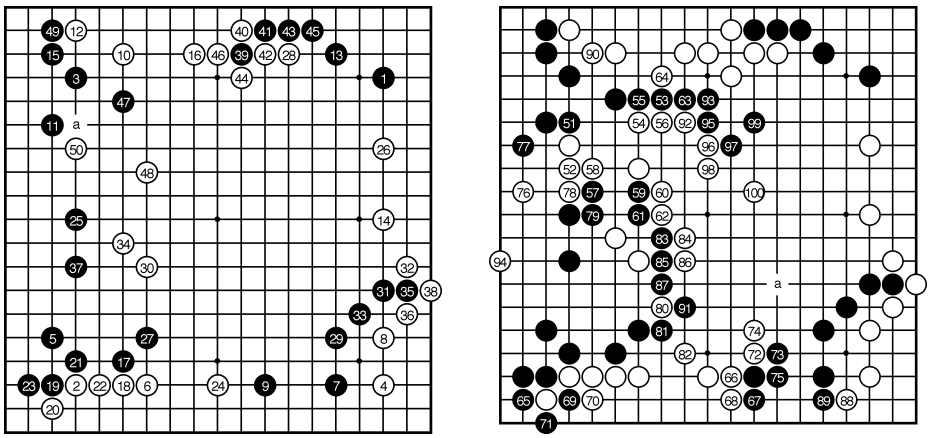

Figure 5: Ing Cup 1999: Goemate (B) - Go4++ (W), moves 1-100

2.1.1 Case Study 1: Ing Cup 1999, Go4++ vs Goemate

Figures 5 and 6 show the deciding game of the 1999 International Computer Go Congress, informally called the Ing cup, played in Shanghai on November 13. Playing white and receiving a *komi** of 8 points, *Go4++* by Michael Reiss won by 13 points over *Goemate*, developed by Chen Zhixing as the successor program of *Handtalk*. This game develops in a tight territorial fashion typical of most current top programs, with little fighting going on. Up to 16, both follow standard opening principles by first surrounding the corners and then expanding to the sides. The *joseki** moves from 16 to 23 are most likely contained in the opening book of both programs. With move 30, *Go4++* starts erasing the large framework that Black has built on the left side. Black invades strongly at 31 and 39, but later lets White connect. Anyway, the result is not bad for Black in both cases. White 52 is a strange shape move, but it succeeds in splitting up Black's left side. Around move 63 the game has already become an endgame contest.

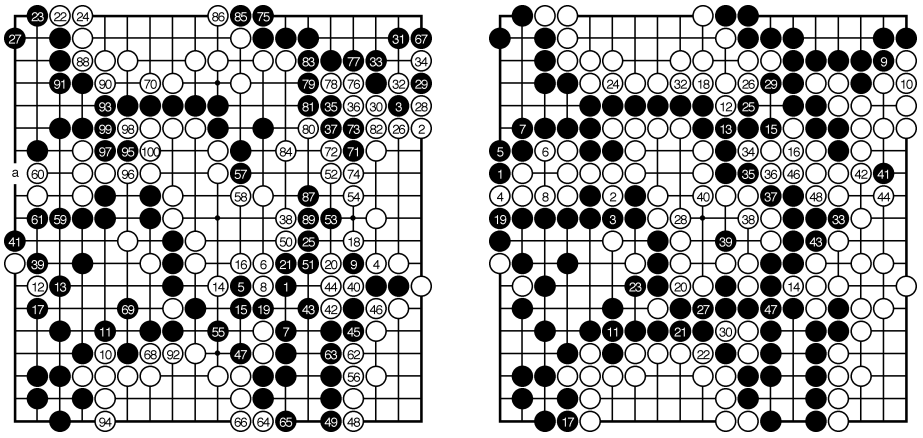

Figure 6: Ing Cup 1999: Goemate (B) - Go4++ (W), moves 101-248. 231 captures below 230, 245 pass

Black 75 is too passive. White 76 threatens to destroy the bottom left side, but Black fails to defend, allowing successive moves at 94 and 112. However, in return Black captures some white stones in the center in the sequence from 83, and until move 130 the game stays very close. 131 is an inexplicable retreat and must be caused by a programming bug. Of course Black should just connect at 132. Up to 137 Black loses more than 10 points. The remaining endgame is uneventful, and White achieves a safe win.

The performance of both programs in this game is very respectable. Their play is rather simple and safe, mostly surrounding territory. While there is still a large number of less-than-optimal moves, there are few really big mistakes. Both programs demonstrate an understanding of many aspects of Go. For example, they can build safe territory as well as large frameworks, and can react early to reduce an opponent's sphere of influence. The programs clearly incorporate important principles of Go, they don't just apply rote patterns as many of the early programs did. This is evident in situations such as move 52, where they program has insufficient detail knowledge to produce a stylish move, but is still able to select a play in the right general area. Programs are also careful to avoid getting weak groups, and play a reasonable endgame. In this game, White is especially skilful in eliminating the opponent's potential *sente** moves.

This game shows computer Go at its best. However, it cannot be denied that the style of play, which is typical of recent tournament games, hides much of the inherent complexity of Go. In contrast, in the next case study another top program is subjected to a more severe test.

2.1.2 Case Study 2: Giving Many Faces a 29 Stone Handicap

In August 1998, at the US Go congress in Santa Fe, New Mexico, a 29 stone handicap game was played between the program *The Many Faces of Go* and the author, with a total of 30$ in side bets riding on the outcome. Like *Go4++* and *Goemate*, *Many Faces* is one of the strongest Go programs in the world, and a few months afterwards it won the 1998 Ing cup. *Many Faces* is regarded as one of the best programs when it comes to tactical fighting. However, in this game its aggressiveness backfires. Despite the huge handicap, the game results in a six point win for the human.

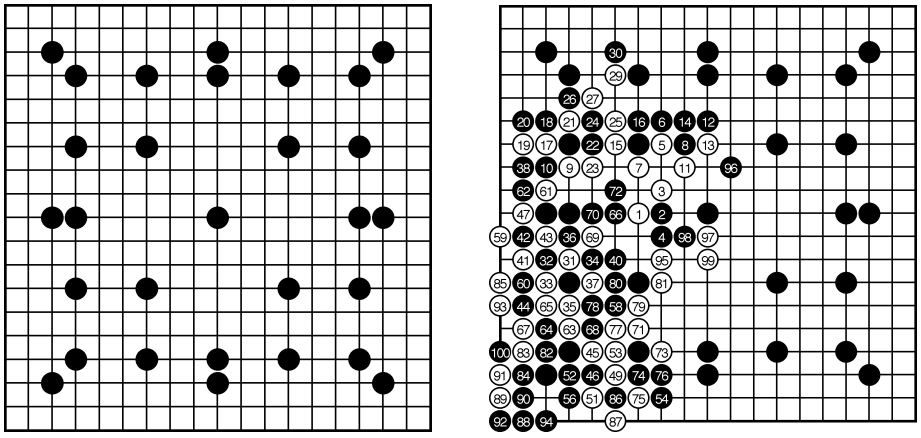

Figure 7: Many Faces (B, 29 stones handicap) - Martin Müller (W), moves 1 - 100. 28 at 21, 39 at 31, 48 at 32, 50 captures below 31, 55 at 31, 57 at 43

Figures 7 and 8 show the starting position and the game record. In the beginning, White sprinkles some stones around the board to probe for weaknesses, but Black defends well. In the bottom left, White uses a confused *ko** fight to get one living group, and continues from this basis, managing to reduce Black's group in the corner to one eye. Black spends too many moves trying to rescue this group in a counterattack against the white stones floating in the center. In the end, White manages to isolate another black group in the lower right corner, and kill it by a nice combination that exploits a hidden dependence between two seemingly safe eye areas. This second big capture makes the game very close, and White easily overtakes Black in the remaining endgame. Throughout this game, most of Black's moves are quite reasonable, but there are just enough mistakes to allow White to grind out a win.

Conceptually, Black's main problem seems to be that the program tries to fight it out with White on even terms, instead of preserving some of its huge initial advantage by playing slow, ultra-safe moves. One might argue that a

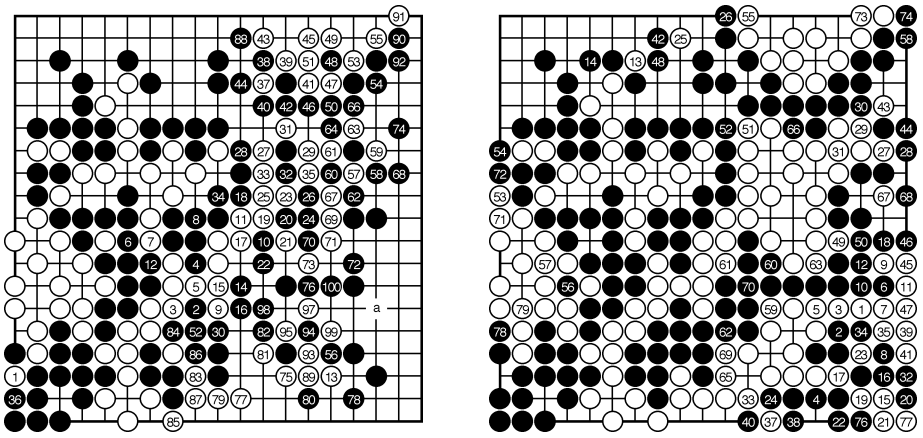

Figure 8: Moves 101-200, 201-279: 165 at 157, 196 connects below 184, 236 at 223, 264 connects right of 256, 275 pass

program playing a safer, more territorial style might be harder to overcome. However, a few weeks before this game, in an exhibition match at the AAAI conference professional player Janice Kim had already beaten *Handtalk* on a similar huge handicap [40].

2.1.3 Assessing the State of the Art

Judged by human standards, play of current program looks 'almost' reasonable, but certainly not impressive. Maybe only someone who has tried to write a Go program him- or herself can fully appreciate the enormous amount of ingenuity and hard work that has gone into building the state of the art systems. It is an indication of just how hard a problem Go is that despite all efforts, the level of programs is not higher than it is today.

2.2 Overview of the Development of Computer Go

2.2.1 Recent Development of Computer Go Programs

While Go programming started in the late sixties, it got a big boost in the mid eighties, with the appearance of affordable PC's on one hand, and of tournament sponsors such as the Ing foundation on the other hand. In early tournaments, Taiwanese programs such as *Dragon* [23] were successful. From 1989-91, Mark Boon's *Goliath* [5] dominated all tournaments, followed by Ken Chen's *Go Intellect* [13, 14, 16, 15] and Chen Zhixing's programs *Handtalk* and *Goemate* [16]. In recent years, David Fotland's *Many Faces of Go* [21], the controversial North Korean program *KCC Igo* and the current top program *Go4++* by Michael

Reiss have also won major tournaments. In total there are about 10 top class programs, including *Haruka, Wulu, FunGo, Star of Poland* and *Jimmy* [82]. Most of these programs are developed and distributed commercially. A step behind the top 10 is a set of about 30 medium-strength programs, often written by university researchers or dedicated hobbyists. Program authors include many strong amateur Go players, and even one professional 9-dan Go player, Tei Meiko. An interesting recent phenomenon is the appearance of open source programs such as the new *GnuGo* [7]. The total size of the computer Go community can be estimated at about 200 programmers, and is growing steadily.

Several milestones have been reached in the short history of computer Go: In 1991, *Goliath* won the yearly playoff between the Ing cup winner and three strong young human players for the first time, taking a handicap of 17 stones. *Handtalk* won both the 15 and 13 stone matches in 1995, and took the 11 stone match in 1997. Programs such as *Handtalk* and *Go4++* have also achieved some success in even games against human players close to *dan** level strength. However, as demonstrated in the previous section, experienced human players can still beat all current programs on much more than 11 stones. *Handtalk* was successively awarded 5, 4 and 3 *kyu** diplomas by the Japanese Go Association *Nihon Kiin* after winning the 1995-97 FOST cups, and *KCC Igo* received a 2 kyu diploma after its success in 1999.

In Japan, in recent years there has been an enormous increase in the number of Go software packages on the market. At an informal survey carried out in 1999 by the author, a local medium-size software store carried no less than 27 Go-related titles, with prices ranging from $5 - -100\$$. All but a few of these packages contained a playing engine.

2.2.2 Literature about Computer Go

The literature on computer Go and relevant topics has grown to the point where it is becoming difficult to read it all. This survey contains a long but by no means exhaustive bibliography.

The development of computer Go has been documented in a number of previous surveys. Wilcox [74] has written extensively about the early US-based Go programs in the seventies and eighties by Zobrist [86], Ryder [60], and by Reitman and Wilcox [54, 56, 55, 73, 74]. Important early papers on computer Go are collected in [38]. Kierulf's Ph.D. thesis [28] contains references for most of the programs that participated in the early computer Go tournaments of 1985-1989, and describes details of the *Smart Go* - *Explorer* - *Go Intellect* line of programs. Erbach [20] gives a good overview of the state of the art in the early nineties. Based on their contacts with the program authors, Burmeister and Wiles have published detailed descriptions and comparisons of several modern Go programs such as *Go4++, Handtalk, Many Faces of Go*, and *Go Intellect* [9, 10].

Information about computer Go programs and tournaments is available from many interconnected web sites, including a computer Go section on the American Go Association's site, *www.usgo.org/computer*, maintained by the author.

An extensive bibliography of online papers about computer Go by Markus Enzenberger is currently located at *home.t−online.de/home/markus.enzenberger /compgo_biblio.html*.

Ph.D. theses about computer Go started appearing 30 years ago, and are recently published at a rate of about one per year [6, 12, 22, 24, 28, 32, 37, 43, 57, 59, 60, 62, 86].

2.3 Go Research in Related Fields

Go has been used as the topic for research in related fields such as cognitive science, software engineering and machine learning [11, 8, 12, 18, 29, 33, 59, 61, 62, 63, 85, 74]. Starting from only the rules of the game, learning programs can pick up basic Go principles, such as saving a stone from capture, or making a one point jump. Methods for learning patterns from master games are discussed in Section 3.1.

A study by Enzenberger [19] describes the integration of a priori knowledge from expert Go modules into a neural network program. Many further tasks such as automatically tuning and expanding an existing knowledge base, or learning new high-level concepts, remain as future challenges.

There are many related books, papers and theses in the field of combinatorial game theory, including [3, 4, 2, 27, 30, 41, 50, 51, 44, 52, 66, 80, 81, 83].

2.4 Some Facts that make Go a Difficult Game for Computers

2.4.1 Differences Between Programs for Go and Other Games

The large search space caused by the great number of possible moves and by the length of the game is often cited as the main reason for the difficulty of Go. However, as Ken Chen points out [16], even 9×9 Go, with a branching factor comparable to chess, is just as difficult as full 19×19 Go, and current Go programs are by no means stronger on a small board than on a big one.

The biggest difference between Go and other games is that static evaluation is orders of magnitude slower and more complicated. Moreover, a good static full-board evaluation depends on performing many auxiliary local tactical searches. In this respect Go can be compared to the single-agent puzzle of Sokoban. Andreas Junghanns' Sokoban solver *Rolling Stone* [26] derives much of its strength from auxiliary searches that solve or approximately solve subproblems, and thereby help to speed up the main search by many orders of magnitude.

2.4.2 Size and Structure of the Problem Space

The search space for 19×19 Go is large compared to other popular board games. The number of distinct board positions is $3^{19\times 19} \approx 10^{170}$, and about 1.2% of these are legal [72]. Such numbers are often cited as the size of the

search space. Strictly speaking, the number of distinct game positions is very much larger because Go rules forbid position repetition. To detect and prevent illegal moves, state information must contain the complete move history, which enormously increases the number of distinct states.

No simple yet reasonable evaluation function seems to exist for Go. This claim is evident to serious students of the game, and is confirmed in theory by the fact that many difficult combinatorial problems can be formulated as Go problems [39, 42, 58]. The situation on a Go board can be extremely chaotic, leaving exhaustive analysis as the only known method of solution. However, a typical position reached during a game is much more regular. In some types of complicated-looking endgame or *semeai** positions, even perfect play is possible with little search and a good theory. This aspect of Go is maybe best exemplified by Berlekamp's endgame studies [4]. To play well in real-game situations, a Go program must therefore combine good theoretical foundations with lots of computing power.

2.4.3 Quality and Quantity of Human Knowledge

Human professional players are amazingly good at Go. They are able to recognize subtle differences in Go positions that will have a decisive effect many moves later, and can reliably judge very early whether a large, loose group of stones can be captured or not. Such judgment is essential for good position evaluation in Go. In contrast, obtaining an equivalent proof by a computer search seems completely out of reach. Skilled players usually know which side is better in a game after a quick glance at the position, and have no trouble reading out sequences many moves ahead. We can contrast Go with other popular games such as Awari, checkers or chess: Humans often get lost in the 'combinatorial chaos' of the game and miss a tactical combination, while machines are able to exploit their superior computing power.

A huge quantity of Go knowledge has accumulated over centuries, much of it implicit in the game records of master players. Thousands of game commentaries, tutorial books and problem collections have been compiled. Pattern knowledge of experts seems at least comparable to that of chess experts, which is already daunting [8, 17, 85]. Patterns recognized by humans are much more than just chunks of stones and empty spaces: Players can perceive complex relations between groups of stones, and easily grasp fuzzy concepts such as 'light' and 'heavy' stones. This visual nature of the game fits human perception but is hard to model in a program. The cognitive models of Reitman and Wilcox [56] were interesting, but today's programs make do with comparatively simple pattern matching [5, 43]. While the knowledge of chess programs is tuned nicely to their searching power, Go programs are still lacking in both quality and quantity of knowledge.

It is well-known amongst Go programmers that 'improving' a program by adding knowledge often makes it weaker in practice. Small changes to a Go program can also have large and unexpected effects, or lead a program into new types of positions that it cannot handle well.

3 Modeling and Representing Go Knowledge: Some Components of Heuristic Go Programs

There are two major ways of incorporating knowledge in a Go program: Patterns, described in Section 3.1, are a very direct way. On the other hand, a structured representation using a hierarchy of components can be used to obtain a more high-level description of a game state. Such representations are discussed in Section 3.2.

3.1 Patterns and Pattern Matching

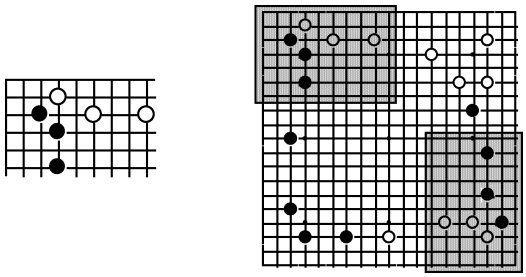

Figure 9: Corner pattern matching in two places

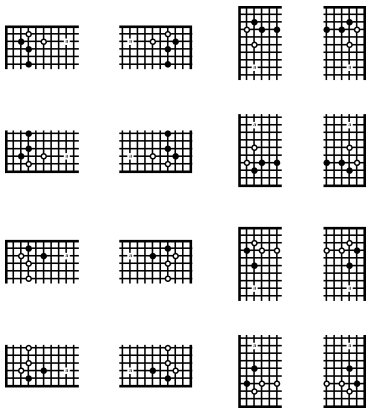

Figure 10: The 16 symmetric instances of a pattern

Patterns are a simple yet powerful way of encoding Go knowledge. Many types of moves, such as *joseki** and *tesuji** are described by patterns in the Go literature. Patterns can be applied in all stages of the game, from opening to endgame. Almost all Go programs contain a pattern database and a pattern

matching subsystem. Most pattern databases are still generated by hand, and contain several thousand patterns. However, methods have been developed that can automatically extract patterns from professional games [84, 32, 33, 34, 65, 67], and such a pattern-learning system is used in at least one of the top programs.

Many variants of pattern matching problems have been studied in computer science. In Go, a large number of 2-dimensional patterns must be compared to a single full board position in the 16 different ways illustrated in Figure 10. Each pattern can appear on the board in many different locations, in 8 different orientations and in two color combinations. It is computationally expensive to match each pattern against the board at each possible location at each turn. Filtering techniques based on hash tables [5, 21] or *tries* [64, 43] enormously reduce the set of candidate patterns that have to be compared with the current board position.

3.1.1 Definition of Patterns

A pattern used in a Go program typically consists of three parts:

- The *pattern map* indicates which points belong to a pattern, and which state among {*Empty, Black, White*} each point in the pattern is allowed to have. It is useful to distinguish between *corner, edge and center* patterns because the edge of the board affects many patterns.
- The *pattern context* specifies additional nonlocal constraints that an overall board position must satisfy to match the pattern. The most important constraints involve the *liberty** count of stones on the boundary of a pattern.
- The *pattern information* contains knowledge which can be applied if the pattern matches, such as good and bad moves or connection information.

3.1.2 Pattern Matching

A typical midgame situation produces about 500 matches from a 3000 pattern database. The effective running time of a pattern matching algorithm depends on a large number of factors:

- Number and size of patterns in database
- Size of the Go board
- Structure of the trie or hash table used for organizing the patterns
- Number of currently matching patterns
- Ratio of matches pruned by trie or hash table
- Speed of pattern-board comparisons, and of context checks

- Optimizations for incremental matching
- Low-level performance tuning

One optimization is very important for game play: a single matching or mismatching pattern typically depends on only a few points on the board. After making a move, most previous (mis-)matches stay valid. To exploit this fact, a dependency set can be kept for each match. After each move matches whose dependency set remains unchanged can be reused. This optimization is very effective in game play. In an experiment with the program *Explorer* on a test suite containing ten complete games, this optimization saved 96% of all center matches, 94% of edge matches and 93% of corner matches [43]. In contrary, on a set of 1000 unrelated positions, the savings were only 10%, 3% and 0.2% respectively. Several further optimizations for tree-based matching are described in [43].

3.2 Knowledge Representation

Go programs contain a number of standard components, which model Go concepts at different levels of abstraction. Selecting such components and choosing suitable representations are two of the main steps involved in building a Go program. Most of the basic concepts surveyed here were already developed by the pioneers Reitman and Wilcox more than twenty years ago. Of course, many refinements and variations have been tried since.

3.2.1 Foundation: Basic Data Types and User Interface

To support programming at a higher level of abstraction, it is useful to build a foundation of basic data types for a Go program, such as lists, trees, or hash tables. Many such toolkits are available for modern programming environments, or even part of the language standard such as STL for C++. A more specialized toolkit can handle data types such as game trees and useful functions such as a general purpose tree search engine and SGF Smart Go Format file input/output. Another indispensable tool for developing Go software is a graphical user interface, which can support development and debugging by a board display with markers and labels on points, tree navigation tools, an overview window showing several boards at the same time, or a tree view showing the structure of the game tree. Recently, several authors have interfaced their Go-playing engine to use the *CGoban* program as a graphical front-end. The *Smart Game Board* [28], which combines a game-independent toolkit with a graphical user interface, was the first and is probably still the most comprehensive such tool. A number of open source initiatives with similar goals have been started recently. An overview with many links is given on the GNU Go web page [7].

3.2.2 Go Board

A Go board is usually implemented as a one-dimensional array, as in Figure 11. Compared to a two-dimensional array, this allows faster access to a point

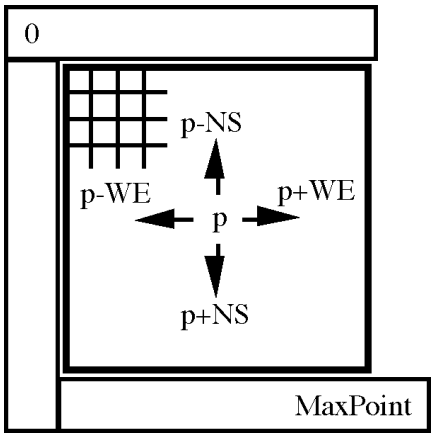

Figure 11: Embedding of board into a one-dimensional array, from [28].

without an implicit multiplication, and is more convenient because a point can be identified by a single number rather than by a pair of coordinates.

The neighbors of a point are calculated by adding or subtracting the constants NS (North-South) and WE (West-East). Given the row and column of a point in the range [1..MaxSize], the array index of the point (row, col) is NS*row + column. Each point on the board has a color *Empty, Black* or *White*, and all points off the board have the color *Border*. Boards smaller than the maximum size can use the upper left corner of the large board. The whole board is surrounded by a one-point border, so that all eight neighbors of any point are simply accessible without needing to check for array boundaries or stepping over to the other side of the board. The borders to the left and to the right of the board can be shared. The following code fragment implements a simple Go board.

```
const int MaxSize  = 19; // maximum board size
const int MaxPoint = MaxSize*MaxSize + 3*(MaxSize+1);
// MaxPoint is large enough to hold board plus three borders

const int WE = 1; // West-East: offset of horizontal neighbors
const int NS = MaxSize+1; // North-South: offset of vertical neighbors

typedef int Point; // point on the board, range[0..MaxPoint-1]
typedef int Position[MaxPoint]; // array to hold Go board

enum {kEmpty = 0, kBlack, kWhite, kBorder}; // states of points

inline Point Pt(Grid col, Grid row) { return NS*row + col; }
// conversion from x,y coordinates.
```

```
int Col  (Point p) const; // get x-coordinate of point p
int Row  (Point p) const; // get y-coordinate of point p
int Line (Point p) const; // get distance of p from edge of board

Position goboard; // example: declare an array to store the board.
goboard[Pt(5,3)] = kBlack; // put a stone on (5,3).
```

Besides integer arrays, *bitmaps* which store 1 bit for every point on the board can be used for elegantly expressing many Go algorithms. For example, *blocks** and surrounded territories can be identified as connected components of points on such bitmaps.

3.2.3 Go Rules, Executing and Undoing Moves

When executing moves, a stack can store all information that is needed for fast undo. For a simple board, this includes the stones played and removed, and Ko status information for checking legality of later moves. A move checker that handles full board repetition is described in [28]. The same stack-based architecture can be used for incremental update of other data. Klinger and Mechner [31] describe a macro-based *revertible object system* that simplifies the incremental maintenance of complex state information during tree search.

3.2.4 Blocks

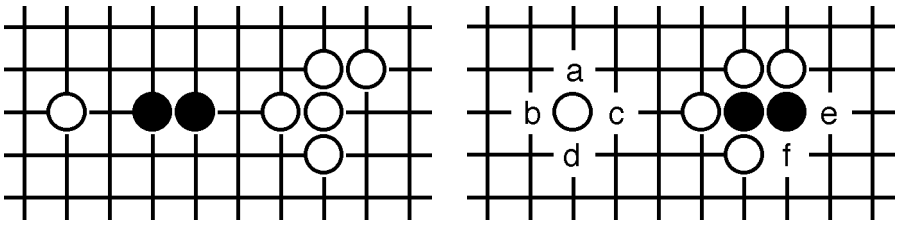

Figure 12: Blocks and liberties

Due to a multitude of languages and traditions, there is no standard nomenclature for Go and computer Go terms. Following Benson [1] and Kierulf [28], we call connected stones of the same color a *block*. Other authors have used terms such as *string, unit, chain* or *worm* for the same concept. Figure 12 shows three sample blocks on the left. A *liberty** of a block is an empty point adjacent to a stone of the block. The liberties of a white and a black block are marked in the right picture. The single white stone has four liberties labeled a, b, c, and d; the black block is partially surrounded by white stones and has only two liberties labelled e and f.

Blocks are the basic elements of board representation. Attributes of blocks include stones, liberties, connections, and references to larger structures such

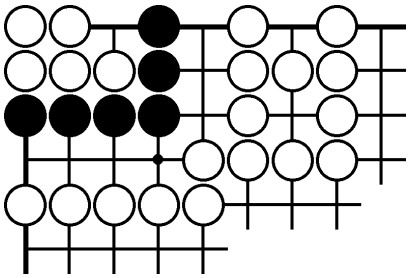

Figure 13: Tactical vs. strategic status of blocks

as territories that contain the block. Liberties are the most important factor for determining the tactical safety of blocks. A standard heuristic used in Go programs, first proposed by Wilcox, is that getting five liberties means tactical safety. The tactical status of blocks with fewer liberties is computed by a capture search, with each player moving first.

Tactical safety can be overridden by strategic considerations. A block can be tactically safe, but strategically dead, or vice versa: In Figure 13, the black block has many liberties and is tactically safe, but it has only one *eye** and is therefore strategically dead. The white five stone block in the corner is tactically captured, but strategically part of a safe white territory. The safety of a block is therefore computed in several phases. The initial value depends mainly on the liberty count and tactical analysis, but it can change after computation of higher-level structures, as in the example above.

3.2.5 Connections, Dividers and Sector Lines

Blocks are solidly connected sets of stones of the same color. Besides providing connection for stones, blocks also serve the purpose of creating walls dividing the Go board and thereby preventing the opponent from connecting. Connections and dividing walls can also be formed more efficiently, by leaving some empty space between stones. Recognizing which loose arrangements of stones are already connected, and which already form a dividing wall, is very important.

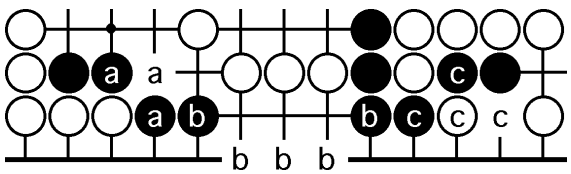

Figure 14: Potential connections by a) joint liberty, b) pattern, c) weak opponent block

Potential connections are places where a connection can be made by a single play. Examples of potential connections, shown in Figure 14, are single shared liberties of two blocks, potential connection patterns from a library of standard shapes, and weak adjacent opponent blocks which can be captured to form a connection.

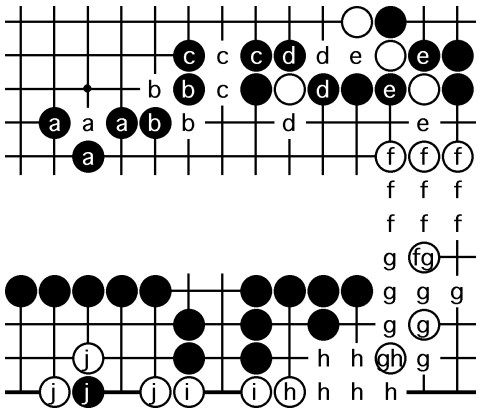

Figure 15: Connections by a) one protected liberty, b) - e) two potential connections, f) - i) connection pattern, j) dead opponent block

Safe *connections* can be formed from potential connections in a number of ways:

- Figure 15a: A single potential connection which the opponent cannot disrupt, for example on a *protected* shared liberty.
- Figure 15b - e: Two independent potential connections such as a diagonal link, bamboo joint, or other combination.
- Figure 15f - h: A connection pattern from a pattern database.
- Figure 15i: A dead opponent block which can be removed to form a connection there.

These connection types are implemented in most Go-playing systems. However, there are a lot of pitfalls caused by unexpected dependencies and by the interaction with tactics. For example, detecting whether a liberty is protected depends on tactical reading. Figure 16a shows a connection threatened by a *ko** fight, 16b and 16c show connections that can be broken because of a lack of liberties. Sometimes, capturing opponent stones does not result in a connection of the surrounding blocks, as in Figure 16d. Connections can be used to define chains of blocks, as in Section 3.2.6.

Dividers [43], also called *links, linkages* or *barriers* [74], are the dual concept of connections. A divider is weaker than a connection: its purpose is to stop

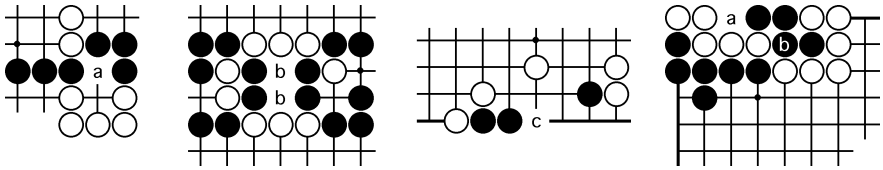

Figure 16: Connections endangered by tactical threats of a) ko, b) - d) lack of liberties

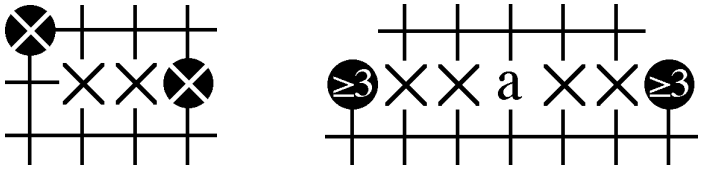

Figure 17: Examples of dividers and potential dividers

an opponent's connection from one side to the other, not to connect one's own stones. Dividers of both players may cross each other. Unlike connections, dividers can also be formed between stones and the edge of the board.

A *potential divider* can be transformed into one or more real dividers by a single move. It can be used to recognize the borders of large frameworks of potential territory. Even more distant relations between stones can be recognized using Wilcox' concept of *sector lines* [74].

3.2.6 Chains

A chain is a set of blocks joined by pairwise *independent* connections. A player can counter all opponent threats to cut off blocks from a chain.

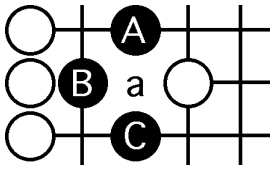

Figure 18: Double threat against two connections

A big problem of defining chains is that connectivity in Go is not transitive. If A is connected to B, and B is connected to C, it does not follow that A is connected to C, since the opponent might have a *double threat* which affects

both connections. Figure 18 shows an example. While either $\{A, B\}$ or $\{B, C\}$ are valid chains, $\{A, B, C\}$ is not, because a move at 'a' can cut off either A or C from the rest. When defining chains, programs must decide which subset of connections is more important.

3.2.7 Surrounded Areas, Potential and Safe Territory

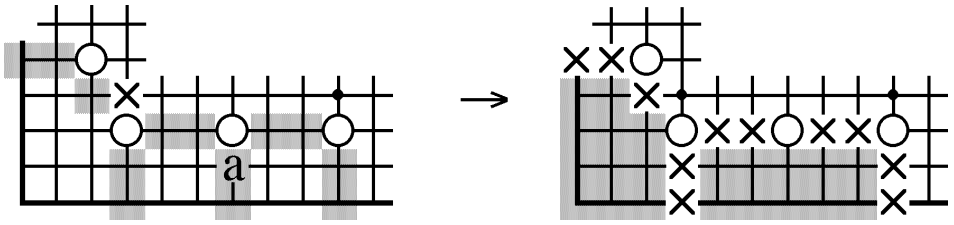

Figure 19: Recognition of territories by dividers

Recognizing board regions surrounded by one player, or *territory*, is of fundamental importance in Go for several reasons. First of all, the player with the larger total area wins the game in the end. Further, surrounded areas provide safe liberties called *eyes** for adjacent stones. Finally, opponent stones must be surrounded in order to capture them.

In the literature there are many different names for surrounded areas and related concepts, often with slightly differing meanings, such as *region, area, eye, territory, zone, potential, moyo, framework*. Territory can be found either by detecting continuous areas of very high influence or by finding boundaries consisting of blocks and dividers. Similarly, potential territory can be identified as regions of moderately high influence or as regions bounded by potential dividers. At an even earlier stage, a program can identify nearby points and generate moves to try to surround them.

For small well-enclosed areas, a definite eye status can be computed [36]. As a heuristic for other areas, a program can compute the minimum and maximum number of secure and of potential eyes [16]. Eyes greatly affect the safety of the surrounding *groups* (see below), and play an important role in *semeai**.

3.2.8 Groups

On top of the basic structures of blocks, chains and surrounded areas, larger-scale aggregates of related stones can be defined in a number of ways. The name *group* is most common; alternative names are *army, unit, dragon*. Go programs recognize groups as contiguous regions of a certain minimum influence [13], by an iterative growing and shrinking process [6], by other distance measures such as *N-th dame* [68], or by using potential connections and dividers [43]. Figure 20 shows an example of groups.

Groups are the basic units of attack and defense [13]. The relative strength of groups determines whether they can survive or will be captured, whether

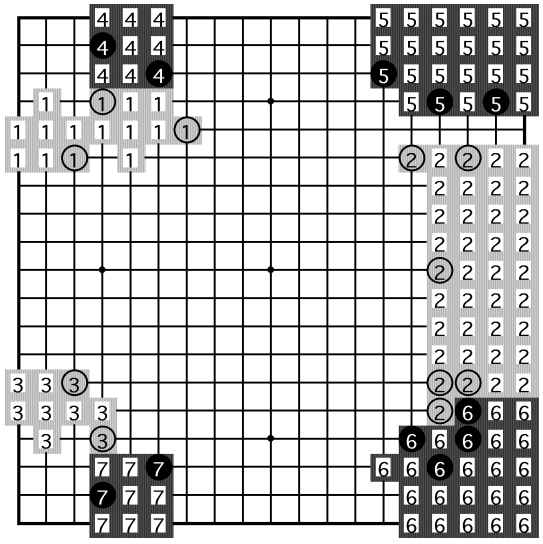

Figure 20: Groups

they can help nearby friendly groups, or attack opponent groups. Measures of group strength are connectivity within the group and towards other groups, the liberties of its blocks, and the eye space and eye potential of the contained areas. Programs typically use extensive static analysis [16], enhanced by goal-directed search for surrounded unsettled groups, to check if they can live, or be killed. Group-related moves are decisive in many games: Adjacent weak groups of the same color can connect to create a single safer group, or be subjected to a splitting attack. Playing at a junction point between adjacent groups of opposite color effectively combines attack and defense.

3.2.9 Global Move Generation

After building a detailed representation of the current state of a Go game by static analysis and goal-oriented search, global move generation and evaluation selects good or promising moves, while at the same time suppressing bad moves. Examples of filters for suppressing bad moves are those that prevent suicidal or otherwise tactically dubious moves, moves in the opponent's sphere of influence that could be cut off, playing on with a dead group, or playing into own or opponent territory.

Move generators usually implement a single abstract concept. Current programs contain thousands of lines of code implementing move generators. Many move generators are bound to a specific object, and propose moves related to that object. The following table contains a small selection.

Type of object	Move generators
Block	escape, capture, stabilize, gain liberties
Group	attack, defend, live, kill, expand, run, cut
Territory, framework	create, extend, reduce, defend, make eye, invade

Other move generators handle interactions between objects, or moves that do not relate to a specific object. Examples are playing a double attack, occupying a junction point between two territorial frameworks, or playing *joseki** from an opening book.

3.2.10 Move Evaluation

There are two basic approaches to move evaluation, both with different strengths and weaknesses. The first way, which is used in most other games, is to play the move and then use static evaluation to compute a full board score. The advantage of this method is that evaluation after the move is usually more accurate, since all effects of the move are taken into account. A disadvantage is speed: computing a full board evaluation is slow. Another problem is greedy play: many necessary long-term defensive moves don't show their effect until much later, and get too low an evaluation.

The second method is direct move evaluation. The value of moves is estimated by the move generator that proposes it, based on heuristics. Moves proposed by several generators can attain a higher total value. The advantages of this method are speed, and more possibilities to fine-tune the values. The main disadvantage is that it is impossible to predict all good and bad side-effects of a move accurately.

After evaluation of candidate moves, many programs perform some extra steps, such as checks to avoid tactical blunders, or special rules to deal with *ko** fights [28].

4 Game Tree Search

Search can be applied to computer Go on different levels and in many distinct ways. Three types of game tree search are commonly used: full-board search, single-goal localized search and multiple-goal search.

Full board search in Go is difficult, for reasons that have already been touched upon in the introduction and will be further developed in Section 4.3. Specialized searches that focus on achieving a tactical goal constitute some of the most important components of current Go programs. A major advantage of goal-directed search over full-board search is that evaluation consists only of a simple test of goal achievement, which is much faster than full board territory evaluation. One use of goal-directed search is to provide locally interesting moves as an input to a selective global move decision process.

4.1 Single-Goal Search

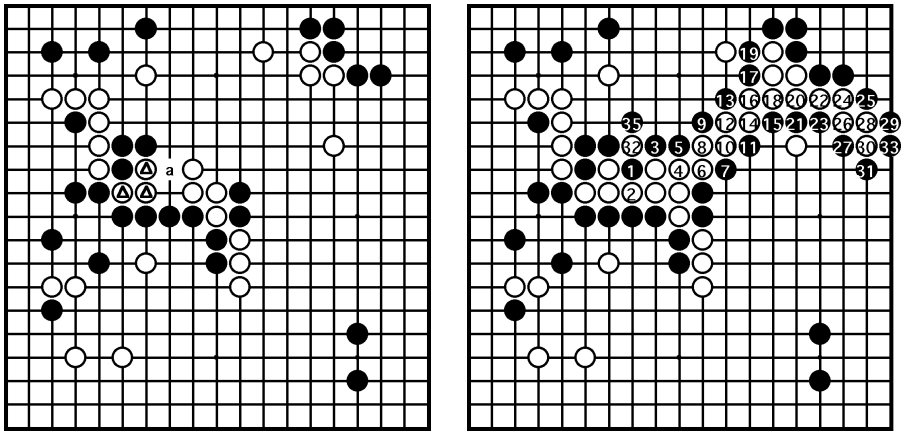

Figure 21: Tactical capture by ladder

Single-goal search uses standard game tree searching techniques for finding the tactical status of blocks, chains, groups, territories, or connections. Most goal-oriented searches are performed twice, once for each player going first, resulting in a tactical status *captured, unsettled* or *safe*. Finer distinctions are possible to account for uncertainty introduced by inconclusive searches, or for outcomes that depend on ko fights.

Knowing the tactical status of items improves the board representation and is a precondition for creating a meaningful scoring function. The simplest example of tactical searches are ladders, which are deep but almost unbranched capturing sequences. Ladders can run across the whole board, as shown in Figure 21. Further topics for which single-goal search has been used are listed in the table below. A typical current program implements many but not all of these goal-oriented searches.

One important difference to standard game tree search is that programs should find *all* moves that achieve some tactical goal, whereas usually search can be stopped after one successful move is found. Searching all moves is necessary in order to find multi-purpose moves that achieve more than one goal simultaneously, and also to allow secondary optimization of territory among all tactically good moves.

Target	Reference
Ladder	[28]
Single block capture	[28]
Life and death	[76, 35]; Section 5.1
Connect or cut	[21]
Eye status	-
Local score	-
Safety of territory	[45]; Section 5.2
Semeai	[21, 47]; Section 5.3

4.2 Multiple-Goal Search

Multiple-goal search tries to achieve an AND- or OR-combination of two or more basic goals. Such combinations represent natural higher-level goals, such as capturing at least one in a set of blocks (OR-combination), or keeping all components of a territory boundary intact (AND-combination). Since it seems infeasible to search each possible combination of interacting goals, heuristics must be used to select the most promising combinations. The table below lists a few common themes. The importance of multiple goals for Go was already emphasized in the early work of Reitman, Wilcox and their co-workers, and as a result an elaborate planning system was implemented for their Go program [54, 56, 74]. In contrast, the implementation of multi-goal search tasks is rudimentary in most current programs. However, there is a number of recent research papers on architectures for adversary and multipurpose strategic planning in Go [24, 25, 37, 75].

Target(s)	Multiple goals
Multiple blocks	save all blocks
Territory boundary	capture block *or* break through divider
Two or more opponent groups	splitting, leaning attacks
Opponent group and open area	attack group *or* make territory
Own group	live locally *or* break out
Own and opponent group	make eyes *or* counterattack

4.3 Global-Level Search and Full-Board Move Decision

There is a great variety of approaches to the problem of global move decision in Go. Many ingenious combinations of search and knowledge-based methods are used in practice. No single paradigm, comparable to the full board minimax search used in most other games, has emerged. Because of the complex evaluation and high branching factor of Go, full-board search has to be highly selective and shallow, and needs many special adjustments [15]. The role of full

board search is often reduced to a kind of quiescence search for weak groups. Most programs use a combination of the following methods:

- Extensive static analysis and evaluation to select a small number of promising moves
- Selective search to decide between candidate moves
- Shortcuts to play some 'urgent' moves immediately
- Recognition and following of temporary goals
- Choice of aggressive or defensive play based on a score estimate

Experimentation with global-level control strategies in Go is likely to continue for at least some time, until a clear preference or standard model can emerge. Local analysis methods based on combinatorial game theory [3] have the potential to replace more traditional decision procedures. However, in practice it appears to be difficult to integrate such techniques with current Go programs [15].

5 Solving Subproblems of Go

For many subproblems of Go, specialized methods have been developed which achieve a much greater heuristic accuracy that general methods, or can even solve a subproblem precisely. Some of these subsystems, most notably those for Life and Death analysis, have been integrated with Go programs, but many others remain as standalone versions that are only usable for specialized analysis tasks. Much work remains to be done in the integration of such expert modules into full-scale heuristic Go programs.

5.1 Life and Death

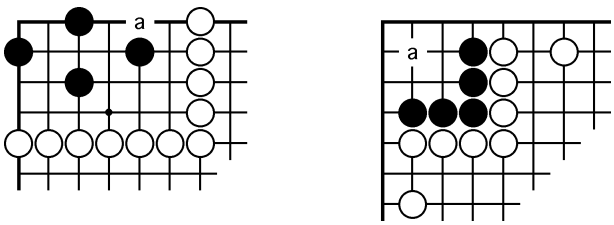

Figure 22: *The diamond* and the *carpenter's square*

Most Go programs contain some Life and Death knowledge, typically as a combination of exact and heuristic rules [35]. The performance of normal programs on Life and Death puzzles is not above their overall level of skill. A program specialized for solving such problems in a small, completely enclosed

region can do much better. Thomas Wolf's *GoTools* has reached the level of strong amateur players [76, 77, 79]. Figure 22 shows two notoriously difficult problems solved by the program. *GoTools* contains powerful rules for static Life and Death recognition, elaborate move ordering heuristics and a refined tree searching algorithm. The problems involved in generalizing such a program to more open types of positions are discussed in [78].

5.2 Safety of Stones and Territory

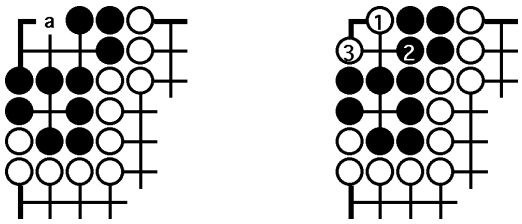

Figure 23: Safe stones, unsafe territory

Proving the safety of stones and territory is similar to Life and Death recognition. One main difference is that safety proofs for large areas cannot use straightforward search since the state space is too large. Another difference is in the treatment of coexistence in *seki**. While stones are safe if the opponent cannot capture them, territory is safe only if it can be proven that no opponent stones can survive inside. Figure 23 shows an example where the black stones are safe but the area that they surround is not.

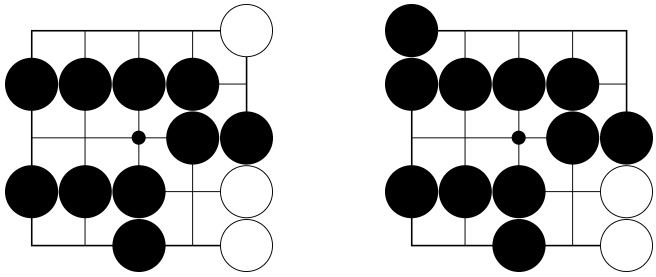

Figure 24: Unconditional safety, and safety by locally alternating play

Benson [1] has given a mathematical characterization of unconditionally alive blocks of stones. Such blocks can never be captured, not even by an arbitrary number of successive opponent moves. For example, all black blocks on the 5×5 board on the left side of Figure 24 are unconditionally safe. Benson's method is

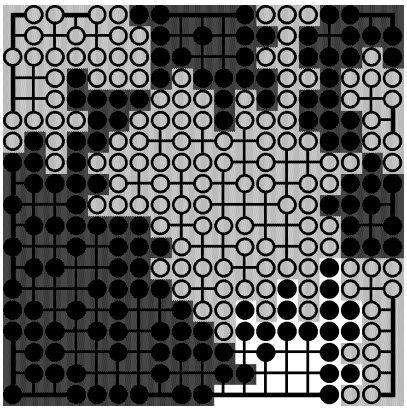

Figure 25: Proving the safety of blocks and territories, from [45].

mathematically elegant but limited in practice, since no defense against threats is allowed. This limitation has led to the development of (less elegant) rules for detecting groups of blocks which are safe under the usual alternating play [45]. The right side of Figure 24 shows an example of blocks which are not unconditionally safe, but safe under alternating play. Combined with a tree search, such rules can be used to prove the safety of many moderately large areas, as in Figure 25.

5.3 Semeai

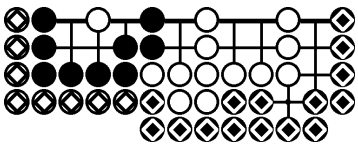

Figure 26: Semeai example, from [47].

*Semeai** are capturing races between blocks of both colors, which do not have two eyes. In order to survive, blocks in semeai must capture the opponent, or at least try to coexist in *seki**. Figure 26 shows an example. The strength of blocks in semeai can be measured by two incomparable quantities, liberty count and eye status. Simple kinds of semeai can be solved by static analysis [47], and more complicated cases by the search technique of *partial order bounding* [49].

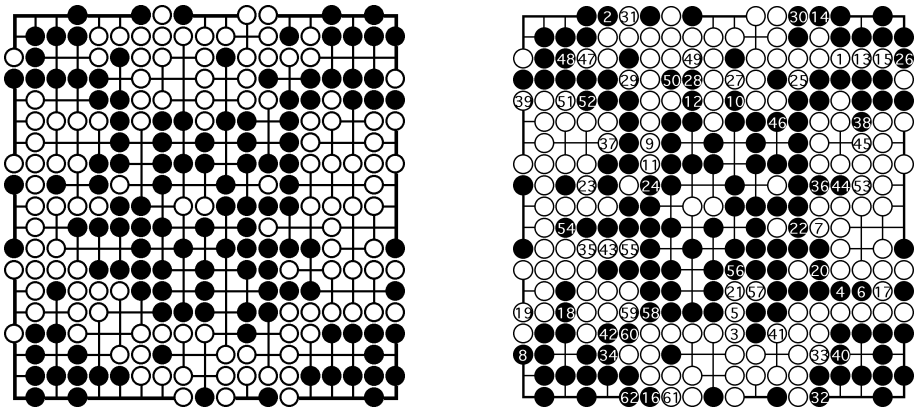

Figure 27: 89 point endgame problem and solution, from [46].

5.4 Endgame

Very late stage Go endgames can be solved by full board minimax search. However, the solution effort grows exponentially with the total size of endgame areas [48]. Endgames that can be decomposed into *independent* local areas can be solved many orders of magnitude more efficiently by the local search technique of decomposition search [46, 48]. This technique implements a divide-and-conquer approach based on combinatorial game theory [3, 4]. Figure 27 shows an 89 point endgame problem, and the computed solution.

Prof. Berlekamp of UC Berkeley and his co-workers have pushed the mathematical analysis of local Go positions much further, to the point where real endgames from professional games can be analyzed. Automating these techniques and applying them to computer Go is a challenge for the future.

6 Challenges for Computer Go Research

Develop a search-bound Go program In contrast to most other games, in Go there has not yet been a clear demonstration of correlation between deeper search and playing strength. Develop such a Go program that can automatically take advantage of greater processing power.

Comprehensive local analysis Develop a search architecture that can integrate all aspects of local fighting and evaluation.

Threats and forcing moves Develop modules that can systematically detect threats and use them for double threats, ko threats, or for forcing moves. Avoid bad forcing moves, which have unexpected side effects.

Test suite Develop a comprehensive public domain computer Go test suite.

Computer Go source code library Unify the efforts to provide a highly (re-)usable library of common functions.

Sure-win program for high handicaps Build a program that can demonstrably win all games on n stone handicap. Then reduce n.

Integrate exact modules in heuristic program Solve the problems of interfacing modules that can understand one aspect of the game very well with a general Go program.

Solve Go on small boards Human players have analyzed Go on many small rectangular boards, but there are few exact proofs [69, 70, 71] Solve Go on small board sizes such as 5×5 or 7×7.

Glossary

block Connected stones of the same color. See Section 3.2.4.

dan Master level. Higher numbers are better. See Figure 4.

eye A surrounded area providing one safe liberty. Two eyes are usually necessary to make stones safe.

joseki A standard move sequence, often in the corner.

ko A single stone capture that can lead to repetition, as in Figure 3.

komi A number of points given to the second player to compensate for the first player's advantage. Usually 5.5 or 6.5 points, but 8 points in tournaments played by Ing rules.

kyu Student level. Lower numbers are better. See Figure 4.

liberty An empty point adjacent to a stone, or a block of stones.

seki Coexistence of Black and White blocks that don't have two eyes.

semeai A race to capture. See Figure 26 for an example.

sente The initiative, the right to play next. A sente move must be answered by the opponent and therefore retains the initiative.

tesuji A skilful tactical move

References

[1] D.B. Benson. Life in the game of Go. *Information Sciences*, 10:17–29, 1976. Reprinted in Computer Games, Levy, D.N.L. (Editor), Vol. II, pp. 203-213, Springer Verlag, New York 1988.

[2] E. Berlekamp. The economist's view of combinatorial games. In R. Nowakowski, editor, *Games of No Chance: Combinatorial Games at MSRI*, pages 365–405. Cambridge University Press, 1996.

[3] E. Berlekamp, J. Conway, and R. Guy. *Winning Ways*. Academic Press, London, 1982.

[4] E. Berlekamp and D. Wolfe. *Mathematical Go: Chilling Gets the Last Point.* A K Peters, Wellesley, 1994.

[5] M. Boon. A pattern matcher for Goliath. *Computer Go*, 13:12–23, 1990.

[6] B. Bouzy. *Modélisation cognitive du joueur de Go.* PhD thesis, University Paris 6, 1995.

[7] D. Bump. GNU Go. http://www.gnu.org/software/gnugo/gnugo.html, 1999.

[8] J. Burmeister. Memory performance of master Go players. In J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 271–286, Maastricht, 2000. Universiteit Maastricht.

[9] J. Burmeister and J. Wiles. An introduction to the computer Go field and associated internet resources. Technical Report 339, Department of Computer Science, University of Queensland, 1995.

[10] J. Burmeister and J. Wiles. AI techniques used in computer Go. In *Fourth Conference of the Australasian Cognitive Science Society*, Newcastle, 1997.

[11] J. Burmeister, J. Wiles, and H. Purchase. The integration of cognitive knowledge into perceptual representations in computer Go. In *Game Programming Workshop in Japan '95*, pages 85–94, Kanagawa, Japan, 1995. Computer Shogi Association.

[12] T. Cazenave. *Systeme d'Apprentissage Par Auto-Observation. Application au jeu de Go.* PhD thesis, University of Paris, 1997. www-laforia.ibp.fr/~cazenave/-papers.html.

[13] K. Chen. Group Identification in Computer Go. In D. Levy and D. Beal, editors, *Heuristic programming in artificial intelligence: the first computer Olympiad.* Ellis Horwood, Chichester, 1989.

[14] K. Chen. The move decision process of Go Intellect. *Computer Go*, 14:9–17, 1990.

[15] K. Chen. Some practical techniques for global search in Go. *ICGA Journal*, 2000. To appear.

[16] K. Chen and Z. Chen. Static analysis of life and death in the game of Go. *Information Sciences*, 121:113–134, 1999.

[17] A.D. De Groot. *Thought and Choice in Chess.* The Hague, The Netherlands: Mouton & Co, 1965.

[18] H.D. Enderton. The Golem Go program. Technical Report CMU-CS-92-101, Carnegie Mellon University, 1991.

[19] M. Enzenberger. The integration of a priori knowledge into a Go playing neural network. cgl.ucsf.edu/go/Programs/NeuroGo.html, 1996.

[20] D. W. Erbach. Computers and Go. In Richard Bozulich, editor, *The Go Player's Almanac.* The Ishi Press, 1992. Chapter 11.

[21] D. Fotland. Knowledge representation in The Many Faces of Go. Report posted on internet newsgroup rec.games.go, 1993. Available by ftp from igs.nuri.net.

[22] K. J. Friedenbach. *Abstraction Hierarchies: A Model of Perception and Cognition in the Game of Go.* PhD thesis, University of California, Santa Cruz, 1980.

[23] S.C. Hsu and D.Y. Liu. The design and construction of the computer Go program Dragon 2. *Computer Go*, 16:3–14, 1991.

[24] S. Hu. *Multipurpose Adversary Planning in the Game of Go.* PhD thesis, George Mason University, 1995.

[25] S. Hu and P. Lehner. Multipurpose strategic planning in the game of Go. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):1048–1051, 1997.

[26] A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search.* PhD thesis, University of Alberta, 1999.

[27] K.Y. Kao. *Sums of Hot and Tepid Combinatorial Games.* PhD thesis, University of North Carolina at Charlotte, 1997.

[28] A. Kierulf. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies.* PhD thesis, ETH Zürich, 1990.

[29] A. Kierulf, K. Chen, and J. Nievergelt. Smart Game Board and Go Explorer: A study in software and knowledge engineering. *Communications of the Association for Computing Machinery*, 33(2):152–167, 1990.

[30] Y. Kim. *New Values in Domineering and Loopy Games in Go.* PhD thesis, University of California at Berkeley, 1995.

[31] T. Klinger and D. Mechner. An architecture for computer Go. Manuscript available on WWW, 1996.

[32] T. Kojima. *Automatic Acquisition of Go Knowledge from Game Records: Deductive and Evolutionary Approaches.* PhD thesis, University of Tokyo, 1998.

[33] T. Kojima, K. Ueda, and S. Nagano. An evolutionary algorithm extended by ecological analogy and its application to the game of Go. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 684–689, 1997. www.brl.ntt.co.jp/people/kojima/research.

[34] T. Kojima, K. Ueda, and S. Nagano. Flexible acquisition of various types of knowledge from game records: Application to the game of Go. In *Proceedings of the IJCAI-97 Workshop on Using Games as an Experimental Testbed for AI Research*, pages 51–57, 1997. www.brl.ntt.co.jp/people/kojima/research.

[35] J. Kraszek. Heuristics in the life and death algorithm of a Go-playing program. *Computer Go*, 9:13–24, 1988.

[36] H. Landman. Eyespace values in Go. In R. Nowakowski, editor, *Games of No Chance*, pages 227–257. Cambridge University Press, 1996.

[37] P. Lehner. *Planning in adversity: a computational model of strategic planning in the game of Go.* PhD thesis, University of Michigan, 1981.

[38] D. Levy, editor. *Computer Games I+II.* Springer Verlag, 1988.

[39] D. Lichtenstein and M. Sipser. Go is polynomial-space hard. *Journal ACM*, 27(2):393–401, April 1980.

[40] D. Mechner. All systems Go. *The Sciences*, 38(1), 1998.

[41] D.J. Moews. *On Some Combinatorial Games Connected with Go.* PhD thesis, University of California at Berkeley, 1993.

[42] F.L. Morris. Playing disjunctive sums is polynomial space complete. *Int. Journal Game Theory*, 10(3-4):195–205, 1981.

[43] M. Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory.* PhD thesis, ETH Zürich, 1995. Diss. ETH Nr. 11.006.

[44] M. Müller. Generalized thermography: A new approach to evaluation in computer Go. In H. Iida, editor, *IJCAI-97 Workshop on Using Games as an Experimental Testbed for AI Research*, pages 41–49, Nagoya, 1997. Also published in the book *Games in AI Research*.

[45] M. Müller. Playing it safe: Recognizing secure territories in computer Go by using static rules and search. In H. Matsubara, editor, *Game Programming Workshop in Japan '97*, pages 80–86, Computer Shogi Association, Tokyo, Japan, 1997.

[46] M. Müller. Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames. In *IJCAI-99*, pages 578–583, 1999.

[47] M. Müller. 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.

[48] M. Müller. Not like other games - why tree search in Go is different. In *Proceedings of Fifth Joint Conference on Information Sciences (JCIS 2000)*, pages 974–977, 2000.

[49] M. Müller. Partial order bounding: A new approach to evaluation in game tree search. Technical Report TR-00-10, Electrotechnical Laboratory, Tsukuba, Japan, 2000. Also submitted for journal publication.

[50] M. Müller, E. Berlekamp, and B. Spight. Generalized thermography: Algorithms, implementation, and application to Go endgames. Technical Report 96-030, ICSI Berkeley, 1996.

[51] M. Müller and R. Gasser. Experiments in computer Go endgames. In R. Nowakowski, editor, *Games of No Chance*, pages 273–284. Cambridge University Press, 1996.

[52] R. Nowakowski, editor. *Games of No Chance*. Cambridge University Press, 1996.

[53] E. Pettersen. The Computer Go Ladder. cgl.ucsf.edu/go/ladder.html, 1994.

[54] W. Reitman, J. Kerwin, R. Nado, J. Reitman, and B. Wilcox. Goals and plans in a program for playing Go. In *Proceedings of the 29th National Conference of the ACM*, pages 123–127, San Diego, 1974. ACM.

[55] W. Reitman and B. Wilcox. Modeling tactical analysis and problem solving in Go. In *Proceedings of the Tenth Annual Pittsburg Conference on Modelling and Simulation*, pages 2133–2148, Pittsburg, 1979. Instrument Society of America.

[56] W. Reitman and B. Wilcox. The structure and performance of the Interim.2 Go program. In *IJCAI-79*, pages 711–719, 1979.

[57] P. Ricaud. *GOBELIN: une approche pragmatique de l'abstraction appliquée à la modélisation de la stratégie elémentaire du jeu de Go*. PhD thesis, University Paris 6, 1995.

[58] J. Robson. The complexity of Go. In *Proc. IFIP (International Federation of Information Processing)*, pages 413–417, 1983.

[59] C. Rosin. *Coevolutionary Search Among Adversaries*. PhD thesis, University of California, San Diego, 1997.

[60] J.L. Ryder. *Heuristic Analysis of Large Trees as Generated in the Game of Go*. PhD thesis, Stanford University, 1971. Microfilm no. 72-11,654.

[61] Y. Saito. *Cognitive Scientific Study of Go*. PhD thesis, University of Tokyo, 1996. In Japanese.

[62] N. Sasaki. *The Neural Network Programs for Games*. PhD thesis, Tohoku University, 1998. In Japanese.

[63] N. Schraudolph. Temporal difference learning of position evaluation in the game of Go. In *Neural Information Processing Systems 6*. Morgan Kaufmann, 1994.

[64] R. Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[65] S. Sei and T. Kawashima. Memory-based approach in Go-program Katsunari. Complex Games Lab Workshop. http://www.etl.go.jp/etl/divisions/~7236/Events/workshop98/, 1998.

[66] W. Spight. Extended thermography for multiple kos in Go. In J. van den Herik and H. Iida, editors, *Computers and Games. Proceedings CG'98*, number 1558 in Lecture Notes in Computer Science, pages 232–251. Springer Verlag, 1998.

[67] D. Stoutamire. Machine learning, game play and Go. Technical Report TR 91-128, Case Western Reserve University, Cleveland, Ohio, 1991.

[68] M. Tajima and N. Sanechika. Estimating the Possible Omission Number for groups in Go by the number of n-th dame. In H.J.van den Herik and H.Iida, editors, *Computers and Games: Proceedings CG'98*, number 1558 in Lecture Notes in Computer Science, pages 265–281. Springer Verlag, Tsukuba, Japan, 1999.

[69] E. Thorp and W. Walden. A partial analysis of Go. *Computer Journal*, 7(3):203–207, 1964. Reprinted in: [38], Vol.II, pp.143–151.

[70] E. Thorp and W. Walden. A computer assisted study of Go on m*n boards. *Information Sciences*, 4(1):1–33, 1972. Reprinted in: [38], Vol.II, pp.152–181.

[71] J. Tromp. Small board Go. Message on computer-go mailing list, 1998.

[72] J. Tromp. On game space size. Message on computer-go mailing list, 1999.

[73] B. Wilcox. Reflections on building two Go programs. *SIGART Newsletter*, 94:29–43, 1985.

[74] B. Wilcox. Computer Go. In D.N.L. Levy, editor, *Computer Games*, volume 2, pages 94–135. Springer-Verlag, 1988.

[75] S. Willmott, J. Richardson, A. Bundy, and J. Levine. An adversial planning approach to Go. In H.J.van den Herik and H.Iida, editors, *Computers and Games: Proceedings CG'98*, number 1558 in Lecture Notes in Computer Science, pages 93–112. Springer Verlag, Tsukuba, Japan, 1999.

[76] T. Wolf. Investigating tsumego problems with RisiKo. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2*. Ellis Horwood, 1991.

[77] T. Wolf. The program GoTools and its computer-generated tsume go database. In H. Matsubara, editor, *Game Programming Workshop in Japan '94*, pages 84–96, Computer Shogi Association, Tokyo, Japan, 1994.

[78] T. Wolf. About problems in generalizing a tsumego program to open positions. In H. Matsubara, editor, *Game Programming Workshop in Japan '96*, pages 20–26, Computer Shogi Association, Tokyo, Japan, 1996.

[79] T. Wolf. Forward pruning and other heuristic search techniques in tsume go. *Information Sciences*, 122:59–76, 2000.

[80] D. Wolfe. *Mathematics of Go: Chilling Corridors.* PhD thesis, University of California at Berkeley, 1991.

[81] D. Wolfe. The gamesman's toolkit. In R. Nowakowski, editor, *Games of No Chance: Combinatorial Games at MSRI*, pages 93–98. Cambridge University Press, 1996.

[82] S.-J. Yan and S.-C. Hsu. Design and implementation of a computer Go program JIMMY 4.0. *Journal of Technology*, 14(3):423–430, 1999.

[83] L. Yedwab. On playing well in a sum of games. Master's thesis, MIT, 1985. MIT/LCS/TR-348.

[84] H. Yoshii. Move evaluation tree system. Complex Games Lab Workshop, http://www.etl.go.jp/etl/divisions/~7236/Events/workshop98/, 1998.

[85] Atsushi Yoshikawa, Takuya Kojima, and Yasuki Saito. Relations between skill and the use of terms - an analysis of protocols of the game of Go. In H. Jaap van den Herik and Hiroyuki Iida, editors, *Computers and Games*, number 1558 in LNCS, pages 282–299. Springer-Verlag, 1999.

[86] A. L. Zobrist. *Feature Extraction and Representation for Pattern Recognition and the Game of Go.* PhD thesis, Univ. of Wisconsin, 1970. Microfilm 71-03, 162.