ELSEVIER

Artificial Intelligence 134 (2002) 145–179

Artificial Intelligence

www.elsevier.com/locate/artint

Computer Go

Martin Müller

Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2E8

Abstract

Computer Go is one of the biggest challenges faced by game programmers. This survey describes the typical components of a Go program, and discusses knowledge representation, search methods and techniques for solving specific subproblems in this domain. Along with a summary of the development of computer Go in recent years, areas for future research are pointed out.

Keywords: Computer Go; Go programs; Game tree search; Knowledge representation

1. Introduction

Of all games of skill, Go is second only to chess in terms of research effort and programming time spent. Yet in playing strength, Go programs lag far behind their counterparts in any other popular board game. While Go programs have advanced considerably in the last 10–15 years, they can still be beaten easily even by human players of only moderate skill. What are the reasons for this state of affairs, and what can we do about it? This survey presents an overview of the specific challenges posed by the domain of computer Go, and the approaches to their solution that have been developed over the years. It also provides a wealth of references for those who want to pursue this fascinating topic deeper.

The structure of this paper is as follows: The remainder of the introduction briefly describes the rules of the game. Section 2 gives an overview of the current state of the art, and discusses the many challenging research issues originating from this domain. Section 3 surveys a wide array of techniques for representing and using knowledge in a Go program. Applications of minimax game tree search in Go are described in Section 4, and Section 5 briefly introduces a number of subproblems of the game for which specialized solution techniques have been developed. Section 6 poses challenging problems for further research

E-mail address: mmueller@cs.ualberta.ca (M. Müller).

0004-3702/02/$ – see front matter
PII: S0004-3702(01)00121-7

in the field and summarizes the paper. Appendix A contains two representative recent games involving Go programs, and a glossary gives brief definitions for the technical terms marked by a star in the text, as in: *komi**. There is an extensive but by no means exhaustive bibliography. A companion web site contains further related information and links to online computer Go resources [60].

1.1. The game of Go

The ancient Asian game of Go, called *baduk* in Korea and *weiqi* in China, 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. Fig. 1 shows the opening phase of a typical game. Play starts on an empty board, and the move sequence is indicated by numbers on the stones. Players can pass at any time; consecutive passes end the game.

The capturing rule states that stones of one color that have been completely surrounded by the opponent, such that no horizontally or vertically adjacent empty point, or *liberty**, remains, are removed from the board. The leftmost picture in Fig. 2 shows two white stones with a single liberty at 'a'. If Black plays there, the two white stones are captured

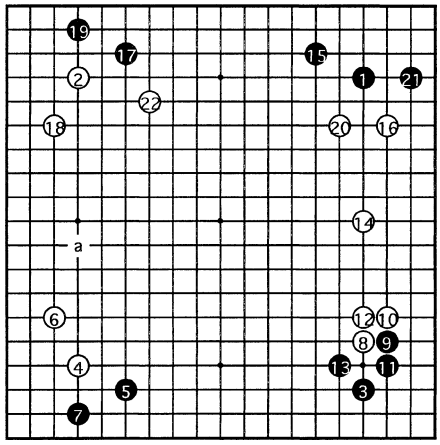

Fig. 1. The game of Go.

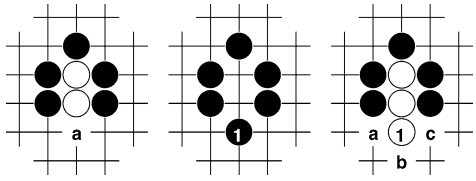

Fig. 2. The capturing rule.

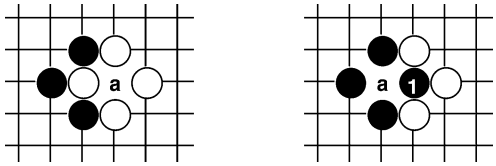

Fig. 3. Ko.

and removed from the board, as shown in the next picture. However, if White were to play on the same point first, as in the third picture, then the white stones have three liberties, and Black would require three moves in a row at 'a', 'b' and 'c' to capture the three stones. If White properly answers Black's attacking moves, the white stones will be able to run away and cannot be captured. Stones can be permanently secured by creating two or more safe liberties, usually in the form of *eyes**.

Capturing and recapturing stones can potentially lead to an infinite loop with repeating positions. The *ko** rule prevents that. A basic ko is shown in Fig. 3. If Black captures a white stone by playing at 'a' in the first diagram, White may not immediately recapture. For the rare, more complex cases of repetition, rule sets differ slightly in their treatment [35].

Human Go players are ranked by kyu (student) and dan (master) grades, starting at high kyu grades of 20 kyu or more, and progressing towards 1 kyu and then on to the lowest master grade of 1 dan. The strongest amateur players have a level of about 6 dan. On top of this scale there is another, separate dan scale for professional players, up to the highest rank of professional 9 dan. Differences in playing strength can be balanced by a handicap system, which allows Black to place several stones on the board before the start of the game. In games without a handicap, Black has to compensate White for the advantage of playing first. This is done by adjusting the final score by a *komi**, usually between 5.5 and 8 points. For detailed information on these and many other aspects of Go, see [7].

2. State of computer Go

The playing level of even the best Go programs is still modest, compared to the great successes achieved in many other games of skill. It is hard to measure the performance of Go programs precisely, since their strengths and weaknesses differ considerably from that of humans. Fig. 4 shows the human ranking scale and a rough estimate of the best programs' playing strengths, which is in the mid-kyu range.

Computer Go has seen an ever-changing succession of big, world-championship-caliber events. The most traditional event, which has been held each year since 1985, is the International Computer Go Congress, informally called the Ing Cup in honor of sponsor Ing Chang-Ki. The Ing Foundation's million dollar prize for a professional level Go program has given a big boost to the popularity of Go amongst games programmers and researchers. While several other top-level international events such as the Japanese FOST cup have been discontinued, the Computer Olympiad has recently been revived and new tournaments are being sponsored by the Korean Internet company *garosu.com* and the

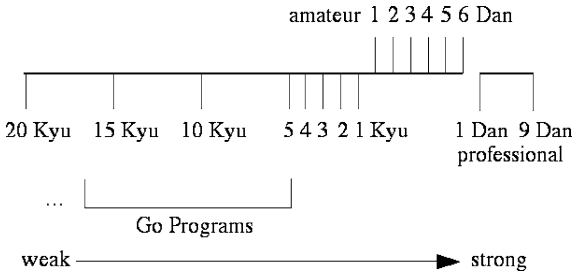

Fig. 4. Go programs on the human ranking scale.

American *IntelligentGo* foundation. In addition, small-scale computer Go tournaments are held every year in Asia, Europe, North America and in an ongoing "ladder" competition on the Internet [65]. 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 and Othello, there is an increasing interest in Go as one of the final frontiers of computer game research.

2.1. The challenge of computer Go

Computer Go poses many formidable conceptual, technical and software engineering challenges. Most competitive programs have required 5–15 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, as every games player knows, just one bad move can ruin a good game. Program performance over a *full* game can be much lower than master level.

The two games given in Appendix A illustrate both sides of this story: on one hand, programs can look quite proficient in the right kind of game, but on the other hand, when matched against a human player who knows how to exploit their weak points, they can still lose games even when receiving a ridiculously large number of handicap stones.

2.1.1. Assessing the state of the Art

Judged by human standards, play of current programs looks 'almost' reasonable, but certainly not impressive. Maybe only readers who have tried to write their own Go program can fully appreciate the enormous amount of ingenuity and hard work that has gone into building state-of-the-art systems. The fact that despite all these efforts the current level of programs is still modest is an indication of just how hard a problem computer Go is.

2.2. Overview of the development of computer Go

This section gives a brief overview of the more recent events in computer Go, and references several previous surveys which contain more information about the earlier programs and competitions.

2.2.1. The recent development of computer Go programs

While Go programming started in the late 1960s, it got a big boost in the mid 1980s, with the appearance of affordable PCs, and of tournament sponsors such as the Ing foundation. In early tournaments, Taiwanese programs such as Hsu and Liu's *Dragon* [32] were successful. From 1989–91, Mark Boon's *Goliath* [5] dominated all tournaments, followed by Ken Chen's *Go Intellect* [16–19] and Chen Zhixing's programs *Handtalk* and his newer *Goemate* [19,20], which currently leads the pack. In recent years, *Go4++* by Michael Reiss, David Fotland's *Many Faces of Go* [27] and *KCC Igo* built by the North Korean KCC team have also won major tournament victories. In total there are about 10 top class programs, including *Haruka* by Ryuichi Kawa, *Wulu* by Lei Xiuyu, Yong Goo Park's *FunGo*, Janusz Kraszek's *Star of Poland* and Yan Shi-Jim's *Jimmy* [99]. Most of these programs are distributed commercially. A step behind the top ten 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 an open source program, *GnuGo* [8], in this category. The total size of the current computer Go community can be estimated at about 200 programmers, and it is growing steadily.

Several important 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*, *Go4++* and *KCC Igo* have also achieved some success in no-handicap games against human players close to dan level strength. However, as demonstrated by test games such as the one given in the appendix, experienced human players can still beat all current programs with much more than 11 stones. As an official but more or less symbolic ranking, *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. As a more realistic estimate, programs are currently ranked at about 15 kyu on the Internet Go servers.

In the big Asian markets, there has been an enormous increase in the number of Go software packages in recent years. In an informal survey carried out in 1999 by the author, a medium-size software store in a small Japanese town 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. More references are available at the online companion site [60].

The development of computer Go has been documented in a number of previous surveys. Wilcox [89] has written extensively about the early US-based Go programs in the 1970s and 1980s developed by Zobrist [103], Ryder [72], and by Reitman and Wilcox [66–68,88, 89]. The important early papers on computer Go are collected in Levy [49]. Kierulf's Ph.D. thesis [39] contains references for most of the programs that participated in the computer Go tournaments of 1985–1989, and describes details of the *Smart Go – Explorer – Go*

Intellect line of programs. Erbach [26] gives a good overview of the state of the art in the early 1990s. Based on information from 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* [10,11]. Chen Zhixing, author of the programs *Handtalk* and *Goemate*, has written a book about computer Go (in Chinese) [20].

Ph.D. theses on computer Go were first published 30 years ago, and currently appear at a rate of about one per year [6,14,28,33,39,43,48,55,69,71,72,74,103]. Markus Enzenberger maintains an extensive bibliography of online computer Go papers [25]. Information about computer Go programs and tournaments is available from many interconnected web sites, which are linked from the companion web site [60].

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 [9,12,14,23,40,44,71,73–75,89,102]. 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 briefly discussed in Section 3.1. A study by Enzenberger [24] 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.

Combinatorial game theory is a branch of mathematics with applications to the analysis of Go endgames, *ko** fights, and determining the value of moves. Many of the publications in this field are relevant to computer Go [2–4,38,41,53,59,63,64,79,97,98,100].

2.4. Why Go is a difficult game for computers

It is easy to write a Go program that can play a complete game. However, it is hard to write a program that plays well. This empirical observation can be explained by comparing Go with other games, by analyzing the search space of Go and by considering the advantages of human players in this domain.

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 [19], even Go on a 9×9 board, which has 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 position 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* [36,37] derives much of its strength from auxiliary searches that solve or approximately solve subproblems, and thereby 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 $3^{19\times19} \approx 10^{170}$, and about 1.2% of these are legal [87]. 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 in all cases, state information must contain the complete move history, which enormously increases the number of distinct states.

No simple yet reasonable evaluation function will ever be found for Go. This 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 [50,54,70]. 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 a good theory and some search [57,58]. This aspect of Go may be exemplified best by Berlekamp's endgame studies [4]. To play well in real-game situations, a Go program must be able to 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 at an early stage 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. Positional judgement of humans is also highly developed. Skilled players usually know which side is better in a game after a quick glance at the position, and can come up with precise estimates of the score.

Since Go stones don't move on the board once placed, it is much easier for people to look ahead and visualize positions with many more stones on the board, compared to chess and many other games. Even amateur players can look 15 or 20 ply ahead in a complex local fight, and 50 ply or more in a *ladder**.

We can contrast Go with other popular games such as Othello, 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 the centuries, much of it implicit in the game records of master players. Thousands of game commentaries, tutorial books and problem collections have been compiled. The pattern knowledge of Go experts seems at least comparable to that of chess masters, which is already daunting [9,21,102]. 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 early cognitive models of Reitman and Wilcox [68] were interesting, but today's programs make do with comparatively simple pattern matching [5, 55]. 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.

A sentiment often heard amongst Go programmers is that 'improving' a program by adding knowledge makes it weaker in practice. Small changes to a Go program can 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. First, *patterns*, as described in Section 3.1, are a direct way. Second, 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. After generating representations such as patterns and higher-level descriptions, knowledge is collected and processed in order to generate good moves. Sections 3.3 and 3.4 deal with the problem of move selection and evaluation in Go.

3.1. Patterns and pattern matching

Patterns are a simple yet powerful way of encoding Go knowledge. For example, Fig. 5 shows a standard corner pattern and a full board position in which this pattern occurs twice, once in its original form and once in a rotated form with reversed colors. Many types of Go moves, such as *joseki** and *tesuji**, are described by patterns in the literature. Patterns can be applied in all stages of the game, from opening to endgame. Almost all Go programs contain a database of patterns 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 [43–45, 77,81,101], and such a pattern-learning system is used in at least one of the top programs.

Many variants of pattern matching 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 Fig. 6. 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.

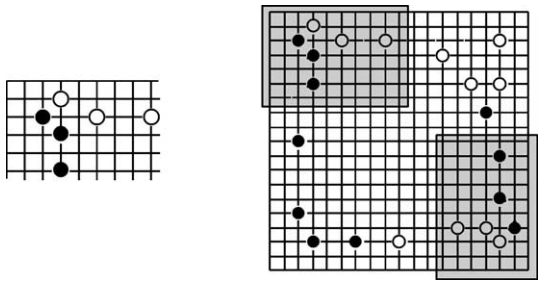

Fig. 5. Corner pattern matching in two places.

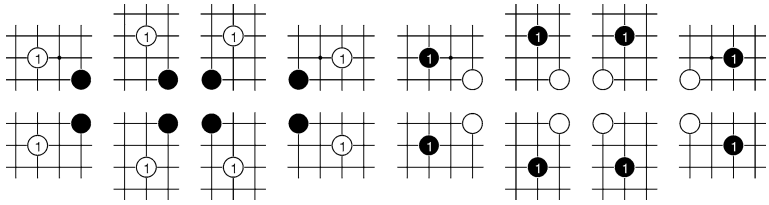

Fig. 6. The 16 instances of a pattern.

Filtering techniques based on hash tables [5,27] or *tries* (radix trees) [55,76] greatly 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 presence of one or two edges of the board affects the validity of 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 and overall safety of stones on the boundary of a pattern.
- The *pattern information* contains knowledge which can be applied if a pattern matches, such as locally good and bad moves, or information about cuts and connections (see Section 3.2.5).

3.1.2. Pattern matching

A typical midgame position produces about 500 matches from a database of 3,000 standard patterns. 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,
- data structure used for organizing the patterns,
- number of currently matching patterns,
- percentage of mismatching patterns pruned by the lookup in the data structure,
- speed of pattern-board comparisons, and of context checks,
- optimizations for incremental matching, and
- 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 author's 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. In contrast, on a set of 1,000 unrelated positions, the savings were only 10%, 3% and 0.2% respectively.

Details of the experiment as well as several further optimizations for tree-based matching are described in [55].

3.1.3. Using pattern information

The main use of pattern information is in move generation (also see Section 3.3). Many programs contain code that give bonuses to moves proposed by a pattern. One example would be moves that create or expand *territory**. In general the selection and evaluation of pattern moves will be controlled by other parts of a program. For example, a pattern move that improves the *eye** shape of a *group** of stones may be extremely important if that group is in danger of being captured, but it may be irrelevant if the group is already safe. Other types of patterns are used as building blocks for higher-level representations such as the connections and dividers in Section 3.2.5.

There are many intricate design and engineering issues involved in dealing with overlapping or even contradictory patterns, and with building and maintaining a high quality database of patterns. In the long term, the use of machine learning techniques seems necessary in this area.

3.2. Knowledge representation

Go programs contain a number of standard components, which model Go concepts at different levels of abstraction. The form of these abstraction differs greatly from the way Go is presented to human students of the game, but is more suitable for computer processing. Defining such components and developing powerful and efficient representations for them are some of the main steps involved in building a Go program. Many 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, and 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 game tree search engine and file input/output in a standard format such as SGF, Smart Go Format [30].

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 adapted their Go-playing engine to use the *CGoban* program as a graphical front-end. The *Smart Game Board* [39], 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 *GnuGo* web page [8].

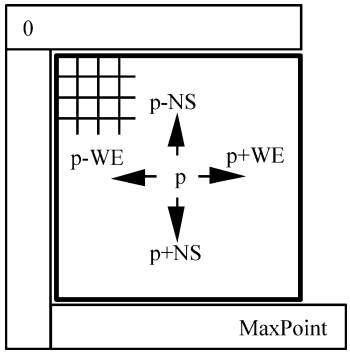

Fig. 7. Embedding a Go board into a one-dimensional array, from [39].

3.2.2. Go board

At the lowest level, the representations of Go data structures are very similar to those used in many other games. A Go board is usually implemented as a one-dimensional array. Compared to a two-dimensional array, this often allows faster access to a point without an implicit multiplication, and is more convenient because a point can be identified by a single reference rather than by a pair of coordinates.

The neighbors of a point are calculated by adding or subtracting the constants WE (West–East), which is set to 1, and NS (North–South), set to 1 larger than the board width. Given the row and column of a point, its array index is NS∗row + column. Each point on the board has a color *Empty*, *Black* or *White*, and points off the board have a special color *Border*. Boards smaller than the maximum size can use the upper left corner of the large board. The board is surrounded by a one-point border on the top, the bottom and between consecutive lines, as shown in Fig. 7. All eight neighbors of any point are easily accessible without the need to check for array boundaries or stepping over to the other side of the board.

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 to support a fast undo operation. For a simple board, this includes the stones played and removed, and Ko status information to check the legality of moves. A move checker that handles full board repetition is described in [39]. The same stack-based architecture can be used to incrementally update other data. Klinger and Mechner [42] describe a general macro-based *revertible object system* that simplifies the incremental maintenance of complex state information during tree search. However, in each specific case it remains a challenging engineering problem to compute and store just the right amount and kind of data to yield a well-informed yet fast search.

3.2.4. Blocks

Due to a multitude of languages and traditions, there is no standard nomenclature for Go and computer Go terms. Following Benson [1] and Kierulf [39], we call connected stones of the same color a *block*. Other authors have used terms such as *string*, *unit*, *chain* or even *worm* for the same concept. Fig. 8 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 at 'a', 'b', 'c', and 'd'; the black block is partially surrounded by white stones and has only two liberties at 'e' and 'f'.

Blocks are the basic elements of board representation. Attributes of blocks include stones, liberties, connections, and references to larger structures such 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 five or more liberties mean tactical safety. The tactical status of blocks with fewer liberties can be computed by a capture search, with each player moving first (see Section 4.1).

Tactical safety can be overridden by strategic considerations. A block can be tactically safe, but strategically dead, or vice versa. In Fig. 9, the black block has many liberties and is tactically safe, but even capturing the five white stones gives it only one *eye**. Black cannot avoid capture in the long term and is therefore strategically dead. The white five stone block in the corner has the opposite status: it is tactically captured, but strategically a part of a large safe white territory including the surrounding black block. In a program, the safety of a block is usually computed in several phases. The initial value depends mainly on the liberty count and tactical analysis, but it can be changed after computation of higher-level structures, such as the strategic status of groups and territories, as in the example above.

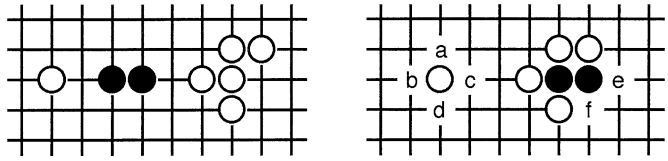

Fig. 8. Blocks and liberties.

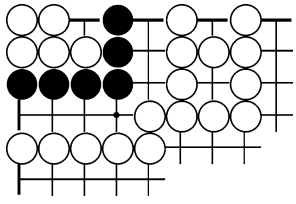

Fig. 9. Tactical vs. strategic status of blocks.

3.2.5. Connections, dividers and sector lines

Besides providing connection for stones, blocks also serve the purpose of creating walls that divide the Go board and thereby prevent the opponent from connecting or expanding. 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.

Potential connections are places where a connection can be made by a single play. Examples of potential connections, shown in Fig. 10, are single shared liberties of two blocks (a), potential connection patterns from a library of standard shapes (b), and places where a weak adjacent opponent block can be captured to form a connection (c).

Safe *connections*, as shown in Fig. 11, can be formed from potential connections in a number of ways:

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

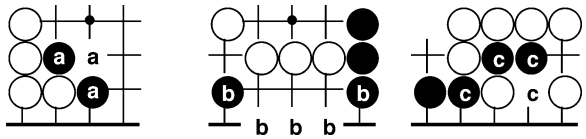

Fig. 10. Potential connections for black blocks by (a) joint liberty, (b) pattern, (c) weak opponent block.

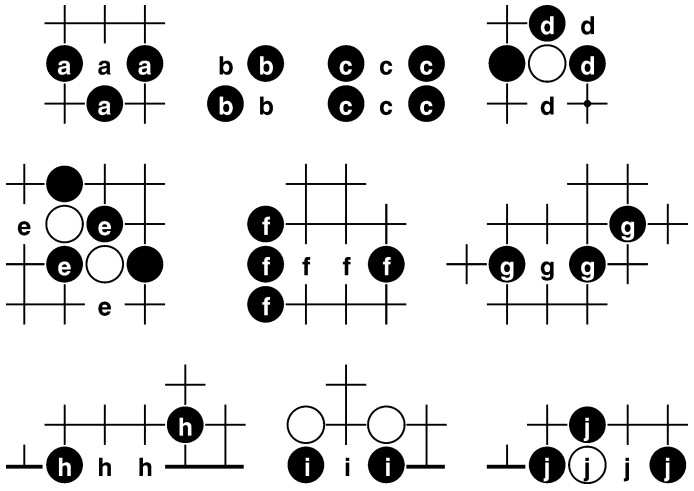

Fig. 11. Forming connections by (a) one protected liberty, (b)–(e) two potential connections, (f)–(i) connection pattern, (j) dead opponent block.

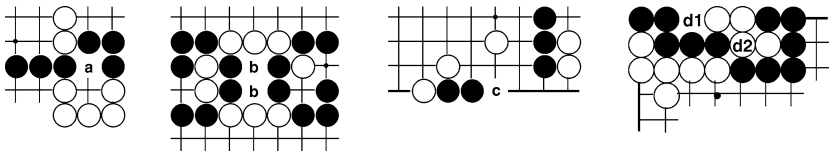

Fig. 12. Connections between black stones endangered by tactical threats of (a) ko, (b)–(d) lack of liberties.

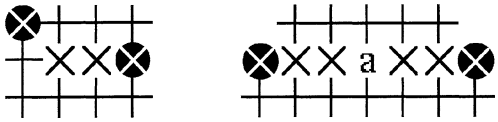

Fig. 13. Examples of dividers and potential dividers.

These connection types are implemented in most Go-playing systems. Connections can be used to define chains of blocks, as in Section 3.2.6. There are a lot of pitfalls caused by unexpected dependencies and by the interaction with tactics. For example, deciding whether a liberty is protected can require tactical reading. Fig. 12 shows connections threatened by a *ko** fight (a), and by a lack of liberties (b,c). Sometimes, capturing opponent stones does not guarantee a connection of the surrounding blocks. Black can capture four stones with d1, but White plays back in "under the stones" at d2 and Black cannot connect.

Dividers [55], also called *links*, *linkages* or *barriers* [89], are the dual concept of connections. See the left picture in Fig. 13 for an example. A divider is weaker than a connection: its purpose is to stop 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*, as in the example in Fig. 13 on the right side, can be transformed into one or more real dividers by making 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* [89].

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. A chain is sometimes also called a group, and many programs actually do not distinguish between chains and groups, as we do here.

A big problem with 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. Fig. 14 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, current programs use heuristics to decide which subset of connections is most important.

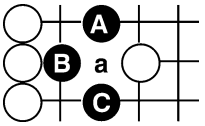

Fig. 14. Double threat against two connections.

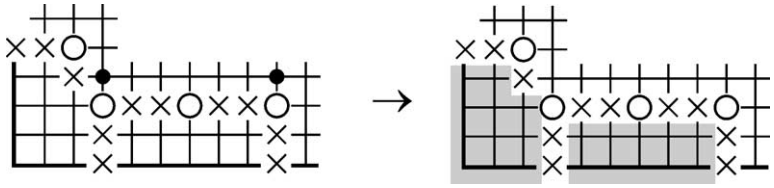

Fig. 15. Recognition of territories by dividers.

3.2.7. Surrounded areas, potential and safe territory

Recognizing board regions surrounded by a single player is of fundamental importance in Go for several reasons. First of all, the player with the larger surrounded 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 names for surrounded areas and related concepts, often with slightly different meanings, such as *region*, *area*, *eye*, *territory*, *zone*, *potential*, *moyo* and *framework*. Territory can be found by detecting contiguous areas of very high influence, or by finding boundaries consisting of blocks and dividers, as in Fig. 15. A third method defines territory in terms of connectivity: a point is territory if an opponent stone placed there cannot connect to other opponent stones which are alive, and cannot live by itself.

As in the case of connections, these definitions are usually not robust with respect to double threats. Even if two regions *A* and *B* are each safe by themselves, there might exist a move that threatens both of them at the same time.

Potential territory can be identified in similar ways as regions of moderately high influence or as regions bounded by potential dividers. In an even less controlled area, 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 [22,47]. As a heuristic for other areas, a program can compute the minimum and maximum number of secure and potential eyes [19]. Eyes greatly affect the safety of the surrounding *groups* (see Section 3.2.8), and play an important role in capturing races or *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* and *dragon*. Go programs recognize groups as contiguous regions of a certain minimum influence [16], by an iterative growing and shrinking process [6], by other distance measures such as *Nth dame* [82], or by

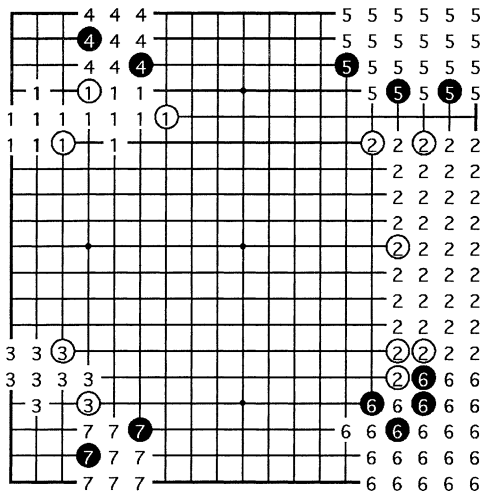

Fig. 16. Groups.

using potential connections and dividers [55]. Other programs just use chains throughout, without defining groups as a separate concept. Fig. 16 shows the groups computed by the author's program *Explorer* for the position of Fig. 1.

Groups are the basic units of attack and defense in Go [16,89]. The relative strength of groups determines whether they can survive or will be captured, whether they will be able to support nearby friendly groups, or can serve as a basis to 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 [19], enhanced by goal-directed search for surrounded unsettled groups, to check if they can live, or be killed. Current implementations of the group concept are reasonable, but they still miss many of the fine points of play. Strong human players are a lot more flexible in the ways they can group stones together during their analysis. Group-related moves are decisive in many games: Adjacent weak groups of the same color can either connect to create a single, stronger group, or be subjected to a splitting attack. Playing at a junction point between adjacent groups of opposite color effectively combines attack and defense.

3.3. Global move generation

Global move generation and evaluation is typically performed in two phases. In the first phase, a detailed representation of the current state of a Go game is built by static analysis as described above, and enhanced by goal-oriented searches (see Sections 4.1 and 4.2). In the second phase, good or promising moves are selected, and bad moves are filtered out. Examples of filters for suppressing bad moves are those that prevent suicidal and other tactically dubious moves, moves in the opponent's sphere of influence that could be cut off, moves that continue to expand a dead group, or moves inside a safe own or opponent territory.

Table 1
Move generation for a single object

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

In contrast to many other games, move generators in Go almost never generate all the legal moves. Instead, a specific move generator usually implements a single abstract concept. Current programs contain many thousand lines of code that implement a large number of different generators. Often, move generators are bound to a specific object, and propose only moves related to that object. Table 1 contains a small selection. 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** moves from an opening book.

Move generation in Go strives for a difficult balance between the two goals of generating all reasonable moves, while filtering out most of the irrelevant ones. This is in marked contrast to the traditional approach in chess, where selective move generation was found too risky [51]. However, in recent years sound new selective move generation methods have been developed for games such as shogi and Othello [13,29].

Besides generating good moves, it is also important to achieve a good move sorting. In an experiment with guessing moves from professional games made 10 years ago and reported in [55], the program *Explorer* ranked about one third of the moves made by professionals among its top three choices. Another third of the moves was ranked between 4 and 20, and the remaining third of professional moves was either ranked below the top 20 or not generated at all. Figures for the current top programs are likely to be a bit higher than that, but programs are still a long way from being able to properly evaluate a large percentage of the moves made by good human players.

3.4. Move evaluation

There are two basic approaches to move evaluation, with different strengths and weaknesses. The first method, which is used in most other games, is position evaluation. A move is played and then a static evaluation function computes a full board score by estimating the status of each point on the board. This estimate is based mainly on territory computed as in Section 3.2.7, and is adjusted by accounting for weak groups (see Section 3.2.8). 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, since 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 do not increase the territorial score immediately. If the evaluation cannot reliably handle subtle nuances in group strength, many such moves will be evaluated too low.

The second evaluation method is direct move evaluation. The relative value of a move is estimated by the move generator that proposes it, based on heuristics. Moves proposed

by several generators can accumulate 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 consequences of a move accurately.

Many programs use a mix of move evaluation and position evaluation [18]. For example, a program based on position evaluation can give a bonus or a penalty to some types of moves that are often misevaluated. Programs based primarily on move evaluation usually perform some extra steps to improve robustness, such as tactical checks to avoid blunders, and also contain special code to deal with *ko** fights [39].

4. Game tree search in Go

Search is used in computer Go on different levels and in many distinct ways. Three types of game tree search are commonly used: single-goal search (Section 4.1), multiple-goal search (Section 4.2) and full-board search (Section 4.3).

Full board search in Go is difficult, for reasons that have already been touched upon in Section 2.4. However, specialized searches that focus on achieving a more restricted 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

Single-goal search uses standard game tree searching techniques to find 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 of *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 components greatly 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 capturing sequences where all attacker moves are threats to capture and all defender moves are forced replies to avoid immediate capture. Very often there is only a single possible move. Ladders can run across the whole board and bend around, as shown in Fig. 17. Black *a* threatens the three marked stones, and White's only escape is at 2 in the diagram on the right. Black keeps chasing White in a zig-zag pattern across the board, and whether or not the attack succeeds is determined by small nuances in the position in a far-away region of the board. The example is taken from a famous recent tournament game in which one of the best players in the world overlooked the ladder and had to resign. However, no Go program would make such a mistake.

Further topics for which single-goal search has been used are listed in Table 2. A typical current program implements many but—for lack of development resources—probably not all of these goal-oriented searches.

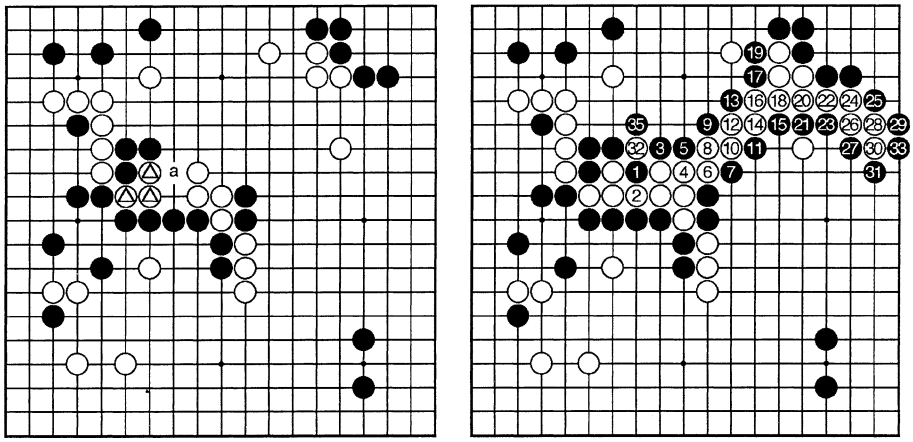

Fig. 17. Tactical capture by ladder (34 at 1).

Table 2
Sample single-goal search tasks

Target	Reference
Ladder	[39]
Single block capture	[15,39,83]
Life and death	[46,91]; Section 5.1
Connect or cut	[27]
Eye status	[22]
Local score	[55]; Section 5.4
Safety of territory	[56]; Section 5.2
Semeai	[27,58]; Section 5.3

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 optimize move selection among all the tactically good moves by using secondary objectives such as good shape or territory.

There is no consensus about which type of search to use for solving tactical problems. Standard iterative deepening alpha-beta search with fractional ply search extensions seems to work well for some types of problems, while proof-number search is better for others. Recently, promising new search methods based on threat analysis have been proposed [15, 83].

Table 3
Examples of multi-goal search tasks

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

4.2. Multiple-goal search

Multiple-goal search tries to achieve an AND- or OR-combination of two or more basic goals. Such combinations can represent 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. Table 3 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 [66,68, 89]. In contrast, the implementation of multi-goal searches is rudimentary in most current programs. However, several recent research papers investigate architectures for adversary and multipurpose strategic planning in Go [33,34,48,90].

4.3. Global search and full-board move decision

There is a great variety of approaches to the problem of global move decision in computer 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 seems to require additional special adjustments [18]. In some programs full board search is mainly used as a kind of quiescence search for weak groups. Most programs use a combination of several 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 of temporary goals, combined with goal-specific move selection, and
- choice of aggressive or defensive play based on the estimated score.

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 still appears to be difficult to integrate such techniques with current Go programs [18,55].

5. Solving subproblems of Go

For many subproblems of Go, specialized methods have been developed which can achieve a much greater heuristic accuracy than general methods, or 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 to integrate such expert modules into full-scale heuristic Go programs.

5.1. Life and Death

Life and Death or *tsumego* is the problem of analyzing whether or not a surrounded, weak group can be made safe from capture, typically by creating two eyes. Most Go programs contain some Life and Death knowledge, typically as a combination of exact and heuristic rules [46]. The performance of normal programs on Life and Death puzzles is not above their overall level of skill. However, programs specialized for solving such problems in a completely enclosed region can do much better. The best such program, Thomas Wolf's *GoTools*, is faster and more accurate than any human in small and medium-size enclosed problems [91,92,95]. Fig. 18 shows two notoriously difficult problems solved by the program [94]. *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 [93]. For open problems and problems involving *semeai**, Wolf estimates the strength of his program at around 1–3 kyu [96].

5.2. Safety of stones and territory

Determining the safety of stones and territory is similar to solving Life and Death problems. 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 on the inside. Fig. 19

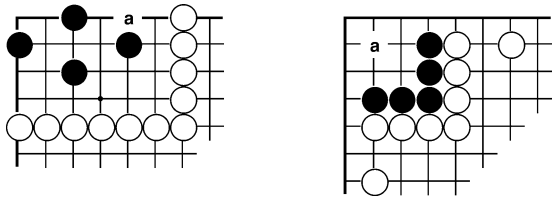

Fig. 18. The *diamond* and the *carpenter's square* (White to play, *a* is the solution).

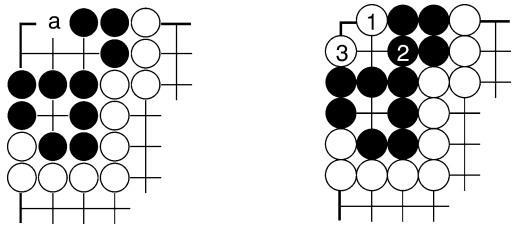

Fig. 19. Safe stones, unsafe territory.

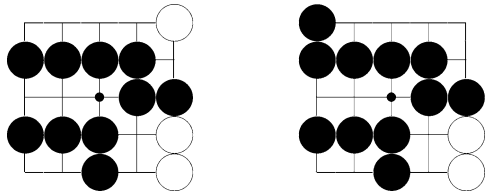

Fig. 20. Unconditional safety, and safety by locally alternating play.

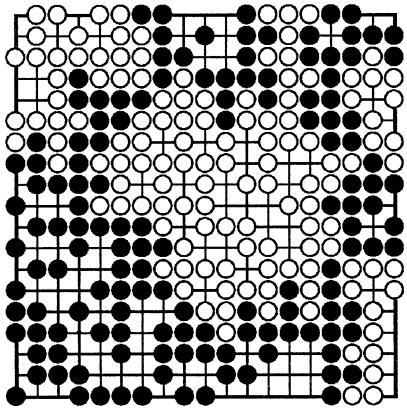

Fig. 21. Proving the safety of blocks and territories.

shows an example where the black stones are safe but the area that they surround is not. White 1 threatens to capture three black stones, so Black 2 is forced. After White 3, no player can continue playing in this area without losing the own stones, so the final result is coexistence, and the black territory has disappeared.

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 Fig. 20 are unconditionally safe. All the black stones always have at least two adjacent empty squares, and White cannot fill them without committing suicide. Benson's method is mathematically elegant but limited in practice, since no defense against any threats is

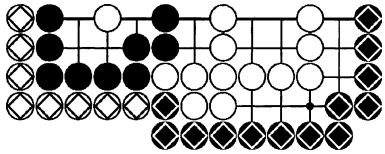

Fig. 22. Semeai solved by static evaluation, from [58].

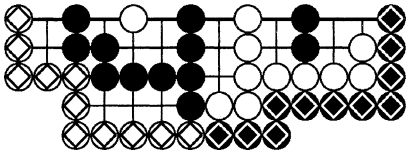

Fig. 23. Semeai solved by search using partial order bounding, from [62].

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 [56]. The right side of Fig. 20 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. In well-subdivided positions near the end of a game, such as the example shown in Fig. 21, every point on the board can quickly be proven safe with current techniques.

5.3. Semeai

*Semeai** are capturing races between unsettled blocks of both colors. In order to survive, blocks in a semeai must capture the opponent, or at least achieve coexistence in *seki**. The strength of blocks in semeai can be measured by two incomparable quantities, liberty count and eye status. Some types of semeai with well-separated small enclosed areas, such as the one in Fig. 22, can be solved by a detailed static analysis [58]. Many more complicated cases, such as Fig. 23, can be solved efficiently by using the search technique of *partial order bounding* [62].

5.4. Endgame

Very late stage Go endgames can be solved by a full board minimax search. However, the solution effort grows exponentially with the total size of the remaining endgame areas [61]. 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* [57, 61]. This technique implements a divide-and-conquer approach based on combinatorial game theory [3,4]. Fig. 24 shows an endgame problem involving a total area of 89 contested points, and the computed solution.

The Berkeley group around Elwyn Berlekamp, Bill Spight and Bill Fraser has pushed the mathematical analysis of local Go positions much further, to the point where real endgames from professional games can be completely analyzed, sometimes more than 100 moves

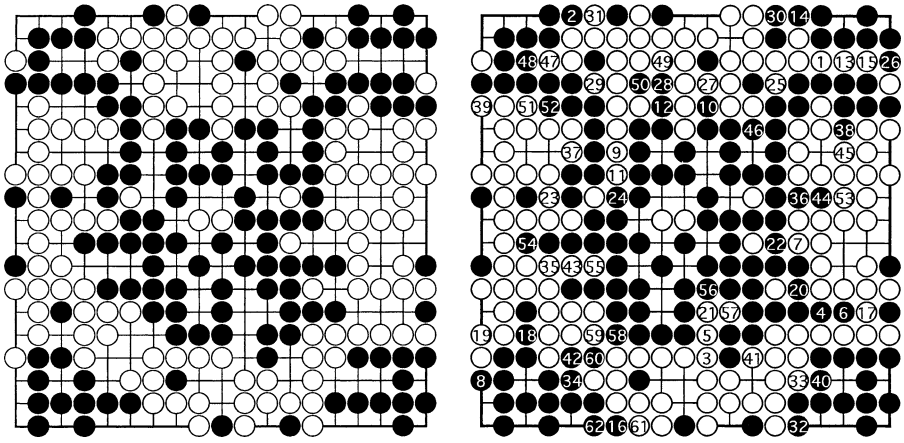

Fig. 24. 89 point area endgame problem and solution, from [57].

back from the end of the game. Their approach uses a mix of human analysis and software tools for evaluation [80]. Further automation of such techniques and their application to computer Go remains as a challenge for the near future.

6. Outlook

Computer Go remains one of the biggest challenges faced by game programmers, and a thorn in the side of AI researchers. This final section contains a long list of challenges for future research in computer Go, as well as a summary of the paper. Some of the research challenges in this domain are:

Develop a search-bound Go program In contrast to most other games, in Go there has not yet been a clear demonstration of a correlation between deeper search and playing strength. Develop 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, maybe along the lines of the *Computer Go Test Collection* [55].

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 with a n stone handicap for some large n. 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, heuristics-based Go program.

Solve small boards Human experts have analyzed Go on many small rectangular boards, but there are few exact proofs [84–86]. Recently, Sei solved the 2×2, 3×3 and 4×4 cases for one version of Japanese rules [78]. Solve Go on other small board sizes. For example, for the 5×5 case the first player should control the whole board, and 7×7 is believed to be a 9 point win for Black.

Count territory Evaluate and contrast approaches for evaluating territory, such as influence, dividers, and connectivity.

Pattern matching Evaluate and empirically test the different approaches to pattern matching and pattern learning in Go. Generalize to more powerful (human-like?) patterns.

Detailed technical evaluation of computer Go methods For many aspects of Go programming there are 3 or 4 reasonable-sounding approaches that are used by different programs. Provide evaluations of the tradeoffs among the choices.

Machine learning Complex systems such as Go programs allow many opportunities for parameter tuning and other, more sophisticated forms of machine learning such as pattern learning. Develop these techniques, using the references in Sections 2.3 and 3.1 as a starting point.

Efficient use of large amounts of memory In single-agent search as well as in other games, large amounts of memory have been used to speed up search [31]. At least some of the leading Go programs already use large precomputed databases of patterns learned from professional games or of eye shapes. Generalize these techniques to make better use of the currently available large memories and disk storage.

Special-purpose hardware Can special purpose hardware be used in Go with as much success as in chess? Which parts of a Go program should be implemented in hardware, and how?

Despite considerable work and much progress in solving specific technical problems, the overall playing strength of Go programs lags behind, compared to most other games. Current programs continue to improve in every aspect of the game, but often prefer to play a peaceful game where these strengths are not so apparent and where the many remaining weaknesses remain hidden. The prevalent style of play circumvents much of the inherent complexity of Go.

A big road block on the way to faster progress is that programs seem to require a large number of tightly coupled components, which leads to software engineering nightmares. It remains unclear whether existing specialized solutions to subproblems can be integrated to create a complete, strong next-generation program, or whether fundamentally new approaches are required.

Acknowledgements

The first version of this paper was written while the author was a postdoctoral researcher at NTT Communication Science Laboratories in Atsugi, Japan. I would like to thank the members of the world wide computer Go community, the Complex Games Lab at ETL Tsukuba, NTT Communication Science Laboratories, and the GAMES group at the University of Alberta for many helpful discussions, and for their valuable feedback and comments on drafts of this paper.

Appendix A. Two representative computer Go games

The two games in this appendix illustrate the state of the art. These and several other illustrative games are available online for replay [60].

A.1. Case study 1: Ing Cup 1999, Go4++ vs. Goemate

Figs. A.1–A.4 show the deciding game of the 1999 Ing Cup, played in Shanghai on November 13, 1999. 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 move 16, both programs follow standard opening principles by first surrounding the corners and then expanding to the sides. The standard *joseki** moves from 16 to 23 are most likely contained in the opening book of both programs. With move 30, *Go4++* starts reducing the large framework that Black has built on the left side. Black invades strongly at 31 and 39, and even though later

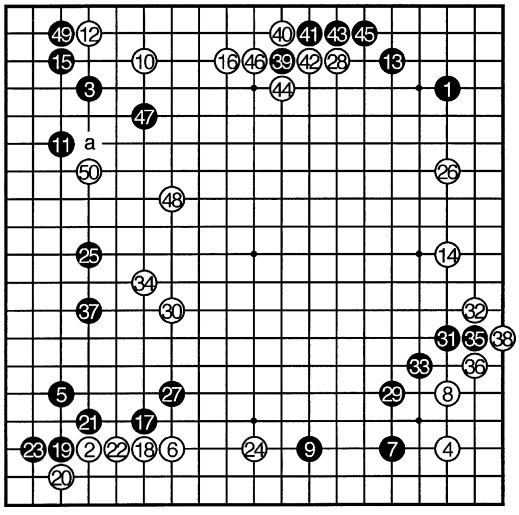

Fig. A.1. Ing Cup 1999: Goemate (B)—Go4++ (W), moves 1–50.

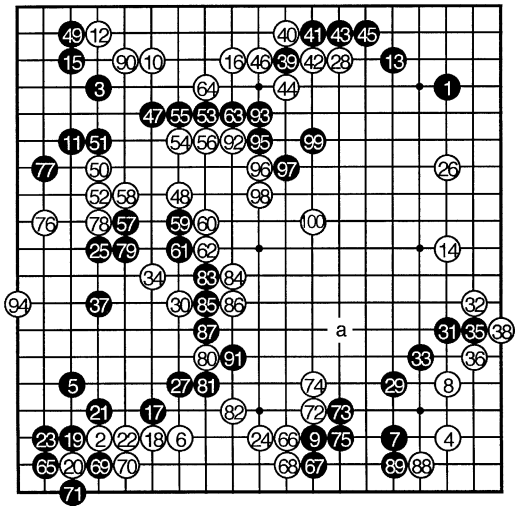

Fig. A.2. Ing Cup 1999: Goemate (B)—Go4++ (W), moves 51–100.

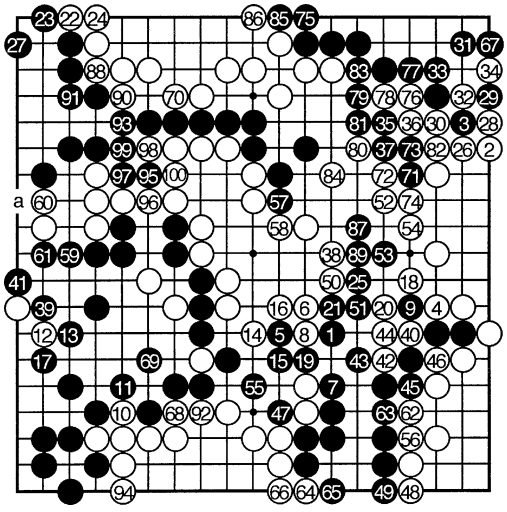

Fig. A.3. Ing Cup 1999: Goemate (B)—Go4++ (W), moves 101–200.

White can connect 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. 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 remains very close. 131 is an inexplicable retreat and must have been caused by a programming bug. Of course Black should just connect at 132. Up

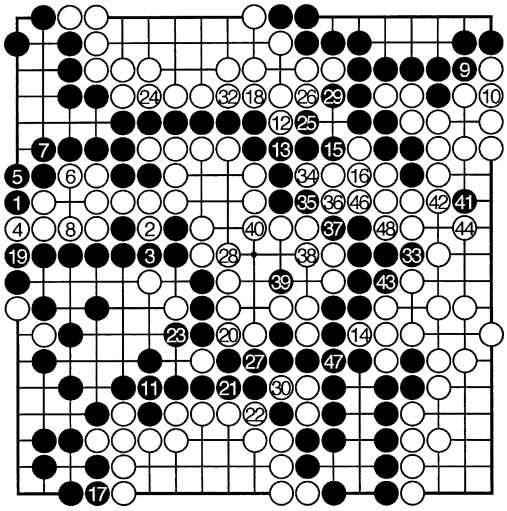

Fig. A.4. Ing Cup 1999: Goemate (B)—Go4++ (W), moves 201–248. 231 captures below 230, 245 passes.

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 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 the early Go programs typically did. This is evident in situations such as move 52, where the program has insufficient detail knowledge to produce a stylish move, but is still able to find the right idea and select a reasonable play in that general area. Top programs are very careful to avoid getting weak groups, and can play a normal endgame. In this game, White was especially skillful in eliminating the opponent's potential *sente** moves.

This game shows current 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.

A.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 David Fotland's program *The Many Faces of Go* and the author, a 6 dan amateur player. Like *Go4++* and *Goemate*, *Many Faces* is one of the strongest Go programs in the world, and a few months after this game 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 case its aggressiveness backfires. Despite the huge handicap, the game ends with a six point win for the human.

Figs. A.5–A.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 corner, White uses a confused *ko** fight to secure one group, then continues to create complications from this basis. By move 85, Black's group in that corner has been reduced to only one *eye**. Black invests too many moves in a failed attempt to rescue this group and in a counterattack against the white stones floating in the center. With move 207

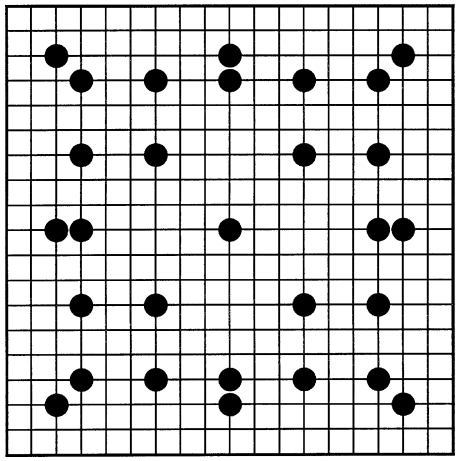

Fig. A.5. Many Faces (B, 29 stones handicap)—Martin Müller (W).

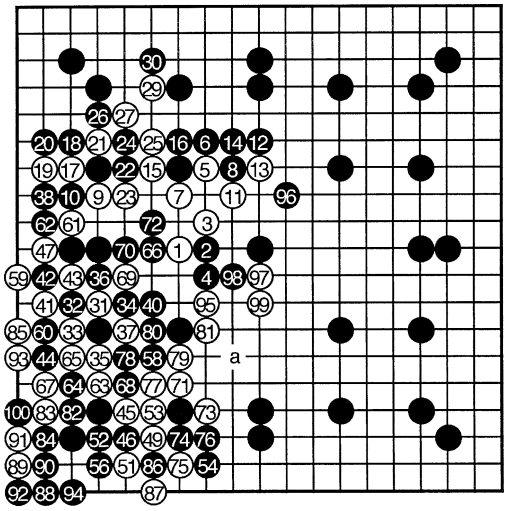

Fig. A.6. Many Faces (B, 29 stones handicap)—Martin Müller (W), moves 1–100. Move 28 is played back at 21, 39 at 31, 48 at 32, 50 captures below 31, 55 at 31, 57 at 43.

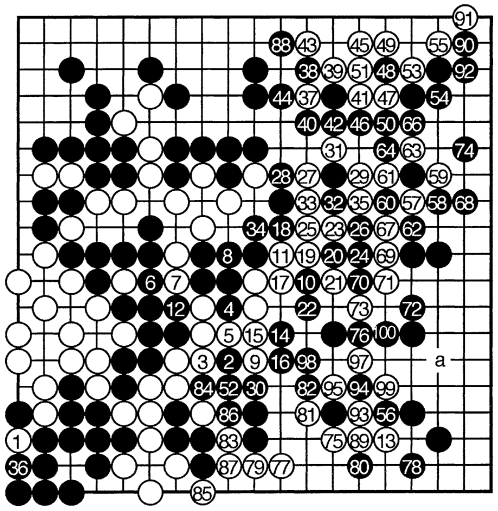

Fig. A.7. Moves 101–200: 165 at 157, 196 connects below 184.

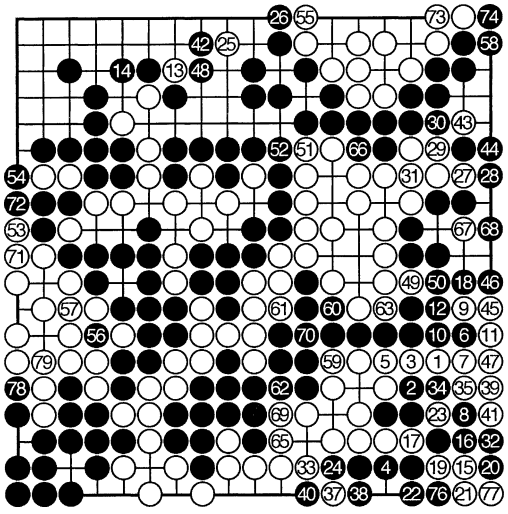

Fig. A.8. Moves 201–279: 236 at 223, 264 connects right of 256, 275 passes.

(7 in Fig. A.8), White isolates another black group in the lower right corner, and kills it a bit later by a combination exploiting a hidden dependency 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 computer 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 a stronger opponent on even terms, instead of preserving some of its huge initial advantage by playing slow, ultra-safe moves. One might argue that a program playing a safer, more territorial style will be harder to overcome. However, a few weeks before this game, in an exhibition match held at the AAAI Conference, professional player Janice Kim had already given *Handtalk* a similar huge handicap and won [52].

Glossary

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

chain Blocks that are virtually connected as in Section 3.2.6.

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

dead Stones that cannot escape capture are called dead, even while they are still on the board.

eye A surrounded area providing one safe liberty. Stones that have two eyes are safe from capture.

group A loosely connected set of blocks of the same color. See Section 3.2.8.

joseki A standard move sequence, often in the corner.

ko A single stone capture-recapture situation that can lead to position repetition, as in Fig. 3.

komi A number of points given to the second player at the end of a game to compensate for the first player's advantage. Usually between 5.5 and 8 points.

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

ladder A basic capturing sequence. See Fig. 17.

liberty An empty point adjacent to a block of stones.

life and death problem The problem whether a weak surrounded group can be made safe from capture, typically by creating two eyes.

seki Coexistence of Black and White blocks that don't have two eyes. See the right side of Fig. 19.

semeai A race to capture. See Fig. 22 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.

territory An area surrounded and controlled by one player. May contain dead opponent stones.

tesuji A skilful tactical move.

References

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

[2] E. Berlekamp, The economist's view of combinatorial games, in: R. Nowakowski (Ed.), Games of No Chance: Combinatorial Games, MSRI Publications, Vol. 29, Cambridge University Press, Cambridge, 1996, pp. 365–405.

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

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

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

[6] B. Bouzy, Modélisation cognitive du joueur de Go, Ph.D. Thesis, Université Paris 6, 1995.

[7] R. Bozulich, The Go Players Almanac, Ishi Press, Tokyo, 1992.

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

[9] J. Burmeister, Memory performance of master Go players, in: H.J. van den Herik, H. Iida (Eds.), Games in AI Research, Universiteit Maastricht, Maastricht, 2000, pp. 271–286.

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

[11] J. Burmeister, J. Wiles, AI techniques used in computer Go, in: Fourth Conference of the Australasian Cognitive Science Society, Newcastle, 1997.

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

[13] M. Buro, ProbCut: An effective selective extension of the alpha-beta algorithm, ICCA J. 18 (2) (1995) 71–76.

[14] T. Cazenave, Système d'apprentissage par auto-observation. Application au jeu de Go, Ph.D. Thesis, Université Paris 6, 1996. http://www.ai.univ-paris8.fr/~cazenave/papers.html.

[15] T. Cazenave, Abstract proof search, in: Proc. 2nd International Conference on Computers and Games, Hamamatsu, Japan, 2000, pp. 81–96.

[16] K. Chen, Group identification in computer Go, in: D. Levy, D. Beal (Eds.), Heuristic Programming in Artificial Intelligence: The First Computer Olympiad, Ellis Horwood, Chichester, 1989, pp. 195–210.

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

[18] K. Chen, Some practical techniques for global search in Go, ICGA J. 23 (2) (2000) 67–74.

[19] K. Chen, Z. Chen, Static analysis of Life and Death in the game of Go, Inform. Sci. 121 (1999) 113–134.

[20] Z. Chen, Dian Nao Wei Qi Xiao Dong Tian (A Kaleidoscope of Computer Go), Zhongshan University Press, 1999 (in Chinese).

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

[22] D. Dyer, An eye shape library for computer Go, 1987. http://www.andromeda.com/people/ddyer/go/shape-library.html.

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

[24] M. Enzenberger, The integration of a priori knowledge into a Go playing neural network, 1996. http://www.markus-enzenberger.de/neurogo.html.

[25] M. Enzenberger, Online computer Go bibliography, 1996–2001. http://www.markus-enzenberger.de/compgo_biblio.html.

[26] D.W. Erbach, Computers and Go, in: R. Bozulich (Ed.), The Go Player's Almanac, The Ishi Press, Tokyo, 1992, Chapter 11.

[27] D. Fotland, Knowledge representation in The Many Faces of Go. Second Cannes/Sophia-Antipolis Go Research Day, 1993.

[28] K.J. Friedenbach, Abstraction hierarchies: A model of perception and cognition in the game of Go, Ph.D. Thesis, University of California, Santa Cruz, CA, 1980.

[29] R. Grimbergen, Plausible move generation using move merit analysis with cut-off thresholds in shogi, in: I. Frank, T.A. Marsland (Eds.), Proceedings of the Second International Conference on Computers and Games, CG2000, Lecture Notes in Computer Science, Vol. 2063, Springer, Heidelberg, 2001, pp. 331–349.

[30] A. Hollosi, SGF FF4—Smart Game Format, 2000. http://www.red-bean.com/sgf.
[31] R. Holte, I. Hernadvolgyi, A space-time tradeoff for memory-based heuristics, in: Proc. AAAI-99, Orlando, FL, 1999, pp. 704–709.
[32] S.C. Hsu, D.Y. Liu, The design and construction of the computer Go program Dragon 2, Computer Go 16 (1991) 3–14.
[33] S. Hu, Multipurpose adversary planning in the game of Go, Ph.D. Thesis, George Mason University, Fairfax, VA, 1995.
[34] S. Hu, P. Lehner, Multipurpose strategic planning in the game of Go, IEEE Transactions on Pattern Analysis and Machine Intelligence 19 (9) (1997) 1048–1051.
[35] R. Jasiek, Go (Weiqi, Baduk) rules; http://home.snafu.de/jasiek/rules.html.
[36] A. Junghanns, Pushing the limits: New developments in single-agent search, Ph.D. Thesis, University of Alberta, Edmonton, AB, 1999.
[37] A. Junghanns, J. Schaeffer, Search versus knowledge in game-playing programs revisited, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 692–697.
[38] K.Y. Kao, Sums of hot and tepid combinatorial games, Ph.D. Thesis, University of North Carolina at Charlotte, NC, 1997.
[39] A. Kierulf, Smart Game Board: A workbench for game-playing programs, with Go and Othello as case studies, Ph.D. Thesis, ETH Zürich, 1990.
[40] A. Kierulf, K. Chen, J. Nievergelt, Smart Game Board and Go Explorer: A study in software and knowledge engineering, Comm. ACM 33 (2) (1990) 152–167.
[41] Y. Kim, New values in Domineering and loopy games in Go, Ph.D. Thesis, University of California at Berkeley, CA, 1995.
[42] T. Klinger, D. Mechner, An architecture for computer Go, 1996. http://www.cns.nyu.edu/~mechner/compgo/acg/.
[43] T. Kojima, Automatic acquisition of Go knowledge from game records: Deductive and evolutionary approaches, Ph.D. Thesis, University of Tokyo, 1998.
[44] T. Kojima, K. Ueda, S. Nagano, An evolutionary algorithm extended by ecological analogy and its application to the game of Go, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 684–689.
[45] T. Kojima, K. Ueda, S. Nagano, Flexible acquisition of various types of Go knowledge, in: H.J. van den Herik, H. Iida (Eds.), Games in AI Research, Universiteit Maastricht, Maastricht, 2000, pp. 221–238.
[46] J. Kraszek, Heuristics in the life and death algorithm of a Go-playing program, Computer Go 9 (1988) 13–24.
[47] H. Landman, Eyespace values in Go, in: R. Nowakowski (Ed.), Games of No Chance, Cambridge University Press, Cambridge, 1996, pp. 227–257.
[48] P. Lehner, Planning in adversity: A computational model of strategic planning in the game of Go, Ph.D. Thesis, University of Michigan, Ann Arbor, MI, 1981.
[49] D. Levy (Ed.), Computer Games I + II, Springer, Berlin, 1988.
[50] D. Lichtenstein, M. Sipser, Go is polynomial-space hard, J. ACM 27 (2) (1980) 393–401.
[51] T.A. Marsland, Computer chess and search, in: S. Shapiro (Ed.), Encyclopedia of Artificial Intelligence, 2nd edn., Wiley, New York, 1992, pp. 224–241.
[52] D. Mechner, All systems Go, The Sciences 38 (1) (1998).
[53] D.J. Moews, On some combinatorial games connected with Go, Ph.D. Thesis, University of California at Berkeley, 1993.
[54] F.L. Morris, Playing disjunctive sums is polynomial space complete, Internat. J. Game Theory 10 (3–4) (1981) 195–205.
[55] M. Müller, Computer Go as a sum of local games: An application of combinatorial game theory, Ph.D. Thesis, ETH Zürich, 1995. Diss. ETH No. 11.006.
[56] M. Müller, Playing it safe: Recognizing secure territories in computer Go by using static rules and search, in: H. Matsubara (Ed.), Game Programming Workshop in Japan '97, Computer Shogi Association, Tokyo, Japan, 1997, pp. 80–86.
[57] M. Müller, Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 578–583.

[58] M. Müller, Race to capture: Analyzing semeai in Go, in: Game Programming Workshop in Japan'99, IPSJ Symposium Series, Vol. 99, No. 14, 1999, pp. 61–68.
[59] M. Müller, Generalized thermography: A new approach to evaluation in computer Go, in: J. van den Herik, H. Iida (Eds.), Games in AI Research, Universiteit Maastricht, Maastricht, 2000, pp. 203–219.
[60] M. Müller, Computer Go survey, 2001. http://www.cs.ualberta.ca/~mmueller/cgo/survey.
[61] M. Müller, Global and local game tree search, Inform. Sci. 135 (3–4) (2001) 187–206.
[62] M. Müller, Partial order bounding: A new approach to evaluation in game tree search, Artificial Intelligence 129 (1–2) (2001) 279–311.
[63] M. Müller, E. Berlekamp, B. Spight, Generalized thermography: Algorithms, implementation, and application to Go endgames, Technical Report 96-030, ICSI Berkeley, CA, 1996.
[64] R. Nowakowski (Ed.), Games of No Chance, Cambridge University Press, Cambridge, 1996.
[65] E. Pettersen, The Computer Go Ladder, 1994–2001. http://www.cgl.ucsf.edu/go/ladder.html.
[66] W. Reitman, J. Kerwin, R. Nado, J. Reitman, B. Wilcox, Goals and plans in a program for playing Go, in: Proc. 29th National Conference of the ACM, San Diego, CA, ACM, New York, 1974, pp. 123–127.
[67] W. Reitman, B. Wilcox, Modeling tactical analysis and problem solving in Go, in: Proc. 10th Annual Pittsburg Conference on Modelling and Simulation, Pittsburg, Instrument Society of America, 1979, pp. 2133–2148.
[68] W. Reitman, B. Wilcox, The structure and performance of the Interim.2 Go program, in: Proc. IJCAI-79, Tokyo, Japan, 1979, pp. 711–719.
[69] P. Ricaud, GOBELIN: Une approche pragmatique de l'abstraction appliquée à la modélisation de la stratégie elémentaire du jeu de Go, Ph.D. Thesis, Université Paris 6, 1995.
[70] J. Robson, The complexity of Go, in: Proc. IFIP (International Federation of Information Processing), 1983, pp. 413–417.
[71] C. Rosin, Coevolutionary search among adversaries, Ph.D. Thesis, University of California, San Diego, CA, 1997.
[72] J.L. Ryder, Heuristic analysis of large trees as generated in the game of Go, Ph.D. Thesis, Stanford University, 1971. Microfilm no. 72-11,654.
[73] Y. Saito, Cognitive scientific study of Go, Ph.D. Thesis, University of Tokyo, 1996 (in Japanese).
[74] N. Sasaki, The neural network programs for games, Ph.D. Thesis, Tohoku University, 1998 (in Japanese).
[75] N. Schraudolph, Temporal difference learning of position evaluation in the game of Go, in: Neural Information Processing Systems, Vol. 6, Morgan Kaufmann, San Mateo, CA, 1994, pp. 817–824.
[76] R. Sedgewick, Algorithms, Addison-Wesley, Reading, MA, 1983.
[77] S. Sei, T. Kawashima, Memory-based approach in Go-program Katsunari. Complex Games Lab Workshop, 1998. http://www.etl.go.jp/etl/divisions/~7236/Events/workshop98/.
[78] S. Sei, T. Kawashima, A solution of Go on 4×4 board by game tree search program, Manuscript, Fujitsu Social Science Laboratory, 2000.
[79] W. Spight, Extended thermography for multiple kos in Go, in: H.J. van den Herik, H. Iida (Eds.), Computers and Games, Proceedings CG'98, Lecture Notes in Computer Science, Vol. 1558, Springer, Berlin, 1999, pp. 232–251.
[80] W. Spight, Go thermography—the 4/21/98 Jiang-Rui endgame, in: R. Nowakowski (Ed.), More Games of No Chance, Cambridge University Press, Cambridge, 2001.
[81] D. Stoutamire, Machine learning, game play and Go, Technical Report TR 91-128, Case Western Reserve University, Cleveland, OH, 1991.
[82] M. Tajima, N. Sanechika, Estimating the possible omission number for groups in Go by the number of nth dame, in: H.J. van den Herik, H. Iida (Eds.), Computers and Games: Proceedings CG'98, Lecture Notes in Computer Science, Vol. 1558, Springer, Berlin, 1999, pp. 265–281.
[83] T. Thomsen, Lambda-search in game trees—with application to Go, ICGA J. 23 (4) (2000) 203–217.
[84] E. Thorp, W. Walden, A partial analysis of Go, Computer J. 7 (3) (1964) 203–207, Reprinted in: [49], Vol. II, pp. 143–151.
[85] E. Thorp, W. Walden, A computer assisted study of Go on $m \times n$ boards, Inform. Sci. 4 (1) (1972) 1–33. Reprinted in: [49], Vol. II, pp. 152–181.
[86] J. Tromp, Small Board Go. Message on computer-go mailing list, 1998.
[87] J. Tromp, On game space size. Message on computer-go mailing list, 1999.
[88] B. Wilcox, Reflections on building two Go programs, SIGART Newsletter 94 (1985) 29–43.

[89] B. Wilcox, Computer Go, in: D.N.L. Levy (Ed.), Computer Games, Vol. 2, Springer, Berlin, 1988, pp. 94–135.
[90] S. Willmott, J. Richardson, A. Bundy, J. Levine, An adversarial planning approach to Go, in: H.J. van den Herik, H. Iida (Eds.), Computers and Games: Proceedings CG-98, Lecture Notes in Computer Science, Vol. 1558, Springer, Berlin, 1999, pp. 93–112.
[91] T. Wolf, Investigating tsumego problems with RisiKo, in: D.N.L. Levy, D.F. Beal (Eds.), Heuristic Programming in Artificial Intelligence 2, Ellis Horwood, Chichester, 1991.
[92] T. Wolf, The program GoTools and its computer-generated tsume go database, in: H. Matsubara (Ed.), Game Programming Workshop in Japan-94, Computer Shogi Association, Tokyo, Japan, 1994, pp. 84–96.
[93] T. Wolf, About problems in generalizing a tsumego program to open positions, in: H. Matsubara (Ed.), Game Programming Workshop in Japan-96, Computer Shogi Association, Tokyo, Japan, 1996, pp. 20–26.
[94] T. Wolf, The diamond, British Go J. 108 (1997) 34–36.
[95] T. Wolf, Forward pruning and other heuristic search techniques in tsume go, Inform. Sci. 122 (2000) 59–76.
[96] T. Wolf, Personal communication, 2001.
[97] D. Wolfe, Mathematics of Go: Chilling Corridors, Ph.D. Thesis, University of California at Berkeley, CA, 1991.
[98] D. Wolfe, The gamesman's toolkit, in: R. Nowakowski (Ed.), Games of No Chance: Combinatorial Games at MSRI, Cambridge University Press, Cambridge, 1996, pp. 93–98.
[99] S.-J. Yan, S.-C. Hsu, Design and implementation of a computer Go program JIMMY 4.0, J. Technology 14 (3) (1999) 423–430.
[100] L. Yedwab, On playing well in a sum of games, Master's Thesis, MIT, Cambridge, MA, 1985. MIT/LCS/TR-348.
[101] H. Yoshii, Move evaluation tree system. Complex Games Lab Workshop, 1998. http://www.etl.go.jp/etl/divisions/~7236/Events/workshop98/.
[102] A. Yoshikawa, T. Kojima, Y. Saito, Relations between skill and the use of terms—An analysis of protocols of the game of Go, in: H.J. van den Herik, H. Iida (Eds.), Computers and Games, Lecture Notes in Computer Science, Vol. 1558, Springer, Berlin, 1999, pp. 282–299.
[103] A.L. Zobrist, Feature extraction and representation for pattern recognition and the game of Go, Ph.D. Thesis, Univ. of Wisconsin, 1970. Microfilm 71-03, 162.