ELSEVIER

Artificial Intelligence 132 (2001) 39–103

Artificial Intelligence

www.elsevier.com/locate/artint

Computer Go: An AI oriented survey

Bruno Bouzy [a,*], Tristan Cazenave [b]

[a] *Université René Descartes (Paris V), UFR de mathématiques et d'informatique, C.R.I.P.5, 45, rue des Saints-Pères, 75270 Paris Cedex 06, France*

[b] *Université Paris 8, Département Informatique, Laboratoire IA 2, rue de la Liberté, 93526 Saint-Denis Cedex, France*

Received 4 May 2000; received in revised form 21 November 2000

Abstract

Since the beginning of AI, mind games have been studied as relevant application fields. Nowadays, some programs are better than human players in most classical games. Their results highlight the efficiency of AI methods that are now quite standard. Such methods are very useful to Go programs, but they do not enable a strong Go program to be built. The problems related to Computer Go require new AI problem solving methods. Given the great number of problems and the diversity of possible solutions, Computer Go is an attractive research domain for AI. Prospective methods of programming the game of Go will probably be of interest in other domains as well. The goal of this paper is to present Computer Go by showing the links between existing studies on Computer Go and different AI related domains: evaluation function, heuristic search, machine learning, automatic knowledge generation, mathematical morphology and cognitive science. In addition, this paper describes both the practical aspects of Go programming, such as program optimization, and various theoretical aspects such as combinatorial game theory, mathematical morphology, and Monte Carlo methods. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Computer Go survey; Artificial intelligence methods; Evaluation function; Heuristic search; Combinatorial game theory; Automatic knowledge acquisition; Cognitive science; Mathematical morphology; Monte Carlo methods

1. Introduction

Since the beginning of AI, mind games, such as Checkers [114,115] or Chess [121], have been studied as application fields for AI. Nowadays, some programs are better than

[*] Corresponding author.

E-mail addresses: bouzy@math-info.univ-paris5.fr (B. Bouzy), cazenave@ai.univ-paris8.fr (T. Cazenave).

human players in most classical games: Deep Blue in Chess [4,67], Chinook in Checkers [117,118], Logistello in Othello [25], Victoria in Go-moku [3]. These results highlight the efficiency of AI methods that are now quite standard.

These methods are very useful to Go programs. However, by themselves, they do not enable the AI community to build a strong Go program. The problems related to Computer Go require new AI problem solving methods. Given the abundance of problems, and the diversity of possible solutions, Computer Go is an attractive research domain for AI. Prospective methods of programming the game of Go will probably be of interest in other domains—for example tree search in classical games is related to AND-OR tree solving, theorem proving, and constraint satisfaction. The current cornerstone in game programming is the Alpha-Beta algorithm. It was discovered in the early stages of AI research, and has been regularly improved ever since. Computer Go programmers are still looking for their cornerstone, which will certainly be more complex than for other games. The Computer Go community has reached an agreement on some unavoidable low level modules such as tactical modules, but specialists still disagree on some other important points. Future programs will probably use the best of all the current possibilities, and link them together in a harmonious way.

The goal of this paper is to present Computer Go by showing the links between existing studies on Computer Go and different AI related domains: evaluation function, heuristic search, machine learning, automatic knowledge generation, mathematical morphology and cognitive science.

To show where the difficulty of Go programming lies, it is first necessary to compare the game of Go to other classical games in a conventional way. In Section 2, we show that the combinatorial complexity is much higher in Go than in other two-player, complete information, games. We also point out that the evaluation of a position can be very complex. Therefore, unlike other games, Section 3 shows that Go programs have poor rankings in the human ranking system and deals with the results obtained when computers compete against human players and when computers play against other computers.

As usual with computer games, we introduce the architecture of a Go program. We examine: the evaluation function, in Section 4; move generation, in Section 5; and tree search, in Section 6. After expounding the key concepts of the evaluation function, based on numerous concepts and viewpoints, we focus on the relationships between tree search and the evaluation function. Tree search is used, both to find a good move in using the evaluation function, and to perform tactical computations useful to calculate the evaluation function.

However, the architecture of a Go program is not composed of these two parts alone. The notion of abstraction plays an important role in Go, and Go programs exhibit structure at different levels, the highest level being the strategic level, and the lowest level being the tactical level. In order to be competitive, every level of a Go program has to be optimized. Therefore Go programmers spend much of their time on optimizations. In Section 7, we present some examples of possible optimizations at different levels of abstraction.

In Section 8, closely related with mathematics, we examine combinatorial game theory [43], which deals with games as the sum of independent sub-games. The game of Go is a global game that can be broken down into many local sub-games. Although local sub-games are generally dependent, this theory offers an appropriate model for the game of Go. More precisely, the strategic level of a program may represent each tactical game by a combinatorial game. This theory is better applied to the end of the game, when local sub-games become independent, and it enables the calculation of the value of the very last moves of the endgame with greater accuracy. In some positions devised to illustrate this theory, some programs play better than the best professional players. The major current difficulty in applying this theory to other sub-games of Go arises from the high dependence between local sub-games.

A problem inherent in Computer Go is that the models with the best results use a lot of knowledge. This need for knowledge makes machine learning, and automatic generation of knowledge, attractive. The best method to obtain an average Go program very rapidly is undoubtedly the temporal difference method. Some symbolic approaches have also been tried, in an attempt automatically to generate tactical knowledge. The two symbolic methods which yield good results are retrograde analysis of small patterns, and logic metaprogramming. By using them, a large number of tactical rules can be generated for the tactical levels of a Go program. We present the different methods that automatically generate Go knowledge in Section 9.

In Section 10, we present a surprisingly effective technique that works quite well for Go: Monte Carlo Go. This technique uses hardly any Go knowledge. However, a very simple program, using this technique, beats classical, and much more complex, programs on small boards (9×9).

The game of Go is a very visual game. Since the beginning of Computer Go, many models of influence have been set up. We provide a formalization of these models with some classical operators of mathematical morphology, in Section 11.

Go is so complex that it can be used to perform interesting cognitive experiments, within a formal setting imposed by the rules of the game. Section 12 centers on the studies carried out on Go, using a cognitive approach.

The material used in this survey is based on existing Computer Go publications and on the authors' own experience of writing Go programs. Unfortunately, programs whose authors do not describe their algorithms and, furthermore, keep them secret, do not explicitly appear in this survey. Nevertheless, for the strongest commercial programs in this category, we tried to gather some personal communications that were sent to the Computer Go mailing list. We could then mention these programs, and give short descriptions of them. We also base descriptions of the main components of this survey on our own experience of writing Go programs: Indigo [15,17,18,20,21], Gogol [28–30] and Golois [32–34]. We think that Computer Go remains a new domain for computer science, and so far, no clear theoretical model has emerged. The domain greatly benefits from studies based on practical experiments. For instance, the Evaluation Function section mainly refers to the Indigo program, and the Automatic Knowledge Generation section to Golois. Nevertheless, we do not limit our descriptions to these programs, and enlarge upon them using the relevant Computer Go publications.

We choose not to include the rules of the game of Go in this paper. If the reader wishes to know them, he can refer to http://www.usgo.org/resources/whatisgo.html where he will find the definitions of “intersection”, “stone”, “string”, “liberty”, “atari”, “ko”, “group”, “eye”, “tsumego”, “life”, “death”, “territory”, “influence”, “handicap”, “kyu” and “dan” which are Go concepts used by our paper.

2. Other games

2.1. Introduction

This section centers on the current achievements in computer implementations of other two-player, complete information, and zero-sum, games, which we also call “other games”. Our aim is to show that the game of Go is more complex than these “other games”. First, we focus on the detailed results for each game. Secondly, we review the theoretical complexity of some of these games. Then we study the space states, and game tree complexity [1] of these games, correlate these with the level reached on the human scale. Lastly, we outline the complexity of Go.

2.2. Results achieved in other games

In this paragraph, we choose several games within the class of “other games”: Go-moku, Othello, Checkers, Draughts, Chess, and Shogi. Although it does not belong to this class, we also add Backgammon to our study.

Go-moku

Go-moku is the game in which you must put five beads in a row—either horizontally, vertically, or diagonally. Several variants exist depending on the size of the board, and the optional use of capture rules. The simplest variant (no capture) is worth considering for implementation as a Computer Game because it is an example of a solved game, since [3] exhibited the winning strategy. Victoria is the best Go-moku program.

Backgammon

Backgammon is not a complete information game (because the players throw two dice) but the techniques used to program Backgammon are interesting, therefore we include it in our set of other games. The best Backgammon program, TD-Gammon, is a Neural Net program, which learned the game only by playing against itself. This program is on a par with the best human players. It was developed at IBM by Tesauro [127,128,130]. It is clearly stronger than other Backgammon programs and it uses the Temporal Difference algorithm [125]. As early as 1980, thanks to some lucky throws of dice, a program [11] beat the human world champion Luigi Villa.

Othello

Logistello [25] is by far the best Othello program. It is based on a well-tuned evaluation function, which was built by using example-learning techniques; on an opening book; and on a selective search. It won numerous computer tournaments. In August 1997, it beat the world champion Mr. Murakami with a straight win 6–0. But in January 1998, Michael Buro, its author, decided to work on different challenges.

Checkers

Computer Checkers met with very early success because. As early as 1959, Samuel developed a Checkers program which won against a “strong” Checkers player [114]. This program was a learning program. However, it is hard to assess its result because the strength of the “strong” player is debatable. This program is also described in [115].

Since 1988, Jonathan Schaeffer, and several researchers at the Alberta University, have been developing the Chinook program [117]. This program is the Computer Checkers World champion which played a match against the human world champion, Marion Tinsley, in 1992 (4 defeats, 2 wins and 33 draws). Marion Tinsley had been the world champion since 1954 and was very fond of Checkers programs. After 6 draws, the 1994 series was interrupted because Marion Tinsley was very seriously ill. Then, Chinook competed against Don Lafferty in 1994 (1 victory, 1 defeat and 18 draws), and in 1995 (1 victory and 31 draws). Ever since, Chinook has been considered as the world champion, both in the human, and in the machine categories. Chinook uses an endgame database, with all the positions containing fewer than 8 pieces. Chinook uses parallelism, an opening book, and an extended Checkers knowledge base. A complete history of this development can be found in [118].

Draughts

The best Draughts program is Dutch. Its name is Truus, and it can be ranked at a national level. New programs (Flits 95 and Dios 97) are now threatening Truus. Flits 95 won the latest two Dutch Computer Draughts championships, although Truus did not participate in the most recent tournament.

Chess

Shannon [121] showed that computer Chess was a good problem for AI to solve because of the cleverness of its rules and the simplicity of the winning goal. For Shannon, this problem was neither too simple nor too difficult. Since then, intensive research has been done, and the paradigm is clearly tree search, and Alpha-Beta [4]. In 1988, Gary Kasparov, the world champion, claimed that a computer had no chance of beating him before the year 2000. At that time, Deep Thought—IBM hardware and software—had in fact only an international Grandmaster rating. Deep Thought was using tree search with Alpha-Beta. It was exploring about 500,000 positions per second [67]. In May 1997, Deep Blue—the massively parallel descendant of Deep Thought—beat Kasparov with a score of 3.5–2.5. It was also using Alpha-Beta, but exploring around one billion positions per second. Given the popularity of Chess, and the increasing use of computers in everyday life, this success made a strong impression on everybody. Moreover, experiments in Chess have established a correlation between tree search depth, and the level of the resulting program.

Shogi

After the success obtained in Chess, the game of Shogi, or Japanese Chess, was, and still is, the next target for Computer Games [84]. Its complexity is greater than the complexity of Chess because of the possible re-introduction, on the board, of previously captured pieces. The branching factor is about 100 in Shogi, as against 35 in Chess. The position is very difficult to evaluate. The best program [143] uses a variant of iterative deepening tree search, and can be ranked at an average level on the human scale.

2.3. Theoretical complexity

When discussing the complexity of games, it is necessary to mention published results about the theoretical complexity of games. Checkers [109] and Go [110] are exponential time complete, as a function of the size of the board. Fraenkel and Lichtenstein [54] have shown that playing a perfect strategy in n by n Chess requires exponential time. Lichtenstein and Sipser [82] have shown that Go is polynomial-space hard. These theoretical results show that Go seems to be even more complex than Checkers and Chess, because these two games have not been proved polynomial-space hard.

2.4. Space states and game tree complexity of other games

By taking the complexity of games into account, a very good classification of two-player, complete information, zero-sum, games has been established by Allis [1] and van den Herik et al. [63]. This section briefly sums up this classification. Allis [1] defined the space states complexity (E) as the number of positions you can reach from the starting position, and the game tree complexity (A) as the number of nodes in the smallest tree necessary to solve the game. For a given game, it is possible to compute these numbers accurately but approximations may provide useful information. Allis gave rough estimations of E and A for each game, as shown in Table 1. In this table, '>' (respectively '>=', and '< <') mean "is stronger than" (respectively "is stronger than or equal to", and "is clearly weaker than"). 'H' represents the best human player.

At first glance, Table 1 shows a correlation between game complexity, and the results obtained by computers on the human scale. Chinook and Logistello are clearly better than the best human player, in Checkers and Othello respectively. Deep Blue has a rank similar to the best Chess player, and Handtalk is clearly weaker than the best human Go players. All these games have increasing complexity. The classical model may be said to consist of the set: evaluation function, move generation, and tree search. This has been used successfully in Chess, Othello and Checkers. Its relative success depends on the complexity of the game to which it is applied. One can observe a correlation between a game's complexity, and a program's results on the human scale. This correlation can be explained by the fact that the same model is being applied to similar games with different complexities.

Table 1

Game	$\log_{10}(E)$	$\log_{10}(A)$	Computer–human results
Checkers	17	32	Chinook > H
Othello	30	58	Logistello > H
Chess	50	123	Deep Blue >= H
Go	160	400	Handtalk < < H

Table 2

Game	$\log_{10}(E)$	$\log_{10}(A)$	Computer–human results
Checkers	17	32	Chinook > H
Othello	30	58	Logistello > H
9×9 Go	40	85	Strongest Go program < < H
Chess	50	123	Deep Blue >= H
15×15 Go-moku	100	80	The game is solved
19×19 Go	160	400	Strongest Go program < < H

2.5. *Complexity of Go*

It is important that the previous correlation be re-evaluated with respect to Go. First, the classical model cannot work in Go without major adaptations. Moreover, we now add two other games—9×9 Go and 15×15 Go-moku—to elaborate Table 2.

We can see that, on the one hand, 15×15 Go-moku is complex by Allis' standards, and yet Allis' program succeeded in solving this game. On the other hand, 9×9 Go is less complex than Chess by Allis' standards, but the 9×9 programs are still weak when compared with human players.[1] The complexity–result correlation has vanished, and this is difficult to explain. Of course, one might argue that Computer 9×9 Go has not been studied enough because of the limited interest that 9×9 Go enjoys compared to Chess. We do not share this viewpoint—9×9 Go is an obstacle for the computer because there is complexity inherent in the Evaluation Function.

For other games, like Othello, Checkers and Chess, good solutions have already been found, using the classical model. To program these games, there is no reason to change the model, which consists in an effective tree search, using a simple move generation heuristic, and a simple evaluation function. With Go, however, researchers have to look for a new model that enables programs to overcome the complexity of the game. They must reverse the model, focus on the complexity of the evaluation function, and on move generation, and only use tree search for verification.

3. Results

This section contains the results achieved by Computer Go since 1960. It first traces the history of Computer Go, and then deals with the current competitions between programs. In a third part, confrontations between man and machine are examined, and lastly we focus on the results obtained with sub-problems of the game of Go such as Tsume-Go and late endgames.

[1] It is difficult to determine which program is the best on 9×9 boards, because of the lack of 9×9 competitions. Nevertheless, Go4++ is at the top of the 9×9 Computer Go ladder on the Internet.

3.1. History of Computer Go

It seems that the first Go program was written by D. Lefkovitz [81]. The first scientific paper about Computer Go was published in 1963 [106], and it considered the possibility of applying machine learning to the game of Go. The first Go program to beat a human player (an absolute beginner at that time) was the program created by Zobrist [147,149]. It was mainly based on the computation of a potential function that approximated the influence of stones. Zobrist made another major contribution to computer games by devising a general, and efficient, method for hashing a position. It consists of associating a random hash code with each possible move in a game, the hash of a position being the XOR of all the moves made to reach the position [148]. The second thesis on Computer Go is Ryder's [111]. The first Go programs were exclusively based on an influence function: a stone radiates influence on the surrounding intersections (the black stones radiate by using the opposite values of the white stones), and the radiation decreases with the distance. These functions are still used in most Go programs. For example, in Go Intellect [36–38], the influence is proportional to $1/2^{\text{distance}}$, whereas it is proportional to $1/\text{distance}$, in Many Faces of Go [50,51].

Since the early studies in this field, people have worked on sub-problems of the game of Go—either small boards [133,134], or localized problems like the life and death of groups [7].

The first Go program to play better than an absolute beginner was a program designed by Bruce Wilcox. It illustrates the subsequent generation of Go programs that used abstract representations of the board, and reasoned about groups. He developed the theory of sector lines, dividing the board into zones, so as to reason about these zones [105,137]. The use of abstractions was also studied by Friedenbach [55].

The next breakthrough was the intensive use of patterns to recognize typical situations and to suggest moves. Goliath exemplifies this approach [13].

State-of-the-art programs use all these techniques, and rely on many rapid tactical searches, as well as on slower searches on groups, and eventually on global searches. They use both patterns and abstract data structures.

Current studies focus on combinatorial game theory [70,90], learning [30,48], abstraction, and planification [65,107,108], and cognitive modeling [15].

The eighties, saw Computer Go become a field of research, with international competitions between programs. They also saw the first issue of a journal devoted to Computer Go, as well as the release of the first versions of commercial programs. In the nineties, many programs were developed, and competitions between programs flourished, being regularly attended by up to 40 participants of all nationalities [53]. An analysis of the current state of the Computer Go community has been published by Martin Müller [93].

3.2. Computer Go competitions

The oldest international Computer Go competition is the Ing cup. It has been organized every year from 1987 until 2000. The winner of the Ing cup plays against young talented Go players (see Section 3.3 below). Year 2000 was, unfortunately, the last year for the Ing competition. A lot of programs were attracted to a recent competition, the FOST cup, which takes place every year in Tokyo (except for 1997, when it was in Nagoya).

Table 3
Winners of Ing cups

Year	Winner
1987	Friday
1988	Codan
1989	Goliath
1990	Goliath
1991	Goliath
1992	Go Intellect
1993	Handtalk
1994	Go Intellect
1995	Handtalk
1996	Handtalk
1997	Handtalk
1998	Many Faces of Go
1999	Go4++
2000	Wulu

Table 4
Winners of FOST cups

Year	Winner
1995	Handtalk
1996	Handtalk
1997	Handtalk
1998	Handtalk
1999	KCC Igo

Other competitions, like the Mind Sport Olympiad, the European, and the American, championships, are organized on a regular basis.

The winners of the Ing cups and FOST cups are shown in Tables 3 and 4, respectively.

As well as the competitions, there is a permanent Internet Computer Go tournament—the Computer Go ladder (http://www.cgl.ucsf.edu/go/ladder.html). It is a “handicap” ladder; where the number of handicap stones that each participant can give to the immediate lower program is explicitly tracked. Whenever the author of a program feels that his program has been improved, he can issue a challenge, either to the program below (to increase the number of handicap stones), or to the program above (to decrease the number of handicap stones). New programs can join the ladder by challenging the program on the “bottom rung” (no handicap). If the new program wins the challenge, it can successively challenge higher programs until it loses. It can then start playing handicap challenges to determine its exact ranking. Challenges are normally played on the IGS (Internet Go Server, http://igs.joyjoy.net/) or NNGS (No Name Go Server, http://nngs.cosmic.org/). IGS and NNGS provide any Go player in the world with an opponent to play games with, as well as the opportunity to watch games, or comment on them, at any time. They are similar to world wide Go clubs. Of course, Go programs may get an account. Many Faces of Go, and GnuGo, are very often connected to these servers.

3.3. Programs versus human players

In addition to the confrontations that are organized every year, after the Ing cup, other confrontations are organized, in an attempt to understand better the strengths and weaknesses of the programs. For example, after each FOST cup, the three best programs play against human players. Handtalk received a Japanese 3rd Kyu diploma for winning its games. However, an opponent who knows the weaknesses of a program can use this knowledge to win easily. For example, during AAAI-98, Janice Kim beat Handtalk, despite an enormous handicap of more than twenty stones. Recently, Martin Müller beat Many Faces of Go, despite a huge handicap of twenty-nine stones. Although Go programs have been improved over the last few years, they are still much weaker than human players.

3.4. Tsume-Go

Most Go programs have Tsume-Go problem solvers. Some other programs are entirely dedicated to Tsume-Go. The best Tsume-Go problem solver is Thomas Wolf's GoTools [140,142]. GoTools is a very strong Tsume-Go problem solver. It can solve 5-dan problems (an amateur 5-dan is roughly equivalent to a professional 1-dan Go player). It has even spotted an error in a dictionary of Tsume-Go problems. It can analyze complex situations completely, and find unique winning moves that Go players find with great difficulty. The problem solver has been used to generate thousands of Tsume-Go problems.

GoTools relies on Alpha-Beta searching, search heuristics, and numerous hand-coded, and tuned, patterns for directing search, and for evaluating positions. Many heuristics used in GoTools, including forward pruning, are well described in [142]. However, GoTools is restricted to completely enclosed problems that contain thirteen or fewer empty intersections [141]. This restriction makes GoTools of little use for programs that play the entire game, and for Tsume-Go problems that are to be solved in real games.

3.5. Combinatorial game theory

In some late endgame positions of the game of Go, where combinatorial game theory applies, Wolfe's program finds a sequence one point better than the sequence found by professional players [8,9]. Müller's [94] is another demonstration of the power of combinatorial game theory applied to Go endgames. It shows how Decomposition Search, a tree search algorithm based on combinatorial game theory, gives clearly better results than Alpha-Beta, when applied to specific endgame positions. Combinatorial game theory has also been used by Howard Landman to find the number of eyes of a group [79], thus enabling a program to break down a life and death problem into a sum of games, so as to reduce its complexity. Furthermore Müller [95] described a method for modeling "fights" in Go, and computing their game values.

4. Evaluation

4.1. Introduction

This section deals with the major difficulty of Computer Go—building the Evaluation Function (EF). The evaluation of a position is necessary for a program that wants to associate a score with a game. Finding a "good" EF is very hard, and is undoubtedly the biggest obstacle in Computer Go. Whenever Chess programmers—very confident in the power and generality of tree search methods, and willing to try their chance in another game—ask Go programmers, very innocently, to give them the EF of Go, they are very surprised to see that Go programmers cannot provide them with a simple, clear, and efficient EF, as is the case in Chess. Instead of tree search optimizations, it is the discovery of the EF for the game of Go that is the main task of Go programmers. Of course, each Go programmer has their own EF. Every EF results from intensive modeling, programming, and testing activities. Consequently, each EF is different from every other one, and no agreed model has clearly emerged in the community. Therefore the task of presenting a Go EF is far from being easy.

To be as clear as possible, we choose to proceed in two steps. First, we focus on the idea that comes naturally to the new Go programmer's mind—the *concrete* EF. It is simple, and quick, but very inefficient when integrated into tree search algorithms. Then, we show the *conceptual* EF of a Go program. More precisely, we choose to present the conceptual EF of the program Indigo [15,17]. This has two advantages. First, it is clear—because we are the programmers of this EF. Secondly, it works—since it is actually integrated into a playing program that regularly attends the current Computer Go tournaments. To simplify the reader's task, we focus only on the main features of this EF. We have intentionally hidden those parts which are needed to make the EF work in practice, but which are not required for an overall understanding. We mention other formal descriptions, such as the best programs' ones [14,36,39], when they have been published. Given that most programmers wish to keep their algorithms secret, descriptions of the best commercial programs are scarce. They are often personal communications [41,52,103,138,139].

4.2. Concrete evaluation

The first idea consists of defining a concrete EF by giving one value to each intersection of the board: $+1$ for black intersections, and for empty intersections with black neighboring intersections only; -1 for white intersections, and for empty intersections with white neighboring intersections only; 0 elsewhere. Obviously, this EF cannot be simpler.

Explicit-control and implicit-control endgame positions

In Fig. 1 the intersections are explicitly controlled: an intersection controlled by one color has the property of either having one stone of this color on it, or the impossibility of putting another color stone on it. Such a position is reached after a large number of moves, and the two players may have agreed on the control of the whole board a long time before. Fig. 2 shows a board where the game stops earlier. In this position, the control is implicit. Human players stop playing in this kind of implicit-control position. When considering the positions that belong to the set of explicit-control endgame positions, the concrete EF gives correct results, and is quickly computed. Unfortunately, this EF is relevant to positions of this set only.

When considering implicit-control endgame positions, this concrete EF gives erroneous results because human players use a large amount of knowledge to recognize them as controlled. The knowledge contained in the concrete evaluation is not sufficient to recognize them as terminal positions. For example, the empty intersections in the bottom right of Fig. 2, and the "isolated" white stone in the same figure are considered as belonging to Black by almost all Go players. Clearly, the concrete EF gives a false evaluation for

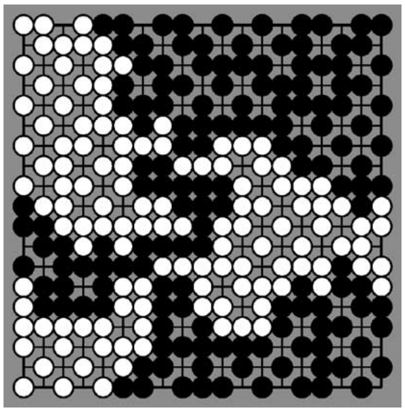
Fig. 1.

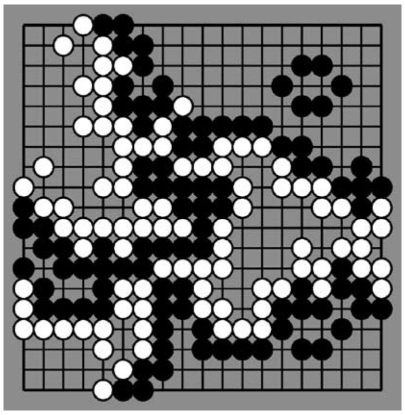
Fig. 2.

Table 5

	Board size		
	9 × 9	13 × 13	19 × 19
Implicit-control endgame depth	60	120	250
Explicit-control endgame depth	160	340	720

them, and the knowledge necessary to explain why these intersections belong to Black would take too long to explain.

But, one should see whether this concrete EF could be used, within a tree search algorithm, so that the EF is invoked in explicit-control endgame positions only. Let us define the depth of a position as the distance between the root node of the game tree and the node of the position. Because human players stop their games on reaching agreement on implicit control, the length of games between human players gives a rough estimate of the depth of implicit-control endgame positions on different board sizes. In addition, a program using the concrete EF, and playing against itself, enables us to estimate the depth of explicit-control endgame positions. Computer experiments show that the average depth of explicit-control endgame positions is twice the board size. These estimates are summarized in Table 5. Although the concrete EF can be computed very quickly, modern computers cannot complete searches down to this depth with the branching factor of Go. As we are again confronted with the combinatorial obstacle, we must give up this approach.

Then, the next step is to try tree search with a *conceptual* EF. This EF will enable the program to evaluate some positions at every stage of the game (in particular the set of implicit-control endgame positions). Of course, because of the numerous possibilities, the next obstacle is the definition of this EF. Anyway, this approach is used by the best current Go programs.

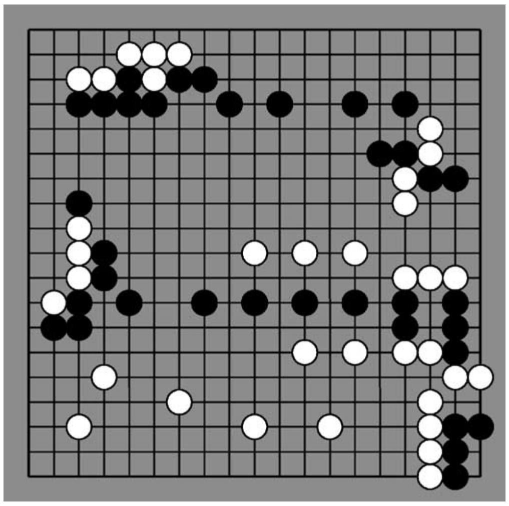

Fig. 3.

4.3. Conceptual evaluation

A worthwhile approach to finding a conceptual EF is to observe human players, to capture the useful concepts, and to transform them into computational ones. Some important human concepts may be equated with their expressions inside game commentaries or Go books. The main Go terms are “group”, “inside”, “outside”, “territory”, “interaction”, “life” and “death”. Other important concepts, such as “inversion” and aggregation” correspond to computational tasks, and we also present them. To illustrate our description, we use Fig. 3.

We present the useful concepts in a bottom-up fashion. We start with small shapes which enable the program to build abstract groups, and we end up with the whole EF. First, we show topological concepts such as “connected group”, then we show morphological concepts such as “territory”, “influence”, “morphological group”, “inside” and “outside”. Finally, we show the concepts of “interaction”, “life” and “death”, together with “inversion” and “aggregation”. These concepts will allow us to finish the presentation with the full conceptual EF. Fig. 3 is used as a reference point to show examples of the different concepts presented in this section.

“Connected group”

In this paragraph, the goal is to define “connected group”. The rules of the game define strings of stones as same-colored 4-connex sets (one intersection has up to 4 neighbors), but, in fact, “connected groups” are what players reason about.

Let us consider two neighboring strings of the same color. Two tree searches may be performed (one search with Black playing first, and another search with White playing first—see “Tree Search” or “Combinatorial Game Theory” sections of this paper) to

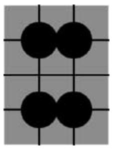
Fig. 4.

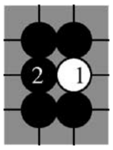

Fig. 5.

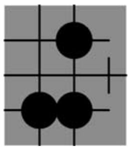
Fig. 6.

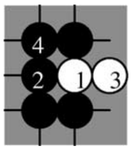

Fig. 7.

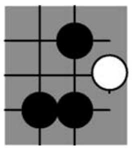
Fig. 8.

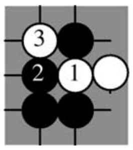

Fig. 9.

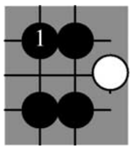

Fig. 10.

determine whether these strings are virtually connected or not. When two strings are virtually connected, they belong to the same "connected group".

In Fig. 4, the two black strings are virtually connected. Even if White plays first, Black will be able to connect (see Fig. 5). If Black plays first, the two strings will obviously be connected. Fig. 4 is called a "connector". Its notation will be '>', so as to indicate that the outcome of this elementary game is an effective connection whoever plays first. More generally, two strings sharing two liberties are also part of the same connected group [36, 38] because if one player plays on one of them, the other player plays on the other one.

The two black strings in Fig. 6 are also virtually connected, as proved by the sequence of Fig. 7. (White 3 is a forced move because White 1 is in "atari" after Black 2.) Fig. 6 is another example of connector '>'.

Fig. 8 is not a connector, as previously described. If White plays first (Fig. 9), the two black strings are actually disconnected by White 1 and White 3. If Black plays first (Fig. 10), the two black strings are connected because Fig. 10 equals Fig. 4. In this case (Fig. 8), the final state of the connector depends on who moves first, and we give the value '*' to the connector.

Then, the "connected groups" are defined as groups of strings linked with connectors '>'. In our example, our program recognizes the connected group of Fig. 11.

A very important point to underline here is the fact that the construction of connected groups implies the use of results from local tree searches having the goal of connection. We will take up this point in the "Tree Search" section of this paper because this is specific to the game of Go: the EF uses Tree Search. This is one important aspect of the EF in Go.

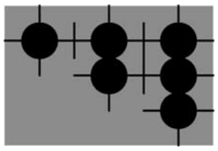

Fig. 11.

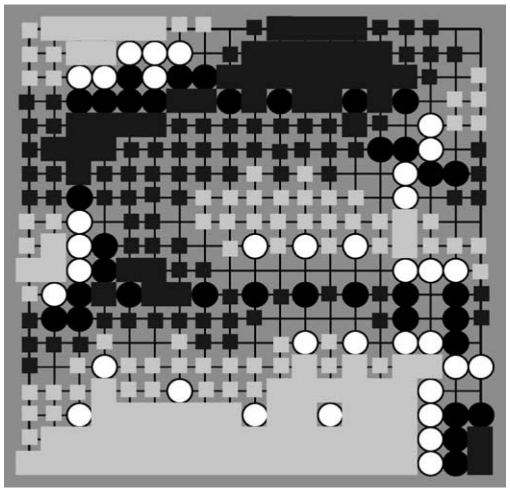

Fig. 12.

“Inside”, “outside” and “morphological group”

The concepts “group”, “territory”, and “influence” are very closely linked to mathematical morphology. We warmly invite the reader to refer to the “Mathematical morphology” section of the paper so as to be familiar with some mathematical morphology operators [120], and with the operators X and Y used in this paragraph.

Let B (respectively W) be the set of black (respectively white) intersections of the Go board. The “morphological groups” are the connected sets of $X(B)$ and of $X(W)$.

Let G be a given morphological group. First, we call $S(G)$ the “skeleton” of G as the subset of G with intersections of the same color as G. Then we define the “inside” of G, $\mathrm{In}(G)$, as the set $G - S(G)$. Lastly, we call $\mathrm{Out}(G)$ the “outside” of G, and define it as the set $Y(S(G)) - G$.

All these operations lead to a morphological view of positions. Fig. 12 shows the morphological view of our example (Fig. 3). The big dark (respectively light) grey squares correspond to the “insides” of black (respectively white) morphological groups, and the small dark (respectively light) grey squares correspond to the “outsides” of black (respectively white) morphological groups. The stones correspond to the skeletons.

The building of groups

The notion of “group” is of paramount importance in modeling a Go position. For human players, the notion of group corresponds neither to the connected group notion nor to the morphological group notion, but to a certain extent to both notions. For programs, the question of knowing which notion is better remains an open problem. Besides, deciding which is the better notion, is a matter of integration within the whole program [14,36,50, 51].

The connected group notion may be used because the connection patterns between stones can be demonstrated by tree searches. But this notion also raises diverse questions. First, the program’s response time may be excessive because of the number of local tree searches that have to be completed to determine the existence of connectors. Nevertheless, this may be speeded up by using patterns, but then the problem is how to handle the patterns database. Some learning techniques may be used to foster the expansion of the patterns database. We shall discuss this point in the “Automatic knowledge generation” section of this paper. Furthermore, another problem linked to the connected group notion is the fact that the connection concept is given undue value. Thus, we might obtain a program which systematically connects its groups before doing anything else.

However, the morphological group notion may also be chosen because it is more intuitive, quicker to compute, and more remote from the connection notion. Unfortunately, it is accurate only in some quiet positions. Therefore tree searches using this notion must reach these quiet positions, if they are, to provide significant results. The more powerful the computer is, the more successful it will be in reaching these quiet positions.

Go4++ uses another idea for building groups: instead of using connection whose values are game-like values: >, *, <, it uses a probability value for each connection and builds a connection map.

Number of “eyes”

The “inside” of a given group is crucial to the life of the group. Its vital contribution is assessed by counting the number of “eyes” of the inside. The number of eyes depends on who plays first. When an inside has more than two eyes, it is alive. For each connected set of the inside, the vital contribution depends on its possibility to be split into other connected sets. On the whole, for size-one-two-or-three connected sets, the number of eyes depends on its boundary, and opponent stones. For sizes from four to about six, it also depends on its shape and on the shape of prisoners. For example, Go players say that “straight 4 is alive” and “square 4 is dead”. For size bigger than about six, the number of eyes becomes greater than two. Each Go program must contain such expertise in counting eyes of a connected set. It is very difficult to define complete and adequate rules for determining the number of eyes of groups. Most of the current Go programs use heuristic rules. The most complete and recent description of these rules are described in a reference paper [39] by Chen and Chen, the authors of two of the best Go programs, Handtalk and Go Intellect. Landman’s [79] is a combinatorial game approach study of eyes. Benson’s [7] is a mathematical study of real life without alternating play.

Furthermore, the heuristic rules must be computed quickly. Therefore, Go programs sometimes use tricks. For example, one heuristic rule says that a connected set whose exterior has a length smaller than 6 has zero or one eye, and when greater than 10, has

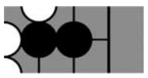
Fig. 13.

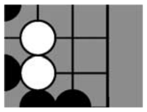
Fig. 14.

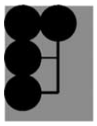
Fig. 15.

two eyes at least [14,39]. After assessing the number of eyes of each connected set, the problem is to count the number of eyes of the whole group. When the connected sets of the inside are not dependent on one another Chen and Chen [39] provide a clever method using binary trees. Each connected set of the inside is a binary tree whose leaves contain the number of eyes, depending on who plays first. The group side selects a binary tree and replaces it by its left sub-tree. The opponent selects a binary tree and replaces it by its right sub-tree and so on. The group side wants to maximize the number of eyes whereas the opponent wants to minimize it. This abstract minimax search is simpler and faster than a regular one on a concrete board.

“Interaction”

A group has friendly and opposing neighbors. For each couple of opposing groups, “interaction” expresses the possibility for one group to dominate the other. The evaluation of interaction between two opposing groups is determined by a set of rules whose expression is very complicated and domain-dependent. Therefore, we shall limit the presentation of an interaction to a single example as simple as possible. Let us examine the black group in the middle right-hand side of the example of Fig. 3 (this group corresponds to Fig. 13) and the white group located just above (see Fig. 14).

These two groups are opposing, their number of eyes and the size of their outside are not sufficient. Therefore, we consider these groups as “weak”. It is important for the program to compare the number of liberties of each weak group. In our example, the white group has four liberties, and so has the black group. The player who plays first will either delete one of his opponent’s liberties, or give more liberties to his own group, or do both. The difference between the number of liberties of the two groups will be crucial in deciding which group dominates the other. When the value (relative to zero) of the interaction depends on who plays first, its value is designated ‘*’. Such is the case in our example. When one group dominates the other whoever plays first, the interaction is ‘>’ for the dominating color. For example, the interaction between the black group (corresponding to Fig. 15) which is located at the bottom right of the position of Fig. 3, and the big white group encircling it, is ‘>’ for White. In the next paragraph, we shall see that this fact will contribute to the “death” of this black group.

“Death”, “inversion”, aggregation

The next important point consists of detecting dead groups—this constitutes the main difference between the concrete EF and the conceptual EF. An error in the judgement of life and death of one group brings about tremendous consequences in calculating the value of the EF. Exact conditions for the death of a group should be given, or when this is impossible, very restrictive conditions should be given to the program. In such conditions,

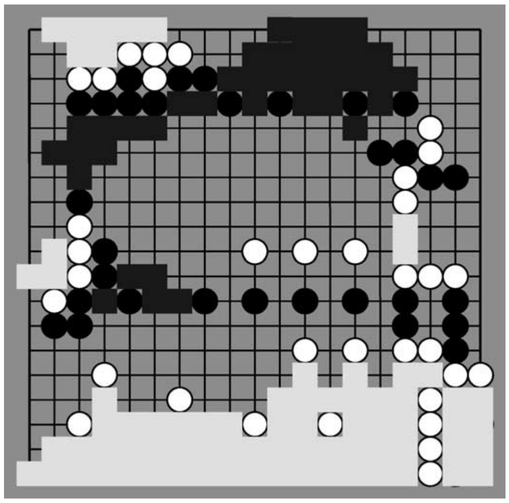
Fig. 16.

the program must not conclude that a group is dead when it is not. A “death” happens when a group answers *all* the following conditions: its number of eyes is not big enough; its outside is not big enough; it has no friendly group to connect with; it has no interaction with an opponent group whose value is different from ‘<’.

In our example, the group in Fig. 15 fulfils all these conditions. Therefore, it is “dead”. On the contrary, the two weak groups in Figs. 13 and 14 have an interaction whose value is ‘*’. They are not dead.

Once a group is dead, an “inversion” happens: the dead group changes its color and merges with its neighbors. Fig. 16 shows the conceptual description following the inversion, consequent upon the death of the black group in the bottom right corner.

Here, the black group has disappeared, becoming part of a big new white group that is located in the bottom edge. After an inversion, the program performs an aggregation to cluster the dead group with its opponent groups into a bigger group.

Summary

Having identified the basic tools, we are now able to describe the general form of a Go EF:

> While dead groups are still being detected, perform the inversion and aggregation processes.
> Return the sum of the value of each intersection of the board (+1 for Black, and −1 for White).

At the end of a loop, if a death is detected, the program inverts and aggregates the dead group to give birth to a new group. If no death is detected, the loop ends and the program computes the addition of the value of each intersection of the board. (+1 if the intersection has a black group on it, -1 if the intersection has a white group on it, and 0 in the other cases.)

Each loop gives rise to one, or zero, dead groups. Successive loops are necessary to detect several dead groups. The evaluation process may alter the life and death status of groups previously examined in response to new information determined in later cycles of the evaluation process.

“Life and death” of groups, “tsumego”...

We have just seen the building process of an EF. However, human players also use the terms “life” and “death”. Furthermore, they call the question of determining whether a group is dead or alive, a “tsumego” problem. It is now time to link these human terms with the previous discussion of concepts of our EF.

Let us first provide a few precise definitions. On the one hand, a *general* tsumego problem arises when a group is weak by our standards. (This means that the number of eyes, and the “outside” of the group, are not sufficient, and that the group has no interaction whose value is not equal to ‘<’.) Such problems consist of finding a solution with the following varying parameters: number of eyes of the inside, outside, and interactions. On the other hand, *basic* tsumegos are problems where the outside and the interactions are attached to insufficient values, hence the basic tsumego problem consists of finding a solution to make only one parameter vary: the inside of the group.

Consequently, it can be easily understood that there is a considerable gap between the two categories of tsumego. General tsumego problems are far more complex than the basic ones. Nowadays, GoTools solves basic tsumego problems at dan level very smartly, and quickly. (Here, basic does not necessarily mean simple: a lot of basic tsumego problems can be situated at a professional level.) Chen and Chen [39] give a set of heuristic rules to count the number of eyes of the inside of a group. However, Go playing programs are not efficient enough to solve general tsumego problems, and unfortunately these problems are the ones that are frequently met in actual games. The inability of Go programs to solve these general tsumego problems constitutes the major obstacle to Computer Go.

4.4. Conclusion

In this section, we have dwelt on the simple case of the *concrete* EF. This EF cannot be used within tree search because of the very small subset of positions in which it gives correct answers: the explicit-control endgame positions subset. Then we presented the *conceptual* EF. We highlighted the main features of an EF in Go: “group”, “inside”, “eye”, “outside”, “interaction”, “life” and “death”. In spite of our effort to describe this function as simply as possible, it remains quite complex. The first reason is to be found in the concept of “group” which is crucial to Go and has many variants. The Computer Go community still holds different views on its definition. The second reason lies in the difficulty in defining intuitive, and visual, concepts such as “inside” and “outside”. Another explanation is the very marked importance of interaction between groups. It is very difficult to determine the state of a group without considering its surroundings. Lastly, as regards tree search, a unique feature of an EF in Go is that it uses local tree searches. This aspect is new when considering the other two-player, zero-sum, and complete information games where tree search simply uses the EF. The setting up of an efficient, correct, and complete, EF is the major difficulty inherent in Computer Go.

5. Move generation

5.1. Introduction

Section 2 showed that a full global search was not possible in Go, and Section 4 pointed out the complexity of the EF. However, the aim of a game playing program is neither to search within trees nor to evaluate boards—these two activities are only means. The aim is to generate moves, and to select only one. Therefore, after leaving tree search and EF aside, the first trend in Computer Go was to generate moves by giving them a "priority" or "urgency", and then to select the move with the highest priority. The first Go programs looked like expert systems which had neither evaluation nor tree search. That was the right approach to try at that time, within such a complex domain [81,105]. Section 5.2 shows that Move Generation (MG) still occupies a special place in current Go programs. In Section 5.3, we show the relevance of Goal Generation (GG). Then we present some examples of MG associated to specific goals in Section 5.4, and we illustrate global MG, with the help of another example, in Section 5.5.

5.2. The specific position of MG in current Go programs

To help MG in the first Go programs, two other components were necessary: an EF to stop playing at the end of the game correctly; and a tree search module to check the tactical status of some local sub-positions, or to find the biggest global move. Therefore, the crucial question is to know the position of MG in connection with the EF and tree search. The EF of a position is complex, and uses a lot of knowledge to describe the abstract objects which are on the board. It is appropriate to try extending this knowledge, beyond descriptions of objects to the description of actions, or moves, performed on the objects. In such a case, the evaluation helps MG, and applies to both positions and moves. This approach is used in contemporary Go programs—for example in Go Intellect [37], and also in Shogi programs [60]. In this case, MG of Go programs is still very important.

5.3. Goal Generation

Moreover, when the programmer makes more extensive use of knowledge to describe moves, he naturally enters the domain of Goal Generation. Instead of generating moves, the program first generates the goals which may prove useful in winning the game. Once the goal has been selected, a specific evaluation function adapted to this goal is chosen, and a specific move generator is associated with this goal. A goal-oriented approach has the advantage of reducing the complexity of the problem to be solved, but the drawback may be the lack of global balance when more than one goal is relevant to winning the game. Current programs such as Go Intellect [37,38], Goliath [14], and Many Faces of Go use such a goal-oriented approach. Other explicit goal-oriented studies in Go are [55,65,66, 108]. Decisive goals to be generated in a position are: “occupy big empty points”, “kill”, “live”, “attack”, “defend” “groups” [38], “expand”, “reduce”, and “invade” “territories”. Furthermore, the sub-goals may be “make eyes”, “destroy eyes”, “connect”, “cut groups”, “capture”, “save strings”, and so on. [52] proposes a hierarchy of goals, which can be used to build a Go program. For each goal, or sub-goal, the program generates special moves, and performs goal-oriented tree search.

5.4. Goal-oriented MG

This subsection provides an example of moves generated according to the context of goal-oriented MG. Let us consider the position in Fig. 3 and to its evaluation shown by Fig. 16. The evaluation identifies three “weak” groups: the middle left white group, the top right black group, and the top right white group. It also identifies territories to be expanded or reduced: the bottom white territory, the middle left black territory, and the top black territory. Each “weak” group, and each territory, generates goals to be pursued: attack or defend a “weak” group; expand or reduce a territory.

Expanding/reducing territories

MG is quite simple in such a case. A program has a pattern database suggesting moves according to this goal. Let us assume the database has the patterns shown in Figs. 17–20. Let ‘X’ be the good move, and ‘Y’ the bad one. Then the MG generates the moves of Fig. 21 for expanding or reducing the territories.

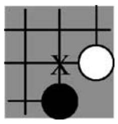

Fig. 17.

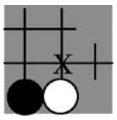

Fig. 18.

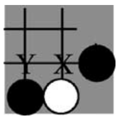

Fig. 19.

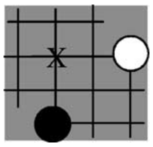

Fig. 20.

When such generated moves are used, tree search allows one to select the best expanding/reducing move. In the case of territory, when assuming that the depth-one positions are still quiet, tree search selects the “best” expanding/reducing move. However, in actual cases, depth-one positions are not necessarily quiet. Therefore, evaluations of territory are not significant, and a quiescence search must be performed [40]. Instead of performing tree search to select one move, another possibility is to refine the patterns for expanding/reducing territory by specializing them as much as possible, and to associate each of them with a value to be used by the move selector. In such cases, the program selects the generated move with the best value. The approach of knowledge refinement is performed manually in most Go programs, when their authors are strong Go players who like introspection. However, this difficult refinement task can be achieved by the computer itself. This aspect will be discussed in the “Automatic Knowledge Generation” section.

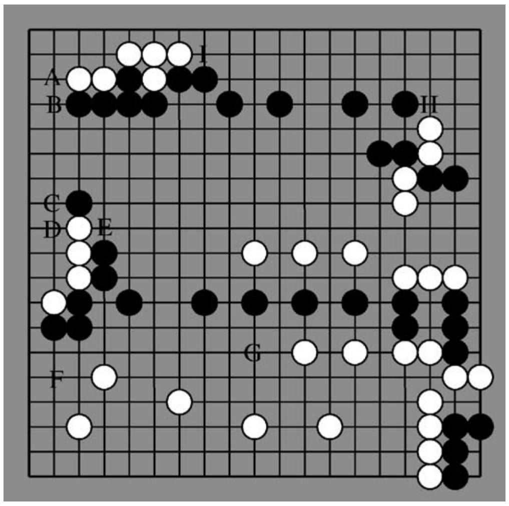

Fig. 21.

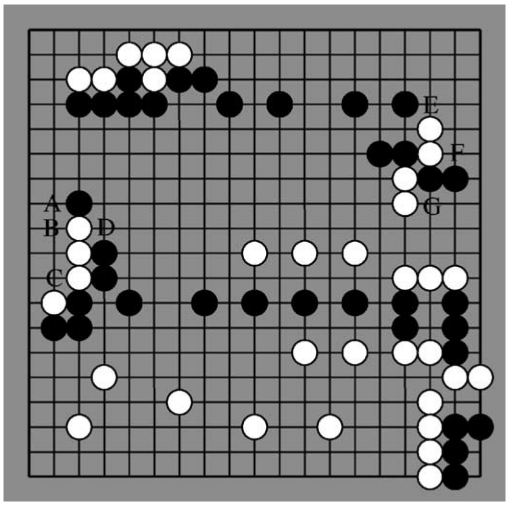

Fig. 22.

Attacking/defending groups

The other relevant goal-oriented MG is the attack and defense of groups. A program contains a pattern database suggesting moves relevant to this goal. Let the database contain the patterns of Figs. 17–20, plus a rule advising to capture/defend the strings which belong to the group, and whose Conway's state is '*' (see Combinatorial Game Theory section). The generated moves are those of Fig. 22.

Move ‘C’ is generated by the rule advising to save the white string whose state is ‘*’, and belonging to the middle left white group. Other moves are generated by the pattern database. Beyond this simple example, Go programs contain many more complex patterns. For instance, the move generation is not symmetrical, and the pattern must also specify which stones of the pattern belong to the “weak” group. A very good description of attack and defense in Go Intellect can be found in [38]. In this section we have presented only simple rules. In fact, rules are more complex in current Go programs: rules may advise a set of moves for each side; they may contain information about the number of liberties of the string; or they can make use of logical expressions.

5.5. Global MG

This subsection illustrates an example of moves generated at the global level. We keep the example of Fig. 16. We assume that the global level considers only the two kinds of goals described in the previous subsection: “expanding/reducing territories” and “attacking/defending groups”.

First, in a Go program that, for speed performance, avoids tree search, the previous MG may be used simply to provide a priority to each goal. For each goal, this priority may be proportional both to the size (or size variation, in the case of territory) of the object associated with the goal, and to a factor specific to the goal class. Therefore, a very rough and simple method consists in selecting the move with the highest priority, which is associated with the goal with the highest priority. This move decision process suffers from a lack of coherence because the program does not verify whether the selected move actually achieves its goal or not.

In our example, we assume that the priority of group is significantly higher than the priority of territory. In such a case, the program decides to attack/defend the middle left white group. Then, assuming that the rule which advises saving the string whose state is ‘*’, has a higher priority than the patterns’ priority, the program chooses to play move ‘C’.

This method may be sharper with the help of a goal-oriented Tree Search so as to eliminate the ineffective moves. For example, move ‘C’ will not work for White and the TS will select ‘A’ or ‘D’. TS may eliminate all the moves of the goal-oriented MG. In this case, the goal is unreachable. For instance, if the program is strong enough, it may want to defend its white group and check that the moves ‘A’, ‘C’, ‘D’ do not reach the goal. Therefore, the program will switch to the top right fight, and try moves ‘E’, ‘F’, ‘G’. Another TS will conclude ‘E’ and ‘G’ are good moves to select at the global level.

5.6. Conclusion

The previous description of MG is similar to the plausible move heuristic, which is well-known in computer games, together with selective search. It was used in the early days of Computer Chess [58]. It is currently used in Computer Shogi [60,69,144], and it will still be used for a long time in Computer Go. An explicit contribution to MG in Go is the reference paper [37], which describes the move decision process of Go Intellect, and fits very well with the above description. Like other programs, Go Intellect contains about 20 move generators. Most of them are goal-oriented and heuristic. A “move coordinator” combines all the values by using linear combinations dynamically determined by the status of the game. A “move checker” is provided to avoid obvious errors in the choice of candidate moves. If a candidate move has a significantly higher value than the other ones, and if it passes the move checker, then it is selected without look-ahead. But, when they are several highly recommended candidate moves, Go Intellect uses global look-ahead on these candidate moves, and selects the best one. Once more, the example of Go Intellect shows how important static MG is. In some cases, MG may be sufficient to select the move without global tree search, or, in other cases, sufficient to order the moves for tree search.

6. Tree search

6.1. Introduction

This section describes the link between Computer Go and Tree Search (TS). In classical games like Chess, the goal of TS is to find a move among many moves, using an EF as a black box. Numerous publications deal with this problem, and study the well-known minimax and Alpha-Beta algorithms, as well as their variants [10,26,73,76,124]. A recent trend has favored new approaches, such as the conspiracy numbers approach [83,116], and proof-number search [2]. Computer Go follows this trend. But in Go, TS is completely different because of the importance of locality. Until now, few methods have emerged. The fundamental issue in Computer Go TS is selectivity. Section 6.2 deals with the different TS methods used in current programs, and Section 6.3 focuses on the new parameters of TS in Go.

6.2. Current use of tree search

To begin with, let us give the relevant information related to TS in the current Go programs (Handtalk, Many Faces of Go, Go4++, Go Intellect, and GoAhead).

Handtalk generates very few (about 4) moves at the global level, and performs only little tree search. Handtalk’s author believes that there are many more important things than TS [42].

The author of Many Faces of Go underlines that his program has multiple TS engines [52]: capturing a string, connecting two strings, making an eye, making a group dead or alive, performing a global quiescence search. An important feature is that TS gives a degree of reliability concerning the result of the search.

Michael Reiss says that Go4++ tries about 40 candidate moves, evaluates the score for each move, and plays the move with the highest value [103].

Ken Chen [40] has recently given some heuristics that enable his program, Go Intellect, to perform an efficient global selective search. His first heuristic consists in cutting off the search at quiescence, returning the value of the evaluation function for stable positions in the global game tree. The second one is to cut off search when a target value is reached. The last heuristic is to associate an urgency value with a move, and to take this value into account when evaluating the position. The author concludes that a balanced combination of global selective search, and decomposition search [94] may well be the best approach to Computer Go.

Peter Woitke [138] says that his program does not calculate variations except for string captures, life and death, and endgame positions.

6.3. Main features

In this section, we depict the main features of TS in Go, concentrating on the features that make TS in Go different from TS in other games. To us, the most important element is locality, since all moves in Go are played at a precise location on the board—in other words, an intersection. We shall later examine other features, such as the goal of TS, the definition of terminal positions, the initiative, abstraction, the dependency between local situations, and finally uncertainty. All these features are linked to TS.

Locality

Given the size of a board, searching by selecting the moves localized on a part of the board only is called a *local* TS. It is very similar to what human players do when they examine different sequences of moves, at different places on the board, in a separate, and independent, way. Local TS is a kind of selective search. It is an approximation of the usual TS, which is called *global* TS. In practice, owing to current computer power, global TS, unlike local TS, cannot be completed, and local TS becomes mandatory. Even with greater computer power, a brute-force global search would not be an efficient approach. There are too many clearly sub-optimal, or even bad, moves in a Go position. A detailed analysis of a position can eliminate many useless moves. A goal-oriented global search, or the splitting of the search into different local searches, are two options which may be worth considering. Splitting the game into sub-games was proved to be useful in the endgame, when used with decomposition search [94]. Applying this technique to the middle game is a promising, but difficult, research area.

However, several obstacles arise when using local TS. First, the locality criterion has to be defined. The problem of defining the ‘distance’, that discriminates between a ‘near’ move and a ‘far’ move, is open. Each Go program employs its own distance. It depends on the data structures chosen in the program. Apart from problem in defining distance, the second obstacle is to know when to stop the local TS. Unfortunately, a local TS often goes beyond its departure point. This happens when all the local moves have been played, and when the local situation is still uncertain and, therefore, impossible to evaluate. In practice, the search has to be stopped. Consequently, the result of a local TS contains some uncertainty. The last problem in using local TS is in the reconstruction of the global result, given the results of local TS. To address this issue, the program simplifies the problem by assuming that they are relatively independent of each other. Although this hypothesis is wrong in theory, it is essential to make this assumption if we are to get results. Assuming independence between local situations, the game of Go can be considered as a sum of independent sub-games. It makes global TS look like a set of local TS. Conway’s combinatorial game theory gives some clues for dealing with local results. One of the most popular strategies among programs is based on thermographs (see the section on combinatorial game theory). It consists of playing in the ‘hottest’ local game. This particularity of Go is also used in decomposition search [94] that drastically reduces the size of the global search.

In short, locality allows TS to obtain a result otherwise impossible with a unique global TS. Current difficulties linked to the locality criterion relate to the definition of a good distance, the criterion for stopping a search, the specification of a measure of uncertainty in the result of search, and the evaluation of the global result once the results of local search are obtained. To circumvent these obstacles, formulating the hypothesis about the independence between local games makes it possible to use Conway's combinatorial game theory.

Use, and goal, of tree search

In classical game programs, TS calls EF at the terminal nodes of the tree. In Go, EF is complex and its value relies on the state of the board's abstract elements. Moreover, knowledge of the state of these elements relies on the results of local and abstract TS. So, EF uses the results yielded by some TS. The usual order—whereby TS calls EF, as in Chess—is reversed in Go. TS, which was the user in Chess, also becomes the supplier in Go. Unlike in Chess, the precision of the result of EF in Go makes it necessary to perform tactical TS, so as to find the state of the elements of the evaluation (connections, eyes, life and death of groups, etc.). Here again, the goal of TS in Go is not, as in Chess, to select a move, but to prove the value of a situation.

Problems specific to Minimax search in Go

Müller [96] identifies difficulties specific to Minimax search in Go: recognizing terminal positions, pass moves, and local position repetitions, or ko. In Go, a position is terminal if no more points are contested, and all points can be classified as black, white, or neutral. Such classification is hard because of the complexity of the evaluation. A terminal position, and a non-terminal position, may be very similar. In Go, passing is legal, and pass moves must be generated during search. In positions where there is no good move, players are allowed to pass instead of damaging their position. Adding pass moves to a tree search, increases the size of the search space. Global position repetitions are forbidden by the rules, but a local tree search must cope with local position repetitions, or ko. A local repetition is produced by the possibility of playing outside the local position at any time during the search. Ignoring the possibility of ko gives misleading results.

Initiative

Given the multiplicity of local situations, either one player or the other can play first in a given local situation. To cope with this property, current programs are required to perform at least two TS on a given local situation, the initiative being taken each time by a different player. We can indicate that a local situation is described by two TS, using a notation such as {B | W} where B (respectively W) is the situation found if Black (respectively White) plays first. When the opponent does not answer to a move in a local situation, but rather plays in a different local situation, the player can play a second move in a row in the first local situation. There is a possibility that a player may play multiple moves in a row on the same local situation. So, theoretically, multiple TS should be performed. In practice, current Go programs do not consider all these possibilities that increase computation time. The combinatorial game theory that takes these problems into account is used as a reference. Recently some progress has been made in the safe cutting of combinatorial game trees [71]. This improvement is comparable to the improvement of Alpha-Beta over Minimax, and is a breakthrough in the computation of the mean, and the temperature of combinatorial games in which there is a dominant move at each stage.

Abstraction and local tree search with a goal

We have shown that the locality criterion reduces the global evaluation to a sum of local evaluations, thus greatly reducing the complexity of the corresponding TS. But, to avoid combinatorial explosion, the Computer Go programmer can also use abstraction in addition to locality. Rather than locally evaluating a position, the program can first identify a typed local goal—for example the connection of two strings, or the life of a group—and then perform a TS with this goal. It reduces the complexity of the EF, as it only takes three values (‘0’ if the goal cannot be achieved, ‘1’ if the goal is achieved, ‘?’ in other cases). The EF is therefore rapidly computed. For a lot of typed local goals—such as connection, eyes, and life and death of completely encircled groups—the corresponding TS is achieved in a short time, and the program has certainties that are better than the non-termination of the local evaluation TS. Moreover, the three-value EF enables the program to use the Proof-Number Search algorithm, which generally obtains better results than Alpha-Beta. However, it is impossible to obtain these abstract results without the inherent counterpart to abstraction: simplification, and consequently inaccuracy.

Fig. 23 gives an example of a proof tree for the goal connect. Black is the friendly color, and the goal is to connect the two black stones. The first move works (the leftmost arrow) and, as it is an OR node, the other branches are cut. The moves at the OR nodes are given by rules terminating on moves that can achieve the goal if two moves are played in a row by the friendly color. This heuristic is used because these moves lead to positions containing threats of winning by the friendly player, and therefore forced moves for the other player.

The rules of Fig. 24 indicate the moves to try at the OR levels of the proof trees. This is visually explained in the first diagram of Fig. 24, where a tree is represented with the color of the moves associated to the branches. The only available information is that Black can reach the goal if it plays two moves in a row (state W of the first diagram). Otherwise,

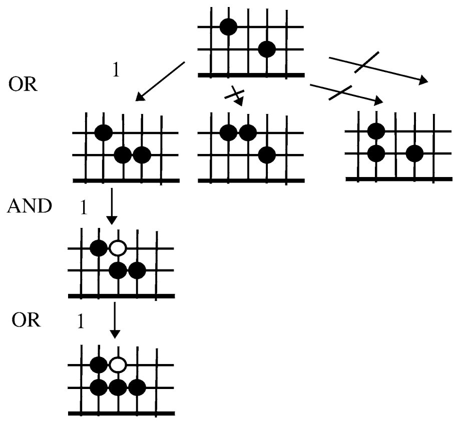

Fig. 23.

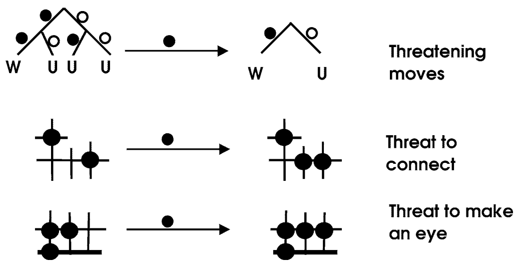

Fig. 24.

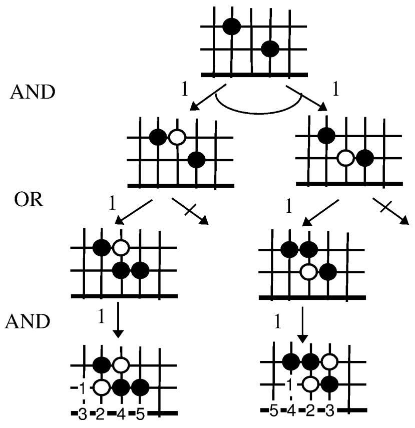

Fig. 25.

the three other combinations lead to unknown situations (state U). When Black plays the moves advised by the rule, it switches to a threatening situation, represented by the tree on the right of the first diagram. Black can now win the goal if it plays one move, and therefore White now has to play to prevent Black from doing so.

The first rule of Fig. 24 is used to find the upper left move of the proof tree of Fig. 23. Black plays on a liberty of the right black string; this liberty is neighboring a liberty of the left Black string. So, if Black plays another move on the liberty of the left Black string, the left and the right Black strings will be connected. The second rule of Fig. 24 advises a move to threaten to make an eye, if Black plays another move on the empty lower right intersection, it will then make an eye.

Fig. 25 gives the second proof tree developed by the Go program when White plays first, and searches with the goal “Disconnect the two black stones”. In order to save space, we have numbered the sequences of moves at the leaves of the tree (odd numbered moves are black moves, and even numbered moves are white, single forced, moves). The two forced white moves—at the root of the proof tree—are refuted by Black. As a result of the two proof trees, Black can connect its two stones even if White plays first: the two black stones are virtually connected.

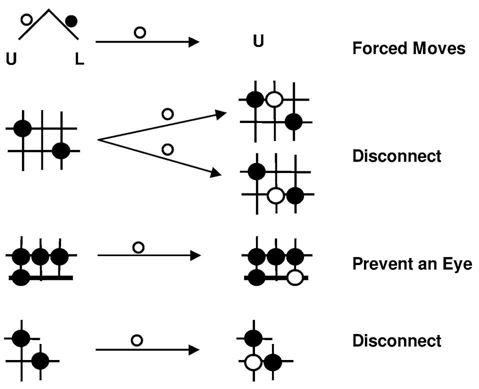

Fig. 26.

The last rule of Fig. 26 is used to find the forced move at the lowest AND node of Fig. 25. The first rule of Fig. 26 is used to find the two forced moves at the root of the proof tree of Fig. 25: these two moves are the only ones to be considered out of all the possible moves. The second rule is used to find the only possible forced move to prevent an eye. The visual definition of a forced move is given in the first diagram: a White move leads to an unknown state (state U), whereas a Black move leads to a lost state for White (a lost state L for White is a winning state W for Black: White loses if Black is connected in our example).

Dependency/independence of local situations

Some local situations are highly dependent on one another—a move can influence two local situations simultaneously. A local TS cannot be reduced to finding a good move, but also has to take into account all the answers to the first move. Otherwise, the global move choice will rely on incomplete information. At the top of the local TS, all the moves have to be played. The dependency of neighboring local situations also implies the existence of moves that do not work in any situation, but that are threats for each of them. These moves enable the player to change one of the situations, thereby they are proved to be efficient. A good program should be able to find them.

Uncertainty

Given the complexity of local situations, numerous local TS terminate without finding the good move, or proving the goal. The results of the local TS remain uncertain. The global level of the program has to handle the uncertainty of the non-terminated TS. There are many ways to represent, and use, uncertainty in Go programs. For example, Gogol [30] represents the uncertainty about the result of a game using a taxonomy of games. The most general game is ‘U’—its value is unknown whoever plays first. A ‘WU’ game is a game where one player can win the game if he plays first, and the result is unknown if the other one plays first. A ‘WU’ game is a sub-class of the U games. ‘UL’ games can be defined similarly. Other sub-classes of games can be defined such as ‘WUUU’ or ‘WL’. This representation of uncertainty in games is useful to describe the elementary sub-games of the game of Go such as the game of capturing a string, the game of making an eye, the game of connecting two strings, and so on. Indigo [18] also represents uncertainty in its evaluation function using the local uncertainties identified in the local fights. Many Faces of Go uses a confidence degree for the result of an evaluation during its TS.

6.4. Which algorithms?

To perform searches, some already existing techniques can be used, each offering its advantages and drawbacks, depending on the context of the TS. For connection, or Tsume-Go, problems, a three-value evaluation function can be used, as well as Proof-Number Search (PN-Search) [2]. The drawback of PN-Search is the time used to copy positions, and the memory required to store them. For all the problems with uncertainties, quiescence search algorithms [6] are recommended. Other TS can use Alpha-Beta. Depth-First search, which offers the advantage of looking at positions in such an order that the results of the evaluation of the former position can be reused in order to evaluate the current position rapidly. The incrementality principle (see the optimization section) can be used. At depth zero, a local TS has to consider all the moves, so that the global TS can detect those moves that may help achieve two goals simultaneously. The game of Go, as a sum of games, shows that programs need both to perform an efficient TS, and to know how to use its results. Combinatorial game theory solves some of the problems that arise when combining the results of multiple TS (see the Combinatorial Game Theory section).

Many heuristics can be used when developing search trees. Thomas Wolf [142] gives some heuristic search techniques used in GoTools. For example, at each node of the search tree, the relevant moves are tried, and their usefulness is then evaluated. Other heuristics consist in giving low priorities to moves that are forbidden for the opponent, in trying as the second move the move of the opponent that has refuted the first move. Some heuristics to improve global search are given by Ken Chen [40].

6.5. Conclusion

In this section, we have shown the specificity of tree search in Go. We have identified many important features of tree search: locality, strong links with the evaluation function, initiative, abstraction, dependency, and uncertainty. Computer Go enriches the classical paradigm of tree search with new viewpoints. Moreover, the large branching factor of Go calls for clever selective search techniques, either in tactical search (GoTools), or in global search (Go Intellect). An increase in computer power will not allow an efficient brute force global search. The special properties of the game of Go make selective [40], and decomposition search [94], much more efficient than brute force global search.

7. Optimization

7.1. Introduction

As in other games, the speed of a program is of paramount importance in the game of Go. Even if current problems are mainly linked to a good modeling of the Go player, it is still very important to do a fast tactical search, and to use knowledge efficiently, to have a competitive Go program. Computer Go optimization techniques range from very low level considerations, like the raw representation of the board, to high level considerations, like the use of abstract strategic representations, so as to choose which search trees to develop. Some mid-level optimizations are also used—for example calculations of the search dependencies, can indicate whether, or not, to recalculate the search tree, if the move played does not modify its result. Optimizations are time-consuming for the programmer. Section 7.2 presents some possible optimization for pattern matching. Section 7.3 gives some hints on how to stop tactical search early. Section 7.4 deals with the ordering of the different loops, and tests that occurs in both manually, and automatically, generated Go programs. Section 7.5 is about the choice of search algorithms. Section 7.6 explains how to optimize the operators of mathematical morphology. Section 7.7 sheds light on the optimization of string capture. Section 7.8 details the use of incrementality, and, finally, Section 7.9 says a word about high level optimizations.

7.2. Pattern matching

The representation of the patterns, and the board to match the patterns on, is a problem that every Go programmer has to face. The patterns represent small parts of the board. One essential property of a string of stones lies in its number of liberties, so that patterns are associated with conditions on the liberties of strings.

Fig. 27 gives examples of some patterns associated with a set of conditions. These patterns represent an eye. Boon described how he optimized the pattern matcher of his program, Goliath [13]. He represented 5×5 patterns, and 5×5 parts of the board, using 32-bit strings, which enabled the program to perform very fast logical operations on integers to match patterns with parts of the board. Other algorithms have been used to optimize pattern matching. For example, the Explorer program uses Patricia trees to match patterns [90]. Gogol [34] represents the patterns in one or two 32-bit integer, and performs a binary search on its sorted patterns list, so as to match them rapidly.

7.3. Stopping search early

The patterns and rules are used, both to select possibly interesting moves, and to stop search early. The following trees are developed to prove that the two black stones are connected. The left tree is developed by using only simple rules to find interesting moves, whereas the right tree is developed by using more clever rules and patterns.

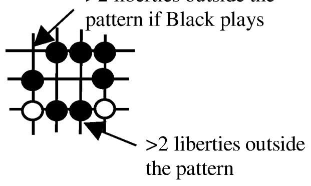
>2 liberties outside the pattern if Black plays
>2 liberties outside the pattern

Fig. 27.

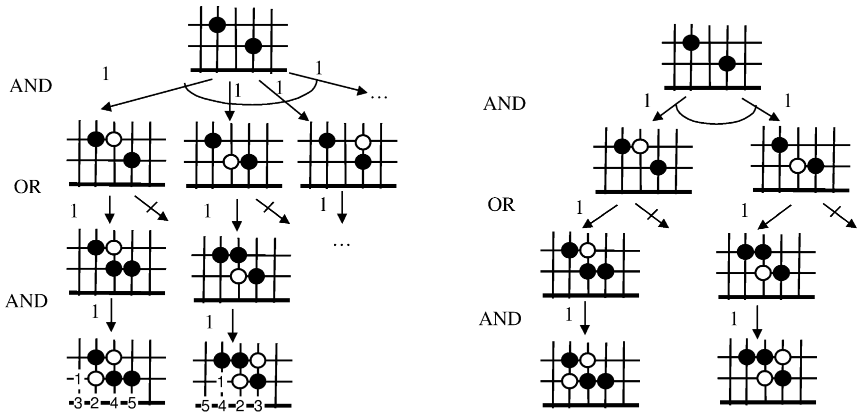

Fig. 28.

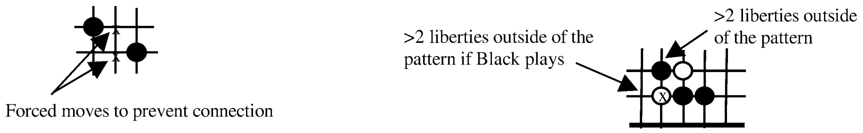
Forced moves to prevent connection
>2 liberties outside of the pattern if Black plays
>2 liberties outside of the pattern
X

Fig. 29.

The left tree of Fig. 28 has 31 nodes whereas the right tree has only 7 nodes. In the game of Go, the overhead of matching simple patterns is largely compensated for by the time gained thanks to search savings. Fig. 29 gives two patterns to cut the right tree. The first pattern shows that there are only two moves to try at the root of the tree, and the second one shows that Black can connect the two stones if he plays first.

7.4. Ordering the conditions of rules

Another approach to speed up the use of rule-based knowledge is program transformation. A metaprogram can make a program faster by automatically ordering the conditions of the rules, or by partially evaluating the rules [28,33]. While reordering conditions is very important for the performance of rules generated by a metaprogram, it is also important when hand-writing rules for a Go program. The following two rules are simple examples that show how important a good order of conditions is.

```
connect(S1,S2,I1):- color_intersection(I1,empty),
                    color_string(S1,C), color_string(S2,C),
                    liberty(I1,S1), liberty(I1,S2).

connect(S1,S2,I1):- color_string(S1,C), color_string(S2,C),
                    liberty(I1,S1), liberty(I1,S2),
                    color_intersection(I1,empty).
```

The two rules give the same results but do not have the same efficiency when `I1` is unknown, because there are more empty intersections than liberties of the string `S1`. When based only on the number of free variables in a condition, reordering does not always work well. Conditions, and therefore variables, are ordered once only, and not dynamically at each match, because it saves more time. Reordering the conditions of a given rule optimally is an NP-complete problem. To reorder conditions in our generated rules, a simple and efficient algorithm can be used. It is based on the estimated number of subsequent nodes, which the firing of a condition will create in the semi-unification tree. Here is a metarule used to reorder conditions of generated rules:

```
branching( ListAtoms, ListBindVariables,
           neighbor(X, Y), 3.76) :-
             member(neighbor(X, Y), ListAtoms),
             member_term(X, ListBindVariables),
             non_member_term(Y, ListBindVariables).
```

A metarule evaluates the branching factor of a condition based on the estimated mean number of facts which match the condition in working memory. Metarules are fired each time the system has to give a branching estimation for all the conditions left to be ordered. When reordering a rule containing N conditions, the metarules will be fired N times—the first time to choose the first condition, and at T time to choose the Tth condition. In the first reordering metarule above, the variable X is already present in some of the conditions preceding the condition to be chosen. The variable Y is not present in the preceding conditions. The condition '`neighbor(X,Y)`' is therefore estimated to have a branching factor of 3.76 which is the mean number of bindings of Y (that is, the mean number of neighboring intersections of another intersection on a 19×19 grid—this number can vary from 2 to 4).

The branching factors of all the conditions to be reordered are then compared, and the condition with the lowest branching factor is chosen. The algorithm is very efficient because it orders rules better than programmers do and because it runs fast, even for rules containing numerous conditions.

7.5. Alpha-Beta or PN-Search: a difficult choice

Another practical problem lies in the use of an appropriate search algorithm for the problem at hand. For example, during a tactical search, Alpha-Beta enables the program to reuse the abstract information of the parent node (such as the numbers of liberties, or the list of adjacent strings), in order incrementally to calculate information about the current node, without occupying too much memory. On the contrary, a Best-First algorithm, like Proof-Number Search, uses a lot of memory if incremental information is kept. Moreover, it takes time to copy the information for each node. However, the trees developed with PN-Search are often smaller than the trees developed with Alpha-Beta, because the sub-games of the game of Go have a variable number of moves at each node. The choice between these two algorithms depends on the data structures used, and on the sub-games at hand. For example, Many Faces of Go uses Alpha-Beta incrementally, and updates the liberties of the strings at each move. On the contrary, Gogol uses PN-Search, copies the raw board, and recalculates only the useful sets of liberties at each node.

7.6. Dilation and erosion

Dilation and erosion are crucial operations in a Go program. They are very often used to calculate liberties, influence, territories, and, more generally, the neighborhood of an object. Howard Landman uses an optimization to calculate the dilatation and erosion operations of mathematical morphology. These operations are useful in calculating both the influence of stones, and their associated territories. To assess dilatations and erosions rapidly, the board can be represented as a bit string—an erosion corresponds to some standard bit-string operations: SHIFT and OR.

7.7. String capture

An optimization used by Goliath [14] is to put many moves simultaneously in a tactical search. When a given program calculates a ladder, it has to play the same two moves repeatedly. Goliath plays all the moves at the same time.

For example in the left position of Fig. 30, to calculate whether the white stone—marked with a triangle—in the lower left can be captured, the program tries, first, to play the liberty of the stone that has the greatest number of second order liberties, and secondly, to play on the other liberty to avoid capture. These same two moves are played repeatedly across the board until the position in the second diagram of Fig. 30 is reached. Instead of analyzing the position at each ply and playing the same moves every two plies, the program can be optimized to go directly to the position of the second diagram of Fig. 30. This optimization enables the program to save time in 80 percent of the ladders calculated by Goliath. Instead of analyzing the position for each move of the ladder, Goliath analyzes the position only once.

Another optimization, specific to string capture, consists of ordering moves during a tactical search. For example, when trying to save a string by playing on its liberties, it is

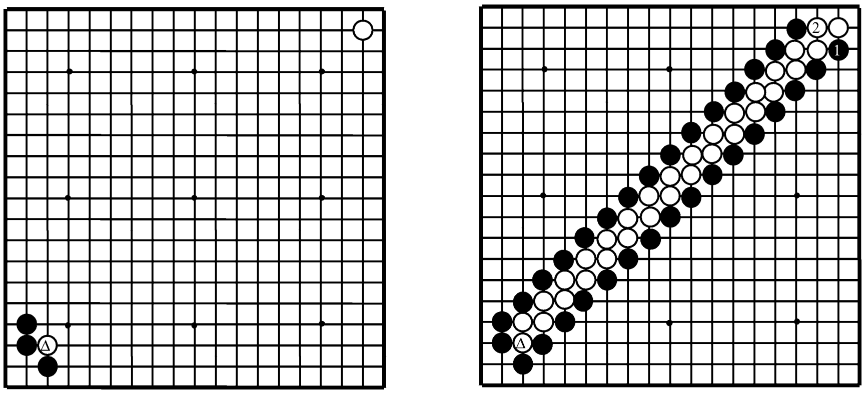

Fig. 30.

usually better to try first the moves on the intersections likely to give the string as many liberties as possible.

7.8. Incrementality

Bouzy [20] introduces the concept of the "incrementality" of a tactical search. Incrementality is an example of a mid-level optimization, which aims at associating a tactical search with a "track" that represents parts of the board involved in the result of the search. If a move is played in the track of an object, then the object must be deleted. If a move is played outside the track of an object, then the object remains unmodified. Whenever a move is played, the program knows which search remains unchanged, and thus saves time. In practice, incrementality of tactical search enables a program to play twice as fast on a 19×19 board. With the track mechanism, the programming task is nicely simplified, but the incrementality problem has become a definition problem. If the programmer defines a track bigger than the optimal one, then the behavior of the program remains correct but is slower than it could be. If the programmer defines a smaller track than the optimal one, then the program forgets to destroy some objects and the program quickly adopts an erratic behavior. The problem of correctly specifying the tracks of objects is difficult to solve.

7.9. High-level optimizations

An example of a high level optimization is the use of an abstract strategic representation of the board that allows the program to focus on some important tactical computations, and to neglect useless ones that might be time-consuming [31,66,107,108].

8. Combinatorial game theory

8.1. Introduction

This section deals with those basic features of combinatorial game theory which have so far been useful to Computer Go. The reader may refer to [43] or [64] to have a mathematical overview, to [44] for a recreational presentation, and to [8] or [89] for the practical applications of the theory to Computer Go. These different viewpoints will allow the novice to understand the surprising differences between this theory and classical games theory [136]. Combinatorial game theory is also called Conway's game theory.

8.2. The results of combinatorial game theory when applied to Computer Go

The results of applying the theory to the problem of playing the whole game are few, and limited. However, with specific sub-problems, like late endgame positions, this theory is highly successful. This section deals with these excellent results, and then points out the difficulties in extending them to other well-identified concepts of Go—"eyes", "ko" or "fights"—and to complete games.

Late endgame positions or "yose"

During the endgame phase, it is partly possible to model a position as the sum of games. We list four key features of such a model. The first feature corresponds to the identification of "groups" and "territories". As a game nears its end, the identification of "groups" and "territories" becomes easier, and thus, the transformation from the position into a list of sub-games becomes possible. In addition, when moves are played, the identification of "groups" and "territories" becomes more stable. Stability is our second feature. It is fundamental because, if not verified, the local tree searches in each sub-game would become pointless. Furthermore, as a game nears its end, the moves played in one sub-game have less influence on other sub-games. Therefore, near the end of the game, the sub-games become independent. Independence of sub-games constitutes the third feature. Besides, as a game nears its end, local searches become shorter. Therefore local searches can be fully completed and the sub-games described. Completion of tree searches is the fourth feature.

Thereby, in positions in which criteria for the use of these four features are fulfilled, Berlekamp and Wolfe have obtained outstanding results. When an abstract description of the positions, in terms of groups and territories, is given to Berlekamp's model, it finds moves that have the same result, or even one point more, than professional players' moves [8,9]. This result had a tremendous impact in the Go community because everybody thought professional players were playing the endgame optimally. Berlekamp's model lucidly demonstrated the contrary. It would take too long to dwell on the mathematical details explaining why the result is so good. In outline, one can simply say that combinatorial game theory employs infinitesimal numbers (like *up*, *down* and *) which describe the sub-games. They transform themselves into one point when the number of sub-games is odd. The proof, and the test positions, can be found in [9].

This result is correct, of course, when the four sets of criteria are fulfilled, and it should be capable of being extended to the endgame with the help of more robust tools than Berlekamp's model. Martin Müller has been working on applying this theory to Go endgames for several years [89–92]. Müller's [94] describes Decomposition Search, which is an algorithm which identifies safe groups, and territories, in a position, finds the sub-games, computes their value, and selects the sub-game. In test positions, Decomposition Search performs much better than a classical alpha-beta. This paper is a very nice demonstration of the power of the combinatorial game theory applied to the endgame in Go.

Recently Kao [71] described a method for computing the mean, and the temperature, of combinatorial games, where each player can have only one option at each local non-terminal position. This method is based on the stable theorem, and on the algorithm MT-Search (Mean-Temperature Search). Although the method can be applied to games other than Go endgames, MT-Search works very well with endgame positions. The improvements in the computation of mean and temperature, due to this method, are comparable to the improvement of Alpha-Beta over Minimax.

"Eyes"

After the success in the endgame, mathematicians have investigated other sub-problems to be found in actual games. Eyes are very useful for life and death problems.

Landman's [79] is the reference paper that shows the link between Conway's theory and the eye concept of the game of Go.

"Ko"

Game theory is far more complex in loopy games. A loopy game is a game whose graph contains loops. In the game of Go, loops are forbidden by the rules, but when you model the whole game as a sum of sub-games, its sub-games may be loopy. For a given sub-game containing a "ko", the first player takes the ko; as the rules do not allow the second player to take the ko again, he plays elsewhere—in another sub-game. If the first player also plays elsewhere with the following move, then the global position has changed, and the second player is now allowed to take the ko in the given sub-game, whose sub-position is the same as two moves before. Therefore, the given sub-game must be considered as loopy under such modeling. Not only does Berlekamp's model encompass ko, but also studies about ko and combinatorial game theory classify the different contexts of ko [92]. Spight [122] extended the thermograph theory to process multiple-ko positions.

"Fights"

Müller [95] described a method for modeling "fights" in Go, and computing their game values. In order to test the validity of the approach, this method was integrated into his program, Explorer, and tested on specific fight positions. The results were very impressive: its system outperformed the current best programs such as Go4++ and Many Faces of Go. This illustrates the power of describing of Go local situations by game values.

The whole game

Unfortunately, the conditions for applying Conway's theory to the whole game are not fulfilled. In brief, even if you have a sub-games model of the global game, the sub-games are greatly dependent on one another. Each model of a global position into sub-games already contains an approximation. Nevertheless, Conway's theory is a source of inspiration to model the game of Go for the computer. This paragraph shows the work accomplished using this perspective.

Goliath [14]—three times world champion in 1989, 1990 and 1991—modeled local searches with switches. Explorer [89,90] is, by far, the program that uses the theory in the most efficient way. For example, Explorer contains tools to compute thermographs of games. In addition, Indigo [15] uses Conway's classification (positive, negative, fuzzy or zero). Unfortunately, local searches in some sub-games cannot be completed, and such sub-games are placed in the 'unknown' category. Furthermore, Gogol [30] does not strictly follow this classification. It defines a taxonomy of games. 'U' games correspond to games whose searches are unknown, or not completed. 'W' games correspond to won games, and 'L' games to lost games. Then, 'U' games can be classified into 'WU', 'WL', or 'UL' games. A 'WU' game is a game where the left outcome is 'W', and its right outcome is 'U', and so on. This classification cannot be justified from a theoretical viewpoint, nevertheless it is preferable to Conway's in practice, because it takes the unknown sub-games into account and gives results, even when a local search is not fully completed.

8.3. Conclusion

Until now, classical game theory [136] has been very well adapted to Chess, Checkers, and Othello, but has lacked power sufficient to model the game of Go. Combinatorial game theory partly makes up for this weak point by giving optimal strategies for playing in some specific endgame positions [9]. It also gives ideas for defining sub-optimal strategies for playing the whole game. Before this theory can be used for the whole game, extensive research has to be done. Future researches might focus on the splitting of the global game into dependent sub-games (rather than independent ones). In addition, experiments should be carried out which relate to searches that may not end. Needless to say, this theory remains to be put into practice, and adapted to all phases of the game.

9. Automatic generation of knowledge

Writing a Go program is difficult because the most efficient programs use a lot of knowledge. Consequently, methods that generate Go knowledge automatically offer great advantages. Research into automatic generation of knowledge in mind games has mainly concentrated on the generation of an evaluation function. However, the evaluation of a position in the game of Go requires many specific concepts, and extensive reasoning. As the game of Go can be divided into subgames, a reasonable goal for knowledge generating systems is to generate knowledge for these subgames, and not for the whole game of Go.

Knowledge can be generated through various techniques. Neural networks learning can be used to learn to select good tactical moves with backpropagation, as shown in Section 9.1. With the help of temporal difference methods, it can also be used to learn to evaluate the probability of an intersection becoming territory. In Section 9.2 we will show how neural networks have learned enough Go knowledge to enable some neural-network-based programs to be competitive. Knowledge is a key component of Go programs—in Section 9.3 we describe the kind of knowledge used in Go programs, and we insist on the difficulty in acquiring and maintaining it. The problems related to knowledge maintenance, and knowledge acquisition, are partially solved by the declarative learning techniques presented in Sections 9.4 (retrograde analysis), and 9.5 (logic metaprogramming). These techniques have proved useful in generating many useful tactical rules that increase the tactical problem-solving abilities of Go programs. Some other techniques for pattern generation have also been tried, such as the ecological algorithm of Kojima [75]—discussed in Section 9.6.

9.1. Backpropagation

Golem [47] is a program that learns to select tactical moves by using neural networks. Golem starts by doing a tactical search to find the status of strings having three liberties, or fewer—the goal of the search being to find whether, or not, the string can be captured. Golem uses a neural network to discard moves when they are too numerous. The inputs of the neural network correspond to those abstract concepts of the game of Go which are likely to influence the capture of strings.

The results of the tactical searches, and some abstract concepts, are used as inputs of another neural network. This network is trained on games between professional players. For each position, the goal of Golem is to rank the professional move above another move chosen randomly. After 2000 attempts, it ranks the moves in the test database correctly 87 percent of times. Golem has learnt to beat Many Faces of Go at the latter's low level.

9.2. Temporal Differences (TD) learning

TD learning has been successfully applied to Backgammon by Tesauro [129]. It might also be applied successfully to other games. Two fundamentally different abilities can be defined in games. The first one is to foresee the likely continuation of a game, either by tree search, or by reasoning. The second one is the ability to assess a position accurately, using patterns and some features of the position, but without calculating explicit move. Backgammon is a suitable game for TD learning because positional judgements are more important. Unlike Chess it does not require the ability to see many moves ahead. The game of Go is also played by using a lot of positional judgement, and knowledge about the shape of stones. The application of TD methods to the game of Go has yielded reasonable results, but Go programs based only on TD learning do not play as well as the best Go programs. However, the TD method has great potential to improve programs that already have a lot of knowledge. Two programs have so far used this method for the game of Go. Chronologically, the first one is the program designed by Schraudolph, Dayan and Sejnowski, the second one—that plays better but relies on more Go knowledge—is NeuroGo by Markus Enzenberger.

The program of Schraudolph, Dayan and Sejnowski

Schraudolph, Dayan and Sejnowski have applied TD learning to Go in the same way that Tesauro applied it to Backgammon [119]. Their first experiment was to train a completely connected 82–40–1 network by letting it play stochastically against itself.

As the game of Go is deterministic, moves are chosen stochastically, so as to explore enough state space. The best moves have an exponentially higher probability of being chosen than the worse ones. The output of the network learned to forecast the number of points by which Black or White would win. This network had to play 659,000 games before beating Wally, the worst public domain Go program. The direct application of TD to Go yields disappointing results in comparison to Backgammon. However, improvements can be made to enable TD to learn better. These improvements concern the architecture, the inputs, and the output of the network.

The first improvement consists of changing the output. The goal of the game of Go is to occupy as much territory as possible. Each intersection occupied by a color, at the end of the game, counts as one point. It is therefore worthwhile to make the network learn the final destiny of each intersection. It is easier to find the destiny of an intersection than the final score. Moreover, the game of Go has properties that make it possible to constrain the architecture of the network. For example, when colors are reversed, or when a reflection, or a rotation, of the board is performed, the properties of the shape of stones remain unchanged. Color symmetry is taken into account by giving opposite values to the inputs for Black and White ($+1$ for Black, -1 for White), and by putting the bias neuron to -1 when White has to play first. Weight adjustments take into account rotations and symmetries by sharing equivalent weights, and by adding the errors resulting from different, but equivalent, intersections.

The improved network was trained against three opponents: a random move generator, as well as the Wally program (a very weak, and simple, public domain Go program), and Many Faces of Go. The program learned to beat Wally after 2000 games, and Many Faces of Go, at its weak level, by playing 1000 games. Another network playing against itself learned to beat Wally after 3000 games. To date, these programs have never participated in any Computer Go competitions.

The NeuroGo program

NeuroGo is a program that uses more elaborate inputs than the former program [48]. In NeuroGo too, the goal of learning is to foresee whether an intersection will be friendly territory at the end of the game. The input of the network is constituted of units. One unit can be either an empty intersection, or a string of stones. The architecture of the network is therefore dependent on the position being considered.

Each unit has its own properties. Each string possesses four properties: having 1, 2, 3, 4, or $\geqslant 5$ liberties; having 1, 2, 3, or $\geqslant 4$ stones; the possibility of being captured if the color of the string moves first; the possibility of being captured if the color of the string plays second. The properties of empty intersections are: Black has 1, 2, 3, 4, or $\geqslant 5$ liberties if he plays on the intersection; White has 1, 2, 3, 4, or $\geqslant 5$ liberties if he plays on the intersection; Black can be captured if he plays on this intersection; White can be captured if he plays on this intersection; eye for Black in 0, 1, 2, or $\geqslant 3$, unanswered moves; eye for White in 0, 1, 2, or $\geqslant 3$ moves.

In addition to calculating the properties of the units, NeuroGo detects groups of stones (sets of connected strings), and the distances between strings (connectable in one move or in two moves). It uses this information to link units with weights corresponding to the relations between the units. The set of units of a position is converted into a graph. This graph enables the program to build relations within the neural network. There cannot be more than one relation between two units. If there is more than one, NeuroGo decides which one is the most important, and selects it. Two neurons of adjacent layers that correspond to two units are linked by the weight corresponding to their relation.

Neurogo has learned to beat 'Many Faces of Go' at an average level (8/20) after 4500 games against itself whereas the program by Schraudolph & al. only beat it at a weak level (3/20). NeuroGo participates in the Internet Computer Go ladder. It plays on small 9×9 boards as well as on 19×19 boards. It is currently situated in the top third of the ladder of Computer Go programs.

Conclusion and future work

To learn how to play well against another program, TD based programs have to play thousands of games, but they have a considerable advantage: they do not require much work from their programmer (NeuroGo is 5000 lines of C). As often with this type of learning, it is facilitated by the quality of the inputs of the neural network. Such is the case with NeuroGo, which learns better than another program which has only a physical description of the board. An interesting study would be to use the elaborate knowledge of the best programs as an input of networks trained by TD methods. This might greatly upgrade the level of programs. A problem might arise with the learning time, which would be much longer than with a simple representation. However, this approach offers the advantage of having the highest ratio Level of the program/Time to develop it.

9.3. Knowledge in Go programs

Knowledge in Go programs is constituted of specialized procedures to compute some specific Go concepts, and of pattern databases. The Go knowledge contained in the procedures ranges from simple concepts, such as the number of liberties of a string, to high level ones, such as the safety of a group. Ken Chen, the author of Go Intellect (one of the best Go programs), provides a good description of the knowledge contained in a competitive Go program [38]. Go Intellect has 3 types of frames: Blocks (i.e., strings in our terminology), Chains (sets of strongly connected strings), and Groups (loosely connected strings), each with about 30, 10, and 50 slots, respectively. The knowledge of Go Intellect is made up of the frame updating procedures, of about twenty goal-oriented move generators, and two pattern libraries. A good example of the heuristic rules used to evaluate the life and death of a group can be found in [39]. The rules combine patterns, conditions on the intersections, and exceptions. Approximately forty rules are described in this paper. They rely on other knowledge, such as knowledge about the possibility of connecting a string to an intersection. A typical Go program contains approximately 3000 patterns. The number of patterns in a program is highly dependent on the design, and the architecture, of the program. For example, Goemate has only 40 patterns, whereas Golois has millions of patterns.

Acquiring this specialized knowledge is very difficult. The traditional way for the programmer is to add knowledge. The difficulty lies in performing introspection. New knowledge interacts with existing knowledge in unpredictable ways. Whenever a programmer tries to improve his program by adding knowledge relating to a particular sub-problem, this new knowledge often interacts with other knowledge in another part of the program, and finally produces bad results. Furthermore, even if the programmer is a very good Go player, he has difficulties in finding rules without exceptions. He often inserts new rules, forgetting the exceptions, and produces bad results again.

A potential solution to the knowledge acquisition problem is declarative knowledge learning. Some Go knowledge can be formalized, and can be considered independently of other knowledge. The insertion of a new rule into a declarative setting does not interfere with previous knowledge. Moreover, the use of automatic generating techniques produces a large amount of knowledge, which would take too long to write down for any programmer. The following two subsections highlight these techniques.

9.4. Generation of rules using retrograde analysis

Retrograde analysis

Retrograde analysis has already been used for one-and-two-player games. Given some final positions, retrograde analysis enumerates all the positions, and associates them with a value that can be Won, Lost, or Drawn for two players games, as well as with the minimal distance to the final position. Well-known results of retrograde analysis are the Chess endgame databases [123,131,132], and the Checkers endgame databases of Chinook. The Chess endgame databases have enabled the Chess community to discover new Chess knowledge [98], and they enable programs to play perfectly some endgames that are hard for human players. The endgame databases of Chinook, for Checkers, have contributed to its world champion title [78,118].

For single agent problems, retrograde analysis has been used to reduce the number of necessary nodes to solve standard 15-Puzzle problems. The generated database enables the program to divide the number of explored nodes by 1000 [45]. Retrograde analysis has also been used to find optimal solutions to Rubik's Cube [77]. The dynamic creation of pattern databases has been used as real time learning to accelerate Sokoban problem solving [68].

Generation of tactical Go rules by retrograde analysis

The generation of simple patterns by retrograde analysis has been performed for the game of Go regarding life, eyes, and connection [27,29]. The work has been extended to tactical rules—in other words, patterns associated with external conditions [34]. The generated patterns (and associated rules) fit in small rectangular shapes (2×2 to 6×3) that represent parts of the board. The external conditions correspond to constraints on relations with objects that are outside the rectangle. The possible objects are friendly strings, enemy strings, and empty intersections.

Each intersection in a pattern can take on three different values (black, white or empty). An empty intersection can be associated with two different external conditions: the minimum number of liberties a friendly move would have when played on the intersection, and the maximum number of liberties an enemy move would have when played on the intersection. Each string in a pattern can be associated with one condition: the minimum number of liberties for a friendly string, and the maximum number of liberties for an enemy string. Each of the external conditions can take on three different values. The three possible conditions associated to enemy strings are : 0 external liberties, <=1 external liberties, and other (>1 external liberties). The three possible conditions associated to friendly strings are: 0 external liberties, >=1 external liberties, and >=2 external liberties. Therefore, there are three possibilities for each string, and nine possibilities for each empty intersection. A pattern that contains two empty intersections on the edge of the pattern and two strings generates $9 \times 9 \times 3 \times 3 = 729$ different rules.

One can compare the number of rules covered by this representation with the number of endgame positions in Checkers [78]. There are 406,309,208,481 endgame positions, with 8 pieces, in Checkers, whereas the number of possible rules in a 5×3 rectangle in the center of a Go board is 59,241,069,331,995. The number of possible patterns is much lower, as there are at most three possible values for each intersection of a pattern (Friend, Enemy, and Empty). Therefore there are 3^{15} possible patterns for a 5×3 rectangle pattern in the center. Table 6 provides the total number of possible patterns and rules corresponding to the size of the pattern and its location.

The algorithms used to generate these rules are slightly different from those used to generate endgame databases.

Usually, when one generates a pattern database, or an endgame database, only one or two bits are used to code each element (these two bits are used to code one of the three

Table 6

Size of the pattern	Location	Total number of possible patterns	Total number of possible rules
2×2	Corner	81	5,133
3×2	Corner	729	184,137
4×2	Corner	6,561	6,498,165
3×3	Corner	19,683	23,719,791
5×2	Corner	59,049	228,469,857
4×3	Corner	531,441	3,238,523,049
6×2	Corner	531,441	8,023,996,893
5×3	Corner	14,348,907	464,991,949,659
3×2	Side	729	541,101
4×2	Side	6,561	18,513,177
3×3	Side	19,683	191,890,599
5×2	Side	59,049	631,651,053
4×3	Side	531,441	20,752,761,345
6×2	Side	531,441	21,555,306,681
3×4	Side	531,441	68,094,804,369
5×3	Side	14,348,907	2,353,796,975,871
3×3	Center	19,683	663,693,159
4×3	Center	531,441	239,111,765,601
5×3	Center	14,348,907	59,241,069,331,995

values: Won, Lost or Drawn) [45,68,77,78]. The bit arrays can be compressed by using standard compressing methods, such as Run-Length Encoding, which is used by Chinook. The positions are sometimes associated with a byte that stores the minimal number of moves before winning, or the maximal number of moves before losing [118,131,132].

Given the large number of possible rules, this representation is not used for databases of rules. Instead, each pattern is represented by a 32-bit unsigned integer, associated with a set of conditions. The only rules that are kept are the rules about won and winning states, since they represent only a small proportion of the possible rules. This allows one to store the generated rules within a reasonable space. Table 7 provides the number of generated rules for eyes on the side.

A simple algorithm to generate pattern databases consists of going through all possible rules, and for each rule testing whether it is in the Won state for the given goal. After each traverse of the possible rules, any new rules can be found only one move before the already discovered rules. So, to discover all the possible rules, the algorithm has to traverse all the possible rules many times, as long as there are new rules discovered during the previous crossing. This algorithm is not suited to Go rules' databases since the number of possible

Table 7

Size of the pattern	Location	Number of won rules	Number of winning rules
3 × 2	Side	11	108
4 × 2	Side	171	1,081
3 × 3	Side	727	5,570
5 × 2	Side	1,661	5,952
4 × 3	Side	38,909	146,272
3 × 4	Side	14,966	62,329
6 × 2	Side	18,194	31,500

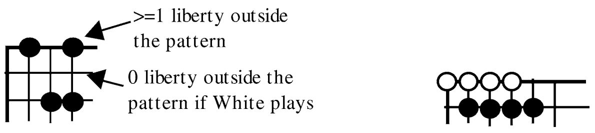
>=1 liberty outside the pattern
0 liberty outside the pattern if White plays

Fig. 31.

rules is very high. A more appropriate algorithm starts from the already generated rules, and undoes the previous rules so as to find the new rules directly, and without looking at all the possible rules to find them.

Fig. 31 shows some generated rules whose aim is to make the black group live in the corner. The use of generated rules greatly enhances the life and death problem solving abilities of Golois for open groups [34].

9.5. Explanation-based generalization and metaprogramming

For domains—like the game of Go—that have a strong underlying theory, a deductive learning method has been developed. It is called Explanation-Based Generalization (EBG) [88], and also Explanation-Based Learning (EBL) [46]. Many studies have focused on the application of this method to planning [86,87]. This type of learning is particularly useful for games. Many programs using deductive learning for games have been described [85,100], the work by Pitrat on Chess [101] marking the onset of these studies.

The application of EBG to the game of Go has been partially formalized by Kojima [74], and has been developed for, and efficiently applied to, many subproblems of the game of Go, using the Introspect system [30].

Explanation-based generalization

The use of EBG for the game of Go has been attempted by Kojima [74] for the capture of stones. As shown in Fig. 32, the program starts with the positions where stones are captured, and undoes moves to find positions where stones are captured some moves ahead.

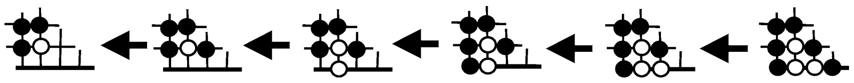
Fig. 32.

The information used by the generated rules concerns only the colors of stones and their positions. There is other, more abstract, and potentially very useful information which is not used to generate interesting rules: e.g., the number of liberties of the strings, or the number of shared liberties between strings. The system does not find the forced moves by itself. On the contrary, they are given to it. Despite its limits, this approach offers the advantage of uncovering a way to apply EBG to the game of Go.

Introspect

Introspect is a system for metaprogramming, and for EBG, that creates tactical rules for multiple games, and more particularly for the game of Go. It uses predicate logic—for example a rule to connect two strings is represented as:

```
connect(S1, S2, I, Color) :-
  color_string(S1, Color), color_string(S2, Color),
  liberty(I, S1), liberty(I, S2).
```

This rule shows that the two strings of stones, `S1` and `S2`, can be connected with a move of color `Color`, on intersection `I`, if `S1` and `S2` are also of color `Color`, and if intersection `I` is a liberty of `S1`, and a liberty of `S2`.

The generated rules deal with the tactical sub-goals of the game of Go: capturing a string, making a string live, connecting two strings, disconnecting two strings, making an eye, and removing an eye. They are used to develop tactical tree searches, and to stop search at an earlier stage, as well as to reduce the number of moves to be examined. Their originality comes from the fact they are theorems of the game of Go. Thanks to this characteristic, the set of forced moves they conclude on is complete. In this way, if we can prove that none of this limited number of forced moves works, then we have proved that no move works. Similarly, when at some node a rule concludes that a goal is achieved, search can be stopped, with the certainty that the goal is effectively achieved whatever the surrounding situation.

The rules that are used to develop the OR nodes of the search trees are created, either by unfolding rules concluding on a won goal, or a winning goal [5,56,126], or by EBG [30, 46,62,88,101].

As well as the classical operations on logic programs that enable Introspect to generate rules for the OR nodes, Introspect also uses metaprograms, which are specific to games, to generate rules about forced moves for the AND nodes of the search trees [32]. These specific metaprograms analyze a winning rule, and find all the moves that invalidate a condition of this rule. All these invalidating moves constitute the complete set of forced moves that prevent the opponent from achieving the goal that the winning rule concludes on.

But EBG systems create too much knowledge, and sometimes useless rules that slow the system down. To avoid this phenomenon, called the utility problem, and to enable the system to limit itself, Introspect only generates rules that contain fewer than 200 conditions, and that do not conclude on sets of more than 5 forced moves. Unlike other domains, where the utility problem is a major obstacle, good results can be obtained for the game of Go with quite simple utility knowledge.

The following rule is generated by Introspect regarding the capture of a string. It concludes on the capture of the string represented by variable `S`. The color of the move to capture is represented by variable `C`, and the intersection on which the move should be played to capture the string is represented by variable `I`:

```
move_to_capture(C, S, I) :-
  opposite_color(C1, C),
  color_string(S, C1),
  number_of_liberties_of_string(S, 2),
  minimum_number_of_liberties_of_adjacent_strings(S, 2),
  liberty_string(I, S),
  minimum_number_of_liberties_if_move(I, C, 2),
  liberty_string(I1, S),
  I = \ = I1,
  number_of_liberties_of_string_if_move(S, [I, C], [I1, C1], 1).
```

The condition `minimum_number_of_liberties_of_adjacent_strings(S, 2)` verifies that all the strings adjacent to the string `S` have at least two liberties. The condition `minimum_number_of_liberties_if_move(I,C,2)` checks that a move of color `C`, on the intersection `I`, has at least two liberties. The condition `number_of_liberties_of_string_if_move(S,[I,C],[I1,C1],1)` verifies that a move of color `C`, on intersection `I`, followed by a move of color `C1`, on intersection `I1` has only one liberty. An illustration of this rule is Fig. 33 where Black can capture the White string by playing one of its liberties.

The tactical programs generated by Introspect are used by the Go program Gogol. They have enabled it to obtain decent results in Computer Go competitions [53]. Once generated, the rules are gathered in a tree of conditions, and compiled into C. When they are integrated into Gogol, they amount to one million lines of C.

Introspect is also a general game, and a metaprogramming, system [100,102] that has generated knowledge for domains other than the game of Go [30].

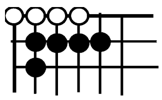

Fig. 33.

9.6. Pattern generation by an ecological algorithm

Kojima has also used an ecological algorithm to discover patterns for the game of Go [75]. He uses games between professional players as examples. The rules contain conditions on the relative positions of the stones (friendly or enemy stones, relative coordinates), and on their location on the board (on the edge or not). Each rule is considered as an individual, and has an activation value. A learning example is considered as food, that can be eaten by the rules which match this example. If no rule matches, then a new rule, that matches the example, is created. When a rule has an activation value above a given threshold, it gives birth to a more specific rule. Each rule eats at each step of the algorithm, so its activation value decreases. The rules with an activation value of 0 die. This approach to pattern generation has not yet generated a competitive Go program.

9.7. Conclusion

Many learning methods have been tried for the game of Go. So far, none has given very astonishing results. However, the temporal differences method, and the program specialization method (either by retrograde analysis or by metaprogramming), give interesting results, when compared to more classical methods for game programming: they are undoubtedly the most promising learning methods for the game of Go.

The automatic generation of knowledge still needs to be investigated more: the game of Go is the domain, ‘par excellence’, where learning, and program specialization, methods can be very useful. One of the principal difficulties in programming the game of Go lies in finding knowledge, and then in writing a relevant program that represents this knowledge. Such is the current task of learning Go programs.

10. Monte Carlo Go

10.1. Monte Carlo methods in physics

The variation principle lies at the heart of theoretical physics. The path taken by a system in a state space is an extremum. The mechanisms used to find extrema are fundamental in classical physics, in relativistic physics, and in quantum physics, as well. Monte Carlo methods [72,99] are derived from statistical physics. A statistical system is composed of a large number of particles. The problem is to know how the speed, and the position, of particles evolve over time. A feature of the evolution of the system is that a quantity, such as the energy, or the activity, is minimized. For example, at high temperatures, a metal is liquid, and its atoms move randomly, but when the metal is cooled, the atoms put themselves into a configuration that minimizes energy—a crystalline structure. This process is called annealing. The longer the cooling, the closer to the minimum of energy the cooled structure is.

To do Monte Carlo simulations, one has to choose a representation of the possible states of a system. Then, one has to define a set of ergodic, elementary, moves, that allows one to go through the state space of configurations, step by step. For a given configuration, moves are generated randomly. The evolution of the system is approximately assessed by choosing a move with a probability that depends on the growth in activity resulting from the move. For example, the probability $p(E)$ that a particle has the energy E at a temperature T is $p(E) = \exp(-E/kT)$, k being the Boltzmann constant. The move that increases the energy by ΔE is accepted with the probability $p = \exp(-\Delta E/kT)$.

10.2. Simulated annealing

For a function of many variables, algorithms that follow the gradient, to find lower values of the function enable us to find a local minimum of the function. However, if we look for a global minimum of a complex, and possibly non-differentiable, function, there is little chance of finding it by using such a method. Simulated annealing is more appropriate to find global minima of complex functions that contain many variables, because it also generates moves that increase the value of the function.

To find a global minimum, simulated annealing plays moves randomly in state space. If the value of the function decreases after the move, the move is accepted. If the move increases the value of the function, the move is accepted with a probability that decreases exponentially with the increase of the value of the function, and with the temperature. The temperature decreases with time, depending on the time given to the algorithm to find a solution.

Simulated annealing has been applied to combinatorial optimization problems like the traveling salesman problem. There are $N!$ different paths between the N cities. Simulated annealing finds a solution, close to the optimal solution, in a polynomial time. The algorithm begins with a random ordering of the cities. There are two types of possible moves. The first type of move is to reverse the order of several cities, which are next to one another on the path. The other type of move is to move several neighboring cities somewhere else on the path. Simulated annealing has also been used to find an arrangement of 25 cards, in a 5×5 table, so that the values of the rows, columns, and diagonals, when interpreted as hands in Poker, are maximized. Simulated annealing is successful, and both much faster, and simpler, than other methods.

10.3. The Gobble program

The Gobble program [22] uses simulated annealing to find an approximation of the best move on a 9×9 board. The principle is the same as the Go-moku program designed by Grigoriev [59]. It consists of randomly developing the different possible games, and in calculating statistics based on the results of the sequences of moves, after each possible next move. The goal of the program is to find an approximation of the value of the different possible moves. To this end, it plays each sequence of moves until the last moves that do not fill eyes. In case of captures, many moves can be played at the same intersection. At the end of a sequence of moves, it counts the number of points for Black, and for White. The final values, associated to an intersection, are the mean of the results corresponding to the sequences of moves in which Black has been the first to play at the intersection.

The problem consists of finding the order in which to play the possible moves in the different sequences. The order of moves must be different, and must bring information for each sequence. Therefore, Gobble uses a different vector of moves for each sequence. The first legal move in the sequence of moves is chosen to continue the sequence, each vector of moves corresponds to a sequence. For all sequences, the initial order of the vector of moves is found by sorting the moves according to their approximate values calculated with the former sequences. This initial order is then modified by going through the vector of moves and by swapping two neighboring moves with a probability that depends on temperature. As sequences are being played, temperature decreases, and the probability that two moves are swapped also decreases.

The moves are sorted according to their values. The value is initialized to 0, for the first iteration, and then it is updated for each sequence of moves. Then, the program goes through the list of moves, and swaps two neighboring moves with the probability p_{swap}. The probability $p(n)$ that a move is moved n steps down is:

$$p(n) = (p_{\text{swap}})^n = \exp(-n/T),$$

so

$$T = -1/\ln(p_{\text{swap}}).$$

p_{swap} decreases linearly down to 0, depending on the number of sequences. Then, for some sequences, p_{swap} remains at 0, in order to find the nearest local extremum. The useful information obtained from random sequences is proportionate to the number of sequences already played. The mean error, Δv, is proportional to the square root of the number of sequences:

$$\Delta v \sim 1/\sqrt{n}.$$

Therefore, to calculate the value of a move to a precision of within one-point, and given that the possible values for the results of the sequences can vary by 100 points, approximately 10,000 sequences have to be played.

Gobble uses two strategies: strategy A: play between 500 and 1000 sequences to find the moves; strategy B: play 400 sequences, retain the 4 best moves, and play 400 sequences for each of these 4 moves. Using strategy A, and despite giving/receiving three handicap stones, Gobble plays equal games against Many Faces of Go. Using strategy B, and despite giving/receiving two handicap stones, Gobble plays correctly against Many Faces of Go.

A specific property of the game of Go, which is taken into account by Gobble to evaluate the moves, lies in the localization of the moves. Locally, the game of Go is more stable than Chess, for example. However, the results are only an approximation of the real result, since the localization of moves is not always verified, and the state space of Go not always regular.

The Gobble approach is original, because it relies on a very limited knowledge of Go—‘do not fill your own eyes’—and yet it results in an average Go program. Given the simplicity of the program, its performances are amazing. This approach can undoubtedly be improved, and conclusions can be drawn from these experiments for other game programming problems.

11. Mathematical morphology

11.1. Introduction

This section highlights the link between image processing and Computer Go. The size of the board (19×19) on which the game is played, is much smaller than the size of the pictures (more than 1000×1000) processed in the pattern recognition domain. Therefore, the complexity of the game of Go is situated far below the complexity of image processing. Nevertheless these two domains have a common, and efficient, tool—mathematical morphology [120]. Hence, this section shows how to use mathematical morphology within Computer Go, and, more specifically, within the EF of a Go program.

11.2. Mathematical morphology and the game of Go

Mathematical morphology is a very useful technique in the field of image processing. For example, some operators enable systems to delete the details whose size ranks below a given scale. Fuzzy mathematical morphology [12] is another refinement that also gives good results. Besides, some Go programmers use it in their program. At the very beginning of Computer Go, the Zobrist model [147], without using it explicitly, already performed mathematical morphology. This model was composed of iterative dilations. It enabled the programs to recognize “influence” as human players do. This model is the ancestor of the refinements used today in Go programs. For example, the Indigo program makes explicit use of mathematical morphology [15,16] for territory-, and influence-modeling. GnuGo [57] also uses this model. This section focuses on “territories”, and “influence”, recognition, and provides information to help understand the Evaluation Function section.

First, let us mention some basic operators of mathematical morphology. Let A be a set of elements, and let $D(A)$ be the morphological dilation of A—composed of A, plus the neighboring elements of A. $E(A)$ is the morphological erosion of A. It is composed of A, minus the elements which are neighbors of the complement of A. ExtBound(A) is the morphological external boundary of A; given by $\mathrm{ExtBound}(A) = D(A) - A$. IntBound($A$) is the morphological internal boundary of A; given by $\mathrm{IntBound}(A) = A - E(A)$. Closing($A$) is the morphological closing of A; where $\mathrm{Closing}(A) = E(D(A))$. Opening($A$) is the morphological opening of A; given by $\mathrm{Opening}(A) = D(E(A))$. The opening and closing operators are very helpful in image processing.

We can then adapt these operators by adding two refinements: numerical inputs and outputs, and two colors (Black and White). We start by assigning values of $+64$ (respectively -64) to black (respectively white) intersections, and 0 elsewhere. The D operator now consists of adding to the absolute value of an intersection of one color, the number of neighboring intersections of this color, provided that all the neighboring intersections are empty, or of that color. For an empty intersection, which has neighboring intersections of the same color, D also adds the number of neighboring intersections of this color to the absolute value of the intersection. D is a numerical refinement of the classical dilation operator mentioned above. Similarly, the E operator now consists of subtracting from the absolute value of an intersection of one color, the number of neighboring intersections whose value is either zero, or whose value corresponds to the

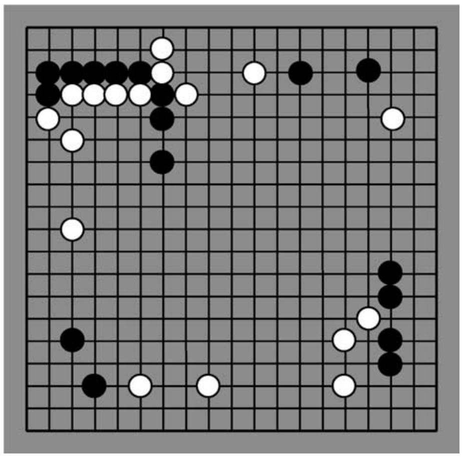
Fig. 34.

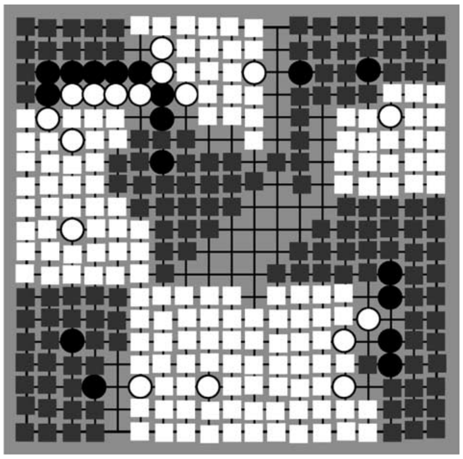
Fig. 35.

opposite color of the intersection. E also makes sure that the sign of the value does not change—otherwise, the value becomes zero. E is therefore a numerical refinement of classical morphological erosion.

Once these refinements are added, we use the operators $X = E * E * \cdots * E * D * D * \cdots * D$, and $Y = D * D * \cdots * D$ where E is composed 'e' times, and D, 'd' times. So as to give the same result as the identity operator in positions where no "territory" is recognized, a link between 'e' and 'd' must be established. For example, in the trivial position, with only one stone located in the middle of the board, X must give the same result as the identity operator. Bouzy [16] has shown that $e = d * (d - 1) + 1$, in which 'd' is a scaling factor. The bigger 'd' is, the larger the scale of recognized territories is.

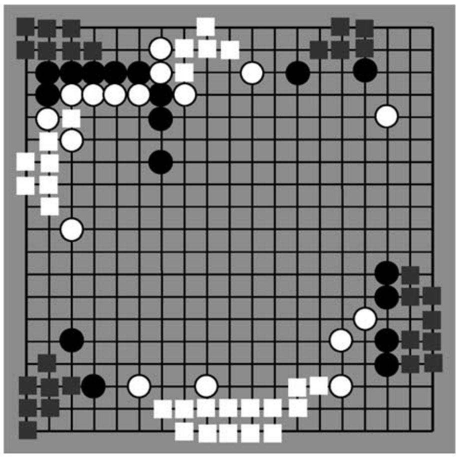
Fig. 36.

Fig. 34 illustrates an example position on which we applied operators X and Y. Fig. 35 shows the result of Y, with $d = 5$, and $e = 0$. Fig. 36 shows the result of X, with $d = 5$ and, $e = 21$. Fig. 35 shows what Go players call "influence" and Fig. 36 shows the "territories" quite accurately. These two points explain the success of mathematical morphology in the game of Go [15]. This technique was part of the EF of the Indigo program [15,16], and now has been integrated in the GnuGo program [57].

12. Cognitive science

12.1. Introduction

This section deals with studies carried out in Cognitive Science which use the game of Go as an investigation aid. After showing that the game of Go is an appropriate domain for cognitive science studies, this section investigates the different studies conducted so far.

12.2. The game of Go is appropriate for carrying out cognitive science studies

Cognitive science experiments, aiming at exploring the human cognitive system when it interacts with the real world, have to be set up inside sufficiently complex domains. Thus, the subjects will be able to use their cognitive faculties that will be studied through the experiment. So as to be effective in practice, the experiments must not be too complex. Therefore, domains whose complexity is neither too low, nor too high, are perfectly adapted. Furthermore, the knowledge used by the subjects during the experiments must be representative of common sense knowledge used by human beings in the real world. Therefore, the domain to be studied must also keep real world features. To sum up, Cognitive Science requires domains which are representative of the real world, and whose complexity lies between the complexity of recreational experiments and the complexity of real-world, but technically ineffective, experiments.

Langston [80] defends the idea that the game of Go is a simplification of the real world while keeping its main features by the following arguments. The Go universe has two spatial dimensions and one temporal dimension whereas the real world has three spatial dimensions and one temporal dimension. In addition, the Go universe is finite—there are 361 intersections on a Go board, and these are endowed with a color which has a discrete value (black, white, or empty). However, the real world is made up of an infinity of points. Furthermore, an infinity of viewpoints describes each point, and each viewpoint takes on a value belonging to an infinite set. Unlike the real world, the Go universe is formal: the rules of the game define the characteristics of the game with great accuracy.

Actually, the main advantage of the game of Go compared to other games is that it is visual: a position with its black, white, or empty intersections must not be perceived simply as such, but rather in an abstract way. In these conditions, the player may identify complex objects. The strength of a player relies on his skill in recognizing complex objects where only concrete information (black, white, and empty intersections) can be found. This aspect does not exist, as markedly, in other games such as Chess, Checkers, or Othello, where the objects of the reasoning process are similarly defined by the rules of the game.

Therefore, it is quite justifiable to choose the game of Go as a domain to perform cognitive experiments [15,112]. We shall now see which studies, using the game of Go, have been done so far within the cognitive science community.

12.3. Related works

The different studies in cognitive science that have been carried out using the game of Go may be classified according to the chosen method: on the one hand measuring response times for re-building observed, and hidden, positions—on the other hand analyzing verbal reports.

Measuring response times

In Chess, Chase and Simon [35] measured time intervals separating the actions of the subjects who were reconstructing positions, once seen, but later hidden. They showed that expert players use information organized in "chunks". A chunk can be defined as a cluster of information. The authors observed that, because Chess experts build actual Chess positions more quickly, they seem to have a greater memory capacity than non-experts. However, with random positions, the authors observed that the experts' and the non-experts' performances were equal. The explanation given by the authors was that the number of memorized chunks is equal for experts and non-experts, but that experts memorize more specialized chunks. This argument would explain the time differences when building actual Chess positions.

In Go, this experiment has been done again [104]. As in Chess, the experiment shows that experts use more specialized chunks than non-experts. Nevertheless, the conclusions were more difficult to reach in Go. Unlike in Chess, the chunks in Go are more complex, as they are not linearly structured, but may be chosen differently, according to different viewpoints.

Burmeister and Wiles [23] describe experiments which use "inferential" information to build Go positions. This contrasts with "perceptual" information which was used in previous experiments [35,61]. A more detailed study of strong Japanese players (6 to 8 dan) was then conducted [24]. The authors conclude that the explanation for moves is an important factor in memorizing positions, and sequences of moves. Thus, strong players clearly remember their games against other strong players, but they have more difficulty remembering their games against weak players, because weak moves have no meaning for them. Consequently, the strong player must use a representation that he is not familiar with—this reduces his memorizing capacities.

Analyzing verbal reports

Scores of cognitive studies rely on natural language production. First, let us sum up the arguments for, and against, both the use, and the study, of such types of information, irrespective of the domain. On the methodological side, Ericsson and Simon [49] propose a method to set up cognitive models relying on verbal reports. The success of the method simply lies in a verbal production model included within the cognitive model. Thus, the validation of the cognitive model is simply performed by comparing the verbal production of the cognitive model with that of the subjects. Concerning the results obtained in analyzing verbal reports, Vermersch [135] defends the idea that it is possible to extract knowledge from many everyday life experiments such as driving a car, and baking a cake. The author thinks that the extraction can be done with such precision that accurate cognitive models could be built. On the contrary, Nisbett and Wilson [97] argue that psychological experiments in general, and verbal reports in particular, are strongly distorted by the experimenters themselves. For the authors, such experimenters uncover only what they are looking for, and if the experimenters could make other experiments aiming at proving the contrary, they would do so. As a result, there are many divergent opinions as to the effectiveness of experiments which are based on verbal reports.

In Go, Saito and Yoshikawa [112,113,115] showed that human players use natural language terms to play their games. The authors recorded the players' voices when they were playing games, or solving Go problems. They also used specialized hardware to record the subjects' eye movements, as they looked at the Go board. The authors observed that friendly and opposing moves are very often named. Small tree searches appear in verbal expressions, and proverbs can be found. The authors conclude that natural language plays an important role in playing Go. They also show that the use of terms depends on the players' rankings [145]. So as to demonstrate this, they developed the "iceberg model" [146], which shows that most knowledge is implicit, and not conscious, while little information is explicitly present in verbal reports.

Bouzy [15] presents a cognitive model of the Go player. It was designed using the following steps. First, a cognitive model was based on strong players' verbal reports. The cognitive model had to be validated with the construction of a computational model. But, unfortunately, the results of that computational model were not conclusive. Therefore, the cognitive model was simplified, in a second stage, by using novice players' verbalizations. Surprisingly, the level of the computational model based on verbal reports by novice players was higher than the level of the program based on experts' verbal reports. This obvious paradox may be explained by the fact that the expert verbalization-based model used high level knowledge, without referring to the low-level knowledge. Therefore, this model—with high level knowledge—was not grounded on solid foundations, and the corresponding program had poor results. The low-level knowledge of the other program enabled it to play at a novice level with better results.

Nevertheless, the most significant conclusion of this work was to be found elsewhere. Of course, the program's implementation required the use of concepts explicitly expressed in the reports, but it also required the use of further concepts, which are called hidden concepts, because they cannot be explicitly identified in the reports. Thereby, it was assumed that a correspondence existed between the hidden concepts, on the one hand, and the implicit knowledge used by human players, on the other hand. This hypothesis bears great similarity to the iceberg model of [146]. This correspondence hypothesis being assumed, Bouzy [15,19] has shown the existence of human players' implicit knowledge such as, the group concept, the inside/outside concept (see "Evaluation Function" section), and the incrementality concept [20] (see "Optimization" section). Incidentally, these concepts are very intuitive, and very useful in helping to play Go. They do not appear very clearly in Go players' verbal reports, thus making it difficult to set up a correct model in Go. Nevertheless, this approach may let researchers obtain insights into the way that humans use implicit knowledge. In [19] the author argues that when one tries a computer implementation in a complex domain, such as Go, one uncovers concepts, which make it easier to model the domain. These concepts do not correspond, either to verbal reports, or to natural language terms in that domain. As these concepts does not correspond to explicit knowledge, the hypothesis is that they correspond to implicit knowledge.

12.4. Conclusion

The game of Go, as a real world simplification that keeps its essential features, is an appropriate domain in which to conduct cognitive studies [80]. Reitman [104] reproduced the Chess experiment [35] in Go; their conclusions were far from being obvious because the chunks were not "linearly" organized, but corresponded to different viewpoints which were difficult to classify. Bouzy [15] has shown that the bulk of Go players' knowledge is not conscious, by extracting this knowledge through the implementation on the computer. Yoshikawa et al. [146] not only argue that natural language plays an important role in Go, but also maintain that Go players' knowledge is implicit, to a large extent.

13. Conclusion

13.1. Summary

In this paper, we have presented Computer Go, and its numerous links with AI. First, we have shown, in Section 2, that the complexity of Go is much higher than the complexity of the other two-player complete information games. This complexity is not only due to the number of possible moves in a given position, but also to the cost of the evaluation of a position.

In Section 3, we mentioned that the best Go programs obtain average results, although it took several years to develop these programs. The best current programs are Goemate, Go4++, Many Faces of Go, Wulu and Go Intellect. However, with sub-problems of Go (life and death problems, and endgame problems), the results are excellent. GoTools [140, 142] solves life and death problems whose level corresponds to the best amateur players. Berlekamp and Wolfe [9] describe a mathematical model that finds move sequences better than professional players' sequences, in some specific late endgame positions.

In contrast to the excellent results obtained with these sub-problems, the weak performance of programs in a complete game might be explained by the difficulty in extending the problem solving tools to become more general tools. It is vital that Go programs find good solutions in cases which occur in actual games. For example, it is difficult to solve life and death problems for groups that are not completely surrounded, although GoTools solves them very well for completely surrounded groups [142]. Section 4 described a Go evaluation function whose complexity is due to its many inter-dependent concepts: grouping, inside and outside, interaction, territory, influence, life, and death. Numerous definitions of these concepts are possible. The choices made in the design of each program bring about a loss of information. Moreover, these choices have to be implemented in an efficient fashion, thus making the design of an evaluation function even more difficult.

As described in Section 4, a characteristic of the evaluation function is the use of local tree searches on simple goals. To build the EF, TS on tactical goals, such as string capture, connection between strings, and eye verification, are commonly performed. The results of these tactical goals are used to build groups at the upper abstraction level. Other TS—on more abstract goals, such as life and death of groups, and interactions between groups—may be performed at a second stage. All these results are processed to build the EF at the global abstraction level. The EF calculation is slow enough to prevent a classical TS. Contrary to other games in which TS uses EF, in Go EF uses TS.

Section 5 mentioned that the first Go programs were expert systems with a single Move Generation module, but with neither an Evaluation Function, nor Tree Search. So far, although Evaluation Function, and Tree Search, modules have been necessary to write the current Go programs, the Move Generation module has occupied and still occupies a special position. Instead of being used by Tree Search, it can be used to select the move directly. Because TS can only be performed locally, or in a goal-oriented way, it must be performed after the computation of a static strategic position evaluation which requires an important amount of knowledge. Therefore, it is worth keeping the knowledge method, and having some static goal, and/or move evaluations, after the position evaluation. This time-saving approach is far from being simple. Move selection undoubtedly remains complex, and needs accurate domain-dependent knowledge. Global move generation has multiple facets, and must be supplemented by a tree search module, which is used to check that goals have been achieved.

Section 6 considered the characteristics of TS. First, the local situations of a position may be viewed as independent of one another, and the global TS may be approximated by several local TS—-that is why a local TS is a selective TS, where moves sufficiently far away from preceding ones are discarded. Another important feature of the game of Go is that, for each local situation, each player may be the first to play, which necessitates the computation of at least two TS for each local situation. The evaluation function is the result of local TS on simple goals, and can contain some uncertainty. A quiet position can be defined as a position in which there is no uncertainty. Quiescence search algorithms are thus appropriate in this approach.

The programmer can choose between many search algorithms. For the goals that use data structures, which can be incrementally updated, Alpha-Beta is the algorithm of choice. When the number of moves for each position varies a lot, it may be better to use PN-Search. For a given sub-problem, the search algorithm is chosen according to its characteristics.

Given the computational complexity of Go playing programs, it is important to perform computations as fast as possible. It is then possible to compute longer sequences of moves, and to obtain more information about the position, so as to make a more accurate evaluation. Section 7 listed the numerous possible optimizations at each level of abstraction of a program. For example, incrementally calculating the liberties of strings has become a widely accepted technique at the lowest level of the program. When computing a tactical search, the same sequences of moves are often repeated. An interesting optimization consists of playing sequences of moves, rather than playing one move at a time, and reanalyzing the position after each move. Another example of low level optimization centers on the use of bit-string operations, to calculate the dilation, and erosion, operators of mathematical morphology. Other optimizations are useful at more abstract levels. The objects related to the results of each local TS can be memorized, and the only local TS to be computed after a move are the ones that contain objects modified by the move. At the highest level, a program can be optimized by eliminating beforehand some computations of the lower levels, leaving the choice of move unchanged.

As presented in Section 8, the possibility of splitting the global position into several sub-positions enables programmers to apply the sum of game theory. Berlekamp achieved excellent results by applying this theory to the late endgame. This result can be explained by the specificity of the test positions in which local situations are totally independent from one another. In real game positions, the local situations are not independent, and the sum of game theory does not apply directly, although several attempts have been performed so far. Explorer is undoubtedly the program which uses this theory in the best way, thanks to thermograph calculations. Other studies on eyes [79], fights between groups [95], or temperature calculations [71] have also been performed.

Neural network learning methods, described in Section 9, and more specifically the temporal difference method, allow the program to automate most of the creation of an evaluation function, and to replace the strategic level of a Go program at a low cost. NeuroGo uses quickly defined, and quickly computed, inputs, and it successfully competes against some programs based on more elaborate methods and concepts. An interesting attempt would consist of using more elaborate inputs, but the learning time may then be prohibitive. So far, programs using EF, based on the TD method, have not reached the level of the best programs. However, they give very good results when comparing the design complexity of classical programs to the low complexity of programs based on this method. Other approaches, which aim to transfer the work of knowledge generation from the programmer to the computer, are logical metaprogramming, and retrograde analysis applied to tactical patterns. These approaches generate millions of rules that enable tactical problems to be solved more quickly, and more accurately. In well-defined sub-problems, like life and death of groups, Golois uses this automatically generated knowledge, and ranks at a similar, if not better, level than the other programs.

Very different problem solving methods may be adapted to the game of Go. In Section 10, Monte Carlo methods surprisingly provides average level Go programs, on 9×9 boards. Based on this method, Gobble is designed in a simple way, but, like NeuroGo, it competes effectively against far more complex programs.

Since the game of Go is visual, it is normal to explore the usual techniques of image processing to see whether they can be useful for Computer Go. This was examined in Section 11. Although 19×19 boards are much smaller than images processed in Computer Vision, mathematical morphology enables Go programs, like Indigo or GnuGo, to recognize territories, and influence by using dilation, erosion and closing operators.

Section 12 showed that Go is a domain, well-suited to the performance of cognitive experiments. In order to obtain significant results, without being confused by the complexity of the real world, cognitive science requires clearly formalized domains like games. So as to extract intuitive, and non-conscious, knowledge from human beings, the domain has to be sufficiently complex. The game of Go, with its intermediate complexity, is a good domain for experiments. The Indigo program has been devised to validate the cognitive model, based on the verbal reports of novice players. Saito and Yoshikawa [112] have shown, on the one hand, that Go players use natural language to guide their thinking process, and, on the other hand, that they use implicit knowledge in a way similar to the "iceberg model". Some experiments, already done in Chess, were repeated, for Go, by Reitman [104]. As in Chess, they show that expert players recognize real game positions better than do novice players, while obtaining comparable results on random positions.

13.2. Future work

Given the present state of Computer Go programming, we may wonder how Computer Go is likely to evolve over the next few years. We have mentioned many different studies, and we can try to figure out which paradigm will result from all these studies.

Computer Go programmers currently agree on very few concepts and tools. All the programs have modules to compute the capture of strings. They also have modules for connecting strings, for killing groups, and for making groups live. But, even at this relatively low level of abstraction, the underlying concepts of the modules differ from one program to another. For example, Many Faces of Go uses an incremental Alpha-Beta to compute strong connections, whereas Go4++ uses a connectivity probability map. Since a Go program is an interconnected whole, it is difficult to argue about the best way to compute a connection. The choice of method for solving a sub-problem depends on the global architecture of the program, and on the choices made in other modules. The problem of comparing the pros and cons of the different architectures currently in use remains unsolved.

The EF computation is complex, because of its many interacting concepts. An attractive approach might be to design a multi-valued EF, each value corresponding to a concept. A bi-valued EF has been applied to fights between groups [95], whereas a multi-valued EF could be applied to other sub-problems of the game of Go. One problem will be to use this multi-valued EF in a TS.

We have shown that an automatic knowledge generator has been successfully applied to the tactical levels of a program. We can ask whether this technique will also give good results at more abstract levels. TD learning techniques, and simulated annealing techniques, have been used in programs using simple data structures, to reach a global goal. Some parts of the best programs might be improved by using such tools. First, they could be used at low abstraction levels, for instance to estimate the connectivity. Second, they could be used at the global levels, using more abstract information. Third, the Monte Carlo methods could be applied to sub-problems, such as life and death.

In 1995, Handtalk—the Computer Go world champion—played moves almost instantly. The speed might be due to assembly coding, and to limited tree search, associated with very good heuristics for finding good moves. The goal of a program is to select a move, TS being one way to find it. Unlike in Chess, TS in the game of Go can be partly replaced by other tools, that select good moves in some positions. In quiet positions, where tree search is not useful, a method based on a very abstract description of the board can give good results.

After obtaining results relating to the late endgame, eyes, and fights, combinatorial game theory will probably be successfully applied to other Go sub-problems. Concerning the full game: as some local computations do not reach their end, the global level of a program needs tools to represent uncertainty. A good program has to use an uncertainty description, which is not to be found in Conway's theory. In addition, the local games of an actual game are not independent of one another. On the contrary, they are very dependent on each other. At the moment, some programs use the simple idea of increasing the priority of those moves which contribute to several local dependent games. But, so far, no study has been performed on this idea, and a formalization of the different independence classes has yet to be set up. An interesting idea would be to formalize, both the use of one goal by another, and the interaction between several games. The problem of performing TS on conjunctions, and disjunctions, of goals remains to be solved.

The promising results of GoTools with life and death problems, containing completely surrounded groups, offer some attractive prospects of developing a life and death problem solver for partially surrounded groups. This kind of problem arises in actual games, and such a problem solver would be very useful. The number of possible moves may increase, while the search deepens, and this constitutes the major obstacle to building this solver. Search in life and death of completely surrounded groups is easier, because the number of possible moves decreases as the depth increases. Another obstacle to be taken into account is the dynamic definition of the goal to be reached (making two eyes, escaping, and fighting).

Another possibility in the evolution of programs might be the use of parallelism. The distributed nature of the game of Go makes this idea appealing. At present, no program uses parallelism.

Lastly, we have to mention the likely evolution of the level of Go programs, in the years to come. In the late 1980s, Mr. Ing decided to award prizes to any programs which could beat a professional player, in a series of even games, before the year 2000. Today, young professional players still give 9 handicap stones to the best programs, and players who are used to playing against programs, are able to give as many as 29 handicap stones to these programs. This difference in terms of number of handicap stones can be explained as follows. During the first few games—when its human opponent confronts the strengths of the computer—the program may give the illusion of being stronger than it actually is, and it plays at its “high” level. Some games later, the human opponent discovers the weaknesses of the computer, and still later, the human opponent identifies almost all the weaknesses of the computer, whose level generally drops to its “low” level. Nowadays, the “high” level of the best programs may be assessed at 5th kyu—this corresponds to an average player in a Go club. However, their “low” level ranks at 15th kyu, namely a beginner level. As long as this gap, between the low and the high levels, is not reduced, it is risky to make any prediction about the evolution of the level of Go programs.

References

[1] L.V. Allis, Searching for solutions in games and artificial intelligence, Ph.D. Thesis, Vrije Universitat, Amsterdam, 1994, http://www.cs.vu/~victor/thesis.html.

[2] L.V. Allis, M. van der Meulen, H.J. van den Herik, Proof-number search, Artificial Intelligence 66 (1994) 91–124.

[3] L.V. Allis, H.J. van den Herik, M. Huntjens, Go-moku solved by new search techniques, Comput. Intelligence 11 (4) (1995).

[4] T.S. Anantharaman, M. Campbell, F.H. Hsu, Singular extensions: Adding selectivity to brute force searching, Artificial Intelligence 43 (1) (1989) 99–109.

[5] J. Barklund, Metaprogramming in logic, UPMAIL Technical Report 80, Uppsala, Sweden, 1994.

[6] D. Beal, A generalised quiescence search algorithm, Artificial Intelligence 43 (1) (1989) 85–98.

[7] D. Benson, Life in the game of Go, Inform. Sci. 10 (1976) 17–29. Reprinted in: D.N.L. Levy (Ed.), Computer Games, Vols. 1 and 2, Springer, New York, 1988.

[8] E. Berlekamp, Introductory overview of mathematical Go endgames, in: Proceedings of Symposia in Applied Mathematics, 43, 1991.

[9] E. Berlekamp, D. Wolfe, Mathematical Go Endgames, Nightmares for the Professional Go Player, Ishi Press International, San Jose, CA, 1994.

[10] H.J. Berliner, The B* tree search algorithm: A best-first proof procedure, Artificial Intelligence 12 (1979) 23–40.

[11] H.J. Berliner, Backgammon computer program beats world champion, Artificial Intelligence 14 (1980) 205–220.

[12] I. Bloch, H. Maître, Ensembles flous et morphologie mathématique, Télécom Paris 92 D007, Département Images, Groupe Image, 1992.

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

[14] M. Boon, Overzicht van de ontwikkeling van een Go spelend programma, Afstudeer Scriptie Informatica onder begeleiding van Prof. J. Bergstra, Amsterdam, 1991.

[15] B. Bouzy, Modélisation cognitive du joueur de Go, Ph.D. Thesis, University Paris 6, 1995, http://www.math-info.univ-paris5.fr/~bouzy.

[16] B. Bouzy, Les ensembles flous au jeu de Go, in: Actes des Rencontres Françaises sur la Logique Floue et ses Applications LFA-95, Paris, France, 1995, pp. 334–340.

[17] B. Bouzy, The Indigo program, in: Proceedings of the Second Game Programming Workshop in Japan, Hakone, 1995, pp. 197–206.

[18] B. Bouzy, There are no winning moves except the last, in: Proceedings IPMU-96, Granada, Spain, 1996, pp. 197–202.

[19] B. Bouzy, Explicitation de connaissances non conscientes par modélisation computationnelle dans un domaine complexe: le jeu de Go, in: Actes du 2eme Colloque Jeunes Chercheurs en Sciences Cognitives Giens, France, 1996, pp. 276–279.

[20] B. Bouzy, Incremental updating of objects in Indigo, in: Proceedings of the fourth Game Programming Workshop in Japan, Hakone, 1997, pp. 179–188.

[21] B. Bouzy, Complex games in practice, in: Proceedings of the Fifth Game Programming Workshop in Japan, Hakone, 1999, pp. 53–60.

[22] B. Brugmann, Monte Carlo Go, ftp://www.joy.ne.jp/welcome/igs/Go/computer/mcgo.tex.Z.

[23] J. Burmeister, J. Wiles, The use of inferential information in remembering Go positions, in: Proceedings of the Third Game Programming Workshop in Japan, Hakone, 1996, pp. 56–65.
[24] J. Burmeister, Y. Saito, A. Yoshikawa, J. Wiles, Memory performance of master Go players, in: Proceedings of the IJCAI Workshop Using Games as an Experimental Testbed for AI Research, Nagoya, Japan, 1997.
[25] M. Buro, Methods for the evaluation of game positions using examples, Ph.D. Thesis, University of Paderborn, Germany, 1994.
[26] M. Campbell, T. Marsland, A comparison of minimax tree search algorithms, Artificial Intelligence 20 (4) (1983) 347–367.
[27] T. Cazenave, Apprentissage de la résolution de problèmes de vie et de mort au jeu de Go, Rapport du DEA d'Intelligence Artificelle de l'Université Paris 6, 1993.
[28] T. Cazenave, Automatic ordering of predicates by metarules, in: Proceedings of the 5th International Workshop on Metareasoning and Metaprogramming in Logic, Bonn, Germany, 1996.
[29] T. Cazenave, Automatic acquisition of tactical Go rules, in: Proceedings of the Third Game Programming Workshop in Japan, Hakone, 1996, pp. 10–19.
[30] T. Cazenave, Système d'Apprentissage par Auto-Observation. Application au Jeu de Go, Ph.D. Thesis, University Paris 6, 1996, http://www.ai.univ-paris8.fr/~cazenave.
[31] T. Cazenave, R. Moneret, Development and evaluation of strategic plans, in: Proceedings of the Fourth Game Programming Workshop in Japan, Hakone, 1997, pp. 75–79.
[32] T. Cazenave, Metaprogramming forced moves, in: Proc. ECAI-98, Brighton, England, 1998, pp. 645–649.
[33] T. Cazenave, Metaprogramming domain specific metaprograms, in: Proceedings of Meta-Level Architectures and Reflection, Reflection'99, Lecture Notes in Computer Science, Vol. 1616, Springer, Berlin, 1999, pp. 235–249.
[34] T. Cazenave, Generation of patterns with external conditions for the game of Go, in: H.J. van den Herik, B. Monien (Eds.), Advances in Computer Games, Vol. 9, Univ. of Limburg, Maastricht, 2001.
[35] W. Chase, H. Simon, Perception in chess, Cognitive Psychology 4 (1973) 81.
[36] K. Chen, Group identification in computer Go, in: D.N. Levy, D.F. Beal (Eds.), Heuristic Programming in Artificial Intelligence: The First Computer Olympiad, Ellis Horwood, Chichester, 1989, pp. 195–210.
[37] K. Chen, The move decision process of Go intellect, Computer Go 14 (1990) 9–17.
[38] K. Chen, Attack and defense, in: H.J. van den Herik, L.V. Allis (Eds.), Heuristic Programming in Artificial Intelligence, Vol. 3: The Third Computer Olympiad, Ellis Horwood, Chichester, 1992, pp. 146–156.
[39] K. Chen, Z. Chen, Static analysis of life and death in the game of Go, Inform. Sci. 121 (1–2) (1999) 113–134.
[40] K. Chen, Some practical techniques for global search in Go, ICGA J. 23 (2) (2000) 67–74.
[41] Z. Chen, E-mail sent on 5th January 1997 to the Computer Go mailing list, http://www.cs.uoregon.edu/~richard/computer-go/.
[42] Z. Chen, E-mail sent on 11th May 1997 to the Computer Go mailing list, http://www.cs.uoregon.edu/~richard/computer-go/.
[43] J.H. Conway, On Numbers and Games, Academic Press, New York, 1976.
[44] J.H. Conway, E.R. Berlekamp, R.K. Guy, Winning Ways, Academic Press, New York, 1982.
[45] J.C. Culberson, J. Schaeffer, Pattern databases, Computational Intelligence 14 (3) (1998) 318–334.
[46] G. Dejong, R. Mooney, Explanation based learning: An alternative view, Machine Learning 2 (1986).
[47] H. Enderton, The Golem Go program, Carnegie Mellon University, CMU-CS-92-101, Pittsburgh, PA, 1992. Technical Report available at ftp://www.joy.ne.jp/welcome/igs/Go/computer/golem.sh.Z.
[48] M. Enzenberger, The integration of a priori knowledge into a Go playing neural network, 1996, http://www.markus-enzenberger.de/compgo_biblio.html.
[49] K.A. Ericsson, H. Simon, Protocol Analysis, Verbal Reports as Data, MIT Press, Cambridge, MA, 1980.
[50] D. Fotland, The program G2, Computer Go 1 (1986) 10–16.
[51] D. Fotland, Knowledge representation in The Many Faces of Go, 2nd Cannes/Sophia-Antipolis Go Workshop, February 1993, ftp://www.joy.ne.jp/welcome/igs/Go/computer/mfg.Z.
[52] D. Fotland, Computer Go Design Issues, E-mail sent on 1st October 1996 to the Computer Go mailing list, http://www.cs.uoregon.edu/~richard/computer-go/.
[53] D. Fotland, A. Yoshikawa, The 3rd FOST cup world-open computer-go championship, ICCA J. 20 (4) (1997) 276–278.

[54] A. Fraenkel, D. Lichtenstein, Computing a perfect strategy for *n* by *n* Chess requires time exponential, J. Combin. Theory, Serie A 31 (2) (1981) 199–214.
[55] 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.
[56] J. Gallagher, Specialization of logic programs, in: D. Schmidt (Ed.), Proceedings of the ACM SIGPLAN Symposium on PEPM'93, ACM Press, Copenhagen, 1993.
[57] Gnu Go home page, http://www.gnu.org/software/gnugo/devel.html, 1999.
[58] R. Greenblatt, D. Eastlake, S. Croker, The Greenblatt chess program, in: Proceedings of the Fall Joint Computer Conference, 1967, pp. 801–810.
[59] A. Grigoriev, Artificial Intelligence or stochastic relaxation: Simulated annealing challenge, in: D.N.L. Levy, D.F. Beal (Eds.), Heuristic Programming in Artificial Intelligence, Vol. 2, Ellis Horwood, Chichester, 1991, pp. 210–216.
[60] R. Grimbergen, A plausible move generator for Shogi using static evaluation, in: Proceedings of the Fifth Game Programming Workshop in Japan, Hakone, 1999, pp. 9–15.
[61] A. de Groot, Psychological Studies, T4, Thought and Choice in Chess, Mouton, La Hague, 1965.
[62] F. van Harmelen, A. Bundy, Explanation based generalisation = partial evaluation, Artificial Intelligence 36 (1988) 401–412.
[63] H.J. van den Herik, L.V. Allis, I.S. Herschberg, Which games will survive?, in: D.N.L. Levy, D.F. Beal (Eds.), Heuristic Programming in Artificial Intelligence, Vol. 2: The Second Computer Olympiad, Ellis Horwood, Chichester, 1991, pp. 232–243.
[64] R. High, Mathematical Go, Computer Go (1990) 14–24.
[65] S. Hu, Multipurpose adversary planning in the game of Go, Ph.D. Thesis, George Mason University, Fairfax, VA, 1995.
[66] S. Hu, P. Lehner, Multipurpose strategic planning in the game of Go, IEEE Transactions on Pattern Analysis and Machine Intelligence 19 (3) (1997) 1048–1051.
[67] F.H. Hsu, T.S. Anantharaman, M. Campbell, A. Nowatzyk, A grandmaster chess machine, Scientific American 263 (4) (1990).
[68] A. Junghanns, J. Schaeffer, Single-agent search in the presence of deadlocks, in: Proc. AAAI-98, Madison, WI, 1998.
[69] G. Kakinoki, The search algorithm of the Shogi program K3.0, in: H. Matsubara (Ed.), Computer Shogi Progress, Kyoritsu Shuppan Co, Tokyo, 1996, pp. 1–23 (in Japanese).
[70] Y.K. Kao, Sum of hot and tepid combinatorial games, Ph.D. Thesis, University of North Carolina at Charlotte, NC, 1997.
[71] Y.K. Kao, Mean and temperature search for Go endgames, Inform. Sci. 122 (2000) 77–90.
[72] S. Kirkpatrick, C. Gelatt, M. Vecchi, Optimisation by simulated annealing, Science 220 (1983) 671–680.
[73] D.E. Knuth, An analysis of alpha-beta pruning, Artificial Intelligence 6 (4) (1975) 293–326.
[74] T. Kojima, K. Ueda, S. Nagano, A case study on acquisition and refinement of deductive rules based on EBG in an adversary game: How to capture stones in Go, in: Proceedings of the Game Programming Workshop in Japan, 1994, pp. 34–43.
[75] 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.
[76] R. Korf, Depth-first iterative deepening: An optimal admissible tree search, Artificial Intelligence 27 (1985) 97–109.
[77] R. Korf, Finding optimal solutions to Rubik's Cube using pattern databases, in: Proc. AAAI-97, Providence, RI, 1997, pp. 700–705.
[78] R. Lake, J. Schaeffer, P. Lu, Solving large retrograde-analysis problems using a network of workstations, in: H.J. van der Herik, I.S. Herschberg, J.W.H.M. Uiterwyk (Eds.), Advances in Computer Chess, Vol. 7, University of Limburg, Maastricht, Netherlands, 1994, pp. 135–162.
[79] H. Landman, Eyespace values in Go, in: R.J. Nowakowski (Ed.), Games of No Chance, Cambridge University Press, Cambridge, 1996, pp. 227–257.
[80] R. Langston, Perception in Go as a problem in AI, Computer Go 6 (1988).
[81] D. Lefkovitz, A strategic pattern recognition program for the game of Go, University of Pennsylvania, The Moore School of Electrical Engineering, Wright Air Development Division, Technical Note 60-243, 1–92, 1960.

[82] D. Lichtenstein, M. Sipser, Go is polynomial-space hard, J. ACM 27 (2) (1980) 393–401.
[83] D. McAllester, Conspiracy numbers for min-max search, Artificial Intelligence 35 (1988) 287–310.
[84] H. Matsubara, Shogi (Japanese chess) as the AI research target next to chess, Technical Report of the Electrotechnical Laboratories, 93-23, Japan, September 1993.
[85] S. Minton, Constraint-based generalization learning game-playing plans from single examples, in: Proceedings of AAAI-84, Austin, TX, William Kaufmann, Los Altos, CA, 1984, pp. 251–254.
[86] S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, Y. Gil, Explanation-based learning: A problem solving perspective, Artificial Intelligence 40 (1989) 63–118.
[87] S. Minton, Quantitative results concerning the utility of explanation-based learning, Artificial Intelligence 42 (2–3) (1990) 363–391.
[88] T.M. Mitchell, R.M. Keller, S.T. Kedar-Kabelli, Explanation-based generalization: A unifying view, Machine Learning 1 (1) (1986).
[89] M. Müller, Games Theories and Computer Go, Computer Go Workshop, Cannes, 1993.
[90] M. Müller, Computer Go as a sum of local games: An application of combinatorial game theory, Ph.D. Thesis of the Swiss Federal Institute of Technology Zürich, 1995, http://web.cs.ualberta.ca/~mmueller.
[91] M. Müller, L. Gasser, Experiments in computer Go endgames, in: R.J. Nowakowski (Ed.), Games of No Chance, Cambridge University Press, Cambridge, 1996, pp. 273–284.
[92] M. Müller, E. Berlekamp, W. Spight, Generalized thermography: Algorithms, implementation and application to Go endgames, TR-96-030 Berkeley, CA, 1996.
[93] M. Müller, Computer Go: A research agenda, in: H.J. van den Herik, H. Iida (Eds.), Proc. CG-98, Lecture Notes in Computer Science, Vol. 1558, Springer, Heidelberg, 1998, pp. 252–264.
[94] 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.
[95] M. Müller, Race to capture: Analyzing Semeai in Go, in: Proceedings of the Fifth Game Programming Workshop in Japan, Hakone, 1999, pp. 61–68.
[96] M. Müller, Not like other games—Why tree search in Go is different, in: Proceedings of the 5th Joint Conference on Information Sciences, 2000.
[97] R.E. Nisbett, T.D. Wilson, Telling more than we can know: Verbal reports on mental processes, Psychological Review 84 (1977) 231–259.
[98] J. Nunn, Extracting information from endgame databases, ICCA J. (1993) 191–200.
[99] R. Otten, L. van Ginneken, The Simulated Annealing Algorithm, Kluwer Academic, Dordrecht, 1989.
[100] B. Pell, Metagame: A new challenge from games and learning, in: H.J. van den Herik, L.V. Allis (Eds.), Heuristic Programming in Artificial Intelligence, Vol. 3, The Third Computer Olympiad, University of Limburg, Maastricht, 1992, pp. 237–251.
[101] J. Pitrat, A program for learning to play chess, in: Chen (Ed.), Pattern Recognition and Artificial Intelligence, Academic Press, 1976, pp. 399–419.
[102] J. Pitrat, Games: The next challenge, ICCA J. 21 (3) (1998) 147–156.
[103] M. Reiss, E-mail sent in January 1995 to the Computer Go mailing list, http://www.cs.uoregon.edu/~richard/computer-go/.
[104] J. Reitman, Skilled perception in Go: Deducing memory structures from inter-response times, Cognitive Psychology 8 (3) (1976) 336–356.
[105] W. Reitman, B. Wilcox, The structure and performance of the INTERIM.2 Go program, in: Proc. IJCAI-79, Tokyo, Japan, 1979, pp. 711–719. Reprinted in: D.N.L. Levy (Ed.), Computer Games, Vols. 1 and 2, Springer, New York, 1988.
[106] H. Remus, Simulation of a learning machine for playing Go, in: C.M. Popplewell (Ed.), Proceedings of IFIP Congress, North-Holland, Amsterdam, 1963. Also in: Information Processing (1962) 428–432.
[107] P. Ricaud, Gobelin: Une approche pragmatique de l'abstraction appliquée à la modélisation de la stratégie elémentaire au jeu de Go, Ph.D. Thesis, Paris 6 University, 1995.
[108] P. Ricaud, A model of strategy for the game of Go using abstraction mechanisms, in: Proceedings IJCAI-97, Nagoya, Japan, 1997, pp. 678–683.
[109] J.M. Robson, *N* by *N* checkers is exptime complete, TR-CS-82-12, Australian National University, Department of Computer Science, 1982.
[110] J.M. Robson, The Complexity of Go, TR-CS-82-14, Australian National University, Department of Computer Science, 1982. Also in: Proceedings of the IFIP, 1983, pp. 413–417.

[111] J. Ryder, Heuristic analysis of large trees as generated in the game of Go, Ph.D. Thesis, Department of Computer Science, Stanford University, 1971.

[112] Y. Saito, A. Yoshikawa, Do Go players think in words? Interim report of the analysis of Go players' protocol, in: Proceedings of the Second Game Programming Workshop in Japan, Hakone, 1995, pp. 118–127.

[113] Y. Saito, A. Yoshikawa, An analysis of strong Go-players' protocols, in: Proceedings of the Third Game Programming Workshop in Japan, Hakone, 1996, pp. 66–75.

[114] A.L. Samuel, Some studies in machine learning using the game of checkers, IBM J. Res. Develop. 3 (3) (1959).

[115] A.L. Samuel, Some studies in machine learning using the game of checkers II, IBM J. Res. Develop. 11 (6) (1967).

[116] J. Schaeffer, Conspiracy numbers, Artificial Intelligence 33 (1) (1990) 67–84.

[117] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, D. Szafron, A world championship caliber checkers program, Artificial Intelligence 53 (1992) 273–289.

[118] J. Schaeffer, One Jump Ahead—Challenging Human Supremacy in Checkers, Springer, Berlin, 1997.

[119] N. Schraudolph, P. Dayan, T. Sejnowski, Temporal difference learning of position evaluation in the game of Go, Neural Information Processing Systems, Vol. 6, Morgan Kaufmann, San Mateo, CA, 1994.

[120] J. Serra, Image Analysis and Mathematical Morphology, Academic Press, London, 1982.

[121] C.E. Shannon, Programming a computer to play chess, Philosoph. Magazine 41 (1950) 256–275.

[122] W. Spight, Extended thermography for multiple Kos in Go, in: H.J. van den Herik, H. Iida (Eds.), First International Conference on Computer and Games 98, Lecture Notes in Computer Science, Vol. 1558, Springer, Berlin, 1998, pp. 232–251.

[123] L. Stiller, Multilinear algebra and chess endgames, in: R.J. Nowakowski (Ed.), Games of No Chance, MSRI Publication, Vol. 29, Cambridge University Press, Cambridge, 1996.

[124] G. Stockman, A minimax algorithm better than alpha-beta?, Artificial Intelligence 12 (1979) 179–196.

[125] R. Sutton, Learning to predict by the method of temporal differences, Machine Learning 3 (1988) 9–44.

[126] H. Tamaki, T. Sato, Unfold/fold transformations of logic programs, in: Proceedings of the 2nd Internat. Logic Programming Conference, Uppsala University, 1984.

[127] G. Tesauro, T.J. Sejnowski, A parallel network that learns to play Backgammon, Artificial Intelligence 39 (1989) 357–390.

[128] G. Tesauro, Practical issues in temporal difference learning, Machine Learning 8 (1992) 257–277.

[129] G. Tesauro, TD-Gammon, a self teaching backgammon program, achieves master-level play, Neural Comput. 6 (2) (1994) 215–219.

[130] G. Tesauro, Temporal difference learning and TD-Gammon, Comm. ACM 38 (1995) 58–68.

[131] K. Thompson, Retrograde analysis of certain endgames, ICCA J. 9 (3) (1986) 131–139.

[132] K. Thompson, 6-piece endgames, ICCA J. (1996) 215–226.

[133] E. Thorpe, W. Walden, A partial analysis of Go, Computer J. 7 (3) (1964) 203–207.

[134] E. Thorpe, W. Walden, A computer assisted study of Go on $M \times N$ boards, Inform. Sci. 4 (1972) 1–33.

[135] P. Vermersch, Les connaissances non conscientes de l'homme au travail, Le Journal des Psychologues 84 (1991).

[136] J. von Neumann, O. Morgenstern, Theory of Games and Economic Behavior, Princeton Univ. Press, Princeton, NJ, 1944.

[137] B. Wilcox, Computer Go, American Go J. 13 (4,5,6) (1978), 14 (1,5,6) (1979), 19 (1984). Reprinted in: D.N.L. Levy (Ed.), Computer Games, Vols. 1 and 2, Springer, New York, 1988.

[138] P. Woitke, New ladder participant, E-mail sent on 11th March 1996 to the Computer Go mailing list, http://www.cs.uoregon.edu/~richard/computer-go/.

[139] P. Woitke, Computer Go summary, E-mail sent on 5th October 1996 to the Computer Go mailing list, http://www.cs.uoregon.edu/~richard/computer-go/.

[140] T. Wolf, The program GoTools and its computer-generated tsume-go database, in: Proceedings of the First Game Programming Workshop in Japan, Hakone, 1994, p. 84.

[141] T. Wolf, About problems in generalizing a tsumego program to open positions, in: Proceedings of the Third Game Programming Workshop in Japan, Hakone, 1996, pp. 20–26.

[142] T. Wolf, Forward pruning and other heuristic search techniques in tsume go, Inform. Sci. 122 (2000) 59–76.

[143] H. Yamashita, Half extension algorithm, in: Proceedings of the Fourth Game Programming Workshop in Japan, Hakone, 1997, pp. 46–54.

[144] H. Yamashita, YSS: About its datastructures and algorithm, in: H. Matsubara (Ed.), Computer Shogi Progress 2, Kyoritsu Shuppan Co, Tokyo, 1998, pp. 112–142 (in Japanese).

[145] A. Yoshikawa, Y. Saito, The difference of the knowledge for solving Tsume-Go problem according to the skill, in: Proceedings of the Fourth Game Programming Workshop in Japan, Hakone, 1997, pp. 87–95.

[146] 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.), Proceedings of the First International Conference on Computer and Games 98, Lecture Notes in Computer Science, Vol. 1558, Springer, Berlin, 1998, pp. 282–299.

[147] A. Zobrist, A model of visual organization for the game of Go, in: Proc. AFIPS Spring Joint Computer Conference, Boston, AFIPS Press, Montvale, NJ, 1969, pp. 103–111.

[148] A. Zobrist, A new hashing method with application for game playing, Technical Report 88, University of Wisconsin, April 1970. Reprinted in: ICCA J. 13 (2) (1990) 69–73.

[149] A. Zobrist, Feature extractions and representation for pattern recognition and the game of Go, Ph.D. Thesis, Graduate School of the University of Wisconsin, Madison, WI, 1970.