Game Theories and Computer Go

Martin Müller

Informatik, ETH Zürich

> The research of Go programs is still in its infancy, but we shall see that to bring Go programs to a level comparable with current Chess programs, investigations of a totally different kind than used in computer chess are needed. John McCarthy 1990

Abstract

Two kinds of theories have traditionally influenced computer game playing: Classical game theory, with its minimax principle, and specialized theories developed for a particular game. In addition to these, combinatorial game theory promises to be useful for computer Go. We propose a model for Go based on this theory and discuss our preliminary implementation. We claim that future Go programs will need powerful theories as well as lots of computing power.

Organisation of the Paper

Section 1: Review of theories used for computer game playing. We discuss classical game theory, game-specific theories and their relation to each other.

Section 2: Brief introduction to the concepts of combinatorial game theory we need for our application to Go. We discuss sums of games and several algorithms for evaluating them.

Section 3: The relation between computer Go and game theories. We survey the current state of the art, taking leading Go programs as examples.

Section 4: Our framework for applying combinatorial game theory to Computer Go. We define local Go games, map them to abstract mathematical games and show how to play Go as a sum of local Go games.

Section 5: Implementation issues of our model and differences to 'standard' Go programs.

Section 6: Limitations of our basic model and extensions designed to overcome these limitations.

Section 7: Summary of our experiences and remaining research problems.

1. Game Theories and Computer Game Playing

Note: in this paper the term game is used exclusively for finite two person games with perfect information, such as Go, Chess, Checkers, Othello, Nine Men's Morris, Shogi, Awari, and many abstract mathematical games. Not included are games of chance, imperfect information, or more than two players, such as Bridge, Scrabble, Blackjack or Poker.

Classical Game Theory

Classical Game Theory was pioneered by von Neumann and others, who formalized and proved concepts game players had been applying for centuries [Neumann 28, Neumann/Morgenstern 44]. In von Neumann's model of two-person games, players alternate moves until they reach a terminal position. At a terminal position, the game value is determined by an evaluation function. The minimax rule can be used to obtain the optimal values for nonterminal positions.

Forward and backward search are practical applications of this theory. Forward search approximates a full game tree by a tree expanded up to some horizon, using a heuristic evaluation function to evaluate the leaves of this tree as if they were real terminal nodes. This tree is evaluated by a minimax procedure.

Exhaustive backward search, or retrograde analysis, constructs huge databases containing information about *all* positions of a game or subgame. Starting with the known values of all terminal positions, retrograde analysis computes the values of all other positions using the minimax rule. A forward searching program can then use a database containing all values from a subgame as if the positions in this subgame were terminal. This effectively reduces the size of the search space.

Both forward and backward search have proven successful in computer game playing. Many refinements have made forward search more practicable, such as alphabeta pruning, transposition tables, parallel search, and selective search expansion. Exhaustive forward searching is currently feasible to a depth of 10 ply in chess, and 20 ply in Checkers and Awari.

Many interesting databases have been computed by exhaustive backward search. In chess, all endgames up to 5 pieces [Thompson 86] and some with 6 pieces [Stiller 89] are solved. Databases exist for up to 8 pieces in Checkers [Schaeffer 92], up to 16 pieces in Nine Men's Morris [Gasser 92] and up to 20 stones in Awari [Gasser 92]. The biggest databases contain values of several billion positions.

Game-specific Theories

Many specialized theories have been developed for solving a particular game, or a subgame such as King, Bishop and Knight against King, or 'KBNK', in chess.

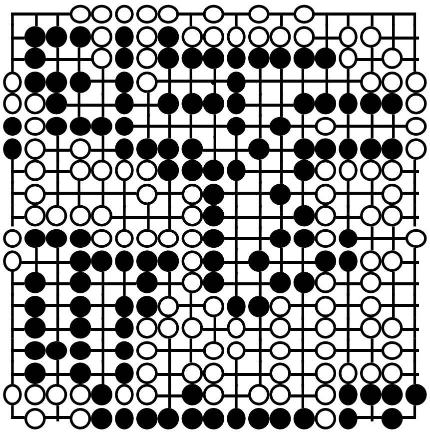
The purpose of such theories is to reduce the search space, both by evaluating nonterminal positions and by supplying pruning rules which reduce the number of moves to be investigated in a forward search. Ideally a theory can solve a game by pure analysis, without any search. Another method applicable in some games is a decomposition into smaller parts which can then be analysed independently.

A theory can be viewed as a compressed database: rules take much less space to implement than exhaustive tables. Successful applications of specialized theories in conjunction with forward and backward search techniques are the solutions to 4-in-a-row and 5-in-a-row by Victor Allis [Allis 88, 92].

Beyond strict theory, there is heuristics. Examples are formalized rules of thumb, or uses of a theory outside the space of guaranteed validity. A vast number of heuristics have been developed for the evaluation of game positions, and for pruning moves during selective search. Still other heuristic rules help decide when to apply heuristic evaluations: *Quiescence search* is used to avoid evaluation of turbulent positions where heuristic evaluations often go wrong.

2. Combinatorial Game Theory

Combinatorial game theory has been developed and applied to many mathematical games by Conway, Berlekamp and Guy among others [Conway 76, Berlekamp/ Conway/ Guy 82]. This theory gives a very elegant mathematical definition of games that is more general than von Neumann's: It removes the condition that players must play alternately. Combinatorial game theory contains minimax theory as a special case. A main topic of the theory is the decomposition of games into independent subgames. This leads to the notion of a sum of games.



Berlekamp's 9-Dan Stumping Problem

Recently, Berlekamp and his students have successfully applied combinatorial game theory to Go Endgames [Berlekamp 91]. Anyone who has studied the theory can now beat professional Go players in carefully designed endgame positions containing lots of similar looking, relatively simple independent local games. We are not aware of applications to traditional board games other than Go. Most games do not decompose into sums.

Basic Definitions and Notation

For an introduction to combinatorial game theory, see [Conway 76], [Berlekamp/Conway/Guy 82], [Berlekamp 91] or [High 92]. Note that we use the terms game, mathematical game, game position and position interchangeably. They all mean the same thing here.

A *game* is played by two players called *Left* (Black) and *Right* (White). It is defined by its left and right *options*, i.e. the game positions to which Left and Right can move.

$$G = \{ L_1,\ldots,L_n \mid R_1,\ldots,R_m \}$$

$$\text{or} \quad G = \{ G_L \mid G_R \}$$

This definition is recursive: L_i and R_j are again games. G_L and G_R denote sets of games here.

The *inverse game* -G is constructed by swapping colors. Its definition is

$$-G = \{ -G_R \mid -G_L \}$$

An integer *number n* can be represented as the game where Left can move *n* times whereas Right has no move, e.g. $0 = \{|\}$, $1 = \{0|\}$, $n+1 = \{n|\}$. Negative numbers are the inverses of positive numbers, e.g.

$$-2 = -(2) = -\{1|\} = \{|-1\} = \ldots = \{|\{|0\}\}$$

There are games corresponding to fractional numbers too, such as $1/2 = \{0|1\}$ and $1/4 = \{0|1/2\}$.

The *Sum* of games *G* and *H* is defined as:

$$G + H = \{ G + H_L, G_L + H \mid G + H_R, G_R + H \}$$

A move in a sum game *G+H* is therefore a move in either *G* or *H*. Games and sums as defined above have the mathematical structure of a group.

Loopy games: It is possible to define a game in terms of itself, as in $G = \{G|G\}$. Most of the theory only applies to *loopfree* games. In Go, loopy games appear only in

Ko fights. We will assume all games are loopfree from now on, except for section 6 when we discuss Ko fights.

Notation: In a sum game $G = A+B+C+\ldots$, the single games $A, B, C,\ldots$ are called *subgames* of G.

To abbreviate notation of games, curly braces can be omitted, and precedence indicated by multiple slashes.

Example: $\{3 \mid \{2 \mid 1\}\} = 3 \mid \{2 \mid 1\} = 3 \,||\, 2 \mid 1$

Some Useful Operations and Functions on Mathematical Games

Comparing Games

We can define the following partial order on games: $G \geq H$ if Left can do at least as well in G as in H under any circumstances, i.e. any sum $G + Rest$ is at least as good for Left as $H + Rest$. This partial order can be used for pruning options of a game.

Dominated and Reversible moves

There are two rules for simplifying a game tree:

1.) Remove dominated moves: If $G = \{A,B,C..|..\}$, and $A \geq B$, then A dominates B, and B can be omitted: $G = \{A,C,\ldots|\ldots\}$

2.) Reverse reversible moves: If $G = \{A,B,C..|..\}$, and A has a right option $A_R \leq G$, then Lefts move to A reverses through Right's move to A_R: If $A_R = \{A_1,A_2,A_3\ldots|\ldots\}$, then $G = \{A_1,A_2,A_3,\ldots,B,C,\ldots|\ldots\}$

Analogous definitions hold for Right's moves.

Applying these rules as often as possible leads to a unique *normal form*..

Cooling

Cooling transforms a game into a simpler one by adding a 'tax' on every move. It is a homomorphism used to simplify a game while retaining much of its structure. Cooling reduces the *temperature* of a 'hot' game. It does not change the *mean value*. Cooling by 1 is an 'almost' isomorphic mapping for Go-like loopfree games [Berlekamp 91].

Temperature

Temperature is an estimate how urgent it is to make a move in a game. It is defined as the smallest *cooling* value such that the *Leftscore* and the *Rightscore* of the cooled game are the same. Cooling a game by more than its temperature yields a number, the *mean value* of the game.

Mean Value

A measure how many points a game is worth on average. This must not be confused with the 'value of moving in a game', which cannot be expressed as a number in general. The mean value is linear: $mean(A+B) = mean(A) + mean(B)$

Incentive

The improvement made by moving in a game, defined as the difference between a game and an option of this game. Incentives are games, not numbers. They are defined to be non-negative (for games in canonical form):

$$Left\ incentive = G_L - G$$

$$Right\ incentive = G\text{-}G_R$$

Leftscore and Rightscore

Leftscore and Rightscore provide some connection to classical game theory: They are the minimax values of a game when players move alternately and Left (Right) plays first.

Switching Games

Switching games are the simplest 'hot' games, with a *temperature* > 0. A *switch* $a|b$, with $a>b$, has mean value $(a+b)/2$ and temperature $(a\text{-}b)/2$. One can play a sum of switches optimally by moving in the switch with highest temperature. This strategy does *not* always work for sums of other games more complicated than switches.

Sente and Gote

The term *sente* is used if a move poses a threat big enough that it must be answered in the same subgame. A *gote* move can be ignored. For example, the switch $n|0$ is a n-point gote move for both players. It is important to understand that sente and gote are relative terms and depend on the size of the threat behind a move. Theory gives a more precise meaning to such notions and can explain some rules of thumb concerning the value of such moves. In the Go literature, special phrases are used for moves that are sente for both players, take away an opponent's sente move, or have a continuation that is sente.

Example: In the game $BIG|n||0$, Left can get n points 'for free' by moving to $BIG|n$ because Right must prevent Left's move to BIG. A Right *gyaku yose* move to 0 is said to be almost twice as big as moving in a simple switch $n|0$. Combinatorial theory confirms this rule of thumb. For example, the temperature of $10|0$ is 5, but rises to 15/2 for the game $20|10||0$ and to 10 for $BIG|10||0,\ BIG \geq 30$.

Good and Optimal Play in Sum Games

Move decision in a sum game proceeds in two stages: First we must select the subgame which contains the move to be played, then choose a move from all options in this game. We can distinguish *global* algorithms that need to compute the sum game to make a move decision from *local* algorithms that do not need the sum.

Evaluation of Sum Games

Before attempting any evaluation, it is a good idea to bring all local games into canonical form, then *cool* them by 1. This simplifies the games considerably and does not change the outcome of the computation.

For simple sums, a brute force evaluation is possible: compute the sum, determine the Leftscore (or Rightscore) of the sum and try all possible moves to find one which retains this score. This method is already better than a brute force global search because the local games have been simplified, and some more pruning may take place during computation of he sum. Still, direct computation of the sum quickly becomes too complex for realistic subgames.
Example: The canonical form of $1|0 + 2|0 + 3|0 + 4|0 + 5|0$ is

$$\{15|14||13|12|||12|11||10|9\}|\{11|10||9|8|||8|7||6|5\}||\{10|9||8|7|||7|6||5|4\}|\{6|5||4|3|||3|2||1|0\}$$

Often we can play perfectly without computing the sum by proving that a move is optimal. *Incentives* of moves can be computed locally: If $G = A+B+C+\ldots$ and we move from A to A_L, the Left incentive becomes $G_L\text{-}G=A_L+B+C+\ldots\text{-}(A+B+C+\ldots) = A_L\text{-}A$. The incentive depends only on the subgame A containing the move. It is easy to prove that we may prune all moves with dominated incentives. Often, only one candidate move will remain.

Otherwise, the optimal move may be hard to find. A simple approximate strategy is to play in a hottest subgame. A better algorithm is Thermostrat [Berlekamp/Conway/Guy 82], [Berlekamp 92]. This algorithm bounds the maximum error by the temperature of the hottest subgame. It is a local algorithm that only needs a relatively simple data structure, the *thermograph* of each subgame.

3. Applications of Theories to Go Programs

State of Computer Go

With the single exception of chess, more programming effort must have been spent on Go than on any other game. But in playing strength measured on a human scale, programs for Go lag far behind their counterparts in Checkers, Awari or Nine Men's Morris. Reasons for this sad state of affairs can be found both in the inherent difficulty of the game and in the limitations of existing programs.

Some facts that make Go a difficult Game

Size and Structure of the Problem Space

The size of the search space for 19x19 Go is probably the biggest of all popular games [Allis et al. 91]. No simple yet reasonable evaluation function exists. This is evident to all serious Go players, and confirmed by the fact that many difficult combinatorial problems can be formulated as Go problems [Lichtenstein/Sipser 78], [Morris 81], [Robson 83].

A Go position can be highly structured, making perfect play possible with very little search and a good theory, as in Berlekamp's problems. But it can also be very chaotic, leaving no alternatives to exhaustive analysis. Therefore a good Go program will need both good theories and lots of computing power.

Quality and Quantity of Human Knowledge

Humans are able to recognize the very subtle differences in Go positions that have big effects later. Professionals know by intuition if a group can be killed or not. Strictly prove this however would take billions of nodes. They know which side is better in a game after a quick glance at the position, a feat that would involve enormous amount of processing on a computer, if it were feasible at all. We can contrast this with a more computer friendly games such as Nine Men's Morris and some chess endgames: Humans get lost in the 'combinatorial chaos' of the game, and machines can use their computing power well [Gasser 91], [Thompson 86].

A lot of Go knowledge has accumulated over centuries. Most of it is implicit in form of expert games. Still, thousands of tutorial books have been written. Pattern knowledge of experts seems at least comparable to that of chess experts, which is daunting enough [De Groot 65]. Patterns recognized by humans are more than just stones and empty spaces: Players intuitively see liberties, or lack of liberties, and fuzzy concepts such as light and heavy stones. This visual nature of the game is easy for human perception but hard to model. The cognitive models developed by Reitman and Wilcox [Wilcox 79] were interesting, but today's programs only use relatively simple pattern matching [Boon 90].

Components of Current Go Programs

A number of standard components of Go programs have evolved over the past 20 years:

- Basic data structures for blocks, groups of stones, and surrounded territories

- Pattern matching module to generate moves, or data structures such as connections
- Some influence function may be used to determine groups and territory
- Connections and boundaries between blocks are defined using patterns or influence
- Move generation and evaluation mainly by static analysis. Multiple evaluations (different motives) for each move are possible. Selection of the best move is done by a linear combination of different evaluations, or by a priority scheme of motives.

Applications of Minimax Search in Computer Go

Global Search

Global search, the move selection method of choice for most game playing programs, runs into a lot of problems in Computer Go. Brute force search leads to combinatorial explosion very soon, and selective search strategies exhibit very irritating behaviour.

When there are several 'hot' local fights in one position, global search tends to switch back and forth between them. Eventually search must stop while some fights are still hot, therefore hot positions must be evaluated. The depth needed to search several fights simultaneously is the sum of all locally needed depths. This leads to an exploding size of the search tree. Tools such as transposition tables cannot solve this fundamental problem. Global search in Go therefore often results in a bad quality of decisions.

Our later case studies of Go programs will show that several attempts at global search have degenerated into specialized searches in one way or another.

Goal-oriented Searches

Most programs use tactical search to classify the status of blocks as dead (is captured, cannot escape), threatened (can be captured, or escape), or safe (cannot be captured tactically). The simplest searches are *ladders*. Move generators for general tactical searches are restricted to moves close to the block and its neighbor blocks.

Similarly, many programs perform a Life-and-Death search for whole groups of several loosely connected blocks of stones. Wolf has built a strong specialized Life-and-Death program [Wolf 91].

Applications of Combinatorial Game Theory

An early attempt to use sums of games for Go endgames is [Miller 76]. No playing program seems to have resulted from this research. The only application of combinatorial game theory in a Go program we are aware of is our program for small endgame positions [Müller/Gasser 92].

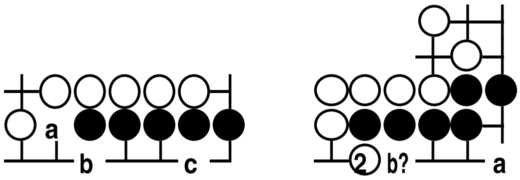
Applications of Special Go Theories

Much Go knowledge is available in form of proverbs. Most of this knowledge is heuristic, such as "Five groups live, the sixth will die" or "Pon-nuki is worth thirty points". Other proverbs are strictly valid under suitable circumstances, for example "On the second line, 8 live, 6 die". Of course the circumstances are not given in the proverb.

Pattern Knowledge

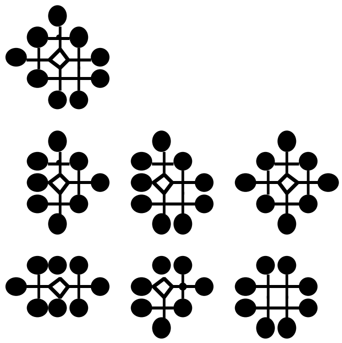
A lot of Go knowledge is encoded as local patterns of stones. Some of these patterns deal with informal concepts such as good shape and thickness, but many patterns have a rigorous meaning. Examples are patterns describing eyes, sure connections, or capturing moves. The expressive power of patterns can be enhanced by adding conditions on stones in the pattern, such as a liberty count [Boon 90].

Prototypical Life and Death problems that often occur in the corners and on the edge of the board can be put into a pattern library. Such specialized patterns greatly speed up Life and Death searches. Still, the constraints for applying such a pattern must be designed very carefully.



Standard corner Life and Death situations

Pattern knowledge is more powerful if it is exhaustive for some subproblem: One can use the fact that no pattern is applicable, too. An example are small surrounded areas that form one eye and might be divided to form two eyes, the so-called *nakade* shapes.



Nakade shapes

A nakade shape is valid only if all surrounding blocks of stones are 'stable enough', i.e. it is not necessary to connect these blocks inside the area.

This list of nakade shapes is exhaustive: Smaller areas (1 or 2 points) form only one eye, all other shapes can form two eyes even if the opponent moves first[1].

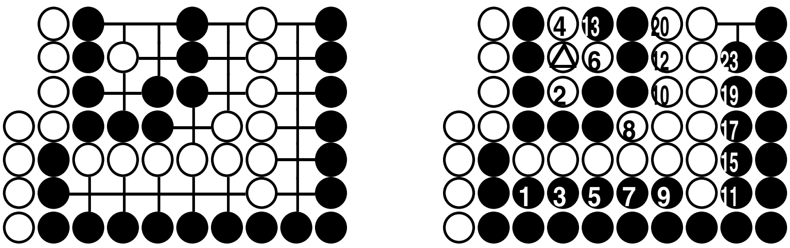
In case of a capturing race, a nakade shape gives additional liberties: For $n>1$, $0 \le m<n$, a n point nakade shape filled by m opponent stones is equivalent to $(n^2-3n)/2+3-m$ outside liberties.

Example: The Semeai Formula

This traditional formula [Lenz 82] gives a static evaluation of a class of simple *semeai* (capturing races) where:

- Each player has one threatened block, the 'target', with 0 or 1 eye. An eye may be a *nakade* shape with additional opponent stones inside, which are counted as outside liberties as explained above.
- Both targets have separate outside liberties and common liberties
- Each target has an eye status: no eye, small eye (less than 3 points) or big eye (3 or more points).

[1]All statements remain valid if some surrounding stones are replaced by the edge of the board, with one famous exception: The 'bent four in the corner' position.



A semeai, and demonstration that Black can win it

14 at Δ, 16 at 2, 18 at 4, 21 at 6, 22 at Δ).

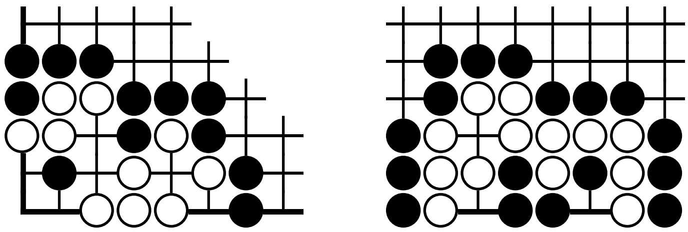
Example: Counting liberties, Black looks absolutely lost with 8 liberties against White's 15. Applying the formula however, we find that both players can win going first. Using tree search for this kind of problem would be very expensive. Assuming each player plays only on opponent's liberties, a problem with *n* liberties for each player has $n!^2$ states. Using a hash table it still has 2^{n+1} states.

Results of simple semeai evaluated by the semeai formula can easily be mapped to mathematical games.

Definitions of Life

Benson has found a mathematical description of *unconditionally alive* blocks [Benson 80]. These blocks cannot be captured even if the player never makes a move and the opponent moves as often as he likes. Popma and Allis studied the restriction where the opponent may move *n* times in a row [Popma/Allis 92]. Their concepts are applicable to Ko fights.

Of more practical interest are algorithms for detecting groups of blocks which are safe with the usual alternating play. Thomas Wolf implemented such an algorithm for his Life and Death program, but published no details [Wolf 91].



White stones recognized as dead (left), and alive (right), by Wolf's static evaluation

Several classes of living groups are introduced in [Müller 89]. Most Go programs contain some Life and Death knowledge, often as a mixture of exact and heuristical rules [Kraszek 88].

Pure Life and Death algorithms have limited usefulness in go playing programs because they rely on the assumption that the target is completely surrounded by safe opponent stones.

Wolfe's Algorithm for Dependent Endgames

David Wolfe has developed a formula for evaluating certain sets of dependent endgames [Wolfe 91a]. This interesting application of combinatorial game theory will be discussed in section 6.

Current Computer Go Programs

Search Methods Used by Current Go Programs

Go Intellect, by Ken Chen, uses a very detailed model of blocks and groups, a complex evaluation function, plus some specialized tactical and Life and Death searches. A very restricted global search provides input for the move selection procedure, but it is not the only criterion used.

Go 4.3, by Michael Reiss, deviates the furthest from the emerging 'standard model'. Reiss claims that it is easier to build a good evaluation function than a good selective move generator. His move selection procedure only acts as a rough filter that selects many moves (typically 30-50). Go 4.3 then does a one ply lookahead on all these moves and chooses the one with the best resulting position. All moves can be evaluated in parallel.

The search used in Goliath, by Mark Boon, is closest in spirit to the approach presented in this paper: In terms of combinatorial theory, it models a Go position as a sum of switches. Goliath identifies goals, such as cutting, attacking a group, or invading a territory. For each goal and each player going first it performs a goal-directed minmax search. The two scores for each goal form a switch. The local move with greatest score difference, i.e. the hottest switch, is played. There are some adjustments to increase the value of *sente* moves.

Go Intellect and Goliath both play some 'obvious' moves instantly without search, such as *joseki* and some tactical patterns.

Summary: Go Skills Present, and Missing, in Go Programs

From games against humans and other programs, we can infer the following remarks hold for all existing Go programs:

- They have knowledge about basic attributes of Go positions, such as blocks, groups, patterns, joseki, simple tactics, territory, Life+Death
- They rely a lot on heuristics which are incomplete, sometimes contradictory
- There is no clearly demonstrated correlation between computing power and playing strength, as exists in chess [Thompson 82]. Of course playing strength is hurt if computing power sinks below some minimum. Some commercial programs implement different levels of skill by simply switching off some components of their evaluation. No correlation between the amount of search and playing strength has been demonstrated.
- They still lack many concepts: recognize or generate double threats, consistency of moves.

4. Computer Go and Combinatorial Theory

Stimulated by the success of Mathematical Go Theory for the late endgame we began to investigate possible applications to Computer Go in 1992 [Müller/Gasser 92].

As a first step, an interface between Wolfe's toolkit for mathematical games and our Go program Explorer was implemented as a term project [Fierz 92]. Using this interface we built a special program that could recognize and solve sums of endgame positions such as *node rooms* [Wolf 91a]. In 'strict mode', we used Benson's algorithm to prove independence of subgames. A 'heuristic mode' allowed analysis of more endgame positions where independence could not be proved.

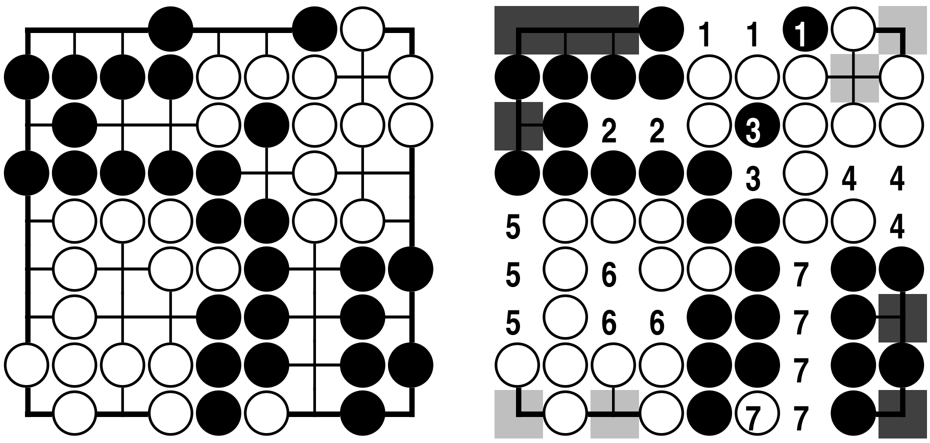
To solve bigger problems, we implemented enhancements such as local hash tables, decomposition of games during search and strict pruning rules. The final version could solve problems with local areas up to size 13. Currently we are interested in the more ambitious goal of using combinatorial game theory throughout a full game of Go.

Why Combinatorial Game Theory is Applicable to Go

As discussed above we cannot expect to succeed with global search in Go. But search has proven a very powerful method for other games, so we would like to use it as much as possible. Our feeling is that current programs fail to exploit the full power of search. Still the success of three time's world champion Goliath indicates that there is promise in localized deep searches. We propose to model a Go position as a sum of many relatively small local search trees, then combine them not by brute force or 'switch' models but using combinatorial game theory.

The Sum Game Model in Computer Go

To apply the theory, we must define a mapping of a Go position to a sum of games. As a first step we must identify subgames. This is done by a *board partition* algorithm that identifies boundaries, territories, potential territories, connections, groups, endgame areas, and open areas. The board partition algorithm decides which structures on the board are independent, or nearly independent, from each other. The result is a set of *local Go games*.



A Go position and its partition into safe territories and independent areas

Each local game is then analysed independently by performing a local tree search. This search allows for several consecutive moves by the same player. Terminal positions are evaluated locally, and the values are backed up in the tree, resulting in a mathematical game that forms an evaluation of the root.

The full board position is modelled as a sum of these evaluations of local games. A sum game evaluation algorithm such as thermostrat is used to find a globally good or optimal move.

Local Go Games

A *local Go game* in a given position p is defined as:

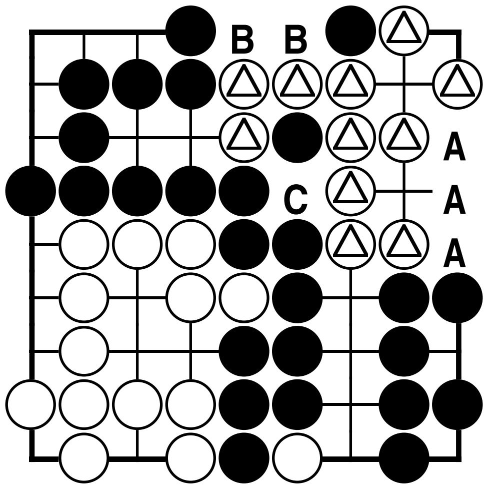
• The *local area*, a subset of points on the board

• A *move tree*, with an *initial local position* (p restricted to *local area*). All moves must lie inside *local area.*

• A numerical *evaluation* of all terminal nodes in the move tree.

Dependencies of Local Go Games

To assure the desired independence of subgames we must prove that a move in one subgame cannot affect evaluation of another subgame. Dependencies between two areas can occur when they share weak boundaries. A move in one game can set up a capture in another game, or even remove the boundary block between the two subgames. Often, some dependencies do exist but have no influence on the game.



Berlekamp's Endgame Problem

Example: This is another endgame problem by Berlekamp, from which our previous example was derived. The main difference is a stone at the uppermost A point. The marked stones are not completely safe because if Black gets all six moves in endgames *A, B* and *C* the white stones will die. This relation does not affect the game value in this case, i.e. the value of the upper right corner is equal to the sum of local games $A+B+C$.

More information on dependencies will follow in the extensions in chapter 6.

How to Find Local Go Games

We will determine which blocks are safe by using an algorithm based on Benson's. These blocks subdivide the rest of the board into independent areas. We can sacrifice accuracy for better board partition by using relatively safe blocks and other probable border lines as boundaries, too.

Surrounded areas can be classified as territory, potential territory, open area. Move generation can be restricted for potential territories, and is not necessary at all for safe territories.

5. Design of a Go Program Using Combinatorial Game Theory

In this section we discuss suitable representations for local Go games and local search. We point out differences to the requirements of normal Go programs.

Data Structures for Local Go Games

We found the following data structures necessary or desirable:

- The local area and all local data such as blocks and territory must be kept.
- The mathematical game evaluation must be built. This can be done in depth first fashion while building the local search tree.
- Moves from the root position must be stored with the options of the mathematical game: In case a move to game G_L from the root node with value G is selected, it must be possible to retrieve the move leading to G_L.
- It is probably a good idea to keep the whole local tree in memory. Construction of the mathematical game can then be done in a separate tree traversal after

expansion is complete. Local trees can be reused during a game if the local area is unchanged. Iterative tree expansion algorithms can work on such a stored tree.

Local Search

In contrast to global search, local search can use special properties of the local area to guide search.

• When the local area is small (less than 10 empty points), exhaustive search is possible [Müller/Gasser 92].

• When specialized theories are applicable, drastic restrictions of the search space may be possible by strong pruning of moves and early static evaluation (compare with the *semeai* example above).

• When heuristics propose a dominant local theme (e.g. a weak group) move generation can be restricted.

• If no special case applies, other heuristics must be used to guide search. This entails all the usual problems of selective search.

Move Generation

There are some differences to move generation for minimax evaluation: we must generate moves for both colors at each node. Furthermore we must distribute computing time over all local games. Clearly some games will be more important, or more complicated, so they need more search.

Pruning rules determine which local move will not be generated. Examples are *dame* moves and moves dominated by other moves [Müller/Gasser 92]. Other rules may be to use only the *n* best moves for each color at a level, with *n* depending on the depth or temperature of the node. It may be possible to implement *sente cutoffs*, i.e. generate only one artificial very high penalty value for not answering a proposed sente move.

Termination rules decide when a node should not be further expanded. The simplest case is when the pruning rules leave no legal moves.

Otherwise we can use a static temperature estimate calculated from the number of unstable points inside the local area. Quiet nodes below a given temperature bound will not be expanded further. This strategy is well suited for an iterative deepening approach using a progressively smaller temperature bound.

Position Evaluation

Evaluation of terminal nodes must map the local area to a number, a count how many points belong to either color. This is easy in real terminal position, where safe and dead stones, territories and neutral points are known exactly.

If termination rules force evaluation of a nonterminal node, we must use heuristic evaluation. To give meaningful results, the evaluation error should lie below the static temperature estimate.

Implementation of Combinatorial Game Theory

Wolfe's toolkit [Wolfe 91b] implements mathematical games with many utility functions and operations such as adding games, converting numbers to games, building a game from its options, reduction to normal form, cooling, computing the temperature, Leftscores and Rightscores and comparing games. It provides conversion of games to text strings and vice versa. Our port to the Macintosh and Smart Game Board [Fierz 92] adds menu commands for accessing the toolkit, storing mathematical games in a Smart Game Board node and converting an evaluated game tree into a mathematical game.

Adapting Explorer to Use the Sum Game Model

Recently we have started to integrate our sum game model into the Go program Explorer [Chen et al. 90]. We found the conversion to be easy in some parts and hard in others.

Easy tasks were:

- To interface Wolfe's toolkit with Explorer
- To implement several sum evaluation strategies
- To replace the existing move decision procedure with a sum game engine

More difficult tasks are:

- Tune the existing board partition algorithm for the needs of sum games
- 'Localize' existing algorithms, i.e. remove references to global (whole board) properties as much as possible.
- Re-use local results from previous moves: store partially expanded local trees, scan and expand old trees in the new global context, garbage collection of irrelevant local trees

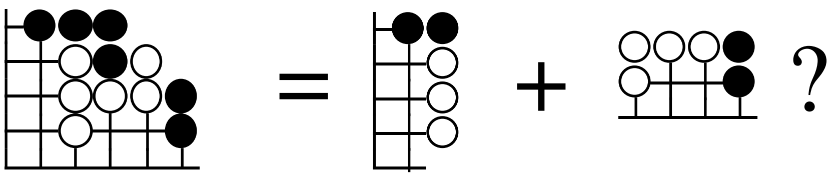
As a preliminary conclusion some non trivial adaptations of Explorer are necessary, but it is certainly better to use it than to rewrite a Go program from scratch. Many data structures are local by nature anyway. Also, the transition can be done step by step with a slight abuse of the sum game model: an old move generator and evaluator can be used to generate a switch game value|0 that approximates the value of a move.

6. Extensions of the Basic Sum Game Model

In this section we discuss various problems with a simple sum game model and how they might be solved.

Problem: With Strict Independence Rules, Local Areas Become Too Big

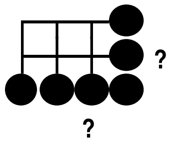
Before the very late endgame, few local games are really independent from the rest of the board. It is hard to develop proper heuristics when games should be considered independent. A frequent case is a territory that can be reduced from several sides. The question is if these reductions can be treated as independent.



One game, or two?

External Dependencies

One way to get better board partition is to allow explicit external dependencies of local areas, such as a number of external liberties of blocks, outside eye space, or connections. Using such a model allows stronger partition, but it is more difficult to recognize and maintain the dependencies. After each change of outside dependencies, a local area must be recomputed.

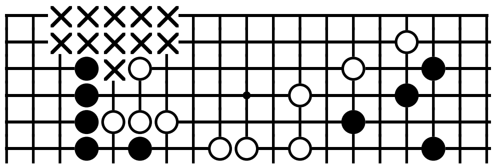


6 Points in the Corner

Example: The evaluation of this 6 point corner area depends only on the number of outside liberties and eyes of the black stones, but not on their exact location on the board.

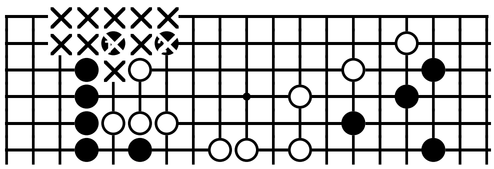
Expanding the Local Area During Search

When using a heuristical board partition algorithm, search may step across the boundaries of the local area.



An open-ended local area

In the example above, board partition might generate a local area consisting of the marked points. The problem is that this area is open toward adjacent black and white territory, and search eventually expands into that territory. If such an open border of a local area is reached during search we must extend the area in that direction.

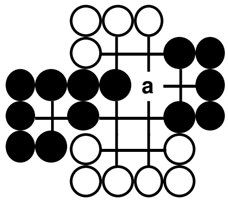


Search extends into neighbor territory

This extension introduces several problems: A basic question is where to limit the expansion of the area. We do not want our search to run around the whole board. One possible solution is to set a maximal area at the very beginning. A problem for evaluation is that it takes place at different point sets now. This must be adjusted too. Another problem is that new unexpected dependencies may show up during search, when areas begin to overlap. More research is needed to find workable solutions to this board partition problem.

Splitting a Local Game

Moves executed during local search often cause more board partition. Therefore we can split our game into a sum of several subgames. In our experiments with late endgames, such 'dynamic splitting' reduced the search space substantially. Very often, some split off parts can be classified statically as territory or neutral points. In any case further decomposition helps fight combinatorial explosion.



Splitting an area into three parts

Splitting the area implies shrinking the local area, which leads to similar problems as expanding it. A surprising implementation problem arose when using hash tables in conjunction with area splitting. Now a real local hash table is needed because the same global position can appear after in different contexts after different sequences of splitting operations.

How to Handle Dependencies

Dependencies between games can occur when the areas of local Go games overlap. This happens when using expanding areas. Another case of dependencies between games is if external dependencies of one local game are related to another game. The effects of such dependencies differ widely: Very often independence does not hold strictly, but it is a useful approximation. In cases such as double threats however, dependencies are crucial.

Three strategies of dealing with dependencies seem possible:

- ignore them.
- Merge dependent local areas, then start from scratch with merged area. Maybe reuse previously generated local trees for further expansions.
- Analyse the interaction between local games, then use a specialized theory to construct a joint game.

A frequent kind of dependency is the *double threat:* the same first move appears in two games which then continue independently. This case can be handled formally by replacing the options $G + H_L$, $G_L + H$ of the sum game $G + H$ with $G_L + H_L$. Other kinds of merging two games by identifying moves can be imagined as well.

[Wolfe 91a] analyses sets of simple endgames *(corridors)* which are adjacent to the same block. This block must be saved from capture after those liberties which are distributed over the endgames have been occupied.

Bounds for Game Evaluation

Conservative evaluation can be used to calculate upper and lower bounds on terminal positions of local games. These values can then be backed up to get upper and lower bounds for the mathematical game evaluation of the root. Such an approach might lead to better algorithms for selective expansion of the local search trees. Games that contribute most to uncertainty of root evaluation can be expanded deeper. It remains to be seen whether significant differences to a temperature driven expansion strategy result. One advantage of such a procedure is that with valid bounds a win could be proven even if the exact game value is not computable.

Local Go Games with Loops (Ko's)

The theory of loopy combinatorial games is still being developed [Berlekamp 92]. It seems that two fundamentally different classes of Ko positions exist. For the 'harmless' class of Ko's much combinatorial game theory can be applied: For example, mean value, temperature, Leftscore and Rightscore of such games exist and they are independent of Ko threats. For the other class of 'hyperactive' games these values depend on available Ko threats, and a general theory seems more difficult.

[Berlekamp 92] gives an extension of thermostrat that incorporates loopy games and simple Ko threats.

Iterative calculation of bounds for games involving Ko is described in [Müller/Gasser 92].

A complete theory of Ko's is hard: for example, it must handle the tradeoff between playing a move for profit and saving it as a later Ko threat, or when to play a point losing threat, or when to generate or remove Ko threats instead of making points, or how to play small Ko threats in a small Ko fight and leave the big ones for a later, bigger Ko.

7. Summary, Outlook and Open Problems

Features of the Proposed Sum Game Model for Go

- The Go knowledge is clearly separated from the evaluation, therefore both can be improved separately.
- There is inherent parallelism on many levels: independent local searches and mapping to math games, operations on math games can be parallelized.
- Stepwise, directed expansion of search is supported. This provides more time control than usual iterative deepening. It is easy to use opponent's time.
- Important Go terms can be expressed formally.
- Locality reduces the complexity of move generation. With the emphasis on evaluation, clever pruning during move generation is not as crucial as in global search because of the reduced search space.
- Added complexity for managing mathematical games, and sets of local trees.
- Board partition and recognition of dependencies must be good.
- The usual problems of selective search appear in complicated local games: missing crucial moves, wrong evaluations.
- Local games may be overlooked completely, such as territories with unstable boundaries.

Future Work

We believe that sums of games provide a useful general framework for Computer Go. Many details must be worked out to make it work.

- More specialized theories, and better heuristics for local games: the better the quality of local games, the better the overall quality of play. Examples are eye detection, Life and Death, or proving the safety of territories.
- More Go knowledge expressed in terms of combinatorial game theory
- Algorithms for dependent games
- Investigate algorithms for the evaluation of sum games: estimate size of errors and computational complexity of different algorithms. Define a class of random go-like mathematical games and test the algorithms on sums of such games. Develop an approximation scheme to calculate progressively better bounds on a game, maybe by computing partial sums.
- Adaptation for parallel computers.
- A long term goal: identify areas where computer is superior to humans, such as chaotic tactical fights, or difficult endgames with many similar summands.

Conclusions

Go is a complex game. The board size, the number of possible moves and average length of a game are greater than in chess or most other games. Still, Go is not completely beyond human comprehension. The game exhibits a lot of logical, geometrical and combinatorial structure which human players recognize and exploit very well. In contrast, today's Go programs handle only the most basic concepts of Go. Our plan for a Go Program is to model the highly structured parts of the game to narrow the gap to humans with their knowledge. Use the detected structure to formulate relevant subproblems. On top of that, use lots of search to outcalculate the humans in the chaotic parts of the game.

To make progress, we feel it is necessary both to develop more Go-specific theories and to apply relevant other theories such as combinatorial game theory to Go.

References

[Allis 88] A Knowledge-based Approach of Connect-Four. The game is solved: White wins. Allis, L.V., M.Sc. thesis, Free University, Amsterdam 1988.

[Allis et al. 91] Which games will survive? Allis, L.V., van den Herik, H.J., Herschberg, I.S. In: [Levy/Beal 91].

[Allis 92] To appear in: Heuristic Programming in Artificial Intelligence 4.

[Benson 80] A mathematical analysis of Go. BENSON, D.B. In: Proc. of the 2nd Seminar on Scientific Go-Theory. HEINE, K. (ed.) Institut für Strahlenchemie, Mühlheim a. d. Ruhr (1979), pp.55-64.

[Berlekamp 91] Introductory Overview of Mathematical Go Endgames. BERLEKAMP, E. In: Proc. of Symposia in Applied Mathematics, Vol. 43, Combinatorial Games, pp. 73-100.

[Berlekamp 92] Mathematical Go: Thermographs find the biggest move asymptotically. BERLEKAMP, E. Unpublished manuscript, 1992.

[Berlekamp/Conway/Guy 82] Winning Ways. BERLEKAMP, E., CONWAY, J.H., GUY, R.K. Academic Press, London 1982.

[Boon 90] A Pattern Matcher for Goliath. BOON, M., Computer Go 13, p.12-23.

[Chen et al. 90] The Evolution of Go Explorer. CHEN, K.; KIERULF, A.; MÜLLER, M.; NIEVERGELT, J. In: Chess - Drosophila of AI? MARSLAND, T. A.; SCHAEFFER, J. (Eds.) Springer Verlag, 1990.

[Conway 76] On Numbers and Games. CONWAY, J., Academic Press, London/New York 1976.

[De Groot 65] Thought and choice in chess. DE GROOT, A. D., Mouton, The Hague, 1965.

[Fierz 92] Go Endgames. FIERZ, W., Semesterarbeit, ETH Zürich 1992.

[Gasser 91] Applying Retrograde Analysis to Nine Men's Morris. Gasser, R. In: [Levy/Beal 91].

[Gasser 92] Gasser, R., personal communication.

[High 92] Mathematical Go. HIGH, R.G. In: The Go Player's Almanac, BOZULICH, R. (Ed.), Ishi Press, Tokio 1992, p.218-224.

[Kierulf 90] Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies. KIERULF, A. Ph.D. Thesis, ETH Zürich, 1990.

[Kraszek 88] Heuristics in the Life and Death Algorithm of a Go-playing Program. KRASZEK, J., In: Computer Go 9, p.13.

[Lenz 82] Die Semeai-Formel. Lenz, K.F. Deutsche Go-Zeitung 57, No.4, 1982.

[Levy/Beal 91] Heuristic Programming in Artificial Intelligence 2. Levy, D.N.L., and Beal, D.F.(eds.), Ellis Horwood 1991.

[Lichtenstein/Sipser 78] GO is Polynomial-Space Hard. LICHTENSTEIN, D.; SIPSER, M. *Journal ACM* **27**, 2 (April 1980), 393-401. (Also: IEEE Symp. on Foundations of Computer Science, (1978), 48-54).

[Miller 76] The End Game of Go. MILLER, J. *In:* Proc. Northwest 76 ACM/CIPS Pacific Regional Symposium (Seattle Pacific College, Seattle, Washington, June 24-25, 1976).

[Morris 81] Playing Disjunctive Sums is Polynomial Space Complete. Morris, F.L., Int. Journal Game Theory, Vol. 10, No. 3-4, p.195-205, 1981.

[Müller 89] Eine Theoretische Basis zur Programmierung von Go. MÜLLER, M. Diploma thesis, Techn. University of Graz, Austria, April 1989.

[Müller/Gasser 92] Experiments in Computer Go Endgames. MÜLLER, M., and GASSER, R. To appear in: Heuristic Programming in Artificial Intelligence 4.

[Neumann 28] Zur Theorie der Gesellschaftsspiele. Neumann, J. von, Math. Ann. 100, pp. 295-320.

[Neumann/Morgenstern 44] Theory of Games and Economic Behaviour. Neumann, J. von, and Morgenstern, O., Princeton University Press, Princeton, 2nd ed. 1947.

[Popma/Allis 92] Life and Death refined. Popma, R.; Allis, L.V. In: Heuristic Programming in Artificial Intelligence 3, van den Herik, j., and Allis, V. (Eds.), Ellis Horwood 1992, p.157-164.

[Robson 83] The Complexity of Go. ROBSON, J.M. Proc. IFIP (International Federation of Information Processing), (1983), 13-417.

[Schaeffer et al. 91] Reviving the Game of Checkers. Schaeffer, J., Culberson, J., Treloar, N., Knight, B., Lu, P., Szafron, D. In: [Levy/Beal 91]

[Schaeffer 92] Man Versus Machine: The Silicon Graphics World Checkers Championship. Schaeffer, J., unpublished manuscript available by *ftp*.

[Shannon 50] Programming a computer for playing chess. SHANNON, C.E. *Philosophical Magazine* **41**, 314 (1950), 256-275.

[Stiller 89] Parallel Analyses of Certain Endgames. Stiller, L. ICCA Journal 12, no. 2, p. 55-64.

[Thompson 82] Computer Chess Strength, THOMPSON, K. In: Advances in Computer Chess 3. CLARKE, M.R.B. (Ed.), Pergamon Press, Oxford, 1982.

[Thompson 86] Retrograde Analysis of Certain Endgames. Thompson, K. ICCA Journal 9, no. 3, p. 131-139.

[Turing et al. 53] Digital Computers applied to Games. Turing, A.M., Strachey, C., Bates, M.A., Bowden, B.V. In: Faster than thought, Bowden, B.V. (Ed.), Pitman, London 1953, p. 286-297.

[Wilcox 79] Computer Go - The Reitman-Wilcox Program. WILCOX, B. *American Go Journal* **14**, 5/6 (1979), 23-41.

[Wolf 91] Investigating Tsumego Problems with RisiKo. Wolf, T., In: [Levy/Beal 91].

[Wolfe 91a] Mathematics of Go: Chilling Corridors. Wolfe, D., Dissertation, Univ. of California, Berkeley 1991.

[Wolfe91b] Games program available via anonymous ftp from milton.u.washington.edu (128.95.136.1), file theory.sh.Z.