Diss. ETH Nr. 11006

Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZÜRICH

for the degree of
Doctor of Technical Sciences

presented by

MARTIN MÜLLER
Dipl.-Ing. TU Graz

born February 3, 1965
citizen of Austria

Accepted on the recommendation of

Prof. Dr. J. Nievergelt, examiner
Prof. Dr. E. Berlekamp, co-examiner
Prof. Dr. R. Marti, co-examiner

Zürich 1995

Meinen Eltern Helmut und Elisabeth

Acknowledgments

I would like to thank my advisor Prof. Jürg Nievergelt for making this research project possible and for providing insights into many aspects of academic life. I also appreciated having Prof. Berlekamp and Prof. Marti as co-examiners.

Anders Kierulf and Martin Dürst carefully read and commented on earlier drafts of the thesis. The past and present members of the Nievergelt group always ensured a pleasant working environment: Anders Kierulf, Peter Schorn, Ralph Gasser, Michele de Lorenzi, Adrian Brüngger, Ambros Marzetta, Markus Julen, Christoph Wirth, Fabian Mäser, Matthias Müller, Nora Sleumer and Thomas Lincke.

Prof. Berlekamp and his group including David Wolfe and Yonghoan Kim developed the mathematics of Go which have been applied in this thesis, and guided me through the intricacies of combinatorial game theory.

Peter Geiser built a prototype of the pattern matching system, Harald Peter extended it to approximate matching and Werner Fierz ported David Wolfe's toolkit for mathematical games to the Smart Game Board. Guido Hächler and Andreas Thurnherr implemented variants of parallel search algorithms.

Finally, I thank all members of the world wide computer Go community for sharing their excitement, expertise, joy and desperation at numerous computer Go tournaments and conferences.

Table of Contents

Abstract

Combinatorial game theory provides an exciting approach to the analysis of games: It allows the decomposition of a game into a sum of local games. In contrast to classical game theory, few applications to computer game playing have been demonstrated.

Game programming has enjoyed the continuous interest of computer science research. Progress has been impressive: several games have been solved by computer, typically by a combination of mathematical analysis and exhaustive search. Programs for many other popular games such as chess challenge the top human players. The notable exception is the ancient oriental game of Go.

This thesis claims that combinatorial game theory can be applied to computer Go. We develop a sum game model for heuristic Go programming and a program for perfect play in late stage endgames. As a case study in Knowledge Engineering, the 'Explorer' program presents new approaches to modeling Go knowledge. We adapt a string matching algorithm for Go pattern matching, develop algorithms for board partition, and create a framework for identifying, searching and evaluating local games.

We extend the Smart Game Board, a workbench for game programmers, with tools for displaying and editing Go-specific data. For measuring the performance of Go programs, we introduce the Computer Go Test Collection containing thousands of annotated positions.

Kurzfassung

Die kombinatorische Spieltheorie liefert einen interessanten neuen Ansatz zur Analyse von Go: sie erlaubt die Zerlegung eines Spiels in eine Summe von lokalen Teilspielen. Im Gegensatz zur klassischen Spieltheorie wurde sie jedoch kaum in Computerspielprogrammen eingesetzt.

Die Entwicklung von Spielprogrammen ist ein aktives Forschungsgebiet der Informatik. Dabei wurden eindrucksvolle Fortschritte erzielt: mehrere Spiele wurden vollständig gelöst, typischerweise durch eine Kombination von mathematischer Analyse und erschöpfender Suche. In vielen anderen bekannten Spielen wie etwa Schach sind Programme starke Gegner für die besten menschlichen Spieler. Die bemerkenswerte Ausnahme ist das alte asiatische Brettspiel Go.

Diese Dissertation vertritt die These, dass die kombinatorische Spieltheorie auf Computer-Go anwendbar ist. Wir entwickeln ein Summenspiel-Modell für die heuristische Goprogrammierung und ein Programm für perfektes Spiel im späten Endspiel. Als Fallstudie im 'Knowledge Engineering' enthält das Programm 'Explorer' neue Ansätze zur Modellierung von Go-Wissen. Wir adaptieren einen Zeichenkettenvergleichsalgorithmus für den Mustervergleich im Go und entwickeln Methoden zur Brettaufteilung sowie zur Identifikation, Suche und Bewertung von lokalen Spielen.

Wir erweitern das Smart Game Board, eine Entwicklungsumgebung für Spielprogramme, mit Werkzeugen zur Anzeige und Bearbeitung Go-spezifischer Daten. Um die Leistung von Goprogrammen zu messen, erstellen wir die Computer Go Test Collection mit tausenden von kommentierten Positionen.

1
Computer Go

> 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

This chapter motivates why computer Go is an interesting research subject. It introduces the specific advantages and problems of this domain. We survey theories related to Go and briefly describe our development shell, the *Smart Game Board*. A final section explains the structure of the thesis.

The ideal reader of this thesis would be a computer scientist familiar with the game of Go and the basic concepts of game theory. Knowledge of combinatorial game theory is helpful, too. An introduction is given in Chapter 3. A glossary of terms is included as an appendix.

1.1 Why Computer Go?

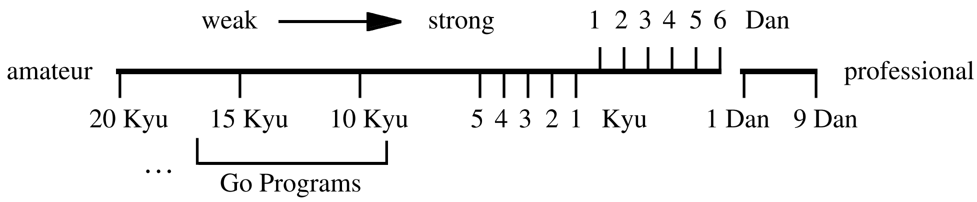
John McCarthy sums up the present state and future direction of computer Go in a few words. Games have long been used as vehicles for research in computer science. Some of the most prominent computer scientists are linked with the beginnings of computer game playing [Neumann/Morgenstern 44, Shannon 50, Turing et al. 53].

Since that time, computer game playing has become a specialized research subject, leading to less insight for computer science in general than had been hoped by the early prophets of *Artificial Intelligence*. In games such as chess, brute force search techniques have proven superior to knowledge based, 'intelligent' programs.

Because of its complexity, Go seems to resist any brute force approach. The promise that computer Go methods will lead to insight into general problem solving makes the topic a challenging one for computer scientists.

1.2 State of Computer Go

With the single exception of chess, more programming effort has been spent on Go than on any other game. Yet in playing strength, Go programs lag far behind their counterparts in Checkers, *Awari* or Nine Men's Morris. Reasons for this deplorable state of affairs can be found in the inherent difficulty of the game, in the structure of the computer Go community, and in the limitations of existing programs, which will be analyzed in Chapter 2.



Go programs on the human ranking scale

The ultimate goal of Go programmers is to create a program of amateur Dan level, or even challenge Ing's million Dollar price in a match against professional players. Currently we are far from reaching this goal. The best programs rank as relative beginners on a human scale, and need a huge number of handicap stones after a player gains a little bit of practice in playing them. Progress has slowed down as the complexity of existing ad hoc solutions increases.

The Computer Go Community

While Go programming started in the late sixties, it got a big boost in the mid eighties with the appearance of the sponsors Mr. Ing Chang-Ki and the computer company Acer Inc. The current size of the serious computer Go community is estimated at about 50 people. There are only a few tournaments. The annual *International Computer Go Congress* (ICGC) has the status of a world championship.

Traditionally, computer Go has been a one person or very-small-team effort. Many researchers in related disciplines such as computer chess, AI, expert systems or machine learning have studied Go and concluded it was 'too hard'. Most existing Go programs have been written by Go enthusiasts, often strong amateur players.

The basis for professional programmers is even smaller than in chess. Computer Go activists are typically university researchers or hobbyists. Though progress is made, most programs remain tied to a single person with limited time budget. The consensus is that a state-of-the art program needs at least 4-5 man-years of effort. Most of the leading programs are veterans approaching their tenth birthday. Few individuals and organizations are willing to make that kind of investment.

Two ways out of the bottleneck are being pursued:

- Start a larger scale project: The obvious way to get more man-years into a Go program. It does not automatically mean better results, though, as new problems of coordination and management crop up. A potentially promising project was recently started by Mark Boon [Boon 94].
- Concentrate on tractable subproblems: Berlekamp and colleagues study the mathematics of late-stage endgames [Berlekamp 94]. Thomas Wolf has developed a strong *tsume Go* program [Wolf 91]. The progress in these domains has been impressive. So far, these approaches have not been integrated into a playing program.

A third way, collaboration between researchers, has proven very difficult in practice. Groups have split or lost members more often than researchers have joined forces.

1.3 Some Facts that make Go a Difficult Game for Computers

Size and Structure of the Problem Space

The size of the search space for 19x19 Go, estimated at 10^{170} positions, is probably the biggest of all popular board games [Allis et al. 91]. No simple yet reasonable evaluation function seems to exist. This is evident to all 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 regular, making perfect play possible with little search and a good theory, as exemplified by Berlekamp's endgame studies [Berlekamp 91]. It can also be extremely chaotic, leaving exhaustive analysis as the only known method of solution. Therefore a competent Go program will have to combine a good theoretical foundation with lots of computing power.

Quality and Quantity of Human Knowledge

Humans are able to recognize subtle differences in Go positions that will have a decisive effect many moves later. Professionals know by intuition whether a group can be captured or not. Proving this by an exhaustive computer search however would take billions of nodes.

Skilled players usually know which side is better in a game after a quick glance at the position, another feat that would involve an enormous amount of processing on a computer, if we knew how to do it. We can contrast Go with more computer friendly games such as Nine Men's Morris, Awari and some chess endgames: Humans get lost in the 'combinatorial chaos' of the game, while machines exploit their computing power [Thompson 86, Gasser 91].

A huge quantity of Go knowledge has accumulated over centuries, much of it implicit in game records of master players. Thousands of game commentaries, tutorial books and problem collections have been compiled.

Pattern knowledge of experts seems at least comparable to that of chess experts, which is already daunting [De Groot 65]. Patterns recognized by humans are more than just stones and empty spaces: Players can perceive complex relations between groups of stones, and easily grasp fuzzy concepts such as 'light' and 'heavy' stones. This visual nature of the game fits human perception but is hard to model in a program. The cognitive models of Reitman and Wilcox [Wilcox 79] were interesting, but today's programs make do with comparatively simple pattern matching [Boon 90, Müller 91b].

While the knowledge of chess programs is tuned nicely to their searching power, Go programs are severely lacking in both quality and quantity of knowledge.

1.4 Goals of Computer Go Research

We have identified three types of research on Computer Go:

1. The 'tournament approach': The goal is to win the International Computer Go Congress and other computer Go tournaments. The long term goal is to win Ing's million dollar prize. Progress is made by improving overall play, which often means finding and fixing the worst bugs.
2. The 'niche approach': The goal is strong or perfect play in a subset of Go positions. Examples are Thomas Wolf's Life&Death program and the endgame component of

Explorer. Progress is made by expanding the range of positions for which exact solutions are feasible, or by adapting the methods to heuristic play.

3. The ‘test vehicle approach’: The goal is to use Go as an example to demonstrate new computer science methods. Go specific results are secondary. This approach is popular with researchers in neural networks and machine learning.

1.5 Theories for Computer Go

For the development of a game program we use theories in two ways:

- To understand the problem domain (e.g. subtleties inherent in the rules) more precisely.
- To provide means of (efficiently) analyzing aspects of a game.

An advantage of Go over other classical board games is that the more regular structure of Go allows better application of mathematical theories.

Classical Game Theory

Classical game theory was pioneered by von Neumann and others, who formalized and proved fundamental concepts of game-playing [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.

Many refinements have been developed for the implementation of classical game theory in computer programs: *Alpha-beta* pruning, transposition tables, and parallel search among others [Marsland/Schaeffer 90]. These techniques lead to enormous success in many board games, yet they fail in all but the most basic Go problems [Thorp/Walden 64]. In place of full-width game tree search, selective search is widely used in computer Go. Examples are goal-oriented search to solve specific tactical problems or tightly constrained global search.

Game-specific Theories

Many specialized theories have been developed for solving a particular game, or for a subgame such as King, Bishop and Knight against King in chess. The purpose of such theories is to reduce the search space: Search depth is reduced by evaluation of nonterminal positions, search width by pruning rules which reduce the number of moves to be investigated. Ideally, a theory can solve a game by direct evaluation of the initial position, without any search.

The solutions of the game Qubic by Patashnik and of Gomoku by Victor Allis demonstrate the power of game-specific theories [Patashnik 80, Allis 94]. Examples of theories for Go subproblems are the traditional ‘semeai formula’ [Lenz 82] and Benson’s algorithm for detecting unconditional life [Benson 80].

Combinatorial Game Theory

Combinatorial game theory (CGT) has been developed and applied to many mathematical games in [Conway 76] and [BCG 82]. This theory defines a game in a very elegant mathematical way that generalizes von Neumann’s: There is no restriction that players must move alternately. A principal goal of the theory is the *decomposition* of games into independent subgames. This leads to the notion of a *sum* of games.

Recently, CGT has been used for pencil-and-paper analysis of difficult Go endgames [Berlekamp 91, Wolfe 91a]. Automating these methods and applying them to computer Go is a main topic of this thesis.

Heuristics

Beyond strict theory, there are heuristics: formalized rules of thumb, or applications of a theory outside its guaranteed range of 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 in which heuristic evaluation is too imprecise.

Almost all the substance of current Go programs is contained in their heuristics for position evaluation and move generation. For example, most *patterns* can be interpreted as heuristic move generation rules.

1.6 Notes on History of Go Theory and Computer Go Endgame

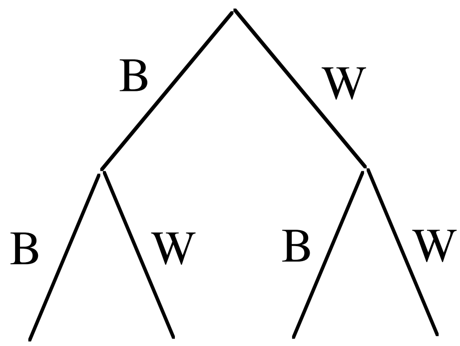
The history of computer Go has been documented in several places [Erbach 92, Kierulf 90, Müller 89, Wilcox 79]. Dissertations dealing with computer Go are [Friedenbach 80, Kierulf 90, Ryder 71, Zobrist 70a]. We complement the literature with a summary of other work that influenced our investigations.

Applications of Combinatorial Game Theory to Go Endgames

An early attempt using sums of games for Go endgames is [Miller 76]. We do not know whether these ideas resulted in a playing program.

Raymond Chen, a student of Berlekamp, has built a demonstration program that plays endgames according to a human-built database of local games [BW 94]. A first step towards automatic generation of such a database was our program for exhaustive analysis of single local endgame positions [Müller/Gasser 94].

Kao Kuo Yuan and Ken Chen studied a simplified version of local endgame play, where players are restricted to a single move in each node. This is a useful approximation whenever a good local move is easy to identify. Finding an optimal move in this *binary tree model* is faster than in the general case [Chen 93].



The binary tree model of Kao and Chen

Safety of Stones and Territory

Benson gave a mathematical characterization of *unconditionally alive* blocks of stones [Benson 80]. These blocks cannot be captured by the opponent even if she is allowed an arbitrary number of moves in a row. Benson's algorithm detects a set of blocks and *regions* that together form two or more eyes. It relies on the assumption that suicide is illegal.

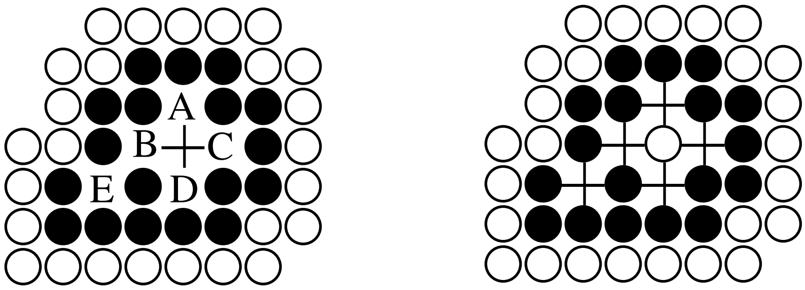
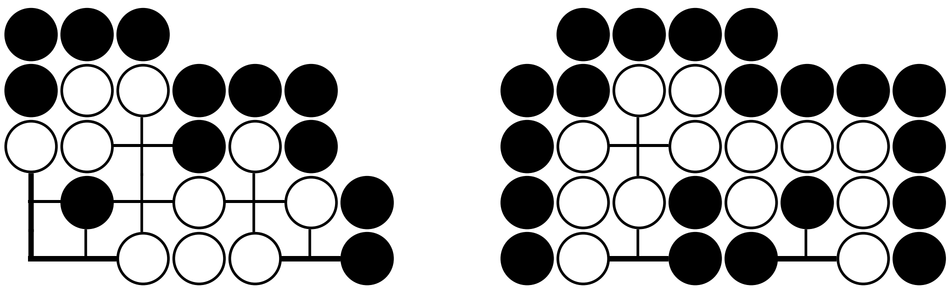


Illustration of unconditional life

The group in the left picture is not unconditionally alive: it can be captured by five successive white moves A…E. The right side group is alive when suicide is forbidden. Of more practical interest are algorithms for detecting groups of blocks which are safe under alternating play. Most Go programs contain such *Life and Death* knowledge, typically a combination of exact and heuristic rules [Kraszek 88].

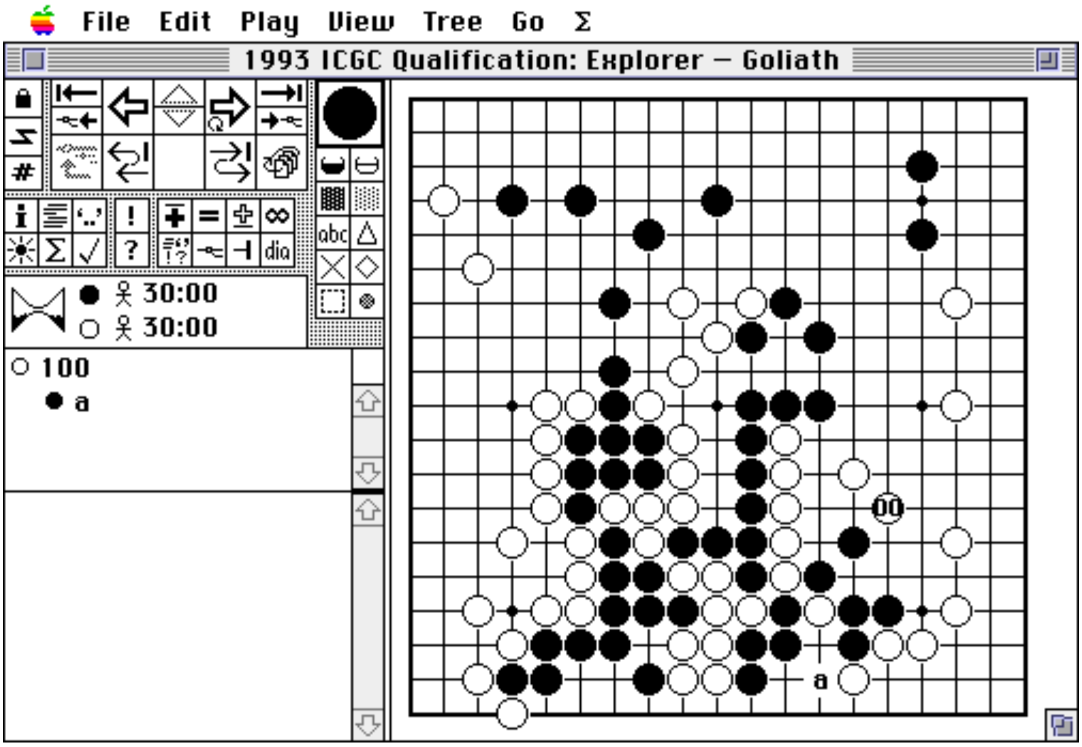
The performance of programs on Life&Death puzzles is in the mid kyu range, maybe slightly above their general level of skill. Specialized programs for solving such problems in a small completely enclosed region do better, reaching the level of strong amateur Dan players [Wolf 91]. Wolf's program contains powerful rules for static Life&Death recognition, elaborate move ordering heuristics and a state of the art tree searching algorithm.



White stones recognized statically as dead (left), and alive (right), from [Wolf 91]

A classification of living groups according to the existence of specific algorithms for making life is given in my diploma thesis [Müller 89].

1.7 Smart Game Board: A Development Tool for Go Programs



Sample screen display

What is the Smart Game Board ?

Anders Kierulf's Smart Game Board is a tool supporting players [Kierulf 90] and programmers [Müller 90] of two player board games. Initially designed for Go [Kierulf/Nievergelt 85], it has been generalized and adapted to half a dozen popular board games.

The Smart Game Board has proven its value for several hundred game players all over the world. In many ways, an electronic game board is more powerful than a conventional board. One example is its ability to keep track of several move sequences and multiple games.

The value of the Smart Game Board for game programmers is evident from all the playing programs written on or ported to the Smart Game Board. This includes two advanced Othello programs [Kierulf 82, 89a], a perfect Nine Men's Morris program [Gasser 90, 95] and several generations of Go programs [Kierulf/Nievergelt 89]. Ken Chen's *Go Intellect* [Chen 89] and our *Explorer* both use the Smart Game Board.

The Smart Game File (SGF) Format has become a widely accepted method of storing Go games and problems. Thousands of amateur and professional games in SGF format are available from anonymous ftp sites such as bsdserver.ucsf.edu (the main Go archive site) and imageek.york.cuny.edu (the IGS archive in subdirectory /igs).

1.8 Project History

The *Explorer* program is one of several game playing programs running under Smart Game Board, which have been developed by Prof. Nievergelt's group at ETH Zürich and UNC Chapel Hill.

At the beginning of this thesis, Explorer was already a five year old project [Kierulf/Nievergelt 85, Kierulf 90]. The software, coded in Modula-2 for the Macintosh, was organized in three layers:

- Smart Game Board kernel, written by Anders Kierulf
- Go Board and Go Tactics, also written by Anders Kierulf
- Explorer, written by Ken Chen

In 1989 Nievergelt and Kierulf returned to Switzerland, while Chen stayed in North Carolina. When I joined the project a few months later, Anders Kierulf soon left the development of Explorer to me. I tried to analyze it by playing with it, reading the source code, and breaking up its evaluation function into components [Müller 90]. Attempts at improving the program had only marginal success. It was becoming clear that it is hard for a non-author to improve such a complex program. To quote Anders Kierulf:

> “Regardless of how muddled the structure of Explorer is, it is the result of half a year of Ken Chen fiddling with parameters, adjustments, and refinements, and it's anyone's guess which parts of the program contribute to its playing strength and which parts are just noise.” [Kierulf 89b]

Starting in 1990 essential parts of Explorer were rewritten from scratch. Knowledge representation was changed from hard-coded Modula-2 to graphical patterns, using a pattern matching engine based on the Patricia method [Geiser 91, Müller 91b].

Recognition of territories and groups was now done with new board partition algorithms developed during my diploma thesis [Müller 89]. The *influence function*, the basic partitioning method of the old Explorer, was replaced by these new algorithms.

A big part of a Go program’s knowledge is not being used directly for generating moves, but for building a good representation of the board. With increasing amounts of knowledge, an old problem of game programming crept up: coordination between different parts of the knowledge base. In the end, more time was spent on relative adjustments between parts of the program than on adding new knowledge. It was time to tackle the major structural weakness of Explorer, the reliance on one huge static evaluation function.

Stimulated by the success of Mathematical Go Theory, in 1992 we began experimenting with a sum game model for computer Go. Wolfe’s toolkit for mathematical games was ported to our development environment as a student term project [Fierz 92]. Using Explorer and this toolkit we built a program that could automatically recognize and solve sums of simple endgame positions such as *corridors* and *node rooms* [Wolfe 91a, Müller/Gasser 94].

This work has been continually extended in two directions: solving a larger class of endgame positions, and building a heuristic Go program using the sum-of-games approach.

Why Focus on Go Endgames?

At first sight, it seems silly to spend much effort on endgame play: at the current level of Go programs, it is irrelevant for 90% of all games played. Yet Go endgames offer a number of advantages for research:

- The complexity of the game often decreases towards the end. This allows the study of Go in a controlled, simplified context.
- An exact solution is possible for some classes of endgame positions.

- The exact solution of parts of a Go board facilitates the analysis of the rest. Reaching the ultimate goal of winning the game is easier when complete information about part of the game is at hand. Such information is useful as additional input for the heuristics that deal with the rest of the board. Human experts use similar reasoning: they observe the score continually from the early midgame, and base their strategic decisions on such an analysis [Takagawa 85].
- Some methods developed for partitioning, searching and scoring during endgame play carry over to the midgame and opening. As programs improve, fewer game-deciding blunders will occur, so the importance of endgame-type calculation is bound to increase.
- On a more philosophical note, the endgame relates to the full game of Go as Go relates to real world AI problems: It provides a simplified, more controlled subdomain that allows the use of stronger theoretical models than the larger, more general problem.

1.9 Contributions of this Thesis

We developed and implemented a computer model of Go as a sum of local games, which proves the feasibility of this divide-and-conquer approach to computer Go. We examined the interaction of search, knowledge, domain-specific theories and abstract mathematics in the 'micro-world' of Go. This gave insights into the modeling of complex domains.

As a verification of the resulting program, we computed solutions to endgame studies of Berlekamp and others [Berlekamp 91, Wolfe 91a], establishing a nontrivial subdomain of Go in which a computer plays perfectly. We also implemented and tested a competitive Go program, using new approaches to board partitioning and pattern matching.

1.10 Structure of this Thesis

Chapter 2 summarizes the state of the art in computer Go programming and introduces the program *Explorer*. Various ways in which Go knowledge is built into *Explorer* are described and compared with the knowledge representation of other well-known programs.

Chapter 3 reviews combinatorial game theory and its application to Go endgame theory, as developed by Berlekamp and others.

Chapters 4 to 6 develop new models for playing computer Go as a sum of local games. Chapter 4 develops a heuristic sum game model for the entire game that aims at reusing components of an existing Go program. Chapter 5 deals with board partitioning algorithms, which are used to split Go positions into sums of independent games. Chapter 6 describes search and evaluation in local games. Chapters 5 and 6 combined yield a program that computes exact solutions to endgame problems.

Chapter 7 deals with implementation aspects of *Explorer*, focusing on the sum game model. Sections on the user interface describe a pattern editor and tools for experiments with sums of local games.

Chapter 8 gives results of endgame computations, and discusses the problem of performance measurement in general. It argues for the necessity of a *Computer Go Test Collection*. In Chapter 9 we summarize the contributions of this thesis to computer science and propose future research topics.

The appendix describes our *Computer Go Test Collection*. A glossary defines concepts of combinatorial game theory, Go and heuristic game programming relevant for this thesis.

Throughout the thesis, algorithms will be given in a pseudocode notation loosely based on the Modula-2 language.

2

Elements of Go Knowledge: a Case Study in Knowledge Engineering

The basic question addressed in this chapter is: How can we put Go knowledge into a playing program? Knowledge Engineering in computer Go means that knowledge must be acquired, selected, represented, applied and maintained.

Using the ‘classic’ non-local versions of Explorer as a case study, we introduce some standard components of Go programs. Several types of search are discussed. In our description of Explorer, we focus on those parts that serve as building blocks for the local game player discussed later: Knowledge representation by *patterns,* and other localized data structures.

2.1 Conventions

Due to a multitude of languages and traditions, there is no standard nomenclature for Go and computer Go terms. We will use the following conventions: A connected set of stones of the same color is called a *block*. The terms *chain* and *group* are used for bigger aggregates of stones.

In diagrams, which were produced using the Smart Game Board, the following markers are used:

	Safe stones and territories
	Dead stones
	Insecure stones
	*Connections*
	*Dividers*

Markers used in diagrams

Partially shown stones at the edge of a diagram are considered safe. Typically these are walls surrounding the area of interest. Other nearby points are sometimes shown dimmed.

2.2 Knowledge Engineering in Computer Go

We survey the steps of knowledge engineering: Acquisition, selection, representation, application and maintenance.

Types of Knowledge used in a Go Program

In a Go program, game-specific knowledge is used for board partition, position evaluation and move generation. Machine-dependent knowledge is needed for the management of resources such as time (search control) and memory. A unique problem of computer Go is allocation of time between multiple search tasks required for the same move.

An important classification of knowledge is facts vs. heuristics. Both coexist in a Go program. There is a tradeoff between computing exact solutions and relying on heuristics for faster approximate solutions.

Knowledge Acquisition and Selection

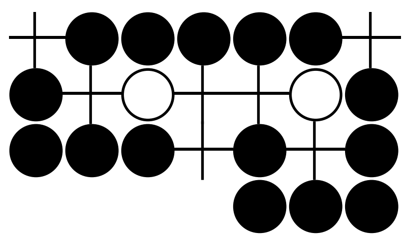
A huge amount of Go knowledge is available in form of literature on theory, Go problems, and master games. The biggest practical problem is finding a suitable subset for an implementation. The selection should follow the principles of consistency, completeness, relevancy and soundness.

- *Consistency* aims at avoiding contradictory advice (e.g. both 'move is good' and 'move is bad'). Some cases of inconsistency can be resolved by a priority scheme, as discussed below.
- *Completeness* (relative to a given standard of performance) assures there are no knowledge gaps that lead to sudden inexplicable drops in performance. An incomplete program will play a sequence of good moves, then suddenly turn elsewhere or commit a blunder that invalidates all previous moves.
- *Relevancy* means that knowledge is applicable in situations that are likely to occur during computer play. The knowledge base must be tuned to the general level of the program. It is counterproductive to include 'high-level' moves if the program cannot follow up correctly. A huge *joseki* database would be of little use to a 10 kyu program, and might even have a negative effect on the program's playing level. If a joseki results in a wall for one player, this wall must fit the surrounding positions. If a joseki ends in a difficult position with many weak groups, invariably the program will blunder in the continuation.
- *Soundness* means giving advice that is good, at least in a large percentage of cases. Soundness is closely related to the other three principles.

Building Knowledge through Search and Inference

New knowledge created by search must be integrated with existing knowledge. We consider only 'special case' knowledge here:

- Proving *lemmas* about a Go position: once the safety of stones and territory in a certain context has been proven, the result is put into a database for reuse. The same procedure is used for partial results such as eye status of an area, and for the perfect evaluation of near end positions.
- Prove bounds on a game's value: By restricting one player's move options, a bound on the real game value can be computed even if the game itself is too complicated or contains loops in the game graph.



Example lemma: The black group can be proven safe using only the section of the board shown

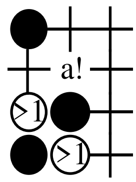
- Inference using rules: e.g. applying the rule that a group of stones is alive if it has two eyes. Results of inference may be treated as lemmas, too, and be put into a knowledge base for reuse.

Goal-Oriented Local Search

Go programs use tactical search to classify the status of blocks as dead (is captured, cannot escape), threatened (can be captured, or can 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 neighboring blocks. Similarly, programs perform Life&Death search for *groups* of several loosely connected blocks of stones [Kraszek 88, Kierulf 90].

Knowledge Representation

Knowledge can be added directly to a program by writing code, building on a library of Go-specific functions, or it can take the form of a database containing patterns or rules in a Go-specific mini-language. The choice of representation is affected by the ease of implementation, the ease of entering and maintaining the knowledge base, and efficiency.



Knowledge representation by pattern, with annotations in a Go-specific mini-language: 'a!' indicates that a *is an urgent move, '>1' is a constraint on the number of liberties required for the block*

As an example of knowledge representation by procedural code, we show the top level of Explorer's move generator for groups:

```
ALGORITHM TGroup.GenerateMoves (toPlay: Color);
(* Generate moves for a group and a player *)
BEGIN
    IF (fTotal ≠ dead) THEN
        IF (fColor = toPlay) THEN Expand () ELSE Reduce () END;

        IF (fTotal ≠ safe) THEN
            IF (fColor = toPlay) THEN Save () ELSE Attack () END
        END;
    END;
END GenerateMoves;
```

Knowledge Application

Knowledge Application involves two phases: During *static analysis* of a position, we match a knowledge base against the current board. During *search*, we use rules in a (different) knowledge base for move generation and evaluation. In both phases, we must *select* relevant parts of the applicable knowledge.

Static Analysis: Matching a Knowledge Base Against the Current Board

Rules consist of a precondition (IF-part) and an action (THEN-part). Rule matching tests which preconditions are currently true. A hierarchical organization of the rule base avoids checking each rule every time. Matching of *patterns* will be described later in this chapter.

Knowledge-Guided Search

Knowledge is used to select moves, and to control tree growth. During selection a program chooses promising move candidates, and prunes unpromising ones. Move order is important for the efficiency of *alpha-beta* evaluation. Tree growth control decides which nodes should be expanded, and in what order. It affects the quality of position evaluation: quiescent positions can be evaluated directly without introducing big errors, turbulent positions should be expanded.

Goal-oriented search is a special case of knowledge guided search: the evaluation function measures progress towards reaching a specific goal. Move generators produce only goal-related moves.

Evaluation during search has to be orders of magnitude faster than the static evaluation of the root position. The knowledge put into these evaluation functions must be tuned for speed, in contrast to the take-all-you-can-get approach of static analysis.

Knowledge Maintenance

Maintenance follows the application of knowledge to debug the knowledge base and achieve the goals of consistency, completeness, relevancy and soundness. Problems discovered while playing through computer games or special test collections provide stimuli for doing maintenance. A good environment with tools for manual and automated testing greatly facilitates maintenance.

2.3 The State of the Art in Computer Go

The evolution of Go programs has produced a set of standard components that are present in most programs. On the other hand, Go programs also share a set of common weaknesses.

Go Skills Present and Absent in Current Go Programs

Standard components present in today's Go programs are:

- Abstractions for representing a Go position, such as *blocks*, *groups* and *territory*
- *Pattern matching* for move generation
- A *joseki* library for standard moves, mainly in the corner
- An *influence function* to determine groups and territory
- *Connections* between blocks, defined by rules, patterns or influence
- *Move generation* and *evaluation* based on static analysis
- Selection of the best move by computing a total move value, using either a linear combination of evaluations, or a priority scheme of move *motives*
- *Specialized search* tasks for deciding whether a block can be captured tactically, and whether a group can make two eyes or not
- Knowledge about the *Ko* rule, and simple *Ko threats* such as atari

On the negative side, the overall standard of play is still low:

- Go programs rely on heuristics which are incomplete and sometimes contradictory.
- In contrast to chess [Thompson 82], there is no clear correlation between computing power and playing strength. The amount of search is only one of many factors limiting playing strength.

- Games are decided by local fighting and the number of blunders equivalent to ‘Pass’ moves.
- Many concepts are still missing: recognition or generation of double threats, or consistency of moves.
- Programs play bad ‘forcing’ moves, which turn out to be *gote* or have bad side effects.

A Survey of Current Go Programs

As a more detailed overview of the state of the art, we give short characterizations of important programs:

Go Intellect (Ken Chen, International Computer Go Champion 1992 and 1994), the ‘brother’ of Explorer, uses a detailed model of blocks and groups, and a complex evaluation function [Chen 89]. It does specialized tactical and Life&Death searches. The resulting move evaluation is dependent on the heuristic value of the blocks and groups involved.

A selective global search involving a few high ranking moves provides additional input to the final move selection procedure. Go Intellect plays some ‘obvious’ *joseki* and tactical moves instantly without search.

Handtalk (Chen Zhixing, International Computer Go Champion 1993) is an amazingly small program, written in assembler for PC. Currently, it seems to be the most solid player, with well balanced knowledge. Unfortunately, little more is known about the program, since no publications are available.

The search used in **Goliath** (Mark Boon, International Computer Go Champion 1989-91) is closest in spirit to the sum-of-local-games approach. Goliath identifies goals, such as cutting, attacking a group, or invading some territory. For each player going first it performs a goal-directed search. Often this search comprises only a sequence of urgent moves. Sometimes a more complete minmax search is done. The difference in scores for each player moving first determines the urgency of the goal. The local move with greatest score difference, i.e. the hottest *switch* in terms of combinatorial game theory, is played. There are adjustments to increase the value of *sente* moves [Boon 91].

Goliath plays many moves instantly, including *joseki* and forced sequences it has computed in advance.

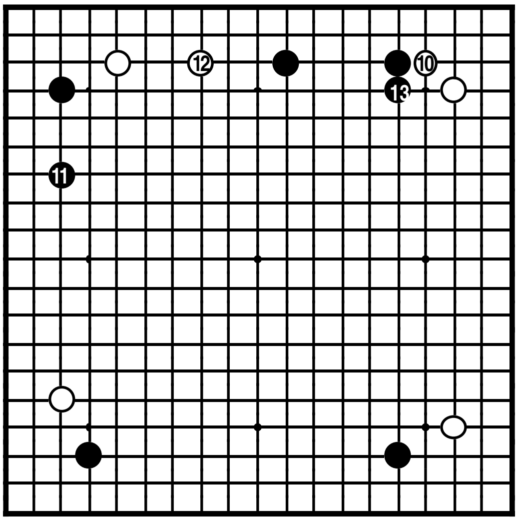
Many Faces of Go (David Fotland, 2nd place International Computer Go Championship 1994) is one of the best known and also best described programs [Fotland 93, Fotland 94]. It has played thousands of games on IGS, the Internet Go Server.

Star of Poland (Janusz Kraszek, 2nd place International Computer Go Championship 1993) has always been one of the top programs. It is based on the paradigm of *homeostasis*, trying to react to changes of equilibrium and keep a balance [Kraszek 90]. Star of Poland is a fast program that is good in Life&Death [Kraszek 88].

A previous version of **Explorer** has won the first Computer Games Olympiad in London [Kierulf/Nievergelt 89]. Other well-known programs include **ModGo** (Knöpfle), **Stone** (Kao), **Go 4.4** (Reiss), **Igo** (Sanechika), **Dragon** (Hsu), **MicroGo** (Scarff), **Nemesis** (Wilcox), and **GOG**, an application developed during the Japanese Fifth Generation Project.

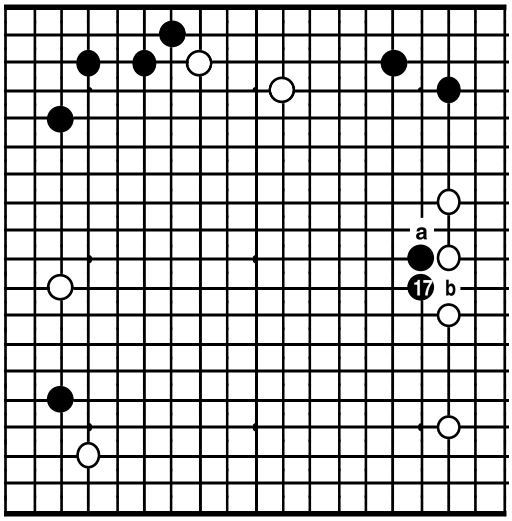
Snapshots from Recent Tournament Games

This section shows the strengths and weaknesses of leading programs in games from recent tournaments.



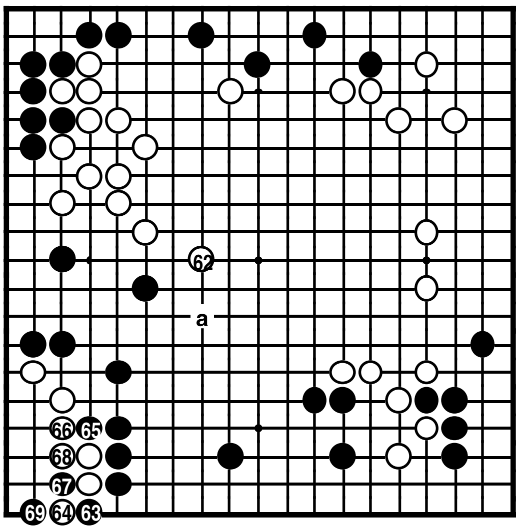
Go Intellect (W) vs. Many Faces of Go (B), US Championship 1994

Intellect's favorite *kosumi* at 10 forces a reply at 13. Yet both programs play a non-urgent move first, before Many Faces finds the 'hot spot'.



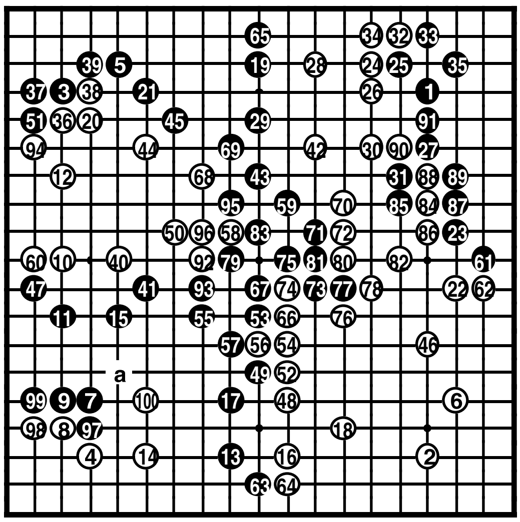
Explorer (W) vs. Many Faces of Go (B), International Computer Go Congress 1993

The wrong choice: Faced with two urgent-looking moves, Explorer chooses *a* and gets into trouble after Many Faces cuts at *b*.



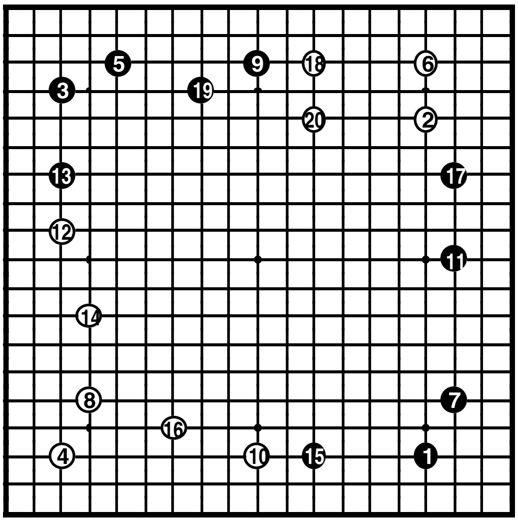
Star of Poland (W) vs. Goliath (B), International Computer Go Congress 1991

The highlight of the Singapore tournament: Goliath kills Star of Poland's fairly safe-looking corner.



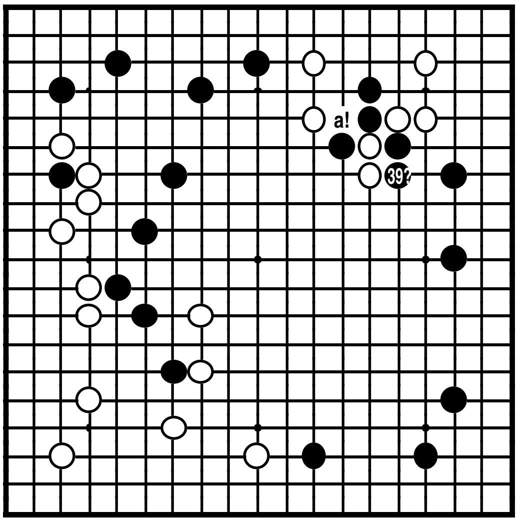
Go Intellect (W) vs. Modgo (B), International Computer Go Congress 1993

A quiet game from the congress in Chengdu: Modgo is a solid program, but Intellect gets far ahead in territory without fighting.



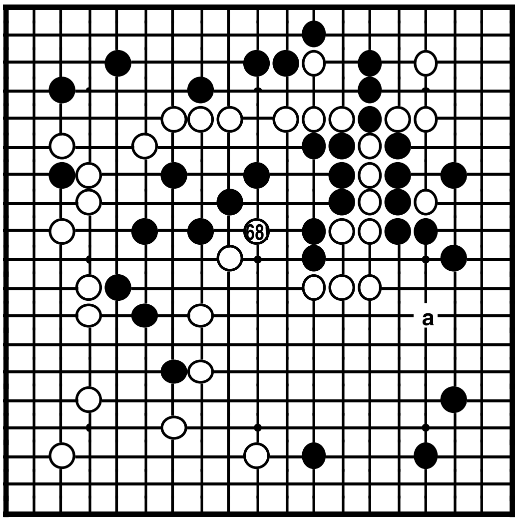
Handtalk (W) vs. Go Intellect (B), International Computer Go Congress 1992

A good opening by both programs, following the principle of extending to the edges from the corners.



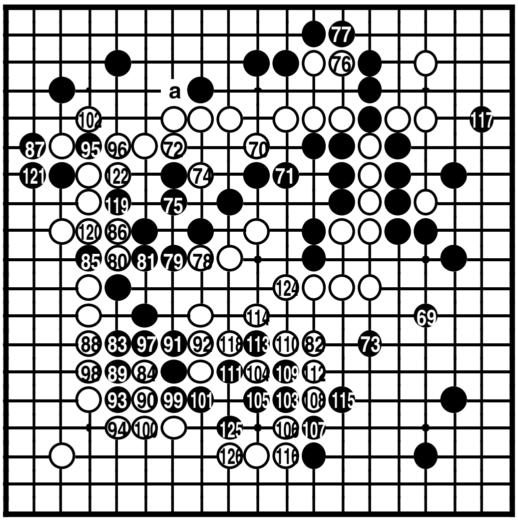
Continuation (1)

Black's tactical error is punished immediately by White's cut at 'a'. A few moves later, the local situation will be reversed.



Continuation (2)

The big black group is neglected…



Continuation (3)

…and finally dies, after missing many chances to live.

Go-related Research in Visual Perception, Machine Learning and Neural Networks

Go has been used as a vehicle for research in visual perception, machine learning and neural networks [Wilcox 79, Enderton 91, Stoutamire 91, Schraudolph 94]. Starting from only the rules of the game, learning programs can typically pick up basic Go principles, such as saving a stone from capture, or making a one point jump.

A most disturbing fact is that since Wilcox' pioneering efforts, none of this research has been applied to a state-of-the-art Go program. The problems of automatically tuning and expanding an existing knowledge base, or learning of new high-level concepts, seem quite different from the ab-initio learning problems that have been researched.

Low level concepts can be learned, but they have already been programmed in the better programs. Neural networks can be trained to play locally good shapes, but have no clue *when* to play it. Such programs fall easy prey to those that know more about tactics and Life&Death.

Parallel Computing

Parallel computing is becoming a popular way of increasing computer power. Several top chess programs use this approach. In Go, the impact of parallel computing is not yet felt: most developers prefer a solid development platform on a PC or workstation to the more experimental environments offered on today's parallel machines. Computer Go tournaments usually require machines on site, which makes it hard to participate with a big parallel machine.

2.4 Overview of the Explorer Program

Explorer is described as an example of knowledge engineering in a Go program. In the course of its evolution, components of earlier versions were continually adapted and reused.

Explorer's Foundation: Smart Game Kernel and Go Modules

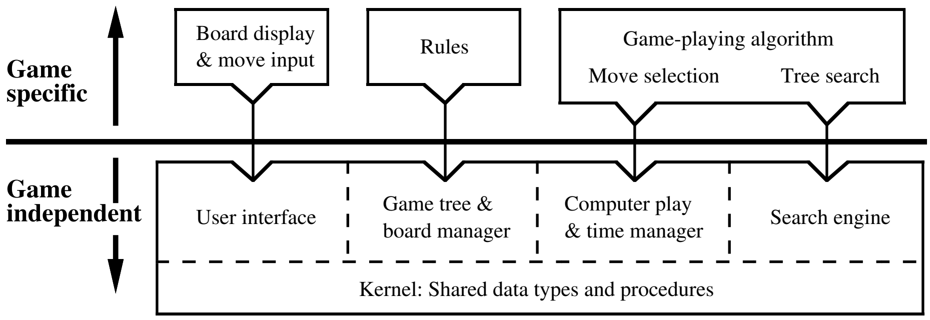


Figure 2: The basic structure of the Smart Game Board [Kierulf 90]

Figure 2 shows the structure of the Smart Game Board. A *kernel* provides basic data types and functions shared by the other modules. The modules of user interface, game tree, board manager, computer play, time manager and search engine contain the game independent parts of a program. These modules provide slots where game-specific functions can be installed. Board display, move input and rules are shared between all programs for the same game. The game-playing part may also consist of several layers: Different algorithms can share a game-specific library. The Kernel and Go modules are essentially those described in [Kierulf 90]. We give a brief summary.

Kernel Data Types

The game-independent Smart Game Kernel offers basic data types and operations: a game *board* consisting of *points*, general purpose *lists*, game *trees* and a *hash table* for detecting transpositions. Graphical user interface elements include a *board display* with markers and labels on points, tree navigation tools, an *overview* window showing several boards at the same time, and a *tree view* showing the structure of the game tree. Game trees created with the Smart Game Board can be *stored* on disk.

Kernel Go Modules

These modules provide Go-specific behavior to the Smart Game Kernel. They are part of both playing (Explorer) and non-playing versions of Smart Go Board. Basic game-specific functions include executing and undoing moves, a legal move checker that handles full board repetition, and a ladder searching routine. The Go kernel's collection of procedures provides a higher level of abstraction, a 'Go language' that is used throughout Explorer.

Explorer Base Modules

These modules expand the 'Go language' with a *point set* type (implemented as a bit map). It provides operations such as computing Connected Components, Border and Interior of a set, and conversion of sets to and from lists of points. Many algorithms can be expressed concisely in this language: In the example, blocks and liberties are computed from sets of all Empty, Black and White points using point set operations.

```
ALGORITHM ComputeBlocksAndLiberties;
BEGIN
    FOR color := Black TO White DO
        blocks [color] := ConnectedComponents (All [color]);
        FORALL block IN blocks [color] DO
            block.liberties := Border (block.stones) * All [Empty];
        END
    END
END ComputeBlocksAndLiberties;
```

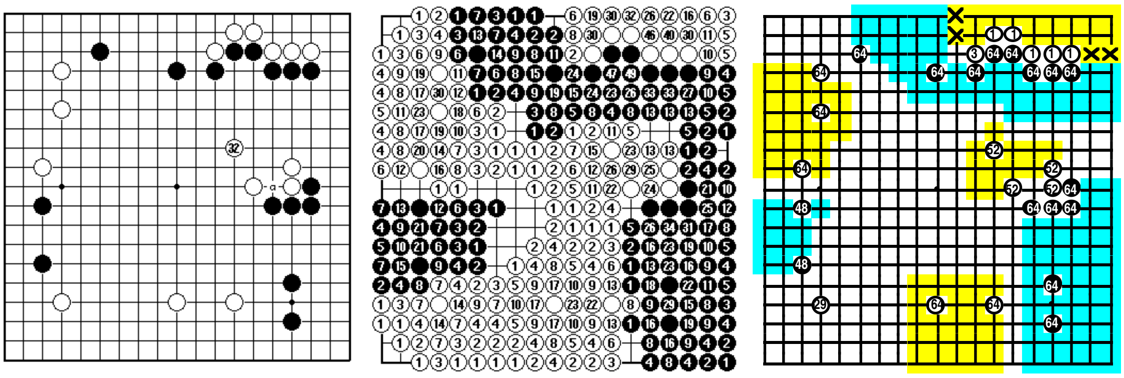
Modules originally designed by Anders Kierulf and Ken Chen [Chen et al. 90] implement *blocks*, *chains*, and specialized search routines for block tactics and Life&Death. Modules added by the author handle *pattern matching*, *dividers* and *connections*.

Blocks and Chains

Blocks are connected sets of stones of the same color. Chains are sets of blocks joined by connections across empty points:

- Two or more common liberties
- One protected common liberty
- A *connection pattern*

The Starting Point: Explorer 1.0-3.3



Influence function of Explorer 3.3, and groups derived from influence function

The old Explorer versions [Chen 89, Chen et al. 90] were based on *influence*: Stones radiated a certain power over nearby empty points. On top of the basic structures blocks and chains, *groups* were defined as contiguous regions of a certain minimum influence. Additional rules handled exceptions near the edge of the board, and made certain that all blocks of a chain ended up in the same group. Localized goal-directed searches were used for tactics (capture/escape block) and Life&Death (kill/save group).

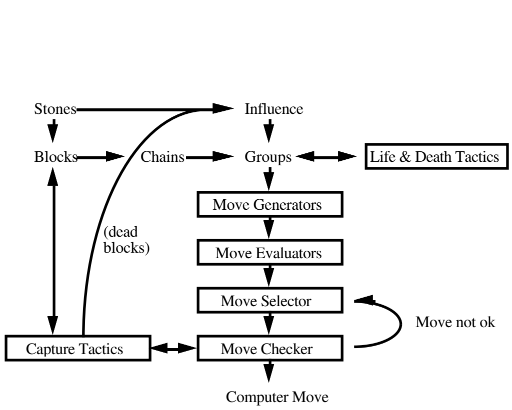
The pattern matching system was implemented as many pages of procedural code. This representation was probably a good decision at the beginning of the project, to get something started quickly. But it had grown to its limits: it was very hard to maintain and improve, especially for people other than the original author.

Transition to Explorer 4

A board partition based on an influence function causes nonlocality and unpredictable behavior. Therefore Explorer 4 introduces a new model based on *dividers* and *zones*. Dividers indicate a gap between stones that is small enough to stop the opponent from connecting through. Blocks and dividers of the same color work together to form the boundary of a zone.

Replacing the other purposes of influence functions, a set of auxiliary board maps stores information for each point and color, such as its distance to the next stone, and sets of near, reachable, dead, or alive points. Hard-coded pattern matching was replaced by a pattern editor and a pattern matching engine. Patterns could now be entered graphically and stored as standard Smart Game Format files.

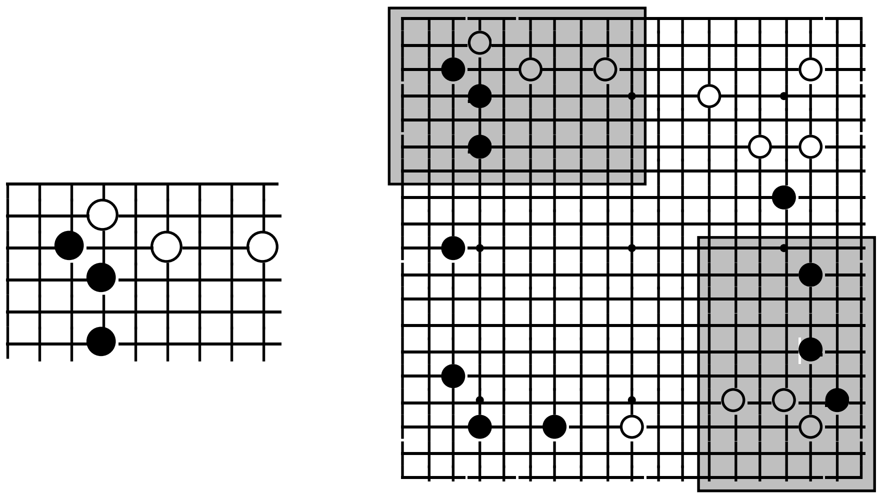
Explorer Main Modules



Control structure of Explorer 1.0-3.3 vs. Explorer 4

These modules implement the top level of the knowledge representation hierarchy. *Zones, groups*, and *fights* represent bigger units on the board than blocks or chains. Explorer modules for pattern matching, local data and move selection will be discussed in the following sections. Even if few of the original components survive till today, the overall control structure of Explorer 4 is still close to that described in [Chen et al. 90] and [Kierulf 90].

2.5 Pattern Matching in Explorer



Corner pattern matching in two places

Patterns are a simple yet powerful way of encoding Go knowledge. Most *fuseki*, *joseki*, and *tesuji* moves are described as patterns in the Go literature. Patterns can be applied in all stages of the game, from opening to endgame. In Explorer, we use patterns for two purposes: To keep a permanent database of standard situations, and to temporarily store results of local analysis.

A *pattern record* has three parts:

- The *pattern map* indicates which state (Empty, Black or White) the points in the pattern must have.

- The *pattern context* specifies additional constraints that a board position must satisfy to match the pattern.
- The *pattern information* contains knowledge which can be applied if the pattern matches.

Definition of Pattern Maps and Matching

We define a *pattern map* as a set of points on a two-dimensional grid, where each point *(x, y)* has a *state* Empty, Black, or White. The set must be connected on the grid graph, but need not have a specific shape. The coordinates *(x, y)* are nonnegative. They are called local pattern coordinates.

The state of a point on a Go board is denoted by *Board (x, y).*

A pattern *p matches* a Go board at location *(x, y)* if the following condition holds: For all $(px, py) \in p$: $state\ (px, py) = Board\ (x+px, y+py)$

Given a set of patterns and a board, we want to find all pairs of patterns *p* and locations *(x, y)* such that *p* matches the board at *(x, y)*.

Symmetries: We want to find occurrences of a pattern obtained by reflecting it about the x-, y-, or diagonal axis, or by swapping colors. Thus a pattern can take up to 16 symmetric forms. We keep only one copy of a pattern and match it against eight copies of the board, which takes care of the mirror symmetries. Color swap is handled directly by the matching procedure.

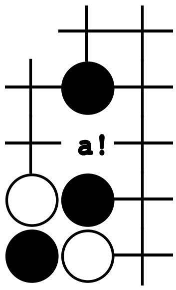
Most Go programs use a hashing scheme for pattern matching. This is efficient if patterns fit into the same small rectangle [Boon 90, Fotland 93]. Our method for matching patterns of variable size (→ Chapter 7, p. 75) uses a *Patricia tree* index [Gonnet 88].

We distinguish three types of patterns:

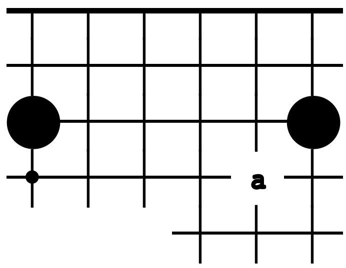
- *Center* patterns can occur anywhere on the board
- *Edge* patterns touch the edge of the board
- *Corner* patterns (e.g. Joseki) are limited to the four corners

Pattern Knowledge

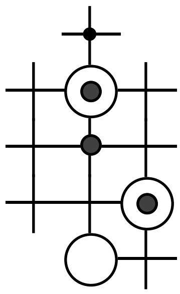
A huge amount of Go knowledge can be encoded in local patterns. Examples are patterns describing eyes, safe connections, or joseki moves. Information stored in patterns includes which moves to play, or *not* to play, e.g. bad shape moves. Move information includes the *motives* for playing a move, e.g. enclose, reduce, make or destroy eye, or urgent shape.



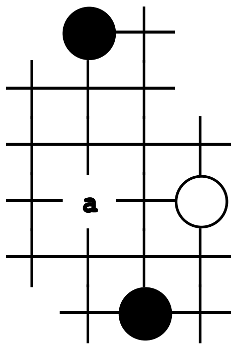
Urgent move



Reducing move



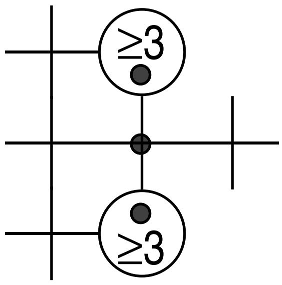
Connection pattern



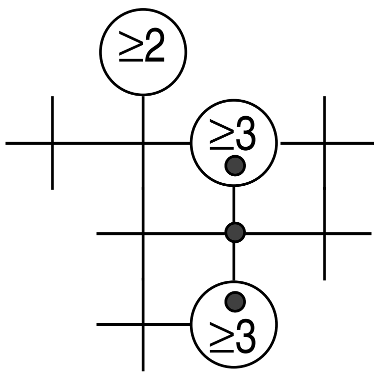
Breakout/Enclosing move

Pattern types

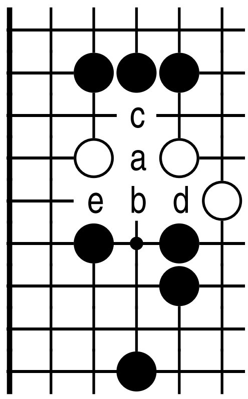
Patterns contain either exact or heuristic knowledge, indicated by a flag in the pattern information. Procedures for proving endgame values or Life&Death status of stones may use only exact patterns.



a) The one point jump, a heuristic connection pattern



b) A supporting stone and a bigger pattern area guarantee safe connection



c) A case where splitting the one point jump is good (Note that pattern (a) matches, while (b) does not)

Example: A one point jump where both endpoints have three or more liberties is usually connected. In theory, the opponent can split the jump by playing on the center point, but that is rarely a good move. Therefore one point jump is a heuristical connection pattern.

Knowledge Acquisition and Maintenance: Building a Pattern Database

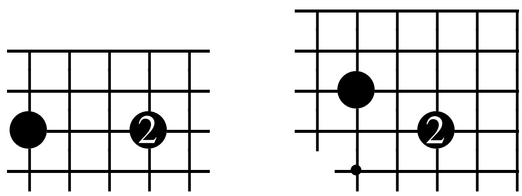
Explorer contains a knowledge base of about 3000 patterns. A pattern editor was developed to build the database. The pattern database slowly approaches a saturation point: few obvious blunders are caused by wrong patterns nowadays. Yet detailed analysis of a typical Explorer game still leads to 5-10 revisions of entries (new patterns, changes to pattern extent, additional evaluation information etc.)

Occasional cleanup work further improves the database by browsing through the Smart Go File describing the database: maintenance tasks include adding continuations of standard sequences, merging similar branches and deleting obsolete patterns.

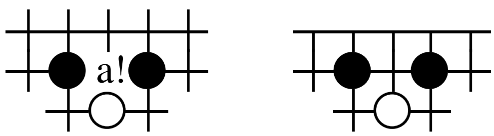
Move Generation using Patterns

A typical midgame situation produces about 500 matches from the 3000 pattern database. Most matches indicate simple connections, dividers or safe extension moves. We increase consistency and relevancy by the following *postprocessing*:

- Use all patterns that define basic structures such as connections.
- Corner patterns are better tuned to the peculiarities of the corner than edge or center patterns, edge patterns are better suited to the edge than center patterns. Therefore ignore edge pattern moves inside a corner pattern, and center pattern moves in an edge pattern.



Standard edge extension is overridden by a specialized pattern in the corner



One point jump is threatened in the center but safely connected near the edge

- Big patterns dominate small patterns. When there is a small pattern for the general case, and a bigger pattern describing a special case, ignore the small pattern.

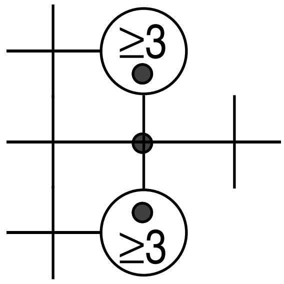
To cut or not to cut…

Example: The crosscut is usually an urgent move, but if the surrounding position is as in the pattern to the right, the cut is not good anymore.

More Postprocessing: Checking the Pattern Context

Constraints [Müller 94] add non-local context to a pattern. The most important constraints involve the liberty count of blocks. In the example, both blocks must have at least three liberties. Since only two liberties each are visible in the pattern map, these constraints are non-local.

Context must be checked for all matching patterns at each move: Even if a pattern map stays valid from one move to the next, the context might invalidate the pattern, e.g. if a crucial outside liberty has been taken.



Pattern context

2.6 Computing Local Data

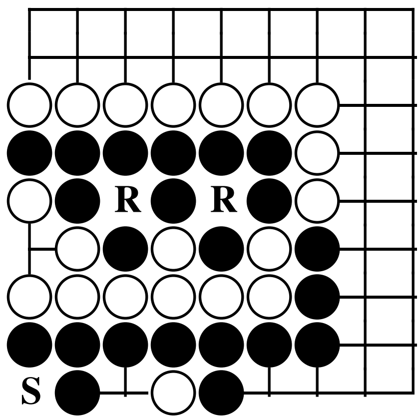
Object Hierarchy

The components of Explorer's board representation are Blocks, Connections, Chains, Dividers, Zones, Groups and Fights. These objects share several attributes which are contained in a common base object type. Fields of the base object type are an ID number, a color, member points and dependency region, a move motive list and a safety estimate.

Basic methods are provided to initialize, free, link, write and display objects. Other methods count the local score, generate moves, and compute the safety estimate. The types of all objects used in board representation are extensions of this base object type. They extend or override behavior of the base type.

Legal and Repetition Moves

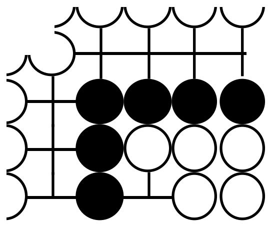
Legal moves are computed using the Smart Game Board function *MoveIsLegal*. This routine detects the cause of illegality, including full-board repetition detection [Kierulf 90]. Repetition moves may be generated by tactical and Life&Death move generators. If such a move gets the best evaluation, Ko threats will be generated (→ Section 2.7, p. 36).



Illegal moves (White to play):
'R' … repetition (Ko),
'S' …single stone suicide.
Moves on all other empty points are legal, including a large scale suicide move

Blocks

Blocks are the basic elements of board representation. Attributes of blocks include stones, liberties, connections, and references to the block's chain and group. The tactical status of blocks with few liberties is computed by capture search, with each player moving first. A block can be tactically safe, but strategically dead, or vice versa:

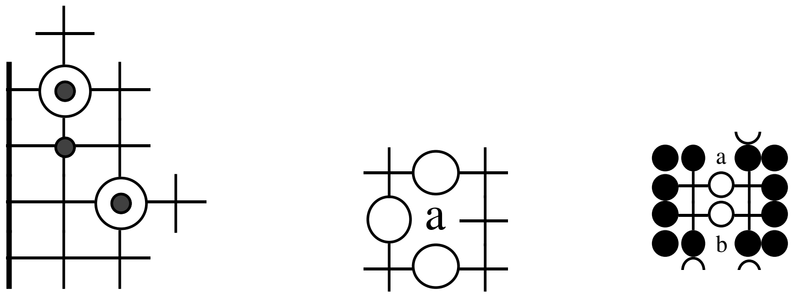


Tactical vs. strategic status of blocks

The black block has many liberties and is tactically safe, but it has only one eye and is strategically dead. The white five stone block is tactically captured by Black, but strategically part of white territory.

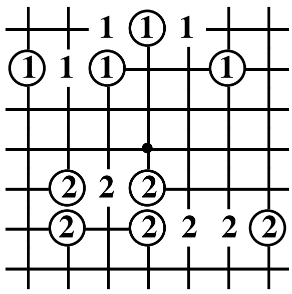
The stability of a block is computed in several steps. The initial stability depends on the liberty count, after computation of chains and groups the stability is updated.

Connections, Potential Connections and Chains



Connection pattern near the edge, connection through safe shared liberty, and connection through two potential connections

Connections across empty points define *chains* of blocks. Explorer recognizes three types of connections: connection patterns, connection through a safe shared liberty, or connection by two independent potential connections. *Potential connections* are shared liberties or *potential connection patterns*.

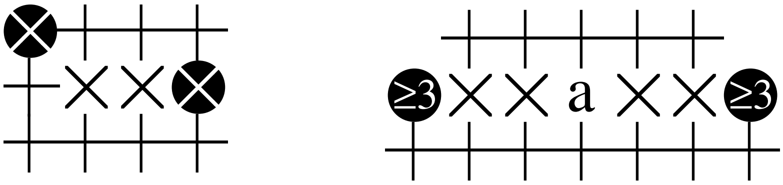


Chains formed by common liberties and connection patterns

A *chain* is a set of blocks joined by pairwise independent connections. A player can prevent the opponent from cutting off any block from a chain. Important attributes of chains are their member blocks and connections, a stability estimate, and the *group* to which the chain belongs.

Boundary Pieces: Dividers and Potential Dividers

Specific small gaps between blocks form a *divider* or *potential divider:*

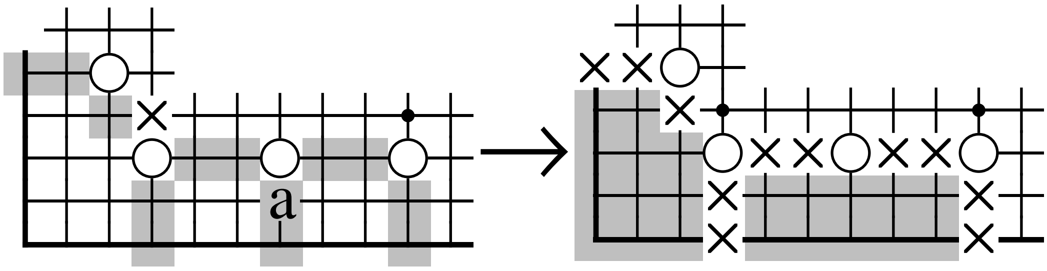


A divider, and a potential divider (move a *generates two dividers)*

A *divider* is weaker than a connection: its purpose is to stop an opponent's connection from one side to the other, not to connect one's own stones. Dividers of both players may cross each other in a crosscut pattern, but by definition a divider cannot cross an opponent's connection.

A *potential divider* can be transformed into real dividers by one move. It is used to recognize the borders of potential territories.

Zones



Dividers, and a resulting partition into white zones
(note the contiguous zone on the lower side)

A *zone* generalizes the notion of (safely surrounded) territory: a connected set of points bounded by blocks, dividers, and potential dividers of one color is recognized as a unit. The interior of small zones might contain more dividers and potential dividers, because it makes no sense to split these areas further. Adjacent small zones with many dividers and potential dividers are merged. The zones adjacent to divider *a* in the figure provide an example.

Zones give a heuristic partition of the board from one player's point of view: A single zone is not too big, because many points including potential dividers are used for partitioning the board into zones. It is not too small because tightly controlled areas (where each point is a divider point) are grouped into a single zone. One zone can contain or overlap several opponent zones.

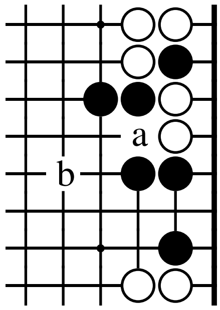
For the eye status of a zone, a total of four estimates is computed: minimum and maximum of existing and of potential eyes in the zone. The safety of zones is classified as safe territory, potential territory, threatened, neutral or dead. Zones are the units of board control in Explorer. Each player strives to secure own zones and neutralize opponent zones. Zone-related moves are decisive in quiet, territory-oriented games.

Groups

Groups are sets of chains and zones of one color enclosed in the same opponent zone. Important attributes of groups are connectivity, safety, and the eye status of member zones.

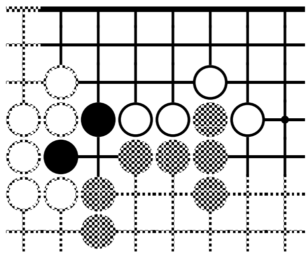
Groups are Explorer's units of defense. Each group needs two eyes to be safe. Surrounded unsettled groups are analyzed by a Life&Death routine [Kierulf 90], to check if they can live, or be killed.

A group may contain several own zones providing eye space for the group. Each group is contained in one opponent zone, which provides the context for attacking and saving the group.



A white move 'a' splits Black into two weak groups; a black defensive move such as 'b' yields a connected, more stable group.

Group-related move motives are decisive in many games: Adjacent weak groups of the same color can connect to create a single safer group, or be subjected to a splitting attack. *Junction* points between weak adjacent groups of opposite color combine attack and defense.



Example of fight (safe stones next to fight are shown dimmed)

Fights

A *fight* is an area containing a set of unstable blocks of both colors. The local situation is too vacillating to analyze it in terms of individual blocks, chains or groups. Fights are analyzed by a variant of capture search: the goal is to capture one of the opponent *vital block's* while saving all own vital blocks in the fight.

2.7 Static Board Analysis and Move Selection

Static analysis creates a full board description by combining all local data. This full board information is used for improving move generation and selection.

Game Stage

Explorer distinguishes four stages of the game: Opening, midgame, endgame and final stage. Some move generators are influenced by the game stage: bad shape moves are strongly penalized during the opening, but only slightly devalued in the endgame. In the final stage only, dame points are filled. Dead groups are captured in the final stage, if it does not cost points under the currently used rules.

Score

Safe territory is counted to find the game score. For potential territories and areas near to one player's stones, a reduced score is used.

Generate Moves and Move Motives

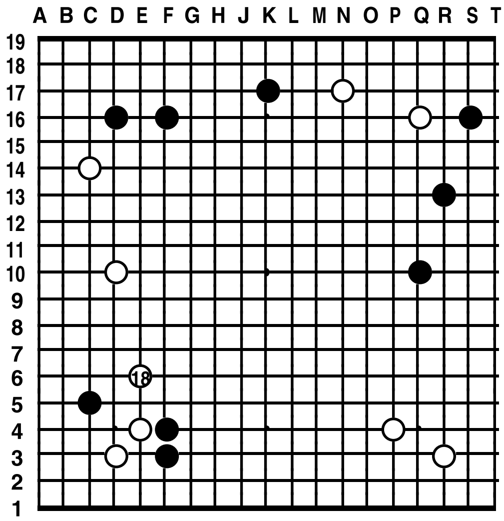
There are two types of move generators: *Local* generators are bound to a specific object, such as a group, a zone or a block. These generators propose moves related to that object, e.g. defending a group, or capturing a block.

Type of object	Move generators
Zone	extend, reduce, defend, invade
Group	attack, defend, expand, cut
Block	escape, capture, stabilize

Type 1 move generators

Global generators rely on full board information. They handle interactions between objects, and moves that cannot be assigned to a single object. Examples are playing a double attack, or occupying a junction point between two territorial frameworks.

The result of calling a move generator is a set of *move motives*. Move motives consist of a motive type, the move, and a value. They are Explorer's units of evaluation.



Move W E6:
Flags:
Normal-Move Possible Legal Junction
No-Check-Move

Motives:
Position Urgent = 400
Zone W K10 Expand = 73
Zone W B7 Expand = 34
Group W C3 Expand = 20
Group W C3 Save-Group = 328
Conn-Extension W E6 Expand = 0

Total Value: 801

Pattern motives:
Connect Urgent Expand Reduce

Modifier: Normal

Move motives and move value

Move Values

After collecting motives from all objects, *filtering* removes misplaced motives, e.g. extension moves inside territory or tactically bad moves. Filters use full-board information unavailable to local move generators.

Motives are sorted by move, and their values are added to find a *total value* for each possible move. To avoid multiple contributions for *similar* motives, the sum of similar motives is replaced by their maximum value. In the example above, only the highest-valued of three *Expand* motives is included in the total move value.

Select Moves

The highest ranking move that survives a *move checker* is produced as best move. The move checker is a last filtering step: its objective is to avoid tactical blunders proposed by non-tactics-aware generators.

If the best move is forbidden due to repetition, its value is reset to zero, and a special Ko threat generator is called. The total values of Ko threat moves are recomputed, and Select Moves is restarted with the new move values. If no move with positive move value can be found, the program *passes*.

3

Combinatorial Game Theory and its Application to Go

In this chapter, we survey combinatorial game theory (CGT) as the study of *sums* of independent games. We introduce the techniques of CGT to simplify game values, evaluate sums, and find a move in a sum game. Of special importance in Go are the technique of cooling and the approximate sum game playing algorithms Thermostrat and Sentestrat. We show that Go endgame play can be interpreted as a combinatorial game theory problem.

3.1 Combinatorial Game Theory, or Playing Sums of Games

Combinatorial game theory investigates the following problem: given two or more (relatively simple) independent games, play the (complex) sum of these games well. If possible, restrict analysis to single games, avoid actually computing the sum.

The complexity of computing a good or optimal move in a sum game may grow exponentially with the input size. Often, though, we can take advantage of the local game structure to dramatically reduce the effort required for solving the sum game.

Basic Definitions and Notation

For introductions to combinatorial game theory, see [Conway 76, BCG 82, Berlekamp 91, High 92]. 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. 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. $0 = \{|\}$, $1 = \{0|\}$, $n+1 = \{n \mid \}$. Negative numbers are the inverses of positive numbers, e.g.

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

There are games corresponding to fractional numbers too, such as $1/2 = \{0 \mid 1\}$ and $1/4 = \{0 \mid 1/2\}$. The *Sum* of games G, 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, inverses and sums as defined above have the mathematical structure of a group. In a sum game $G = A+B+C+\ldots$, the single games $A, B, C, \ldots$ are called *subgames* of G.

Loopy games: It is possible to define a game recursively in terms of itself, as in $G = \{G \mid G\}$. The strongest results of the theory apply only to *loopfree* games. In Go, loopy games appear in Ko fights. We will assume all games are loopfree from now on, except for those sections where we explicitly discuss Ko.

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

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

Comparing Games

The following definition gives a 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 is used for pruning dominated options (see next section).

Canonical Form

Any loopfree game can be brought into a unique canonical form by removing dominated options and reversing reversible options.

1) Remove dominated moves: If $G = \{A, B, C, \ldots \mid \ldots\}$, and $A \geq B$, then A dominates B, and B can be omitted: $G = \{A, C, \ldots \mid \ldots\}$
2) Reverse reversible moves: If $G = \{A, B, C, \ldots \mid \ldots\}$, and A has a right option $A^R \leq G$, then Left's move to A reverses through Right's move to A^R: If $A^R = \{A_1, A_2, A_3 \ldots \mid \ldots\}$, then $G = \{A_1, A_2, A_3, \ldots, B, C, \ldots \mid \ldots\}$.

Analogous definitions hold for Right's moves. An example of a reversible move is shown in Chapter 4, p. 50.

Leftscore and Rightscore

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

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 as follows:

$$\text{Left incentive} = G^L - G$$
$$\text{Right incentive} = G - G^R$$

Incentives can be used to find an optimal move in a sum game without computing the sum, by proving that a move is optimal. The *incentive* of a move can be computed locally: If $G = A+B+C+\ldots$ and we move from A to A^L, the Left incentive becomes $G^L - G = A^L+B+C+\ldots-(A+B+C+\ldots) = A^L-A$. Thus the incentive depends only on the subgame A containing the move. It is easy to prove that all moves with dominated incentives can be pruned: Assume $A^L-A > B^L-B$. Then $A^L+B+C+\ldots = G+(A^L-A) > G+(B^L-B) = A+B^L+C+\ldots$, so the move to B^L is dominated. We may also prune all but one of several moves with identical incentives. Often only one candidate move remains.

Example: In the game $1|0 + 2|0 + 3|0 + 4|0 + 5|0$, the incentive for the move from 5|0 to 5 or 0 dominates all other incentives. Therefore it is an optimal move for both players in this sum game.

Mean

A measure how many points a game is worth on average, when playing many copies of the same game. This must not be confused with the value of a move which is measured by the incentive. The mean is linear: *mean* (A+B) = *mean* (A) + *mean* (B)

Cooling

Cooling is a technique for simplifying games 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*.
The game G cooled by the amount t is denoted by G_t and is defined as follows [BW 94, p. 50]:

$$G_t = \{G_t^L - t | G_t^R + t\},$$

unless for some $\tau < t$, $\{G_t^L - \tau | G_t^R + \tau\}$ is infinitesimally close to a number x,

in which case $G_t = x$

The special significance of cooling for Go comes from the fact that *cooling by 1* is an order preserving one-to-one mapping of *even elementary* Go positions (→ [BW 94, p. 52/53], or Glossary, p. 102) to simpler ones. Cooling turns typical endgame values into infinitesimals, sums of which are well understood.

Many of these games sum well, avoiding the usual combinatorial explosion of sum computation. A partial order on incentives of infinitesimals is often enough to assure optimal play without computing the sum.

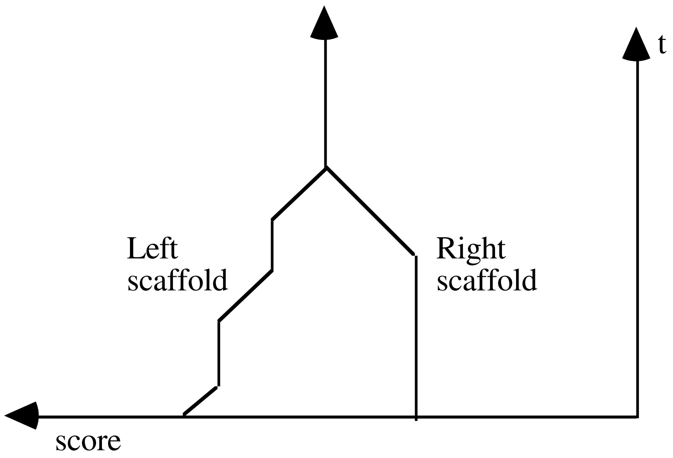
Cooling by 1 helps less for hot summands. It is most effective when games already have low temperature.

Temperature

Temperature is an estimate of how urgent it is to 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.

Thermographs

A *thermograph* is a graph showing the scores of a game against all amounts of cooling, i.e. *Leftscore* (G_t) and *Rightscore* (G_t) for $t \geq 0$. The score axis is traditionally drawn with values increasing to the left.



Sample thermograph

Switches

Switches are the simplest 'hot' games, with a *temperature* > 0. These games end after one move by either player. A *switch a* | *b*, with numbers *a*>*b*, has mean (*a*+*b*)/*2* and temperature (*a*-*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.

Symbols for Special Games

The game *0* | *0* is called * in combinatorial game theory, *dame* in Japanese Go jargon. It is a game with mean *0* and temperature *0*.

Game Types in Combinatorial Game Theory

The mathematical theory of games defines four basic types of games *G*:

$G > 0$	positive game	Black wins no matter who plays first
$G < 0$	negative game	White wins no matter who plays first
$G = 0$	null game	The second player wins
$G \not\gtrless 0$	fuzzy game	The first player wins

Game Trees of Combinatorial Game Theory

In contrast to the game trees of minmax search, game trees for combinatorial game theory must contain successive moves by the same color, because the opponent might *tenuki*, i.e., play in another local game. Move generation is locally exhaustive in principle. *Terminal* or *stopping positions* are positions without a good move. In Go this usually happens when all points are occupied or have become territory.

Approximate Algorithms for Sum Game Evaluation

We distinguish *global* algorithms that may need to compute the sum game to make a move decision from *local* algorithms that work on local games, without computing sums of games. A local algorithm cannot always determine the optimal move, but it may be much easier to compute. Furthermore, some local algorithms such as *Sentestrat* or *Thermostrat* guarantee a result close to the optimal score [BCG 82].

Because summing games is a complex operation, we want approximate algorithms that can play a sum game without computing the sum. The goal of approximate algorithms is to find a good move in a reasonable time.

The Hottest Subgame Algorithm

This elementary algorithm can play a sum of many types of simple games (such as switches) correctly. It fails to play perfectly, or even guarantee a result close to the mean value, in sums of more complicated games. Berlekamp gives an example in [Berlekamp 92, p. 60]: the game $27 \mid \{33 \parallel 1 \mid -3 \parallel\mid -16\} \parallel -1$ has mean 0 and temperature 1. If Right follows a hottest game strategy, Left can win by an ever increasing amount in a sum of more and more copies of the game, while all these sums have mean 0 and temperature ≤ 1.

```
ALGORITHM HottestMove (sum: TSumGame): MOVE;
BEGIN
    hottestTemp := -infinity; hottestGame := NIL;
    FOREACH game IN sum DO
        temp := Temperature (game);
        IF temp > hottestTemp THEN
            hottestTemp := temp; hottestGame := game
        END
    END
    RETURN MoveAtTemperature (hottestGame, hottestTemp);
END HottestMove;
```

Improved Temperature Based Algorithms: Thermostrat and Sentestrat

Thermostrat [BCG 82, Berlekamp 92] improves on the simple hottest game algorithm. It guarantees play close to the optimum if the temperature is low. It bounds the maximum error by the temperature of the hottest subgame. Thermostrat is an algorithm based on the *thermographs* of subgames. It is asymptotically optimal: the error can be bounded by a constant for playing a sum of arbitrarily many copies of a game.

Sentestrat [Berlekamp 92] is a simpler algorithm which is also asymptotically optimal. All opponent moves that raise the temperature above an *ambient* are treated as *sente* and answered. Compared with Thermostrat, this reduces the complexity of the algorithm. Sentestrat needs only the top part of a thermograph.

3.2 Go as a Sum Game

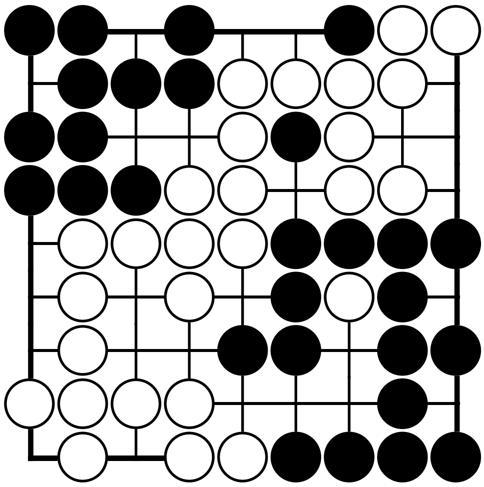
Computing the game-theoretical value of a Go position by exhaustive search is feasible only for the simplest positions [Thorp/Walden 72]. Even with today's advanced search techniques, the effort grows exponentially in the number of possible moves. If there are more than 20-30 possible moves, exhaustive search takes too long. Yet a late endgame position that looks trivial to an experienced Go player often contains over 100 reasonable moves that must be checked to prove the value of a position.

A Go position usually consists of several local scenes that can be analyzed individually. In the opening, these scenes can be far apart, and their influence on each other may be weak. A better partition occurs late in the game, when there are walls of stones dividing the board. A move cannot have any influence across a wall of safe stones.

Independence holds only for games where Ko loops are irrelevant for local play. The influence of Ko on the theory is discussed in Section 3.3, p. 43.

Board partition improves when many safe walls are on the board. In the opening and midgame, any partition can be approximate at best. In the endgame, the partition gets more precise when the status of all big groups has been settled, and the outlines of territories are clear. When stones become *immortal*, significant parts of the board definitely belong to one of the players. The *connected components* of the rest of the board form *local games* that are independent from each other.

If each local game is simple enough to analyze completely, combinatorial game theory can compute an optimal move for the full board position.



Late endgame position suited for exact analysis

To apply combinatorial game theory to Go, we make the following assumptions: Positive scores are good for Black, negative scores good for White. Most of our analysis will be identical for all popular variants of the rules. When necessary, we will discuss rule-specific differences. Per default, suicide is forbidden.

We model play from a given starting position to the last valuable move: better than *dame* in Japanese rules, better than filling territory in Chinese rules. Local games with integer value, or integer plus a number of *dame*, are treated as terminal positions.

What Kinds of Positions can be Analyzed By the Local Games Approach?

The sum-of-local-games approach can handle a broad spectrum of situations, from a heuristic combination of selective local search (Chapter 4) to mathematically exact analysis of late endgames (Chapter 5 and 6). Differences between exact and approximate approaches appear in rules of board partition, exhaustive vs. selective local tree search and precise vs. heuristic evaluation of local terminal positions.

Positions Suited for Exact Analysis

Berlekamp and other mathematicians have analyzed local and full-board endgames by hand. [BW 94] contains a set of more than 100 *node rooms* and their mathematical values. The book contains an analysis of *corridors*, groups invading several corridors that need a connection at a later stage, and several other generalizations. A set of full-board problems illustrates play in sums of local games.

The analysis of full board positions as a sum of local games depends on the independence of local areas, or at least on an observation that dependencies do not affect the value of the sum game. In many cases in [BW 94], this is not obvious, and has to be established through case-by-case analysis. In our model, we will pose strict conditions to ensure independence of local games. For our *Computer Go Test Collection*, we designed a set of problems with endgames equivalent to those in [BW 94] and with clearer independence of local games.

Playing Computer Go as a Sum of Local Games

We can now outline a procedure for playing Go as a sum of local games:

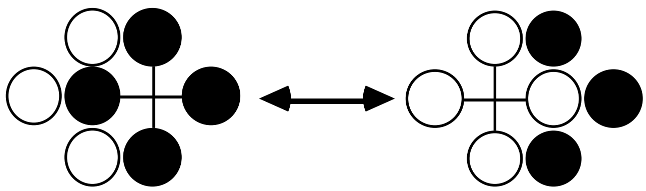
- Board partition: find safe blocks, safe territories, and local areas
- Generate local game trees in each area
- Evaluate local terminal positions
- Transform local game trees into mathematical games (and simplify games)
- Find an optimal move in the sum game and play it

We will develop algorithms following this outline in the rest of this thesis: heuristic algorithms for playing the entire game, and exact algorithms for late endgame positions.

3.3 Local Games with Ko Loops

The classical model of combinatorial game theory assumes loopfree games to derive its theorems. Games with loops, which are called Ko in Go, have a 'completely different' theory [BCG 82, Chapter 11]: for example, the mathematical group structure is gone for sums involving such games. Since Kos appear so often in local game analysis, we need to deal with it in our program.

Developing the theory of loopy combinatorial games and applying it to Go is an ongoing research effort [Berlekamp 92, Moews 93, BW 94, Berlekamp/Kim 94]. It seems that two fundamentally different classes of Ko positions exist. Many concepts of combinatorial game theory can be extended to an astonishingly large class of Kos: mean value, temperature, Leftscore and Rightscore of such games can all be defined independently of Ko threats for these games. For the other *hyperactive* class of games, these values depend on available Ko threats, and a general theory seems more difficult. [Berlekamp 92] extends the algorithms Sentestrat and Thermostrat to incorporate loopy games and absolute Ko threats.



Ko

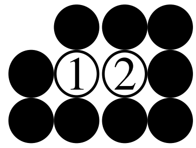
The number and size of Ko threats are side effects of endgame play that are not visible in the mathematical game values. Using Chinese rules, victory may depend on playing the endgame (including dame) in such a way as to leave the maximum number of threats for the final half point Ko fight [Berlekamp/Kim 94].

A complete theory of Kos would have to deal with many intricate problems: for example, it must handle the tradeoff between playing a move for profit and saving it as a later Ko threat, when to play a point losing threat, when to generate or remove Ko threats instead of making points [Popma/Allis 92], or how to play small Ko threats in a small Ko fight and leave the big threats for a potential later, bigger Ko.

Removing Loops to Calculate Bounds on Ko

Simple-minded backup of values through a loopy graph leads to an infinite loop even if loopy moves are irrelevant for computing the game value. To calculate bounds on loopy games we must find loopfree games that approximate the loopy game.

As a preliminary step, we remove loops where one player makes all the moves. In such loops the player needs to commit suicide with as many stones as there are moves in the loop. Single-player loops therefore only happen if suicide is allowed.



3 at 1
Suicidal two-move-White-only loop

Since a single-player loop just loses stones, there must be at least one bad move in the loop. The naive way of disabling one of the suicide moves is dangerous, because it may be a good move. A safe way of removing a single-player loop is to disable the last move closing the loop. Each remaining loop will contain moves by both players.

Sidling [BCG 82, Chapter 11] is an iterative algorithm to improve upper and lower bounds on a loopy game. In its general form the algorithm starts with loopy games **on** and **off** that are not handled by our software. By restricting moves of one player we obtain other bounds on the game value that can be used to initialize sidling [Müller/Gasser 94]. If after sidling the two bounds are identical, good local play does not depend on Ko and we have determined the exact game value.

4

A Heuristic Sum Game Model for Computer Go

In this chapter we develop a heuristic sum game model for Computer Go through a systematic study of *approximations* to board partition, exhaustive search, scoring and evaluation of sum games. The result is a new framework for embedding the heuristic knowledge of Go programs discussed in Chapter 2. The basic concepts of this model have been implemented and tested in a prototype Go program, Explorer 5. The implementation of Explorer is discussed in Chapter 7. A complementary view of this chapter is as a study of extensions to the exact model for endgame play that will be developed in Chapters 5 and 6.

4.1 A Sum Game Model for the Entire Game of Go

When applying a local game model to the opening or midgame, the biggest obstacle is the fact that the board cannot be partitioned into independent subgames. Few areas are completely surrounded, and surrounding stones are not yet invulnerable. We develop heuristics for board partition that split the imperfectly surrounded areas of the opening and midgame.

An unavoidable side effect of heuristic partition are dependencies between the resulting local games. Simple strategies for dealing with dependencies are pretending the games are independent, or re-searching a bigger merged game. We discuss two more sophisticated techniques for dependency analysis: proving that dependencies do not affect the game value, and Wolfe's analysis of blocks adjacent to many corridors [Wolfe 91a]. *Ko* fights are another source of dependencies: They add cycles to local game graphs, and introduce non-local effects through the repetition ban rule.

When local games are independent, evaluated local game graphs contain all relevant information. To deal with dependencies between games, we introduce the notion of a *context* of a local game: an abstract representation of the relevant features of the rest of the board.

Even after heuristic partition, areas remain that are too big for an exhaustive search by a computer Go program, which must produce a move in a few seconds or minutes. In such big areas we use selective search. Tree growth is controlled by limiting the number of moves generated, and by stopping the search before reaching a terminal position.

Evaluation of local midgame positions is more complex than the territory evaluation used for the quiet terminal positions encountered in the endgame. To account for issues such as Life&Death and tactical stability of stones, we develop an iterative search model where modules covering different aspects of the game work together to expand and evaluate nodes of the same local game graph. The model is inspired by quiescence search, a method popular in minmax search programs.

There are two levels of search control: for the whole sum game, search time must be distributed between local games. For each local game, we must decide which nodes to expand and which expert modules to use for search and evaluation. A postprocessing stage handles detected dependencies between games.

Computing the exact sum of hot games is impractical, and may even be useless when dealing with heuristic local game values. Faster approximate algorithms for sum game evaluation such as those of Chapter 3 better match the needs of our framework.

A Sum-of-Games Framework for Computer Go

Based on our experience with both a specialized program for exact endgame calculation and a program for the entire game of Go, we state the following goals for a program based on the sum-of-games approach:

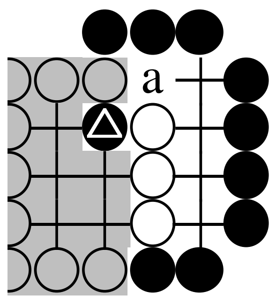
- It works in all stages of the game, not only the late endgame.
- It replaces most of the guesswork of a static evaluation function, which invariably leads to worthless moves and other blunders, by local search. The effect of moves is compared with the effect of the opponent's *local* moves.
- It improves locality and modularity of the program through the collaboration of modules for different aspects of the game.

4.2 Context and Constraints: Augmenting the Local Game Information

The *context* of a local game is an abstraction of relevant features on the rest of the board, and of the history of the local game, especially prisoners and Ko status. Varying the context leads to *versions* of a local game: different sets of nodes in the game graph are valid, and the evaluation of nodes may change, too.

Constraints model a *dependency* between a game and its context. An evaluation of a node that relies on constraints becomes invalid if one of these constraints is violated during play on the rest of the board. Typical constraints are predicates involving the number of outside liberties, outside eyes, or potential connections of blocks to safe blocks.

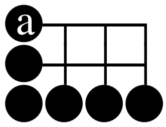
Constraints are used to guide search. They are generated by one expert module for use by the others. For example the Life&Death expert sets up conditions that need to be respected during endgame search. We call a constraint *endangered* if it can be violated by the next move.



Context affects play and evaluation of a local game

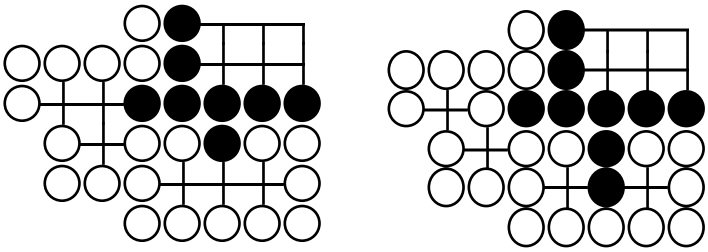
In the example, the value of the white territory depends on outside moves: Three different games result depending on whether:

- *a* is occupied by White
- *a* is occupied by Black but other outside liberties remain
- all outside liberties are occupied by Black



Context =
OutsideLiberties (a, 2)
& OutsideEyes (a, 0)

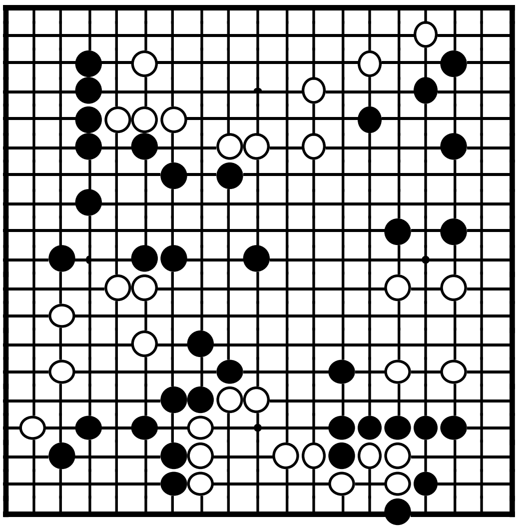
Local game with context information: Six points in the corner



Two matching instances of the same local game

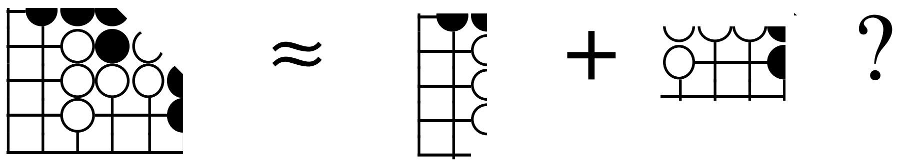
The next example shows that context is a useful abstraction mechanism: The evaluation of the six point corner area depends on the number of outside liberties and eyes of the black stones. Their exact location on the board is irrelevant. A small number of different contexts covers all possible developments on the outside.

4.3 Heuristic Board Partition



In this position from [Kageyama 78], safety of blocks and territories cannot be proven

The requirements for proving blocks and territories safe are very strict, and are rarely fulfilled before the endgame. In the fairly typical example position, no block or territory on the board can be proven safe. The whole board becomes a single 'local' game. Such *partition failure* is in contrast to the capabilities of human players and heuristic programs, who can produce a reasonable approximate partition.



A good approximate partition?

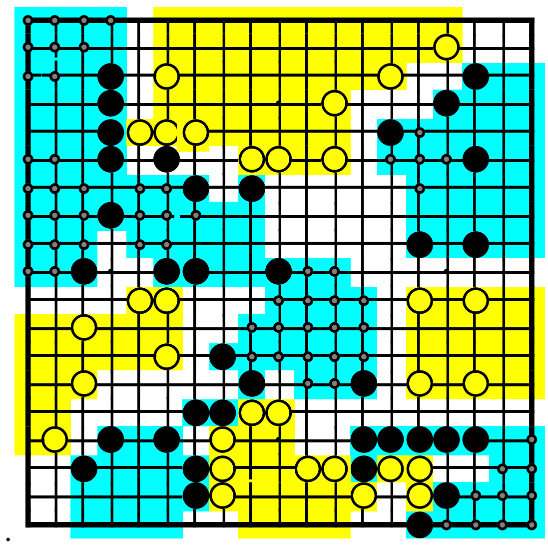
How can we find a good heuristic board partition? Safe stones, territories and dame points make ideal boundaries because they are not affected by play on the rest of the board.

For approximate partition we choose additional empty points that have little interaction with the rest of the board, and strong stones that are unlikely to be captured later. Suitable empty points are those classified as *neutral* by a Go program, or points in a *divider* such as a bamboo joint. Strong blocks that help to partition the board are those that have many liberties, or are connected to safe blocks in a chain.

Another partitioning method relies on conservative constraints: the assumption that a player will always defend the boundaries of a loosely surrounded area. These boundaries can then be used for board partition.

Algorithm for Heuristic Board Partition

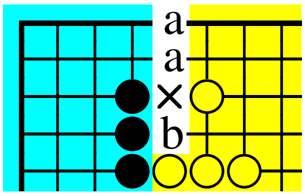
To keep the partition as exact as possible, we first execute the exact board partition algorithm (→ Chapter 5, pp. 61-65) to find *immortal* blocks, safe territories and independent small endgame areas. In a second step we apply heuristic partition to the remaining big areas.



Explorer's heuristic partition of Kageyama's position

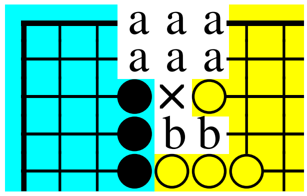
```
ALGORITHM ComputeHeuristicBoardPartition;
BEGIN
    ComputeExactBoardPartition (safe, dame, smallAreas, bigAreas);
    (* smallAreas are already suitable for search, big areas need to be partitioned heuristically *)

    boundaries := ComputeHeuristicalBoundaries;
    FOREACH bigArea IN bigAreas DO
        FOREACH heuristicArea IN ConnectedComponents (bigArea - Area (boundaries)) DO
            DefineLocalGame (heuristicArea, bigArea) (* Find type of area, initialize *)
        END(*FOREACH*)
    END(*FOREACH*)
END ComputeHeuristicBoardPartition;
```

Endgames Adjacent to Loosely Bounded Territory



Endgame areas a and b between adjacent territories.

The divider marked × separates the two areas

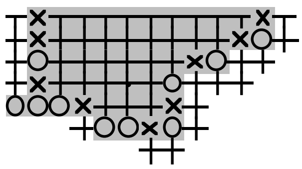


Expanding the initial area

The endgame areas next to loosely bounded territories as computed by board partition are too small. For search and evaluation we extend the *initial areas* of endgames into adjacent territories. The exact initial boundary between territory and endgames is not critical because we allow endgame play to expand into the territory dynamically during a search.

Loosely Bounded Territories and Potential Territories

Territories loosely surrounded by blocks and dividers are common results of heuristic board partition. They can be reduced by successive moves from adjacent endgame areas.



Territory loosely surrounded by blocks and dividers

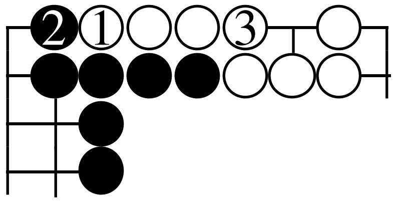
Endgame areas adjacent to the same territory are usually dependent: a multi-move invasion sequence from one endgame will eventually reach the area of another endgame. Often, dependencies between these sequences will not affect the value of the initial position much.

In practice, we can assume that these endgames are independent for an initial analysis. If the overlap turns out to be critical, we will recognize it during postprocessing. Otherwise we can treat the total area as the approximate sum of a constant territory in the middle plus several adjacent endgames. We make the *constant context assumption* that during analysis of one endgame no other adjacent endgame is played.

Another type of dependency concerns endgames adjacent to two or more territories: A possibly complex adjacency graph of territories and endgames results. Again, an independence assumption allows us to approximately analyze such situations.

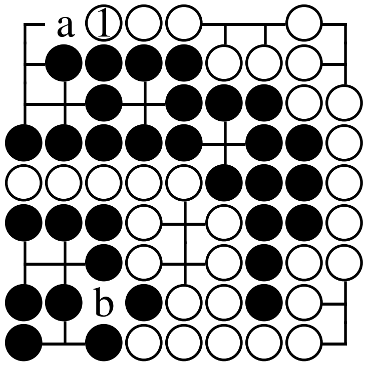
Reversibility Improves Independence

If an invader's move can be blocked 'in sente', that assures that no deeper invasion occurs in normal play.



A reversible move

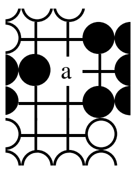
If White plays *1*, she should answer Black's move *2* with *3*, because the game after the *1-2* exchange is worse for White than the original game. This means that Black can complete his wall either at *1* or at *2* as a 'free' side effect of endgame play, without danger of losing points.



Reversible move as a Ko threat

Such reasoning is not applicable any longer when Ko fights are involved: In the lower left corner Black has just taken the Ko to the right of *b*. Now a white move at *1* works as a threat. If Black blocks at *a*, White captures at *b* and kills the Black group because Black has no Ko threats. Therefore Black cannot play his 'free' move at *a*, but must connect the Ko. White plays at *a* and wins the game by one point.

Splitting a Local Game

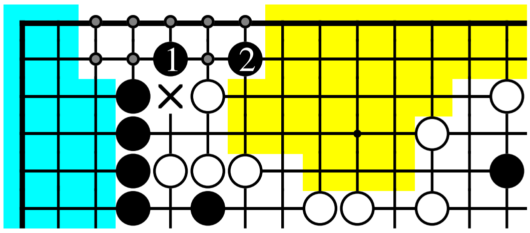


Black 'a' splits an endgame area into three parts

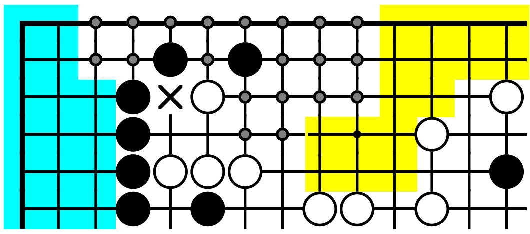
Moves executed during local search often cause more board partition: a local game is split into a sum of several subgames. Split off parts are smaller, and of simpler structure. Many parts can be statically classified as territory or *dame*. This decomposition helps to fight combinatorial explosion.

Expanding the Local Area During Search

When search reaches the boundary between an endgame and territory we must extend the area of the endgame.



Search reaches neighbor territory …



…and the endgame area expands

Formally, an extension operator *ExtendArea (area, territory, move)* calculates a new *area* that takes the move's effects into account. This new *area*:

- Contains only points of the previous area and the territory
- Is big enough to contain all follow-up moves of *move*
- Allows computation of a good approximate score in case search is stopped here
- Includes blocks that have become unstable because of *move*

The *total area* of a local game is defined as the union of areas of all nodes in the game graph.

When using local games computed at earlier moves, the constant context assumption may have been violated by moves in some area extension. That part of the game graph has become invalid and must be re-searched in the new context. Shrinking and expanding the area of local games affects the implementation of transposition tables used for local search and evaluation. This topic is discussed in Chapter 7, p. 81.

4.4 Local Search Control and Move Generation

Local Search State

A local search state contains information that is needed for local search and evaluation. It is much smaller and faster to update than the full board state.

Search Control Strategies

Search is controlled at the sum game and at the local game level. On the sum game level, the available time is distributed over all local games. Some games will be more important, or more complicated than others, so they need more search. Expansion of nodes is controlled by an *urgency* estimate: Quiet nodes below a given urgency bound are not expanded further. Control repeatedly iterates over all local games using a progressively smaller urgency bound.

```
ALGORITHM SearchSumGame (sum: TSumGame) (* current nodes of local games *)
BEGIN
    limits.urgency := InitialUrgency ();
    time := TimeLeft ();
    WHILE (time > 0) AND (limits.urgency > 0) DO
        sumExpanded := TRUE;

        FOREACH node IN sum DO
            IF NOT (IsConstantGame (node) OR HasFlag (node, allSearched)) THEN
                (* search the game *)
                limits.maxTime := GetTimeLimit (node, limits.urgency);
                …
                (* set other search limits based on urgency value *)
                …
                generator := SelectGenerator (node, limits);

                localState := StateOfNode (node); (* context switch to local state *)
                (*next iteration of local search *)
                localState.Search (generator, limits);

                IF NOT HasFlag (node, fullyExpanded) THEN sumExpanded := FALSE END;

                FilterNodesInSubgraph (node) (* prepare subgraph for the next iteration *)
            END (*IF*)
        END (*FOREACH*);

        time := time - TimeUsed ();
        IF sumExpanded THEN limits.urgency := ReduceUrgency (limits.urgency, time) END
    END (*WHILE*);
END SearchSumGame;
```

Expansion Urgency

Urgency is a heuristic value used for sorting moves in selective search. It is not a measure of move quality: For example, a risky move in a tactically unstable situation will have *high urgency*. If search refutes the move it gets a *low evaluation*.

One factor contributing to urgency is the estimated temperature of the local area: it would be risky to evaluate a *hot* local position statically. A simple bound on the temperature is the number of unstable stones plus the number of empty points that are neither surrounded nor *dame*.

Selective Search Extension

It would be ideal if search could be controlled by a single urgency value. In practice, exponential growth lurks behind every corner: No matter how well we design the urgency function, there will be games where lowering the urgency by a small amount will add uncontrollably many nodes. We control each search iteration by additional bounds on:

- Time
- Number of node expansions
- Number of moves generated for each single node

Search stops when any of these limits is reached. During search, local nodes are marked as:

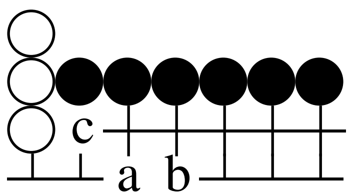
- *Terminal node* if the value of the node can be determined statically
- *Unexpanded node* if the node has been reached during search, but no moves have been generated
- *Partially Expanded node* if some but not all moves have been generated on this urgency level
- *Fully Expanded node* if all moves at the current urgency level have been generated

We call a move generator *complete* if search with urgency 0 is equivalent to exhaustive search. Given enough resources, the iterative algorithm eventually lowers the urgency to 0 and searches the full local game graph.

```
METHOD TLocalState.Search (generator: TGenerator; VAR limits: TLimits);
BEGIN
    (* fCurrentNode is a field of TLocalState *)
    IF HasFlag (fCurrentNode, fullyExpanded) THEN (* recursive call in subtree *)
        FOREACH node IN Siblings (fCurrentNode) DO
            ExecuteNodeMove (node); (* executing a move changes fCurrentNode *)
            Search (generator, limits);
            UndoMove
        END (*FOREACH*)
    ELSIF ExpandNode (fCurrentNode, limits) THEN
        SetFlag (fCurrentNode, partExpanded);
        newMoves := GenerateMoves (generator, limits);
        FORSOME move IN newMoves DO
            IF NOTContinueSearch () THEN ABORT (* stop FORSOME loop *)
            ELSIF ExecuteMove (move) THEN
                (* executing a move changes fCurrentNode *)
                IF NOT HasFlag (fCurrentNode, partExpanded) THEN
                    (* avoid multiple expansions of same node *)
                    Search (generator, limits)
                END (*IF*);
                UndoMove
            END (*IF*)
        END (*FOREACH*);
        IF AllMovesExpanded () THEN SetFlag (fCurrentNode, fullyExpanded) END
    END (*IF*)
END Search;
```

4.5 Move Generation

Heuristic move generation is restricted to few moves per color, and is subject to constraints from search control: A maximum number of moves for the node, or a minimum urgency for a move.



Selective move generation: {a, b, c} for White, {c} for Black

Two Approaches to Move Generation

For move generation in a local game we either adapt parts of an existing program or write new generators specialized for local search.

Local generators (→ Chapter 2, p. 35/36) can be reused directly. *Global* generators rely on full-board information that is not available locally. According to the paradigm of replacing guesswork by search, we adapt global move generators by removing global conditions whenever possible: We use constraints of the local game to model the global context. We try ‘risky’ moves anyway, and rely on search to show whether they are good.

New move generators specialized for local search can be cleaner and faster, but need a bigger development effort. In practice we have preferred careful review and stepwise rewriting of old generators.

Using Constraints and other Filters in Move Generation

Endangered constraints are used as filters in move generation: if a constraint set up to protect a territory from invasion becomes endangered, only invasion stopping moves are generated.

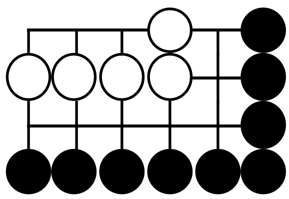
Other filters prevent 'uninteresting' branches from being searched:

- Nodes where constraints have already been violated
- Dominated moves: If a move on an adjacent point is at least as good as a move, we can prune the move, e.g. to secure territory on a bigger scale.
- Moves that 'seem bad', such as suicide, self-atari, unconnected moves, *bad-move* or *bad-shape* pattern moves

Heuristic filters depend on the urgency level. With decreasing urgency, filters eliminate less moves.

Imitating a Local Game: A Quick Adaptation of Goal-directed Search Modules

Goal-oriented minmax search, such as capture search or the Life&Death routine, returns two success values, one for each player going first. As an intermediate solution, these value pairs are converted into a *switch* (→ Chapter 3, p. 40): If a block or a group can be captured or saved depending on who plays first, we construct a switch *B-score* | *W-score:* After capture, the whole area belongs to the attacker, which gives an exact local score.



Local game with Life&Death situation approximated by a switch

For estimating the score after saving the block or group we use the local territory count, which assumes that unsettled stones are alive, as an approximation of the resulting game value. This method is similar to Goliath's goal-oriented approach.

4.6 Cooperation of Move Generators, Filters and Evaluators

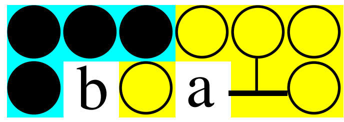
We describe a complementary set of move generators, filters and evaluators. Search starts with one generator, after filtering unneeded moves we apply another generator, etc. until time runs out or search is complete.

Characteristics of Move Generators, Evaluators and Move Filters

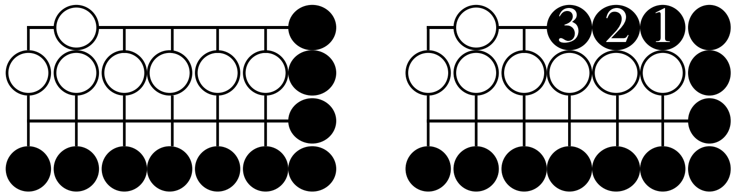
- A *Life&Death generator* identifies unsettled groups and generates eye-making or killing moves. *Life&Death evaluation* is perfect if the group is dead, a crude estimate if the group lives.
- The goal of a *tactical move generator* is to capture or save stones. *Tactical evaluation* measures the strength of blocks, it does not evaluate territory or influence.
- An *endgame move generator* tries to optimize the local score, but it may overlook Life&Death issues because of its limited scope and knowledge. *Endgame evaluation* is tuned for precision.

- A *dame and cleanup move generator* is an endgame generator for the very last moves.
- A *context checking filter* is run before searching a graph generated at an earlier move. It prunes all moves that are illegal or violate constraints in the current context.
- A *tactical move checker* prunes tactically bad moves. It skips moves marked as a *sacrifice*, e.g. eye-stealing moves from Life&Death.

In the example, White has one point of territory whoever goes first, so endgame would not generate a play here. The cleanup generator will eventually connect at *a* and fill the dame point at *b*. The Life&Death generator might play on *a* to make or destroy an eye if the white group is unsettled.



A territorially worthless move affecting eye shape



Endgame interaction with Life&Death

In this example, the context for endgame search assumes that the white group is heuristically safe. When the Life&Death module re-searches the tree generated by the endgame, it detects nodes where White is dead and the Life&Death evaluator reevaluates them. In this case, the incentive for playing move 3 above increases dramatically.

4.7 Dealing with Dependencies Between Local Games

This section describes several approaches to the handling of dependencies within a sum game model.

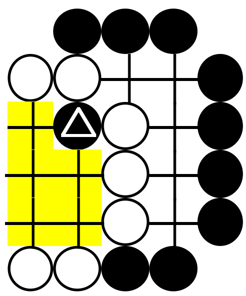
Dependencies Between Local Games

Dependencies between games occur when the areas of local games overlap during area expansion, and when constraints of one local game are related to another game. The effect of such dependencies differs widely: often it is so small that independence is a useful approximation. In cases when a move works as a *double threat* however, dependency analysis is crucial.

We classify dependencies as one-way or two-way:

- In a *one-way dependency of G on H*, play in H has an effect on G, but G does not affect H.
- In a *Two-way* or *mutual dependency* play in G affects H, and vice versa

An example of a one-way dependency is a territory whose size and safety depends on play on the outside. Mutual dependency occurs between endgames adjacent to the same potential territory. Strategies for dealing with dependencies are:



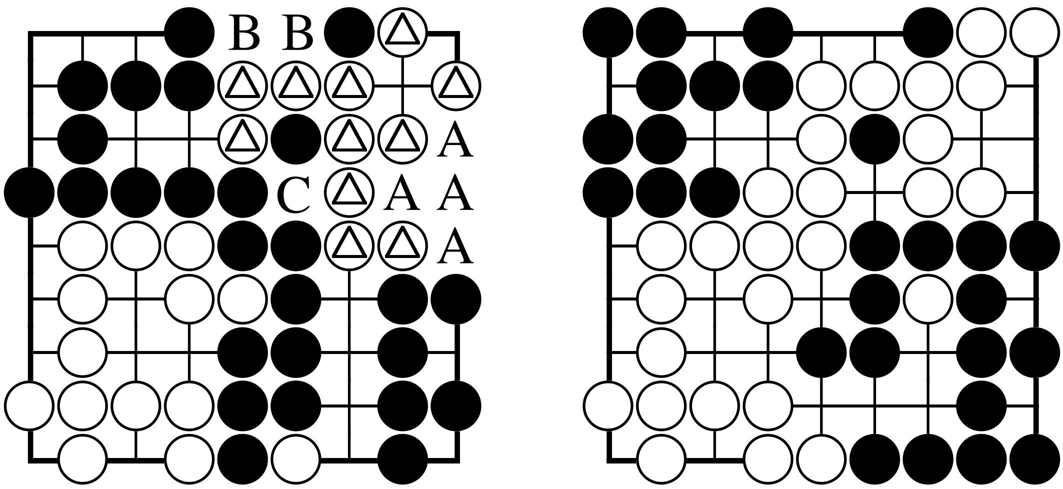
One-way dependency of territory on outside endgame area

- Ignore the dependency, treat games as independent
- Prove that the dependency does not affect the value of the sum, or play of the sum game
- Merge mutually dependent local games, then re-search the combined game, possibly using previously generated information on single games
- Analyze the interaction between local games, then use a specialized theory to compute the joint game value

Strategy 1: Ignoring Dependencies

This strategy is of course the simplest, and the most dangerous. Further research is needed to develop good heuristics for deciding when dependencies can be ignored.

Strategy 2: Proving that Dependencies do not Affect the Sum Game



Berlekamp's Problem C.8 vs. its counterpart in the Computer Go Test Collection

The endgame areas of both examples above are equivalent. Yet the marked white stones are not *immortal*: if Black gets six moves in a row in endgames A, B and C, the white stones will die. Therefore these endgames are related. We could choose to ignore this relation, regard the marked stones as safe and independently analyze the areas A, B and C. In this sum game, it seems impossible that the marked stones will be killed if we play any reasonable endgame sequence. Still, such *vabanque* play is unsatisfactory. Berlekamp and Wolfe [BW 94, p. 62-64] prove that the white stones can be made safe as a 'bonus' during endgame play. Their analysis yields a coordinated strategy for playing these three games such that the value is the same as if the games were independent.

A different analysis of dependencies that are irrelevant in a *specific* sum game is given in [Berlekamp/Kim 94]. Best-case and worst-case assumptions produce upper and lower bounds on local games, which bound the effects a dependency could have. It is shown that incentives computed using both kinds of bounds fit into the same place in the partial order of all incentives of a sum game. Further analysis shows that the game score is identical for both bounds, which proves that the dependency can be ignored in the given context.

Application of these techniques in a computer program is another challenging research topic.

Strategy 3: Merging Games

The safest way to deal with dependencies between games is to merge the games and re-search the merged game. In heuristic search, we want to take advantage of the information on single games for searching the merged game. Of special interest are double threat moves that work as a threat in two of the games to be merged.

Given games $G_1 \ldots G_n$, we define the merged game G = Merge $(G_1, \ldots, G_n)$ by:

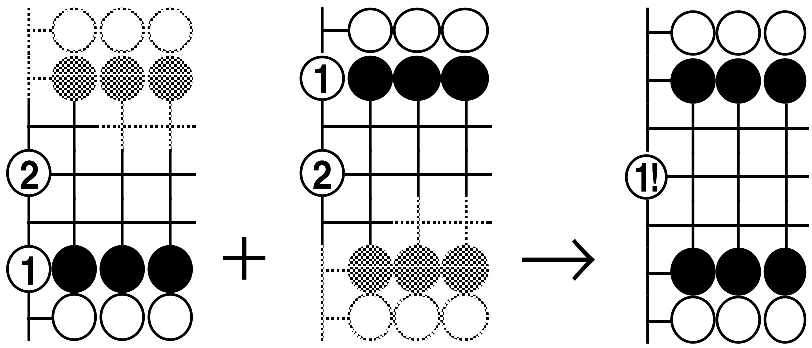
Area (G) = Area $(G_1) \cup \ldots \cup$ Area (G_n),

Constraints (G) = Restrict (Constraints $(G_1) \cup \ldots \cup$ Constraints (G_n), Complement (Area (G)))

Constraints inside the area of G can be deleted: they are not external dependencies of the merged game.

Double Threats

Several important concepts in Go are applications of the double threat theme, which can be interpreted as an interaction between two local games. If a move has an effect on two subgames, and the opponent can only reply in one game, we can play two moves in a row in the other game. Many surprising *tesuji* can be understood as a common move of two dependent local games.



The identical move 2 of two dependent endgames is a tesuji: it threatens to connect in two independent ways

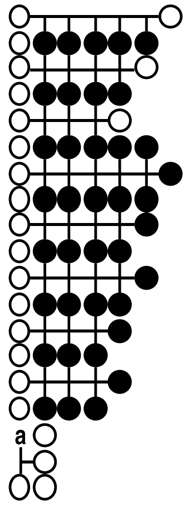
- A *Ko threat* is a local threat, and also threatens to take the Ko again by breaking the repetition ban in the subgame containing the Ko
- A *ladder breaker* is both a local and a long distance threat
- *Aji*-exploiting moves are usually threats in both of two adjacent games

As a heuristic for move generation in merged games, we propose to search common second moves of Black-Black or White-White sequences in two overlapping single games and use these as candidate moves of the merged game.

Strategy 4: Formal Dependency Analysis

Wolfe discusses sets of local games that are adjacent to the same blocks [Wolfe 91a], [BW 94, Chapter 4.7]. These blocks must be saved from capture at some stage, after liberties which are distributed over all endgames have been taken.

For the special case of a single block invading many *corridors* he develops an algorithm to determine the value of the combined game from local game values, without searching the merged game. These results are extended to trees of multiple blocks with multiple connection points (*sockets*).



Unconnected block invading many corridors [BW 94, p. 80]

4.8 Playing Algorithm of Explorer 5

The main program of Explorer 5 handles playing the sum game and cleaning up in the final phase of the game.

Generating Moves in a Sum Game

For generating a move, we must find the current local games, search them and evaluate the resulting sum game. For selecting a move in a sum game, we can choose any of the heuristic algorithms discussed in Chapter 4.

```
ALGORITHM PlayGo (state: TState; toPlay: Color; parameters: TParamaters) : MOVE;
(* Generate a move for 'toPlay' in the current state (board, history), *)
(* with given parameters: sum algorithm, time, memory *)
BEGIN
    sum := FindMatchingNodes (DB, state); (* find precomputed local game nodes in database DB*)
    sum := sum + NewGames (sum, state); (* Generate new games for rest of board *)
    ExcludeConstantGames (sum); (* no effect on computations, omit for efficiency *)

    SearchSumGame (sum);
    EvaluateGames (sum);
    IF ActiveKoBan () THEN GenerateKoThreats () END;

    move := FindSumGameMove (sum, toPlay);
    IF move = Pass THEN move := FindFinalMove (state, sum) END;
    RETURN move
END PlayGo;
```

Final Moves After the Sum Game

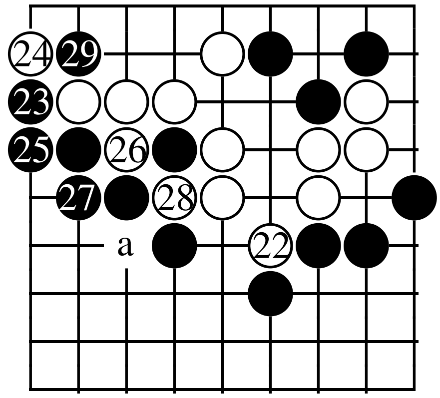
After a sum game is over, there can still be moves that need to be played before the corresponding Go game is really finished:

- Dame pairs and other constant games that have been removed from the sum game computation
- Extra moves to capture dead stones and secure territory in Chinese rules

A constant game can be 'reactivated' by the opponent, e.g. by playing a threat or a bad move there. In that case, the game will be reinserted in the sum after the opponent move.

Comparison of Playing Styles: Explorer 4 vs. Explorer 5 Prototype

With local search, the Explorer 5 prototype plays less blunders equivalent to *Pass* or *very small* moves than Explorer 4. The biggest problem of the prototype program is *greed*: it does not recognize the value of 'thick' moves. Maximizing territory leads to 'thin' positions that are vulnerable to cuts and double threats. An additional non-territorial evaluation seems necessary to overcome this weakness.



Game fragment: Explorer 4 (White) vs. local player prototype (Black)

This game fragment illustrates the problem. White 22 creates cutting points in Black's shape. After White 28, a defense by Black becomes necessary. Explorer's static evaluation recognizes the weakness, and it gives protecting moves such as 'a' a higher evaluation than the cut at 29. The local player however fails to detect the urgency of protecting in its shallow local search and greedily cuts at 29.

4.9 Summary: A Combinatorial Game Theory Framework for Approximate Play of Sum Games

We propose to replace the 'standard model' of computer Go by a sum game model. By combining heuristic computer Go methods with combinatorial game theory, do we get the best of both worlds? We see the following benefits and problems of the sum game model for computer Go:

Benefits:

- The model is well suited to the knowledge and style of current Go programs: they concentrate on local fighting and surrounding territories.
- Locality reduces the complexity of move generation and position evaluation. With the emphasis on evaluation, clever pruning during move generation is not as crucial as in global search because of the reduced search space.
- Stepwise, directed expansion of search is supported. This provides more time control than the usual iterative deepening. It is easy to use opponent's time.
- Important Go terms can be expressed or refined in terms of the theory, and need not be programmed separately.
- The reuse of local game analysis improves efficiency while playing a game.
- Parallelism emerges naturally on many levels: local searches are independent, evaluation and operations on mathematical games can be parallelized.

Problems:

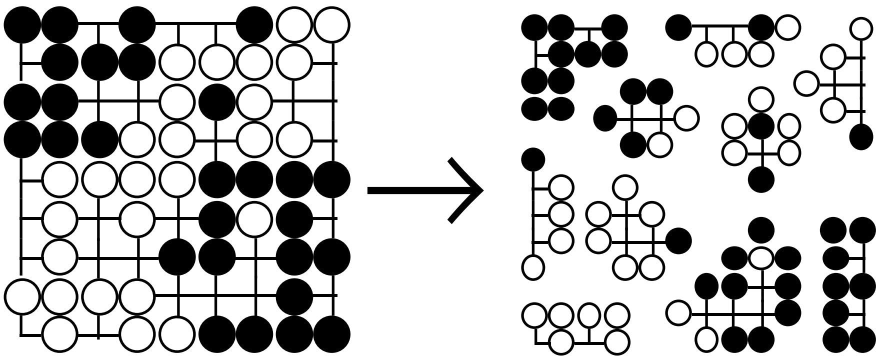
- Managing incrementally changing sets of local graphs in place of a single position adds complexity.
- Accurate board partition and recognition of dependencies is crucial.
- The usual problems of selective search appear in complicated local games: missing crucial moves leads to wrong evaluation.
- Non-constant local games may be overlooked completely, such as insecure 'territories' which can be destroyed from the inside.
- In this model there is no concept of long-range full board plans. This is probably not a big issue until programs reach amateur *Dan* or even professional level.

5

Go Knowledge for Exact Board Partition: Determining Safe Blocks, Safe Territories and Local Endgame Areas

To play Go endgames perfectly, we must replace heuristic methods by exact ones. We decompose a full board position into a sum of independent local games. Independence is guaranteed by boundaries built from *immortal* blocks of stones. A block is immortal if it has two safe liberties.

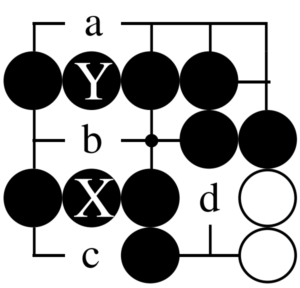
Given a board partition, we can judge whether a position is solvable on current computers, i.e. whether it decomposes into small enough fragments.



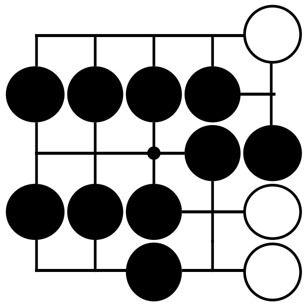
A Go position as a sum of local games: its partition into safe territories and independent endgame areas

5.1 Benson's Criterion for Unconditional Life

Benson's classic analysis of unconditionally alive blocks [Benson 80] provides a starting point for finding safe blocks and territories. In Benson's definition, a *region* is a connected set of points on the Go board. A *color*-enclosed region is a region whose adjacent points are occupied by blocks of *color*. The *interior* of a region is the subset of points not adjacent to an enclosing block. A *small color-enclosed region* has an interior that is either the empty set or filled with opponent stones.



Benson's flawed example A1



A similar position where Black is unconditionally alive

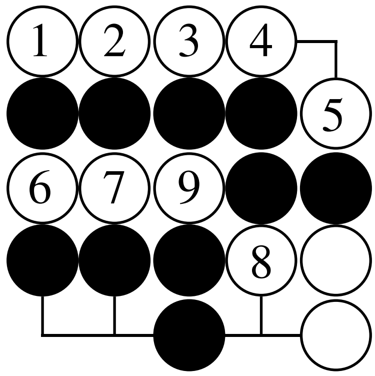
In Benson's example A1, *b, c* and *d* are small Black-enclosed regions. However, there is an instructive error in region *a*: the interior, which consists of a single point in the top right corner, is neither the empty set nor filled by white stones. Therefore *a* is not a small Black-enclosed region. An example where all regions are *small,* that fixes Benson's (or the editor's) oversight, is shown on the right side.

A small color-enclosed region is called *healthy* for a block if the block is adjacent to all empty points of the region. Regions *b, c* and *d* are healthy for block *X*, only region *b* is healthy for block *Y*.

Given a set *B* of blocks, a region *r* is *vital* to a block *b* in *B* if

- *r* is healthy for *b*
- All blocks adjacent to r are in the set *B*

A set *B* of blocks is called *unconditionally alive* if each block in *B* has two vital regions. Benson proves that unconditionally alive blocks cannot be captured by the opponent even with an unlimited number of moves in a row. On the other hand, any block that is not unconditionally alive can be captured by the opponent. The figure shows how to capture block *Y* in Benson's example. Note that if the upper right corner were occupied by White, she could not fill all the liberties in this region.



Capturing a block that is not unconditionally alive

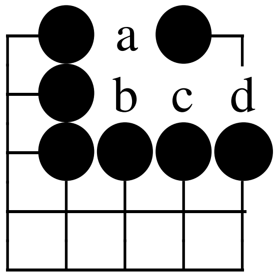
5.2 An Algorithm for Recognizing Safety under Alternating Play

In contrast to Benson's analysis, we are interested in blocks and territories that are safe under *alternating play, with the opponent moving first and winning all Ko fights.* This covers a much larger class of blocks and regions.

The crucial condition for safety is that each block must be guaranteed two liberties. In Benson's algorithm, each healthy region provides a liberty for a block. A region can never contribute more than one liberty, because each region can be almost-filled by opponent stones, with one point providing a liberty for the opponent stones.

Under alternating play, the definition of vital regions must be changed. Vital regions can now guarantee either one or two liberties for an adjacent block. We define a region to be *1-vital (2-vital)* for a block if the region guarantees *one (two) liberties under* *alternating play*, with the opponent moving first. A region can be 1-vital or 2-vital for a block even if it is not vital in Benson's sense. Each vital region is also 1-vital.

We define the *accessible liberties* of a region as the set of liberties of all enclosing blocks in the region. In the example, The top right corner is only a liberty of a block in the interior of the region, so it is not an accessible liberty.

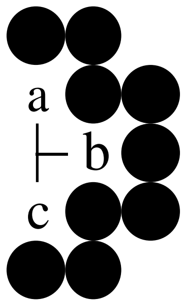


Accessible liberties of a region

Static Recognition of 1-Vital Regions

In the example, White could play a, b and c to take away all accessible liberties of the block in this region. Under alternating play, the black block has a safe liberty here, so the region is 1-vital.

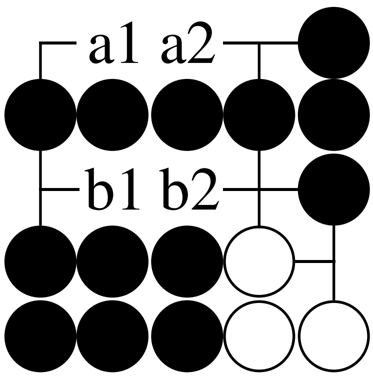
We will call interior points that are not occupied by the opponent the *potential opponent eye space*. A region with one adjacent block is 1-vital if each point in the potential opponent eye space can be uniquely assigned two adjacent accessible liberties. If the opponent plays one liberty, we take the other, which assures that the opponent cannot surround any point.



A region that is not vital, but 1-vital

Static Recognition of 2-Vital Regions

A 2-vital region provides two liberties for each adjacent block. The following condition statically recognizes regions that are 2-vital for *all* adjacent blocks:



Examples of two-vital regions

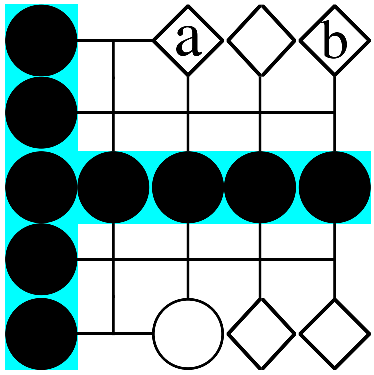
A *color*-enclosed region is *2-vital* for all adjacent blocks if all empty points are adjacent to *some* adjacent block and it has two *intersection points*. An intersection point is an empty point *p* such that *region - {p}* is not connected and *p* is adjacent to *all* blocks.

Each 2-vital region can be transformed into two vital regions in Benson's sense in two different ways. If the opponent plays one intersection point, we take the other. The resulting single merged block has two vital regions.

Safety of Areas Surrounded by Safe Blocks

The regions used in the block safety proofs above are all safe territory. Safe territory means the opponent cannot get safe stones inside. We can prove the safety of regions surrounded by safe blocks by proving that the opponent cannot live inside. The opponent *can live inside* if the potential opponent eye space contains two points *e1, e2* that are not adjacent.

In the example, White can live inside by surrounding *a* and *b*. In the area below the black wall, White's potential eye space is too small.



Can White live inside? The potential eye space is marked.

Summary of Safety-detection Algorithm

We computed a set of *safe* blocks and territories with following properties:

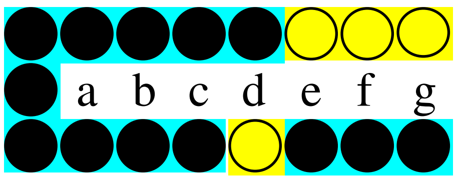
- For each safe block there is an algorithm that guarantees two liberties against any opponent attack
- All safe territories are surrounded by safe stones
- The opponent cannot live inside a safe territory
- Moves on non-safe points on the rest of the board have no effect on the *safe* classification of points

Proving Safety by Search

A static algorithm for detecting safety can be used in a search for proving a larger class of blocks and regions safe. The algorithm is similar to standard Life&Death search, but may use only information on local features of the board.

Dame Points

Single *dame* points, with value *, are the smallest possible independent regions. The algorithm FindDamePoints calculates a set of dame points:



Computation of dame points: unsurroundable = {d, e, f, g}, dame = {e, f}

```
ALGORITHM FindDamePoints (empty: POINTSET; safe: ColorSET): POINTSET;
(* given a set of empty points and sets of safe points, compute subset of dame points *)
BEGIN
    unsurroundable := NeighborPoints (safe [Black]) * NeighborPoints (safe [White]) * empty;
    dame := {p ∈ unsurroundable | (EmptyNeighbors (p) ⊆ unsurroundable)};
    RETURN dame
END FindDamePoints;
```

Proof: We show that a stone on a point computed by FindDamePoints cannot possibly contribute to a capture, or to surrounding territory. Consider any local sequence of play that involves a move on a point $p \in dame$. Removing p from the sequence only affects liberties of safe stones, so there is no change in which blocks are captured. It does not change size or status of territories either: only surroundable points can become part of territory, and all neighbors of p are unsurroundable by construction. For the same reason, p cannot change the safety of any blocks surrounding territory.

There may be other *dame* points that cannot be detected by this static algorithm. The value of playing an unsurroundable point such as *d* in the example is greater or equal than playing a *dame*: the stone is safe from capture, and it might help to surround own or reduce opponent territory.

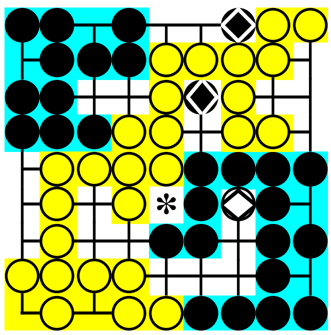
The example also illustrates that dame points are useful for board partition: The area *{a, b, c, d}* is surrounded by safe blocks and dame points and therefore an independent subgame.

5.3 Summary: Algorithm for Exact Board Partition

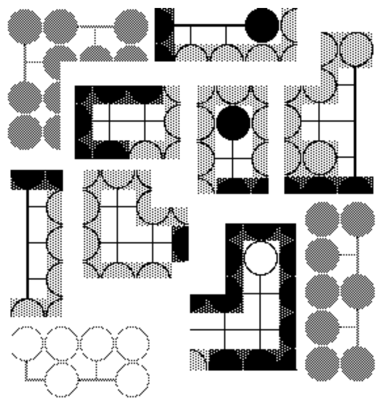
Board Partition

Given *safe* and *dame* points, we compute *connected components* of the remaining *unsafe* points. Each component constitutes a local area. The play in one area cannot affect the play or the score in other areas because the areas are separated by safe blocks. If any area is *too complex* the algorithm indicates a *potential complexity problem*. *Too complex* is an estimate of the cost of searching the area, a simple definition is:

Number of legal moves > 10



The constant part of a game: Safe blocks, territories and a dame point



Endgame areas (territories are shown dimmed)

The Algorithm for Exact Board Partition

The algorithm ExactBoardPartition computes safe and dame points, then partitions the rest of the board into connected components. If the *bigAreas* list remains empty, we will attempt to solve the endgame exactly.

```
ALGORITHM ExactBoardPartition (  VAR safe: ColorSET; VAR dame: POINTSET;
                                 VAR smallAreas, bigAreas: LIST)
BEGIN
    smallAreas := EmptyList ();
    bigAreas := EmptyList ();
    FindSafe (safe);
    dame := FindDamePoints (All [Empty] - safe, safe);
    FORALL area IN ConnectedComponents (AllPoints - safe [Black] - safe [White] - dame) DO
        IF TooComplex (area) THEN
            Append (area, bigAreas)
        ELSE
            Append (area, smallAreas)
        END
    END
END ExactBoardPartition;
```

6

Local Search and Evaluation in the Endgame

Following board partition, the algorithm for converting a Go endgame into a combinatorial game consists of generation of *local game trees*, scoring of local *terminal positions*, and *evaluation* of local games as mathematical games. Finally, a *move* in the resulting sum game must be selected, and identified with a move in the Go position.

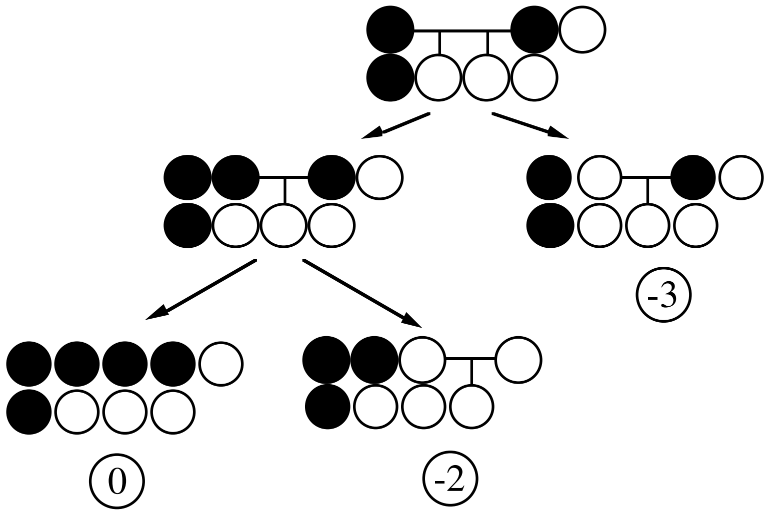
After describing the basic algorithm, we give several enhancements, such as pruning rules, cooling, and using incentives. The result is a program that can solve a wide range of late Go endgames.

Safe territories and dame points are games that can be evaluated statically. Other local games are analyzed by local game tree search. Terminal positions are evaluated, and the values backed up in the tree, resulting in a mathematical game evaluation of each node.

The result of search and analysis is a complete description of possible endgame plays that makes perfect play possible. We save the results of local analysis in a *database of local positions*. During play, each full board position corresponds one-to-one to a set of local positions, one from each local game. Positions and their values are retrieved from the database.

The value of a full board position is the sum of local position values. A *sum game evaluation* algorithm selects a *sufficiently good* or *optimal* move. We also consider an algorithm with lower memory requirements, which stores only a subset of the positions of a local game in the database. In this case, if play reaches a position not stored in the database, we must re-search the corresponding local game.

6.1 Local Search



Local game tree with evaluation of terminal nodes

An endgame area consists of unsettled stones, and of empty points not yet surrounded by either color. Safe stones, usually of both colors, surround the area. During endgame play, unsettled stones either become safe or are captured. Empty points will become occupied, safe territory or dame. A rare case are 'untouchable' empty points in seki.

Move Generation

We generate all legal moves for both colors, except in a terminal position or if moves can be pruned. Examples of *pruning rules* are restricting the number of *dame* moves generated to at most one, and pruning moves *dominated* by other moves. *Termination rules* decide when a position can be evaluated statically, without further expanding the tree.

Terminal Positions

We recognize the following terminal positions:

- No legal moves
- No good move (all points are territory, or dame)
- Value of position known from transposition table, pattern, or local position database

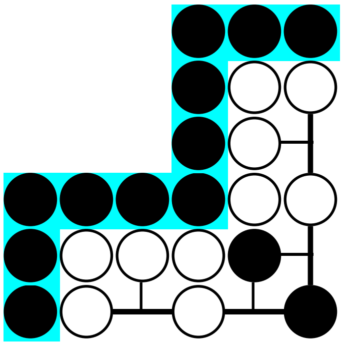
The first two types of terminal positions evaluate to a number, or number-plus-star (odd number of dame). If the value of a position is known from another source, it is terminal in the sense that there is no need to continue search. The value itself can be any mathematical game value. Non-terminal positions may have a constant value as well, if the outcome is the same no matter who plays first, but this has to be proven using search.

Scoring

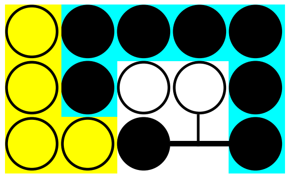
Scoring assigns a number, or number plus dame, to a terminal position. In Chinese rules, scoring measures how many stones and empty points belong to either color. In Japanese rules, territory and prisoners are counted. Both kinds of scoring are straightforward in the endgame because safe and dead stones, territories and neutral points are known exactly.

Scoring Positions with no Profitable Moves

Positions where all moves are useless or bad indicate either seki or territory. Games such as $\{n\text{-}1 \mid n\text{+}1\}$ or $\{n\text{-}1 \mid\}$ typically appear when n point territories are not recognized early enough. The player who moves there just loses a point under japanese rules, or in the best case keeps the score at n, if the opponent must answer. Fortunately, these games evaluate to n. Even in less obvious cases, we have always found the mathematical score consistent with Go tradition. An example is the *snapback* in the figure below.

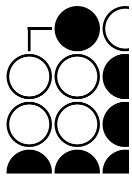


Seki: 22 | -8 || 24 | -6 = 0



Snapback: 4 || 6 | 0 = 5

Other games without profitable moves usually indicate *seki*. The relation between scoring using different Go rules and combinatorial game theory is discussed in detail in [BW 94]. Several examples, including the famous *Three Points Without Capturing*, are analyzed there. In complicated positions with no profitable moves our algorithm indicates a *potential scoring problem*.



Three Points Without Capturing

Exact Play on a Subset of the Board

When board partition indicates that *big* areas remain, or search cannot solve all *small* areas, the user interface offers three options:

- Stop playing
- Ignore the unsolvable big areas
- Use heuristic play in the big areas

Even if an original position has been solved, suicidal opponents can cause problems later: they might endanger their ‘immortal’ stones, which violates our initial assumptions on board partition and generates new unstable positions where optimal play might be very complicated.

Sufficiently Good vs. Optimal Play

Sufficiently good moves guarantee at least the optimal score of the initial position. We play optimally as long as the opponent plays well, but we might fail to exploit all mistakes the other player makes.

Saving Results of Local Search: A Database of Local Game Positions

Positions encountered during local search are stored in a *database of local games*. An important question is what to store: There is a tradeoff between recomputation time and required storage. For building a permanent database, it makes sense to store only 'difficult' moves in the database, and recalculate the rest if needed.

It is efficient to store which moves are locally good or bad, but omit *refutations*: the trees proving that bad moves are inferior. Such a database is sufficient for playing against a good opponent. Only in the unlikely case of locally bad opponent play, we must recalculate one subtree to find a refutation. The following table lists some possibilities:

Type of database	Content
Full	All positions encountered during exhaustive search
Color-complete (Black or White)	All positions reachable by non-dominated moves of *color* and arbitrary opponent moves (pruned bad moves of *color*)
Optimal	At least one position corresponding to each *non-dominated option* of each reachable position, guarantees optimal play from each position in database
Ko-complete (Ko-optimal)	Complete (optimal) + possible Ko threats and answers
Sufficiently good	Guarantees optimal score from a starting position, might fail to exploit some opponent mistakes

6.2 Mapping Game Trees to Mathematical Games

A local game tree with evaluated leaf nodes can be transformed into a mathematical game by the algorithm TreeToValue. Moves are evaluated recursively and sorted by color to find the options of a game.

```
ALGORITHM TreeToValue (node: NODE) : TMathGame;
VAR game, option: TMathGame; col: Color;
BEGIN
    IF IsTerminal (node) THEN
        game := StaticEvaluation (node)
    ELSE (* inner node *)
        options [Black] := []; options [White] := [];
        FORALL son IN Sons (node) DO
            option := TreeToValue (son);
            col := NodePlayer (son);
            options [col] := options [col] ∪ {option};
        END (*FORALL*);
        game := {options [Black] | options [White] }
    END (*IF*);
    RETURN game
END TreeToValue;
```

Pruning Game Trees

The evaluation of nodes leads to pruning the game tree: moves to dominated options are marked as locally bad moves. With the usual domination test for '≥', all but the first of several equally good moves are pruned. A test for '>' keeps the full set of equally good moves.

6.3 Finding a Move in the Sum Game

The algorithm BruteForceFindMove finds an optimal move by computing the Minmax score of the sum game:

```
ALGORITHM BruteForceFindMove (games: LIST; toPlay: Color) : MOVE;
VAR sum, game: TMathGame; foundMove : BOOLEAN; opp: Color; move: MOVE;
BEGIN
    sum := TMathGame (0);
    FOREACH game IN games DO sum := sum + game END; (* compute sum *)
    score := Score (sum, toPlay); (* min-max score of sum game*)

    foundMove := FALSE; move := Pass;
    opp := Opponent (toPlay);
    FORSOME game IN games DO                         (* loop through games' options …*)
       FORSOME option IN Options (game, toPlay) DO   (*…gives set of 'reasonable' moves *)
          IF Score (sum - game + option, opp) = score THEN (* found an optimal move *)
             move := GetMove (game, option, toPlay); (* move from game to option *)
             foundMove := TRUE;                      (*ABORT stops FORSOME …*)
             ABORT                                   (* …iteration over options *)
          END
       END (*FORSOME*);
       IF foundMove THEN ABORT END                   (* abort FORSOME over games*)
    END (*FORSOME*);
    RETURN move;
END BruteForceFindMove;
```

This method should already be more efficient than a global minmax search because local games have been simplified, and more simplification takes place during summation. Still, direct computation of the sum of 'hot' endgames soon becomes prohibitively expensive. If during sum evaluation a time or memory limit is reached, the algorithm stops with *summation failure* (not shown in the algorithm above).

Faster Methods for Sum Evaluation

Processing sum games can be speeded up by omitting constant games, dame, and miai points from the sum game. With Chinese rules, the value of *dame* is $\{1 \mid -1\}$, and they may not be omitted. These moves will be played in a finishing phase after the sum game is over.

Using Incentives

Two techniques of combinatorial game theory introduced in Chapter 3 help to speed up move selection in a sum game: For games of relatively low temperature, *cooling by 1* (p. 39) is very effective and allows easier computation of the sum.

The use of *incentives* (p. 38) is often decisive: if one option has an incentive that dominates all other incentives of all local games, it is an optimal move that can be played without computing the sum.

```
ALGORITHM FindMoveByIncentive (games: LIST; toPlay: Color) : MOVE;
VAR sum, game: TMathGame; foundMove : BOOLEAN; opp: Color; move: MOVE;
BEGIN
    best := EmptyList (); (* list of (incentive, game, option) tuples *)

    FOREACH game IN games DO
        FOREACH option IN Options (game, toPlay) DO
            incentive := Incentive (game, option, toPlay);
            IF Dominates (incentive, Incentives (best)) THEN
                best := [(incentive, game, option)];
            ELSIF NOT IsDominated (incentive, Incentives (best)) THEN
                Append ((incentive, game, option), best);
            END
        END
    END;

    move := Pass;
    IF ListLength (best) = 1 THEN
        (incentive, game, option) := FirstElement (best);
        move := GetMove (game, option, toPlay);
    ELSIF best ≠ EmptyList THEN
        move := BruteForceFindMove (games, toPlay)
    END (*IF*);
    RETURN move;
END FindMoveByIncentive;
```

The following example illustrates an exponential explosion of summation, which is avoided by the use of incentives.

Sum game	Canonical Form of Sum
$1\|0$	$1\|0$
$1\|0 + 2\|0$	$3\|2\|\|1\|0$
$1\|0 + 2\|0 + 3\|0$	$6\|5\|\|4\|3\|\|\|3\|2\|\|1\|0$
$1\|0 + 2\|0 + 3\|0 + 4\|0$	$\{10\|9\|\|8\|7\|\|\|7\|6\|\|5\|4\} \mid \{6\|5\|\|4\|3\|\|\|3\|2\|\|1\|0\}$
$1\|0 + 2\|0 + 3\|0 + 4\|0 + 5\|0$	$\{15\|14\|\|13\|12\|\|\|12\|11\|\|10\|9\} \mid \{11\|10\|\|9\|8\|\|\|8\|7\|\|6\|5\} \mid\mid \{10\|9\|\|8\|7\|\|\|7\|6\|\|5\|4\} \mid \{6\|5\|\|4\|3\|\|\|3\|2\|\|1\|0\}$

Partial Search

Incentives of moves are a local concept, so they cannot take a specific sum game into account. Minmax search can resolve which move is best in the context of the current sum. Combinatorial game theory is used as a (very strong) pruning method for minmax tree search.

Example: In the sum game $5 \mid 0 + 6 \mid 4 \parallel 0$, Left to play can either move to 5 in the first game or to $6 \mid 4$ in the second game. The incentives $5 - 5 \mid 0$ and $6 \mid 4 - 6 \mid 4 \parallel 0$ are incomparable, so we try playing out both moves:

$$\begin{aligned} Leftscore\ (sum) &= Max\ (Rightscore\ (5 + 6 \mid 4 \parallel 0),\ Rightscore\ (5 \mid 0 + 6 \mid 4)) \\ &= Max\ (5, 6) = 6. \end{aligned}$$

Only the move to $6 \mid 4$ retains the minmax score. In contrast, in the sum $2 \mid 0 + 5 \mid 0 + 6 \mid 4 \parallel 0$, the move to 5 would be best, assuring a *Leftscore* of 7 in contrast to 6 for the other moves.

Mapping a Move in the Abstract Game to a Go Move

As a last step, we must find a Go move corresponding to the selected option in the abstract game. We select the first move with sufficient incentive.

Ko During Search

Loops often occur during search, even if they do not affect optimal play or the game value. As described in Section 3.3 (p. 43/44), we compute bounds on the game score by disallowing one player's moves that lead to a loop (this player is said to *lose all Kos*). If optimal play does not depend on Ko, upper and lower bound are identical and the game value has been determined. Both players have an optimal strategy that does not involve a Ko fight. If the bounds differ, the algorithm stops with a *Ko-failure*.

6.4 Speeding up Local Search

Ignoring illegal moves and captures, the number of possible plays in an n point area is approximately $2n$ (n for each player), and play generates a n-1 point area. A rough estimate for the size of the game tree is therefore $2n \cdot 2(n-1) \cdot \ldots = 2^n n!$

Due to this combinatorial explosion, even fairly small endgames become prohibitively expensive to compute using this approach. In the following, we look at a number of techniques for reducing the size of the search tree.

Transposition Table

A transposition table detects identical board positions, reducing the size of the search space from $\approx 2^n n!$ to 3^n states. The implementation of a transposition table for local search is described in Chapter 7, p. 81. Adaptation of search and evaluation algorithms is straightforward.

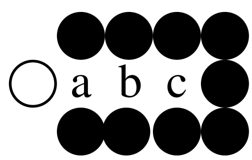
n	$2^n n!$	3^n
1	2	3
2	8	9
3	48	27
4	384	81
5	3840	243
6	46080	729
7	645120	2187
8	10321920	6561
9	185794560	19683
10	3715891200	59049

Reduction of search space by transposition table

Pruning Moves

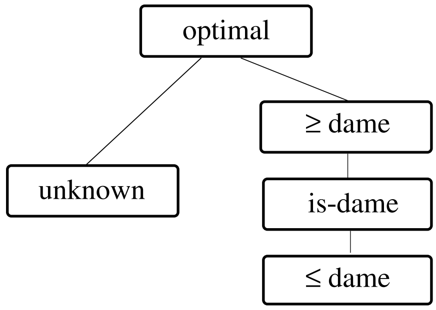
The *width* of the search tree can be further reduced by pruning moves. In contrast to selective search, we may only eliminate moves that are *provably* worse-or-equal than others. If a move achieves all points of the local area, or if any other move would give the opponent an answer which achieves the maximum score, the move is optimal, and we can prune all other moves.

For *corridors* the move at the entrance of the corridor is optimal for both players. In the example, we need only consider move *a*.



Corridor with unique best move

Dame points are recognized by the algorithm *FindDamePoints* (p. 64). Comparing the incentive of moves relative to the value of a dame move leads to the following partial order for pruning:



Partial order for pruning

A move causes pruning of all moves in lower categories (relative to the partial order). When a category contains a unique value, all but one move in this category can be pruned (the *is-dame* and *optimal* categories in this example). This rule can easily be extended, e.g. by including *unsurroundable* points (p. 64), or by recognizing *corridor* values and simple switches.

6.5 Summary: Using Combinatorial Game Theory to Solve Late Endgames by Computer

What we can do

- Perfect computer play in late endgame
- Find the game-theoretic value of a position long before the end
- Evaluate the opponent's endgame moves
- Find moves that are *good enough* even with a reduced local game database

Limits of the 'Strict' Approach

Strict analysis of endgames is currently not possible if one of the following limits is reached:

- Partition: too few blocks can be proven safe independent of endgame play. Therefore some areas become too big for complete search.
- Summation: no move with dominating incentive exists, and both summing and partial search take too long.
- Ko: Standard combinatorial game theory exploits the independence of local games. In the case of Ko, the independence is broken (a locally bad move may be globally best if it serves as a Ko threat). Generalizations of the theory to handle Kos are a topic of current research.
- Added complexity in 'obvious' situations: In cases where the focus of play is obvious (i.e. only one local situation is relevant), combinatorial game theory introduces additional complexity by investigating moves for both players in this and each other position. Reversibility (p. 38) and *thermographic reversibility* (not discussed in this thesis) can be used to eliminate much of this extra complexity.

7

The Explorer Program: Implementation Issues

We describe the implementation of selected Explorer components: pattern matching, local games, and additions to the Smart Game Board user interface.

Pattern matching in Go is mapped to a text searching problem which is solved using a *Patricia* tree index. Three optimizations important for the application to Go are described.

The implementation of local games and their sums consists of the following parts: A *local game* contains summary information on the game and *game graph. Nodes* of the game graph can be found through pattern matching, after storing a *pattern* in a *local game database*. A *local state* provides a context for efficient local search that supports operations such as move generation, and fast Execute/Undo of moves.

A sum game changes incrementally during a game. By storing local game information in a database we avoid recomputation. An analogous database is also used for caching the results of tactical minmax search.

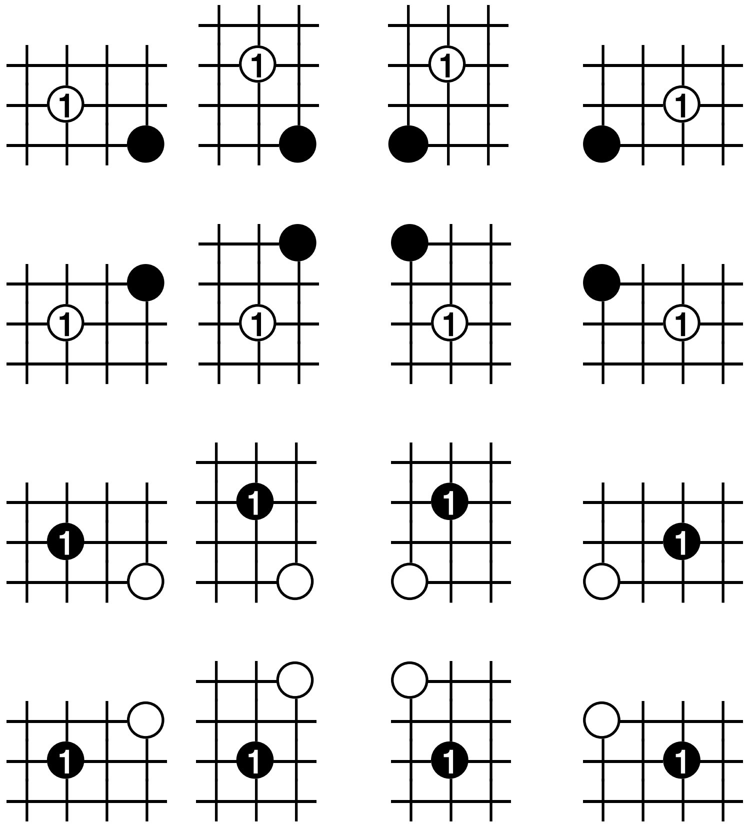
Operations on *combinatorial games* are handled by Wolfe's toolkit [Wolfe91b] which has been ported to the Smart Game Board [Fierz 92] and adapted for use in Explorer. A *sum game* contains references to nodes of local games that match a full board position. It provides methods for sum evaluation and move selection. Besides supporting the sum of local games model, Explorer 5 uses new mechanisms for memory management and time control.

Support by the user interface is crucial for the exploratory kind of program development typical of game programs [Kierulf et al. 90]. Explorer reuses and extends components of the Smart Game Board. The tool palette of the Smart Game Board has been extended by a *pattern* and *local game editor,* which supports building and maintaining a pattern library and experimenting with sums of local games.

The *Analyze* tool tests a Go program in selected problems. Assertions input in text form or through the graphical interface are checked during analysis. Further tools are available that show, write, edit and check the program's data.

7.1 Pattern Matching

Pattern Matching in Go



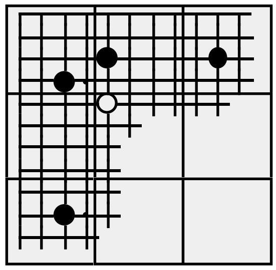
The 16 orientations of a pattern

Many variants of pattern matching have been studied in computer science. In Go, we need to match a large number of 2-dimensional patterns in 16 orientations against a single board. It is too expensive to match each pattern against the board at each possible location at each turn. We use a *Patricia tree* as a filter that reduces the set of possible matches to a few candidates. Only candidate patterns have to be compared with the current board position. A number of further optimizations are shown which dramatically improve performance.

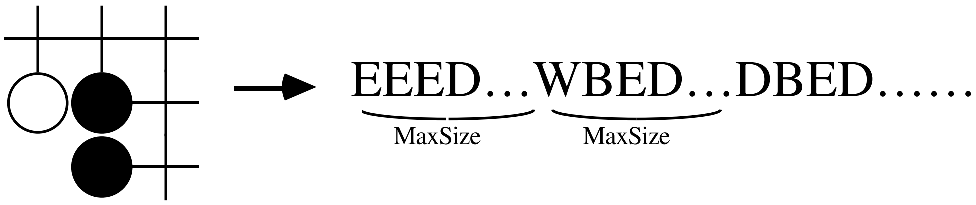
A Text Searching Algorithm for Pattern Matching

The Patricia method [Sedgewick 83, p. 219-223] is a radix trie searching method that has been used for searching large text databases such as the Oxford English Dictionary [Gonnet 88]. We develop an algorithm specialized for pattern matching in Go:

- Each pattern is covered by a grid of 4×4 tiles. For small patterns, one tile is sufficient. Full board patterns on the 19×19 board (e.g. for openings) need 25 tiles.
- For each point in a tile, we store the value Empty, Black, White, or DontCare (out of pattern). This takes 2 bits. The 4×4 tile size has been chosen to fit one tile into a 32 bit word.

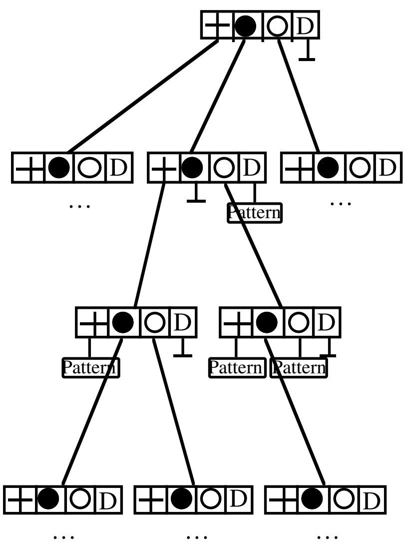


Covering an irregularly shaped pattern with 4x4 tiles



Converting a pattern to a sistring

- For linearization, all points in the pattern are indexed. The index of the point with local coordinates *(x, y)* is *MaxSize·x+y*, with a constant *MaxSize* equal to the maximum board size, e.g. *19*. Unused indices correspond to DontCare points. The result is a *semi-infinite string* (*sistring*) over the alphabet *{E, B, W, D} = {Empty, Black, White, DontCare}*. All but a finite number of characters in the string are *'D'*.
- A Patricia tree index for differentiating the sistrings of all patterns is built. The inner nodes of the tree contain an integer index and references to four subtrees. Patterns contained in the same subtree share the same color at the indexed point of their sistrings. The leaves of the tree contain either a reference to a sistring (pattern leaves) or NIL, indicating that there is no matching sistring in the library.
- For each starting location *(x, y)* and each orientation, we perform a *match*: we compare the board against the Patricia tree at the index points of nodes. From each node, we traverse both the subtree with matching color and the DontCare subtree.
- Matching the board against the Patricia tree ends in leaf nodes, which are either NIL (a *mismatch leaf*) or *pattern leaves.* A pattern leaf contains a reference to a pattern that matches the board at all sampling points and has to be compared with the board. Because DontCare points in patterns match any board state, there may be more than one matching pattern per (location, orientation) pair.



Patricia Tree

Comparing a Pattern with the Actual Board

For comparing the board against candidate patterns, we compute tiles for each starting location and orientation of the board. These *tile boards* are incrementally updated before matching. If all pattern tiles match the board tiles, we have found a match. As soon as two tiles mismatch, we stop the comparison with a *comparison mismatch*.

Factors Affecting the Speed of the Pattern Matching Algorithm

The effective running time of the pattern matching algorithm depends on many factors:

- Number and size of patterns
- Size of the Go board
- Structure of the Patricia tree
- Number of patterns
- Speed of Patricia tree and of full pattern-board comparisons
- Ratio of mismatches caught by the Patricia tree

Optimization 1: Incremental Matching

One optimization is very important for game play: The result of a single match or mismatch, the list of matching patterns, typically depends only on a few points on the board. After making a move, most matches stay valid. To exploit this observation we keep a *dependency set* for each match, which contains all the points that were tested in the matching process. After each move we must update only those matches whose dependency set is affected by a move. This optimization is very effective in game play. It has little effect when successive full board positions are unrelated, e.g. during analysis of unrelated test positions.

In a test on complete games, namely the set *'Pro 10 Games'* of the *Computer Go Test Collection*, this optimization saved 96% of all center matches, 94% of edge matches and 93% of corner matches. In contrary, on the test *'Pro 1000 Positions'* of unrelated positions, the savings were only 10%, 3% and 0.2% respectively. Most of these 'accidental' savings were probably due to matching empty areas.

Optimization 2: Adaptive Tree

Hitting a mismatch leaf (NIL) in the Patricia tree is cheaper than detecting a mismatch during the final pattern-board comparison. After such a pattern-board mismatch, the *Adaptive Tree* algorithm replaces the pattern leaf with a new node branching at the index where the pattern-board mismatch occurred. The new node has four siblings: the pattern leaf and three NIL leaves, one of which will correspond to the board color at the index. This way the tree grows and catches more mismatches in the tree, which saves time if the same query occurs again.

A disadvantage of the Adaptive Tree algorithm is increased memory requirement. Another potential disadvantage, degeneration of the tree structure, did not occur with our data.

Optimization 3: Balancing the Patricia Tree

Analysis of Patricia trees for our pattern library showed poor balance: near the root, most branches carried few patterns. The bulk of patterns was contained in very few branches initially, and the tree became bushy only at about depth 10. The reason is that we built the tree with low indices close to the root. But at low indices, most patterns were very similar: almost all points were Empty, with occasional DontCare points.

Ambros Marzetta implemented a tool to rotate the patterns so that a stone would occur as soon as possible in the pattern. This simple change in data representation effectively doubled the speed of the algorithm.

In our *Best Balance* algorithm we go a step further: we repeatedly find the index with the best balance between subtrees: we maximize the number of non-NIL subtrees, then maximize the number of patterns in the smallest non-NIL subtree.

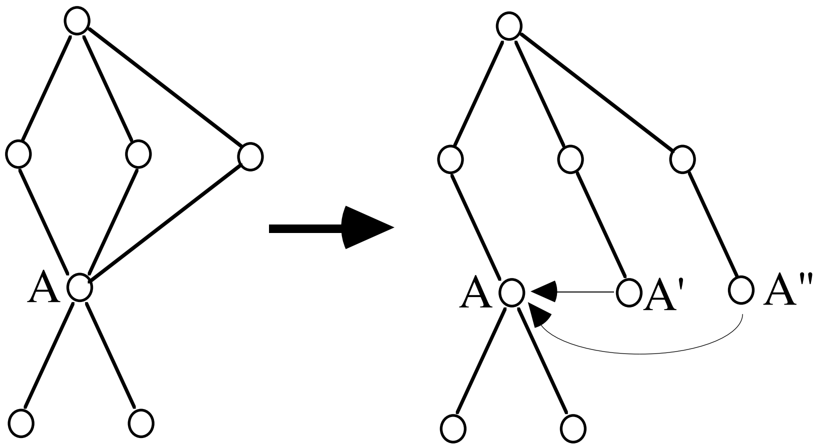
7.2 Local Games

This section describes the implementation of local games, game graphs and their nodes, mathematical games, and thermographs. The same implementation is used for both the heuristic and the exact local game model.

Design Principle: Store Search Tree in Memory

In typical minmax search programs, the search tree is not stored explicitly. Results can be reused easily through a transposition table. A single tree is searched deeper following the actual line of play at each move. In a sum of games model, the result of local analysis may stay valid for many moves. It is feasible and efficient to store computed local results in memory and retrieve matching local situations during a game.

Game Graph

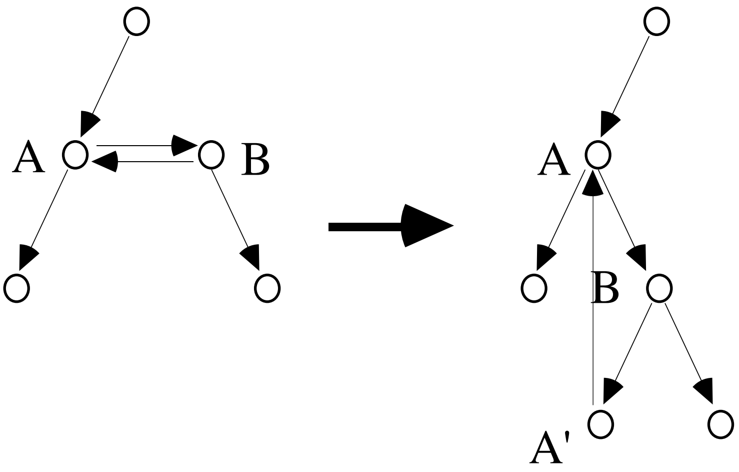


Representation of game graph as a tree with transposition links

The *game graph* is a rooted, directed, possibly cyclic graph. It has been implemented using a Smart Game Board tree plus extra *transposition links*. Nodes of game graphs are accessible by traversing the graph through the tree structure and transposition links. Loopy graphs are represented in the same way by one or more transposition links.

The transposition link approach was chosen for the simplicity of its implementation on top of the existing tree structure. Its main disadvantage is that it leads to additional nodes in the graph. There is no duplication of subtrees below these nodes, however.

Only one predecessor of a node is easily accessible through this structure. Access to other immediate predecessors is not required for our application. The only operation that traverses the graph bottom-up is finding the root node of the graph, which is supported by the underlying tree.



Representation of loopy game graph

We distinguish three kinds of nodes: *Interior* nodes, *terminal* nodes, and *transposition* nodes that contain only a reference to the real node.

Representation of a Local Situation: Nodes of the Local Game Graph

A node of a local game graph can be viewed as an extension of the Smart Game Board NODE type [Kierulf 90, p. 19]. Local game specific attributes are available in addition to standard Smart Go Board attributes:

- The area of the local game at this node
- The color of all points in the local area
- The context: prisoners, Ko status, outside liberties, connections, eyes
- Flags indicating loops in the subgraph, status of move generation, evaluation, etc.
- The quality of evaluation: Proved value, heuristic value, or entered by human
- Mathematical game evaluations: exact value, upper and lower bounds
- Mean, temperature, or a heuristic temperature estimate

Local Game

A *local game* is identified with the root node of a game graph. It contains information which applies to the whole graph. Such attributes include playing rules, initial constraints for the game, and statistics on the game.

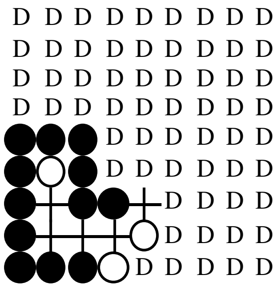
Local Node Database

The database stores patterns describing the nodes of local game graphs. By matching the database against the current board we retrieve a set of matching nodes. The function of the database is to find local game nodes which have been computed at an earlier stage during a game. This prevents recomputation of games.

Local Pattern

Maps for local node patterns are defined differently from usual pattern maps: To find all local node patterns in a single match operation, the location and orientation of pattern maps is fixed. All maps are extended to the full board by DontCare points.

Patterns of local nodes are stored in a database separate from the standard pattern database. As with standard patterns, several nodes may share the same pattern map. This happens when the same local situation occurs in different contexts.



Full board map for local game

A Database for Caching Local Minmax Search Results

Task definitions and solutions [Kierulf 90] for tactical and Life&Death search are also stored in a local pattern database, which replaces the fixed size table used for this purpose in former Explorer versions.

Local State for Search and Evaluation

Local search and evaluation operate in a context which may not match the current board position: the local game might be in a different Ko status, or search may be (re-)started in a local situation that differs from the current board.

A *local state* describes the *current node* of a local game. It contains all information on a local game that is needed for move generation and evaluation. For fast executing and undoing of moves, it supports a stack that tracks incremental changes of the local situation. This stack contains complementary information to the basic Smart Go Board move stack, such as changes in the safety status of stones during search.

Local state information is used for creating new nodes in the game graph during search and for traversing an existing graph during evaluation. In the *read-only* mode of evaluation, trying to execute a move that is not in the graph will fail. A status field of the local state tracks any error, such as overflow or an illegal move, that occurred during graph traversal.

Typically one state per local game is used, though the design with a local state separate from the local game graph allows concurrent read-only access by several local state objects to the same local game.

Context Switch Between Local Games

The design of the Smart Game Board [Kierulf 90] encourages multiplexing all local games on the same unique board and move stack. When switching from one local game to another, all moves of the first game are undone, then the other game is set up from the empty board. The root node of each local game contains information to set up the context of the local game.

The obvious disadvantage of the multiplexing approach is that a context switch between local games is slower than keeping a separate board and move stack for each local game. The switch causes an overhead comparable to switching between two open games in the Smart Game Board. The main advantage is that much less memory is needed. Timing tests showed that the context switching time is negligible compared to the total search time. Most local games have relatively shallow search trees, which keeps setup and undo times low.

Hashing for a Local Node Transposition Table

When the context of a local situation remains constant, hashing is a faster alternative to pattern matching, but it must be used carefully in conjunction with local games. For example, the same situation may occur with a different number of captured stones, which yields a different final score.

We use the Smart Game Board's built-in hash table, which computes a full board hash key [Kierulf 90]. Depending on the application, hash entries are references to nodes or mathematical evaluations of nodes. The same position can appear in different contexts, which must not be confused by hashing. We experimented with two different solutions:

- A general solution is to construct a new code from a context code and the board hash code.
- When the needed information is composed of a purely local part and a context dependent part, we can store only the local information in the hash table and compute the effect of context on the fly. For example, endgame evaluation can be computed as the sum of a hashed local score and a prisoner count that is computed incrementally during graph traversal. An advantage of this method is that more transpositions are found than when hashing the same local situation in different contexts to different hash table entries.

Writing Local Games to a File

Local games can be stored as standard Smart Game Format files. Context and other information on the local game is stored as game-specific properties in the root node. Other nodes contain only the move, and an evaluation if one exists. Transposition links and context information for non-root nodes are not stored in the file, but rebuilt after reading the file. The transposition links are restored using hashing.

Sums of Local Games

Explorer's *Sum Game* type is based on a *collection* type which has the following operations: create empty collection, add item, delete item, find item and enumerate items. Additional operations for sum games are summation, move selection strategies such as Sentestrat and Thermostrat, writing and display methods and error checking.

7.3 Combinatorial Game Theory

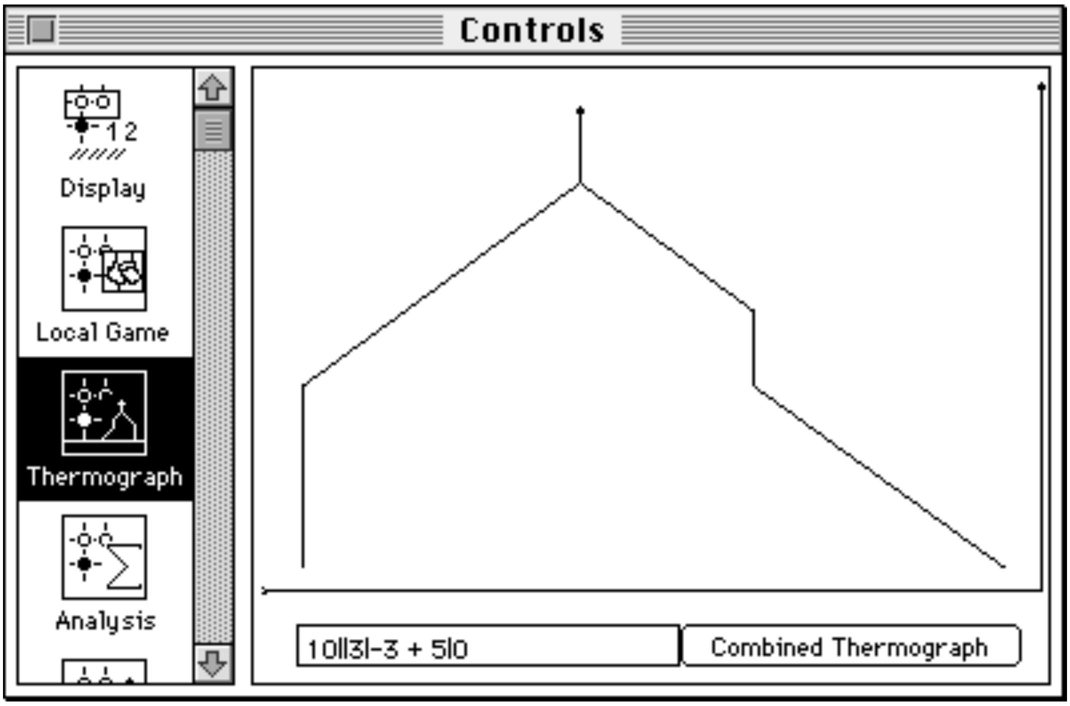
Integrating Wolfe's Toolkit into Explorer

David Wolfe's toolkit [Wolfe 91b] implements mathematical games and many functions and operations such as building a game from its options, reduction to canonical form, adding games, converting numbers to games, cooling, computing the temperature, Leftscores, Rightscores and comparing games. The toolkit also provides conversion between games and text strings.

Werner Fierz' port of Wolfe's toolkit to the Smart Game Board [Fierz 92] has been extended with interface elements to access the toolkit functions through the Smart Game Board, a facility to store mathematical games in a Smart Game Board node and procedures for converting an evaluated game graph into a mathematical game.

Thermographs

For experiments with the *Thermostrat* algorithm, we have implemented a *thermograph* data structure and display. Explorer can compute and show thermographs of a single loopfree game, or the combined thermograph of a sum game.



The Thermograph Display

7.4 Time and Memory Management

Memory Management of Objects and Local Trees

In a program with limited memory, we need to decide which local game nodes and other calculated results to keep, and which to dispose during the course of play. Which items are obsolete, and which can probably be reused later? Only heuristic rules for memory management are possible, because the future actions of the user are unpredictable.

During the course of a game there is a gradual shift of relevance. Starting a new game leads to radical change, just about all computed results are useless in the new game. Some flexibility for users undoing moves should be provided.

As a solution we define a *forced substate,* a subset of all points on the current board. The substate consists mainly of safe-looking stones. We delete all objects that do not match the forced substate. For supporting undo's, we exclude all points affected by the last two moves from the points of the forced substate. Some items are exempt from automatic disposal: user inputs, and objects explicitly calculated on a user's demand.

Time Control for Tournament Play

Time control determines how much time to use for each subgame, for tactics, and for Life&Death problems. The main time control mechanism is contained in the iterative search method described in Chapter 4. The same algorithm with smaller time slices could be used for computing in opponents time. A fast mode is used in time trouble: only minimal time is used for subproblems such as tactics calculations and Life&Death.

7.5 Extending the Smart Game Board as a Tool for Go Programmers

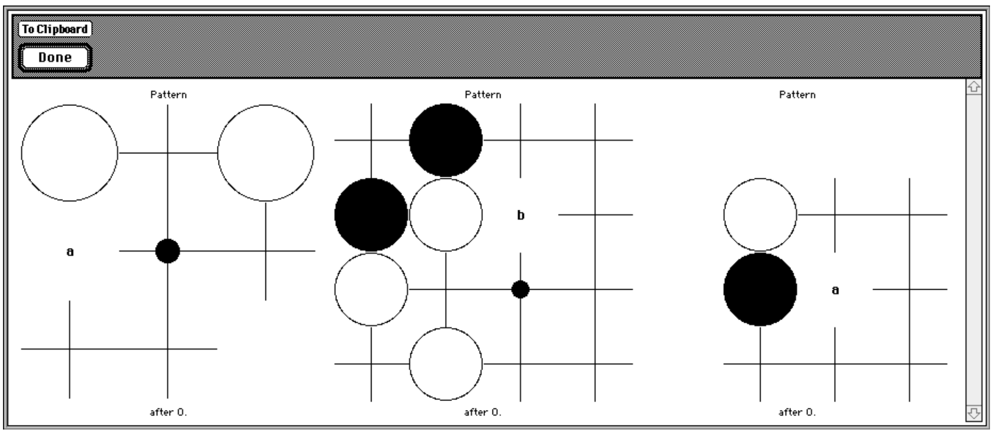
Game Programmers as Special Users of the Smart Game Board

Besides writing the code for a computer player, game programmers are also users of the Smart Game Board. Playing through computer games and analyzing them, or testing their program's performance in collected problems is part of their everyday work. Standard functions of the Smart Game Board yield additional benefits for programmers:

- The *game tree editor* can be used to store a search tree, including additional information such as evaluation of terminal nodes and backed up values.
- A multitude of *views* provides an excellent environment for testing and development.
- *Game collections* are useful for archiving computer games and collecting test problems.
- When the program plays a test game, all variations tried by the opponent are automatically kept for later study.
- A tester can *mark* critical moves and add *comments* for the programmer.
- *Smart Go Format* files can be easily exchanged by e-mail.
- *Diagram printing* facilitates the creation of documentation and reports.

Evolution of Smart Game Board since 1990

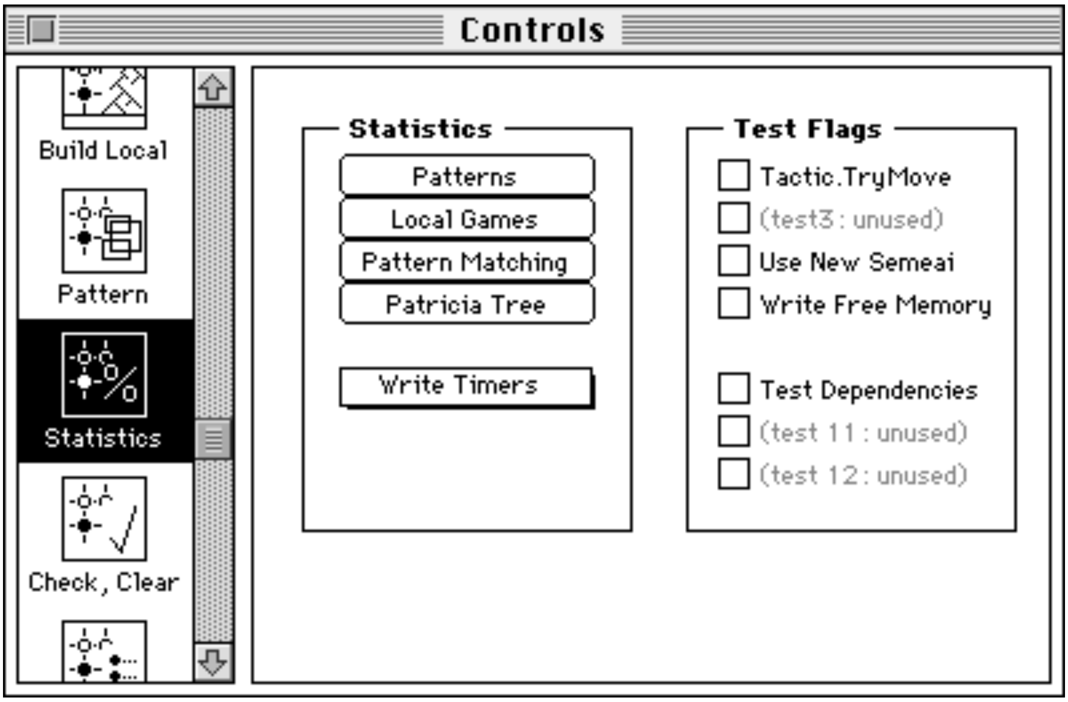
Christoph Wirth has implemented many extensions to the Smart Game Board since 1990 [Kierulf et al. 91]: New views are available to show the *game tree* in graphical form, an *overview* of critical game positions, and a *text protocol*. Game programmers can define *game-specific* properties and use the new *dimmed* and *hidden* properties for dimming or hiding parts of the displayed board. A game can be output in the standard Rich Text Format (RTF) complete with diagrams and annotations.



Pattern overview

The new display facilities have been used in Explorer. In the example, the overview window shows patterns for selected moves.

Controls

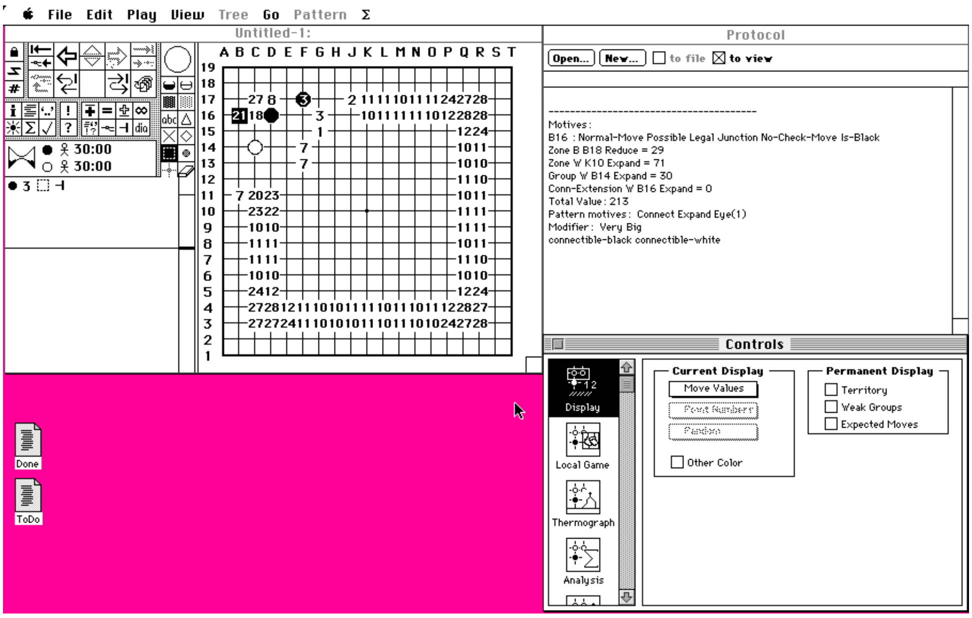


Controls view

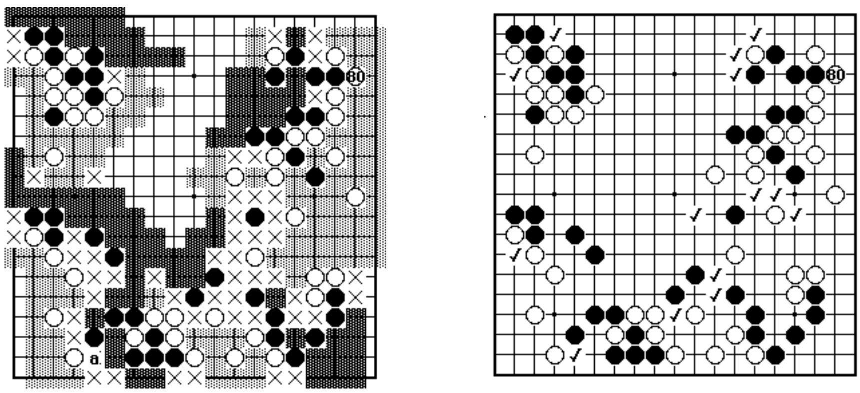
The *Controls view* was built by reusing Christoph Wirth's *Preferences* view. It provides access to Explorer functions for display, operation on local games, thermographs, the Analyze Tool, patterns and pattern matching, statistics, checking and debugging tools.

Display Tools

We use the Smart Game Board components *board view, tree view, overview, protocol,* and *game collection* for displaying Explorer data. Displays are selected from a menu in *Controls*. The display of territory, weak groups, and expected moves can be made permanent. This information is shown in addition to the currently selected display.



The display selected in Controls *is shown in board view and protocol view*



Maps showing connectable *points, and* urgent *pattern moves*

All data structures, such as Blocks, Chains, Groups, Zones, and Local Games, can be displayed. Full board *maps* show sets of points that are safe, near, or junction points, and sets of moves such as legal or pattern moves.

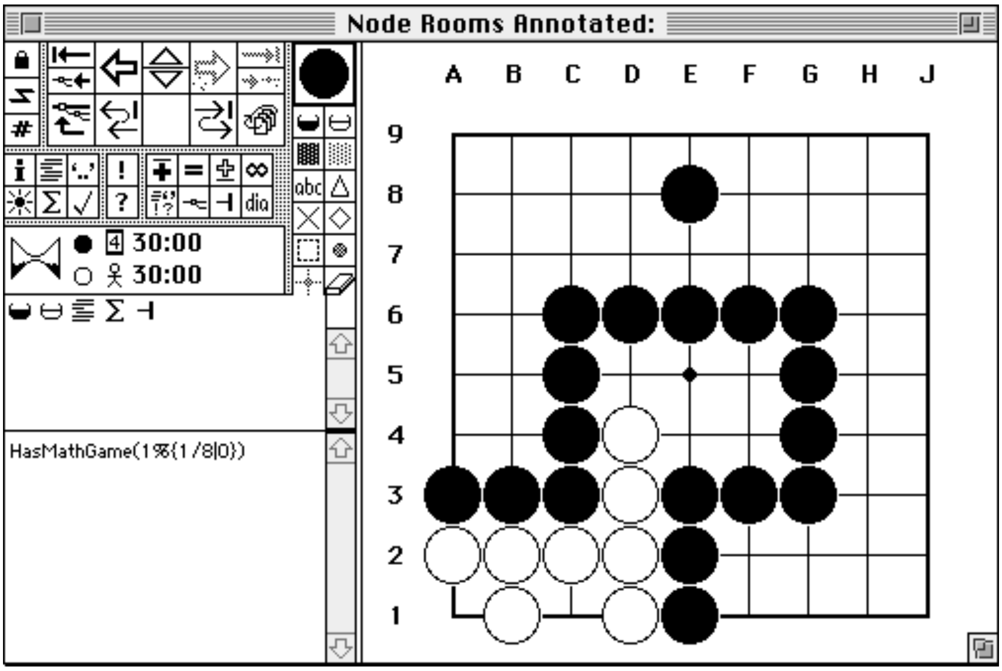
Extending the Analyze Tool

The *Analyze* tool compares the program's computation with annotations stored in a Smart Game Board tree. As an extension of the tool described in [Kierulf 90], we introduce *commands* and *assertions*. Commands are used for setting parameters such as rules and program version during analysis:

Sample Commands:
Version(local)
SumStrategy(sum)
Evaluation(territory,static)

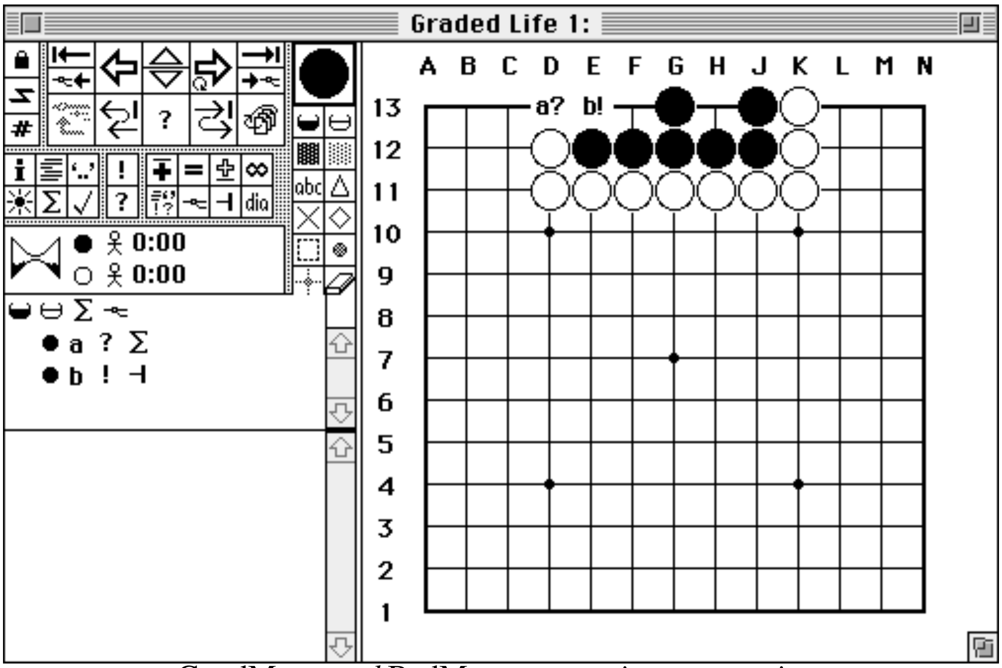
Sample Assertions:
IsBlock(K14, dead)
IsMove(G3, good)
AreSame(group, A1, T19)

Sample commands and assertions



Endgame position with assertion in comment view

Assertions are statements about the current Go position, which are compared with Explorer's analysis. All nodes with assertions that fail are marked during analysis. Statistics show totals over all assertions checked during analysis. Frequently used assertions, such as IsMove (p, good), IsMove (p, best), IsMove (current, blunder) are assigned to properties and can be entered graphically using icons in the Smart Game Board's panel. A complete list of commands, assertions and their corresponding properties is given in [Müller 94].



GoodMove *and* BadMove *properties as assertions*

8

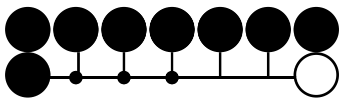
Results

In exact endgame play, we show examples of solved local games and full board positions. We discuss the problem of performance measurement in general and introduce the *Computer Go Test Collection* as a tool to measure performance accurately.

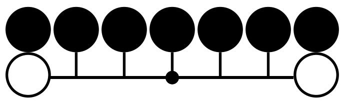
8.1 Exact Endgame Calculations

Single Endgame Positions

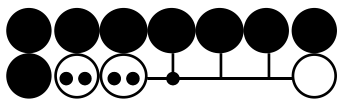
Corridors [BW 94] are among the most basic exercises for an endgame program. Hash tables and exact pruning rules lead to substantial savings even in this easy case. The figure shows a variety of corridors, both with and without stones in the corridor.



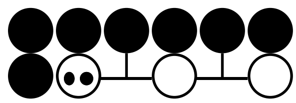
Blocked corridor, value 1/16



Unblocked corridor, value 1/4

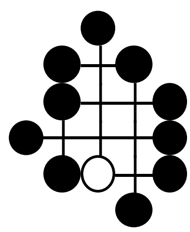


$0|+_2$



*Hot game with two stones in corridor, value 2|**

In [Müller/Gasser 94], we verified the values of the 103 *node room* positions with 3 to 7 empty spaces in [Wolfe 91a, BW 94]. For positions where optimal play does not depend on Ko, we obtain identical results. For the positions where optimal play may depend on Ko, our program returns bounds. The most complex Ko position 'the rogue' and our computed bounds are shown.

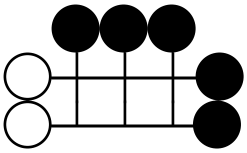


The 'rogue'

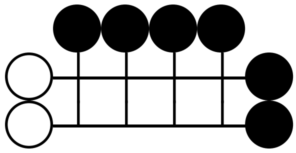
Upper bound:
Black Wins Kos:
$5|\{4|||3||2|1*\}|||2|1*||*$
cooled: $49/16|7/4$
$\mu = 77/32$, $t = 53/32$

Lower bound:
White Wins Kos:
$\{5|||4||3|*\}|||2|1*||*$
cooled: $3|3+_1||7/4$
$\mu = 76/32$, $t = 52/32$

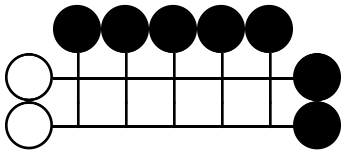
The following examples, also from [Müller/Gasser 94], are given with *cooled* game values, mean, and the temperature of the *original* game (the temperature of the cooled game is 1 less). For some nine point corner positions we obtained very complicated bounds.



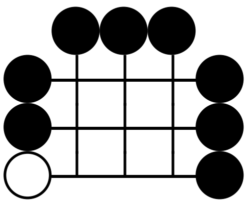
2|1
μ: 3/2, temperature: 3/2



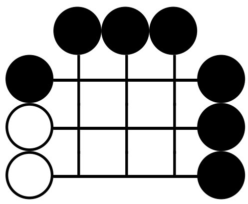
4|3/2
μ: 11/4, temperature: 9/4



Upper Bound: 6|{3|2}, $3-_{2*}$
μ: 17/4, temperature: 11/4
Lower Bound: 6|{3|2},{$6+_{4|4,\{4|2\}}$|{5|4‖2*},{4,4↑*|3‖3}‖2}*
μ: 17/4, temperature: 11/4



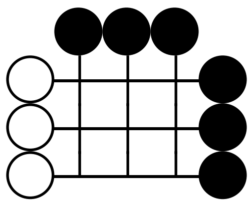
Upper Bound: 7|17/8,{5|3↑•2‖3*|||2}*
μ: 73/16, temperature: 55/16
Lower Bound: 7‖$2-_{1}$|2
μ: 9/2, temperature: 7/2

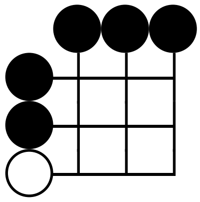


Upper Bound: {5|||3↑•3‖$2-_{1}$‖1}|1*
μ: 9/4, temperature: 9/4
Lower Bound: 3↑•3|1
μ: 2, temperature: 2

Upper Bound: 3/2↑,{$3+_{5/2}$|1/2,{5/2|1/2}}|1/2
μ: 1, temperature: 3/2

Lower Bound: 9/8|1/2
μ: 13/16, temperature: 21/16





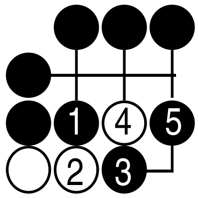
Upper Bound:
$$\{5|5/2\},5+_{4-_{1}}|\{5|1+_{1}|||1,\{5,\{5|1\}|1\}|1||0\},\ \{\{5||3|5/2\},\{5|3+_{6+_{4}}\}|5/2,\{3|5/2\},\{3,\{3|5/2\}|\{5/2|\{5/2|1-_{1}\},\{2,\{7/2|1-_{2*}\}|1-_{1}\}\},\{3|5/2||2|-1\}\}||-1\}$$

μ: 3, temperature: 3

Lower Bound: $5|2*|||1-_{2}||1/2|0$

μ: 2, temperature: 19/8

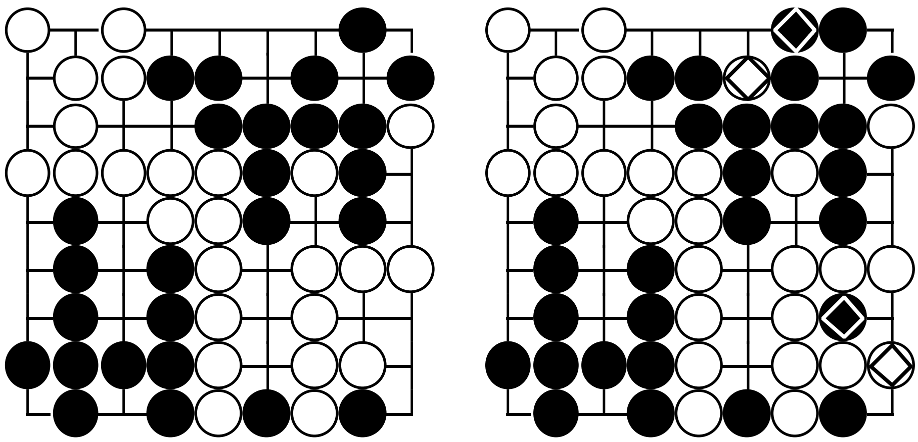
The value of the last position depends on interesting variations. For example, if Black wins Kos, she has a surprising defense of the corner: Black gives way once, then blocks at 3. If White cuts, Black plays and wins a Ko with 5. Such sequences are easily overlooked by many human players.



Black's clever defense of the corner

Full Board Endgame Problems

Explorer solves full board endgames that decompose into small enough independent fragments. Appendix C of [BW 94] contains a set of full board problems. Many of these problems can not be partitioned properly using the exact algorithms of Chapter 5. The independence of subgames is not immediately provable, and has to be established individually through careful analysis. We have therefore designed a corresponding set of problems with identical local endgame values, where the safety of surrounding blocks is easier to prove.



Berlekamp's problem C.7, and its modification (added stones are marked)

Problem C.7 of [BW 94] is an example. The program cannot prove the safety of Black's top right and of White's lower right group. In the top corner, only one eye is independent of endgame play. In the lower right corner, there is a possible Ko fight in case White gets surrounded. In the modified version, both groups are absolutely alive. The added stones do not affect the score or the local game values.

The complete table of modified problems is given in the appendix.

8.2 Performance Measurement for Heuristic Go Programs

Human players measure their Go strength on a widely accepted rating scale (→ Section 1.2, p. 9). It is possible to enter a program in a human tournament and earn a rating, but such a number is a doubtful measure of performance [Kierulf et al. 90]. To measure a program's performance more accurately we consider computer analysis of master games and test position collections, and play against a variety of opponents. Together these tests give a more adequate overall picture of a program's strengths and weaknesses than one single test.

Candidates for measurements are the quality of moves, accuracy of scoring and board partitioning, recognition of features such as eyes, connections, or dividers, or efficiency in terms of speed, number of nodes searched, or memory used. Such tests are supported by the *Analyze Tool*, by Smart Game Board elements such as timers, and by an annotation format for game positions [Müller 94].

In the following, we focus on measuring the quality of moves played by a Go program. Except in the endgame, where exact analysis is possible, the quality of moves will be judged heuristically by a human player who is much stronger than the program. We give examples how test results can be used for improving a program.

The Ranking Problem

The ranking obtained by playing an opponent once in a tournament cannot be compared with a human's ranking. People learn to exploit a program's specific weaknesses. Experiments show that after a few games against the same player, program performance typically deteriorates by 10 or more levels. Although Goliath has won the 17 stone match against humans in the 1991 International Computer Go Congress, experienced players still beat any Go program easily on 17 handicap stones and more in 1995.

Computer vs. Computer

Computer Go tournaments give an indication of the relative strength of Go programs. However, some features of a program do not appear in a game against another computer:

- Current Go programs have little knowledge, so the program is tested only in the restricted set of positions that programs can generate. For example, programs cannot play double threats or Ko fights well, while many humans strive to create them. Computers also fail in their judgement of situations where people rely on 'vague' concepts.
- Each game explores only one branch in the tree of possible moves. A program may be lucky in a particular game, while it would collapse had the opponent tried a different variation. Playing through one sequence in a game does not guarantee stability of the program.
- The *bug of the day* often determines the eventual winner, even if the losing program plays better Go on average. To take an extreme example, a program that fills its own eyes at the very end will lose no matter how well it plays before that bug appears.

The best test opponent is not always the best program: play against a very weak or random player can uncover more program weaknesses, because unforeseen positions come up that are beyond the program's repertoire.

Human vs. Computer

One advantage of human opponents is that they play widely varying styles. The games exhibit a range of positions not seen in computer vs. computer play. Another important advantage is that players often go back in a game and try different variations, until they finally find a gap in the program's defenses. Protocols from such games are ideal for solidifying a program.

Finding Suitable Test Positions

There are many ways to set up test positions:

- Specialized problem sets for Life&Death, capturing stones, the endgame etc.
- Find-the-best-move problems: Positions where a single best move has been recommended by expert players.
- Computer games: Positions in which a Go program has gone wrong before.
- Anti-Computer traps: situations where the 'obvious' move is wrong (e.g. a worthless 'urgent' pattern move)

Because the designer of a test suite has complete freedom on the set of positions included, this testing method is very powerful. With carefully selected test sets one can get independent measures of performance in different stages of a game, leading to a detailed profile of program behavior. While setting up a test suite certainly takes a lot of work, it is easy to run it over and over again automatically.

The popularity of test collections such as the Bratko-Kopec test in computer chess [Kopec/Bratko 82] indicates the importance of a widely accepted set of test positions. Our *Computer Go Test Collection* is a first step to develop such a set for computer Go.

Computer Go Test Collection

The Computer Go Test Collection is designed to test Go programs in all aspects of the game. Details of the collection are described in the Appendix of this thesis, and the *Readme* text that accompanies the files. Version 1.0 can be retrieved from the World Wide Web at http://nobi.ethz.ch/martin/cgtc.html. We encourage all Go programmers to try their program against the problems, publish their results, and contribute tests to future releases of the collection.

Tests using Master Games

Ideally, the moves of a program should be analyzed by a strong player, or by the programmer who is familiar with the playing algorithm. Such analysis takes a lot of effort and cannot be automated. For automated testing, we can make use of expert knowledge: In a game between master players, the overwhelming majority of moves will be good. Comparing a program's evaluation with these master moves yields a measure of its knowledge and evaluation.

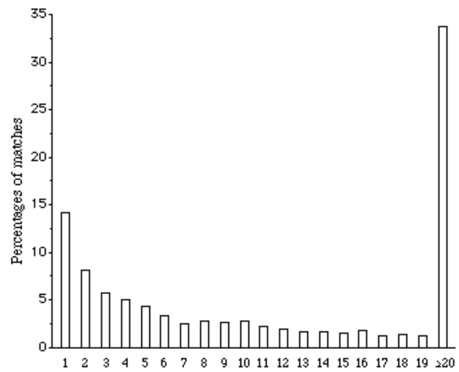
Testing the Static Full-Board Evaluation

In [Müller 91a] I tested Explorer's static evaluation value on a set of 3000 moves from randomly selected master games. For that test, Explorer sorted all legal moves by static evaluation value. Categories 1 to 19 contain the master moves that were ranked in first to 19th position by the program, while category 20 contains all moves with rank 20 or worse.

Results

The results are summarized in the figure. Explorer ranked 14% of all master moves as its best move, 8% as second best, 6% in third position and so on. 34% of master moves were in category 20, being ranked twentieth or worse. From the statistics, I distinguished three types of master moves:

- About one third of all moves were ranked in first, second or third place. The program correctly recognized that these moves are very important.
- One third of all moves was ranked from 4th to 19th place. It seems that the program 'knew' about these moves, but did not recognize their full value.
- The last third of moves were 'too difficult' for Explorer at the time. This may have been due to missing knowledge about high-level plans, or other errors in the evaluation of the move.



Ranks of master moves according to Explorer, from [Müller 91a]

Because of changes in the move selection procedure, it is not feasible to repeat this experiment with the current version of Explorer. Only the move ranked first is accessible now.

A Guessing Game for Humans and Programs

For comparison, I tried a similar test on humans and several computer Go algorithms. All participants were presented the game *'Pro 1 Game'* from the *Computer Go Test Collection*. Their task was to predict the next move. We allowed up to three tries per position for humans. The figure below shows the performance on the 253 moves of the game:

	Player	Rank 1	Rank 2	Rank 3	Rest (≥4)
	Human A (9 Kyu)	32.4%	9.9%	5.5%	52.2%
	Human B (11 Kyu)	31.2%	16.2%	6.3%	46.2%
	Intellect 5.0	13.8%	n.a.	n.a.	n.a.
a	'Hint'	12.3%	n.a.	n.a.	n.a.
	Explorer 4	13.4%	n.a.	n.a.	n.a.
b	Explorer 3	14%	8%	6%	72%
c	Random	0.8%	n.a.	n.a.	n.a.
d	Random	0.5%	0.5%	0.5%	98.6%

Remarks:

a Intellect's 'Hint' function is a faster version of Intellect, with a greatly reduced amount of search.

b These results were obtained on a different test set [Müller 91a] and are therefore not directly comparable. They were included because they are the only available data on Rank 2 and Rank 3 moves by a program.

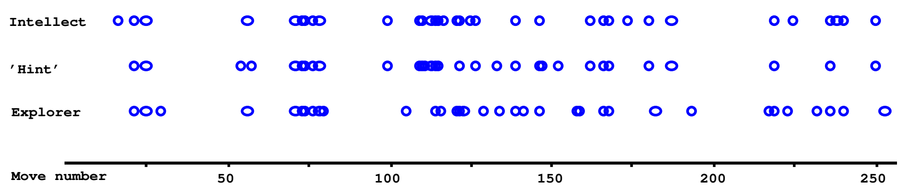
c Result of one experiment with Explorer's Random Player (2 correct guesses)

d Computed expected values

Interpretation of the Results of the Guessing Game

The statistics show a correlation between the percentage of correct guesses and playing strength. It would be interesting to extend the experiment to stronger players. For programs, the performance on specific test collections may be a better indicator of strength.

Testing programs on master games causes some peculiarities: In a test on a complete game such as 'Pro 1 Game', the program sometimes gets addicted to a particular move and wants to play there for many moves, while both master players consistently ignore the area. The same mistake in evaluation thus affects many positions. The problem would not occur in sets of unrelated positions.



Correct predictions of master moves

The figure shows a plot of the moves that were guessed correctly by the three tested algorithms. The program's performances are very similar, with short periods of frequent hits followed by long stretches of complete disorientation. Forced tactical sequences lead to many hits, while tempting but unplayed ataries or cuts can lead the programs astray for a while.

There are two factors which make the guessing game a bit unfair:

- Equally good moves
- Ties in the program's evaluation

Even if the program's proposed move is as good as the master move, it is counted as a miss in the guessing game. Many forcing moves can be played at any time during a long period without affecting the final score.

Of all moves which get the same best evaluation by the program, only one can be selected for play. This does not matter much in 'hot' positions, since the evaluation is sufficiently fine-grained there, but in quiet positions many moves will get the same evaluation. Even if the master move is one of those best-scoring moves, it is improbable that it will be selected.

One factor that gives humans an advantage over programs in this test is context information: Humans learn to 'trust' the professional player and give up on a particular move after a few unsuccessful tries. A program will try the same move much more often, whenever its evaluation tells it to do so.

The similarity in scores of Intellect, Intellect's *'Hint'* function and Explorer came as a big surprise. It seems that knowledge about good moves is comparable. The observed difference in playing strength must come from the kind of bad moves played, which is not measured in this test.

Graded Go Problems for Beginners

We measured the performance of Explorer on all problems of Graded Go Problems for Beginners, Vol. 1 [Kano 85]. The problems were split into categories as follows:

Category	Problems
Tactics	1-30, 35-40, 51-52, 61-92, 97-98, 113-115, 179-208, 212-215, 221, 237
Strategy	31-34, 50, 93-96, 216-220, 236
Life&Death	41-46, 99-112, 116, 121-178, 222-233
Endgame	56, 58, 60, 120, 238-239
Opening	47-49, 118, 234-235

Problems 53-55, 57, 59, 117, 119, 209-211 were omitted because they contain neither a good nor a bad move that the program could check.

The table below gives the results of Explorer on this collection. The number of positions in the table is higher than the number of problems, because many problems contain follow-up moves and variations that are also checked in the test.

Collection	# Pos.	Solved	%
Tactics	147	123	83.7
Strategy	17	14	82.4
Life&Death	335	224	66.9
Endgame	12	7	58.3
Opening	6	5	83.3
Total	517	373	72.1

For comparison, David Fotland claims ‘about 95% correct’ for his program *Many Faces of Go* [Fotland 94]. His test was restricted to the original problems, without variations and follow-up moves. Many Faces was set to a special problem-solving mode for the tests.

Using the Tests to Improve Explorer

The traditional way of improving a Go program is to play through games, stop at bad moves and try to fix them. This process is tedious and unsystematic. The *Computer Go Test Collection* in conjunction with the *Analyze Tool* has proved helpful in finding the following types of program errors:

- Explorer moves that are evaluated much higher than master moves indicate errors in the evaluation function.
- Master moves with low evaluation show gaps in the program’s knowledge.

Analysis results can be used directly to debug and improve a program. One method is to work specifically on master moves ranked second best by the program, comparing the evaluations of the first two moves and trying to adjust parameters to reverse their relative importance. Another way of improving the program is to analyze the moves it misses completely and write new code to generate such moves.

9
Thesis Summary and Future Research

We discuss the contributions of this thesis to computer science in general and computer Go in particular. A final section suggests interesting future research topics.

9.1 Contributions of this Thesis

Contributions for Computer Science in General

- Scientists are fascinated by problems which can be stated simply, yet are hard to solve. Computer Go is a prime example. We have brought the divide-and-conquer approach, a fundamental paradigm of computer science, to bear on computer Go.
- The application of a sophisticated mathematical theory to computer Go provides an example of algorithms for a nontrivial decomposition of a complex problem.
- We have established a new link between theory and practice, admittedly in an esoteric domain.

Contributions for Computer Go

- We have implemented a late endgame player, a niche where program play surpasses human play in both speed and exactness. We did this by applying concepts from combinatorial game theory to Go. The program plays a wide variety of late endgame positions perfectly.
- We have designed a model of Go that integrates exact and heuristic types of knowledge, closing the gap between the mathematical theory of Go and the practice of computer Go.
- We have developed algorithms for board partition and dependency analysis. The central idea of board partition has been used both in a program following a traditional model, and in a program based on the sum game approach.
- Go Knowledge has been modeled in a variety of ways, including patterns, boundaries and zones. A pattern system based on a text matching algorithm allows to efficiently match patterns of arbitrary size and shape.
- We have refined the Smart Go Board, a powerful tool for Go programmers and Go players with a pattern editor, tools to analyze games and problem collections, and a variety of displays.
- We have collected a *Computer Go Test Collection* which provides a freely accessible test suite for Go programmers.

9.2 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, human intellect seems to handle Go well. The game has a lot of logical, geometrical and combinatorial structure which human players can recognize and exploit. In comparison, today's Go programs comprehend only the most basic concepts of Go.

Combinatorial game theory captures an essential part of what Go is about. I think that in one form or another, it will become a key component of all successful future Go programs. To make progress, I feel it is necessary both to encode more Go-specific knowledge and to push forward the application of theories such as combinatorial game theory to Go. After more than 5 years of working on the subject, the depth of the game still astounds me: it has remained as much of a mystery as ever.

9.3 Future Research Problems

As in any active research area, each answered question gives rise to new problems. Interesting research topics encountered during work on this thesis include:

A Complete Go Program Based on the Sum Game Model

- Improve local move generation.
- Incorporate 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 semeai, Life&Death, or tactical search.
- Improve algorithms for dependent games, and interaction of local games. Specifically, search for and generate double threat moves.
- Add an evaluated library of standard sequences.
- Handle Ko and Ko threats.

An Extended Sum of Games Framework

The sum of games framework developed in this thesis can be improved in many ways:

- Investigate algorithms for the evaluation of sum games: estimate the quality 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.
- Port the program to parallel computers.
- A long term goal: identify further areas where computer power is superior to human knowledge, such as chaotic tactical fights, or other positions that need a lot of tedious calculation.

Pattern Learning

Research on pattern learning in computer Go has unfortunately been restricted to ab-initio learning of the most basic patterns from zero knowledge. Interactive or automatic tuning and expansion of a state-of-the-art pattern base is another fascinating research topic.

Move Generation and Evaluation From Precomputed Database

Evaluations in terms of combinatorial game values could easily be added to the pattern database containing standard sequences for the opening, midgame and endgame. Each match will yield an accurate description of a local area without requiring any further analysis or search.

Build a Search Index for Large Game Collections

A Patricia tree index is well suited for search in very big databases. Only the relatively small tree must be kept in memory. The data such as patterns or full board positions can be represented in the tree by a reference to their disk location [Peter 93]. For huge databases, even parts of the tree could probably be swapped to disk without a big loss in efficiency. Such a search index would be great for enhancing electronic Go books, finding examples of textbook *joseki, fuseki* or *tesuji* in real games.

Extend the Computer Go Test Collection

Add more problems or create new types of collections to enhance the value of the collection. Problems from standard text books on Tesuji, Life&Death and endgame could be added.

Appendix
The Computer Go Test Collection

A wide variety of test problems have been collected, instrumented, and made available to the public as the *Computer Go Test Collection*.

Version 1.0 of the Computer Go Test Collection can be retrieved through the World Wide Web, from http://nobi.ethz.ch/martin/cgtc.html.

A.1 General Purpose Test Sets

These tests are suitable for all-round testing of a program. Most types of collections are available in several sizes, for fast and for thorough testing.

Name	Description
Computer 100 Positions	100 unrelated positions from Computer-Computer games
Computer 1000 Positions	1000 unrelated positions from Computer-Computer games
Pro 100 Positions	100 unrelated positions from professional games
Pro 1000 Positions	1000 unrelated positions from professional games
Pro 1 Game	A complete professional game
Pro 10 Games	10 complete professional games
IGS 138 Games	138 amateur Dan games played on the Internet Go Server
IGS 31 Counted Games	The subset of 'IGS 138 Games' games that have been finished and scored

Professional and IGS games have been cleaned up by removing Pass plays at the end, and adjusting the final count when necessary.

A.2 Test Sets for Specific Features

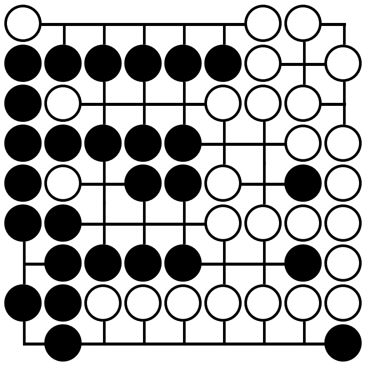
These sets are designed to test a special theme or phase of the game. Collections for a theme contain both positive and negative instances: positions where following the theme leads to a good, or to a bad move. Collections on similar or orthogonal themes may overlap to some degree.

Name	Description
Perfect Play 1	all problems from [BW 94], Appendix C.
Node Rooms	all node rooms from [BW 94], pp. 71-76.
Local Game 1	Situations for testing partition and local search
Loose Ladders 1*	10 Loose Ladder problems from textbooks
Graded Tactics 1*	Stone capturing problems from [Kano 85], Vol. 1
Graded Tactics 2*	Stone capturing problems from [Kano 85], Vol. 2
Graded Life&Death 1*	Life&Death problems from [Kano 85], Vol. 1
Graded Life&Death 2*	Life&Death problems from [Kano 85], Vol. 2
Tactics 1	Stone capturing problems from computer games
Life&Death 1	Life&Death problems from computer games
Cut&Connect 1	Cutting and connecting problems from computer games
Ko 1	Ko fights from computer games
Territory 1	Securing and invading territory from computer games
Endgame 1	Endgame problems from computer games
Threats 1	Good and bad threats from computer games
Double Threats 1	Double threats from computer games
Final 1	Final stage moves, filling dame, defend etc. from computer games
Blunder 1	Blunders from computer games

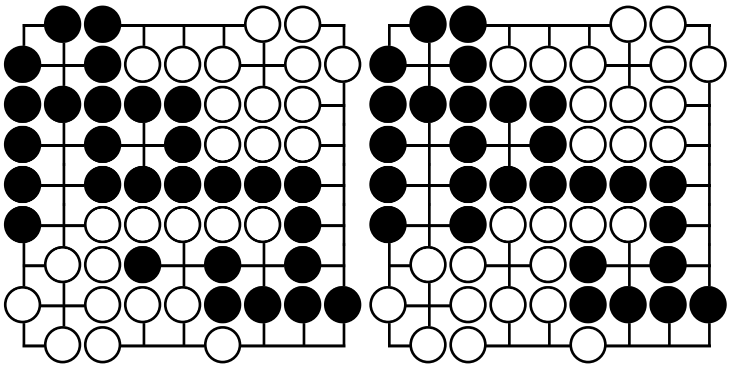
Files marked with a * are excluded from distribution because they are based on material from copyrighted books. The Nihon Kiin's *Graded Go Problems for Beginners* series [Kano 85] provides a thorough test of tactics and Life&Death. It contains many basic but interesting problems. Sources for further computer games are the program authors, tournament bulletins, and the *Computer Go Ladder* that was recently started on IGS [Pettersen 94].

A.3 Table of Full Board Endgame Problems

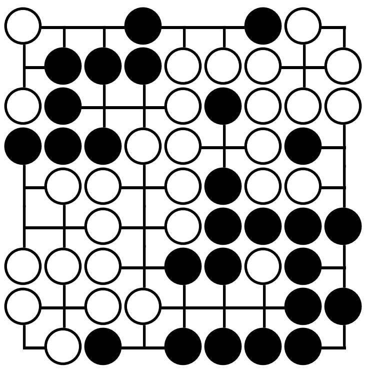
We show modifications of the 9x9 problems from [BW 94, Appendix C], with brief comments on the changes made. Problems C.1 and C.14 could be partitioned 'as is' and no modifications were made. C.20 is not really an endgame problem and is omitted here.



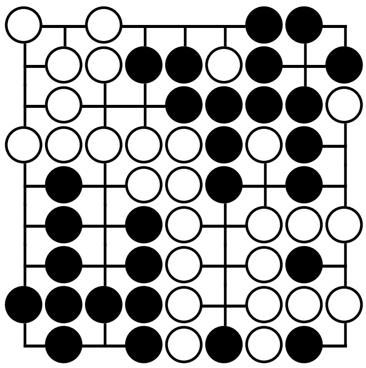
C.2: Added second eye for black group



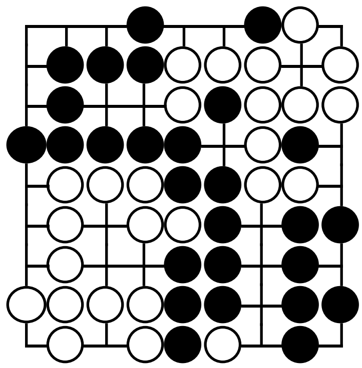
C.4 and C.5: Needed many changes to make blocks alive and separate endgames. Had to replace an unblocked 5 point corridor by a blocked 3 point one.



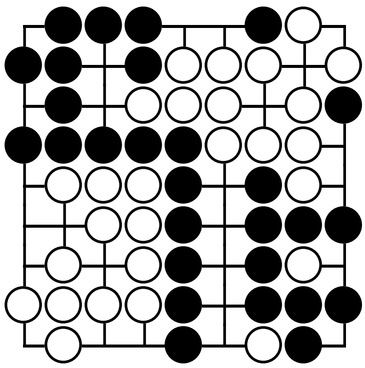
C.6: could not find a good conversion. The version shown here is equivalent regarding endgames and territories, but it has four more white stones than the original.



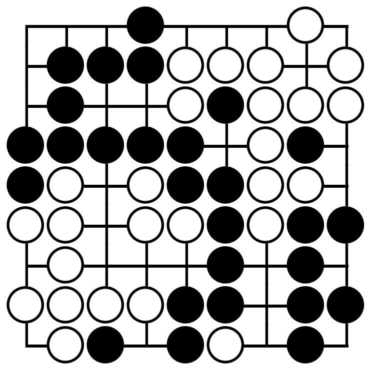
C.7 The example used in Chapter 8



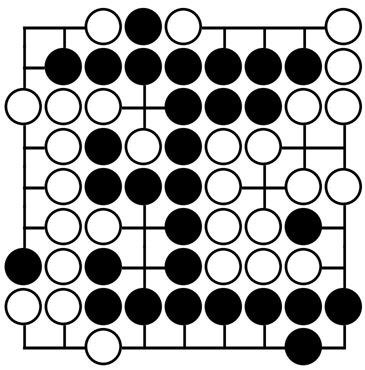
C.8



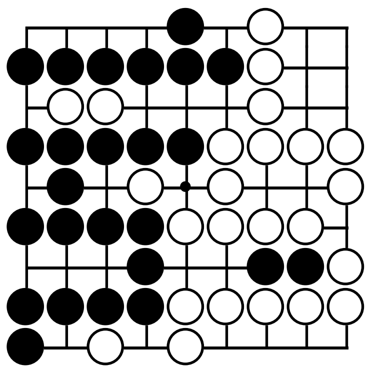
C.9



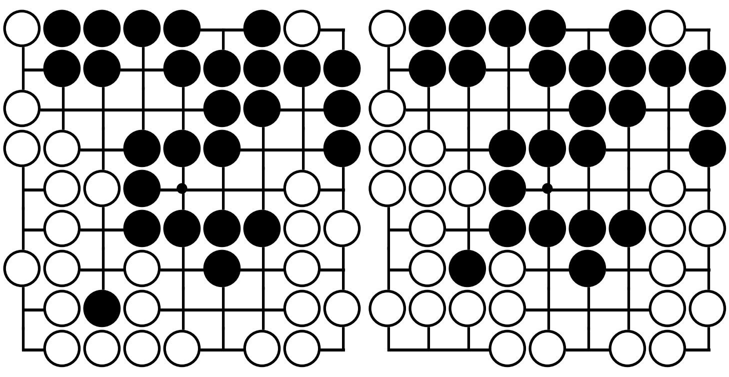
C.10



C.13



C.21



C.15 and C.16 Changes to top left corner, to avoid possible Ko under Chinese rules

Glossary

Most terms in the glossary were assigned to one or more categories:

[CGo]	Computer Go
[CGT]	Combinatorial game theory
[CS]	Computer science
[Go]	Go
[GT]	Game theory
[Ex]	Explorer

More detailed explanations of Go terms can be found in [Bozulich 92].

Term	Description
*	→ Dame [CGT, Go]
Area	A connected set of points on the Go board [Ex]
Awari	An African board game. Many variants exist under names such as Kalah, Kalaha, Wali, Wari etc.
Block	Connected stones of the same color [Ex]
Canonical Form	Loopfree games have a unique canonical form → Chapter 3, p. 38 [CGT]
Chain	Connected set of blocks, joined by connections [Ex]
Chinese Rules	→ Rules
Connection	Unbreakable link between two or more blocks [Ex]
Constraint	*Constraints* model a *dependency* between a game and its context [Ex]
Context	For a local game: an abstraction of relevant features on the rest of the board, and of the history of the local game, especially prisoners and Ko status [Ex]
Cooling	A technique for simplifying games by adding a 'tax' on every move → Chapter 3, p. 39 [CGT]
Corridor	A long narrow area open at the end [CGT]
Dame	A neutral point under Japanese rules, equivalent to the game $\{0 \mid 0\} = *$ in CGT. Under Chinese rules, the value is $\{1 \mid -1\}$. An odd number of dame is equivalent to a single dame, an even number of dame has value zero. [Go]
Dan	Rank for professionals and strong amateur players [Go]
Dependency	Dependencies between local games occur when the areas of local Go games overlap during area expansion, and when external constraints of one local game are related to another game [Ex]
Divider	A small gap between stones of one color, where the opponent cannot connect through. Used to define zone boundaries [Ex]
Endangered constraint	An *endangered* constraint can be violated by the next move [Ex]
Even elementary	Go position with even *parity* involving no Kos or *odd sekis*. → [BW 94, p. 52/53]. [CGT]
Fight	A cluster of weak stones [Ex]
Fuseki	The opening phase of the game [Go]
Geta	A loose encircling move which captures stones [Go]
Global generator	Global move generators need a full board context, local generators operate on local data only [Ex]
Global search	Search on the full Go board. Not recommended in general [CGo]
Gote	A move that can be ignored, that has no urgent follow-up move [Go]

Group	A non-separated set of chains of one color, contained in one opponent zone. The unit of Life&Death considerations [Ex]
Healthy region	Area providing a safe liberty for a block in Benson's algorithm → Chapter 5, p. 62 [CGo]
'Hot' game	Used informally for a game with high temperature [CGT]
Immortal Stones	Stones which are safe independent of endgame play [CGT]
Incentive	The improvement made by moving in a game. → Chapter 3, p. 38 [CGT]
Influence	A heuristic measure for estimating the relative control of both players over points on the board [CGo]
Japanese Rules	→ Rules
Joseki	Standard opening sequence in the corner. More general, any standard sequence (as in *midgame joseki*) [Go]
Kyu	Rank of weaker amateur players [Go]
Life&Death	Procedure to check whether a group can survive by making two eyes (or →seki) [CGo]
Leftscore	Minimax value of a game when Left plays first → Chapter 3, p. 38 [CGT]
Local game	A part of the Go board that is proven or assumed independent from the rest of the board. [CGT, Ex]
Local generator	Local move generators operate on local data only [Ex]
Loopy Game	Game with loops in the game graph → Chapter 3, p. 43 [CGT]
Mean	How many points a game is worth on average → Chapter 3, p. 39 [CGT]
Odd Seki	A *seki* with odd *parity* [CGT]
Option	The positions to which a player can move in a game → Chapter 3, p. 37 [CGT]
Parity	A Go position or local game has even (odd) parity when the number of empty points plus the number of prisoners captured is even (odd). [CGT]
Pattern	A local arrangement of stones and empty spaces [CGo]
Prisoner	Captured stone [Go]
Region	A candidate eye area in Benson's algorithm → Chapter 5, p. 61 [CGo]
Rightscore	Minimax value of a game when Right plays first → Chapter 3, p. 38 [CGT]
Rules	Japanese rules count empty points and prisoners, Chinese rules count own stones and empty points. → [Bozulich 92, BW 94] [Go]
Seki	A stalemate between two or more eyeless groups [Go]
Semeai	Race to capture between non-alive groups [Go]
Semi-infinite string	A string with a start but no end. Used for representing Explorer's patterns [CS, Ex]
Sente	A move that is answered by the opponent. More general, a move sequence is sente if the opponent makes the last move locally [Go]
Sente and Gote	Sente and gote are relative terms and depend on the loss involved in ignoring a 'sente' move. Combinatorial game theory gives a more precise meaning to such notions and can explain some rules of thumb concerning the value of such moves. [Go, CGT]
Sentestrat	An approximate algorithm for sum game play → [Berlekamp 92] [CGT]
Sidling	An algorithm for improving bounds on the value of a loopy game → Chapter 3, p. 44 [CGT]
Sistring	→ Semi-infinite string
Static analysis	Analysis by use of knowledge, without search. [CS, GT]
Stopping Position	→ Terminal Position [CGT]

Sufficiently good move	Move in a sum game that guarantees a score at least as good as the optimal score at the beginning of analysis. In contrast to an optimal move, it might not exploit all opponent mistakes, but is easier to compute. [CGT]
Switch	Simple 'hot' game of the form $a \mid b$ → Chapter 3, p. 40 [CGT]
Temperature	A measure of how urgent it is to move in a game → Chapter 3, p. 39 [CGT]
Temperature estimate	A heuristic upper bound on the local game temperature [Ex]
Terminal Position	A leaf in the game graph that is evaluated statically [GT]
Territory	In heuristic play, a loosely surrounded area. In exact play, a safe area surrounded by immortal blocks. Territory may contain prisoners. [Go, CGo, Ex]
Tesuji	A skillful move [Go]
Thermograph	A graph showing Leftscore and Rightscore of a cooled game → Chapter 3, p. 39 [CGT]
Thermostrat	An algorithm for sum game play based on the *thermographs* of subgames → [Berlekamp 92] [CGT]
Thick and thin	Terms describing the strength of one player's position. Thick positions are strong (no cutting points, many liberties), thin positions are vulnerable to attacks. [Go]
Tsume Go	Life&Death problems, usually in a small completely surrounded area [Go]
Zone	An area surrounded by blocks and dividers of one color, a generalization of territory [Ex]

References

[Allis et al. 91] Which games will survive? ALLIS, L.V., VAN DEN HERIK, H.J., HERSCHBERG, I.S. In: [Levy/Beal 91].

[Allis 94] Searching for Solutions in Games and Artificial Intelligence. ALLIS, L.V., Dissertation, University of Limburg, Maastricht 1994.

[BCG 82] Winning Ways. BERLEKAMP, E., CONWAY, J.H., GUY, R.K. Academic Press, London 1982.

[Beal 89] Advances in Computer Chess 5. BEAL, D.F. (Ed.), North Holland, Amsterdam, 1989.

[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/Kim 94] Where is the “$1,000 Ko”?. BERLEKAMP, E., KIM, Y. Go World No. 71, pp. 65-80, Ishi Press 1994.

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

[Boon 91] BOON, M. Personal communication.

[Boon 94] BOON, M., Message posted on internet newsgroup rec.games.go, 1994.

[Bozulich 92] The Go Player’s Almanac, BOZULICH, R. (Ed.), Ishi Press, Tokyo 1992.

[Bramer 83] Computer Game-Playing: Theory and Practice. BRAMER, M.A. (Ed.) Ellis Horwood Ltd., Chichester, West Sussex, 1983.

[Brügmann 93] Monte Carlo Go. BRÜGMANN, B., Report available by ftp from bsdserver.ucsf.edu.

[BW 94] Mathematical Go: Chilling Gets the Last Point. BERLEKAMP, E., WOLFE, D. A K Peters, Wellesley 1994.

[Chen 89] Group Identification in Computer Go. CHEN, K. In: [Levy/Beal 89], pp. 195-210.

[Chen 93] Binary Game Forest - An approximation model for Go. CHEN, K., unpublished report, 1993.

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

[Clarke 82] Advances in Computer Chess 3. CLARKE, M.R.B. (Ed.), Pergamon Press, Oxford, 1982.

[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.

[Enderton 91] The Golem Go Program. ENDERTON, H.D., Technical report CMU-CS-92-101, Carnegie Mellon University, 1991. Report available by ftp from bsdserver.ucsf.edu.

[Erbach 92] Computers and Go. ERBACH, D.W. In: [Bozulich 92].

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

[Fotland 93] Knowledge Representation in The Many Faces of Go. FOTLAND, D. Report posted on internet newsgroup rec.games.go, 1993. Available by ftp from bsdserver.ucsf.edu.

[Fotland 94] Many Faces of Go. FOTLAND, D., World-Wide-Web page http://cgl.ucsf.edu/go/Programs/ManyFaces.html, 1994.

[Friedenbach 80] Abstraction Hierarchies: A Model of Perception and Cognition in the Game of Go. FRIEDENBACH, K. J. Ph.D. Thesis, Univ. of California, Santa Cruz, 1980.

[Gasser 90] Heuristic Search and Retrograde Analysis: their Application to Nine Men's Morris. GASSER, R., Diploma thesis, ETH Zürich 1990.

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

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

[Gasser 95] Efficiently Harnessing Computational Resources for Exhaustive Search. GASSER, R., Dissertation, ETH Zürich, 1995.

[Geiser 91] Pattern Recognition in the Game of Go. GEISER, P.M., Diploma thesis, ETH Zürich 1991.

[Gonnet 88] Efficient Searching of Text and Pictures. GONNET, G.H., Extended Abstract, UW Center for the New Oxford English Dictionary, June 1988.

[High 92] Mathematical Go. HIGH, R.G. In: [Bozulich 92], pp. 218-224.

[Kageyama 78] Lessons in the Fundamentals of Go. KAGEYAMA, T., Ishi Press, Tokyo 1978.

[Kano 85] Graded Go Problems for Beginners. KANO, Y., Nihon Ki-in, Tokyo 1985.

[Kierulf 82] Brand - an Othello Program. KIERULF, A. In: [Bramer 83].

[Kierulf 89a] Peer Gynt vs. Bill. KIERULF, A. Othello Quarterly 11, 2, pp. 6-8, 1989.

[Kierulf 89b] Quo Vadis Go Explorer. KIERULF, A. unpublished manuscript, ETH Zürich, 1989.

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

[Kierulf et al. 90] Smart Game Board and Go Explorer: A case study in software and knowledge engineering. KIERULF, A.; CHEN, K.; NIEVERGELT, J. Comm. ACM, February 1990.

[Kierulf et al. 91] Every interactive system evolves into hyperspace: The case of the Smart Game Board. KIERULF, A.; GASSER, R.; GEISER, P.; MÜLLER, M.; NIEVERGELT, J. In: Proc. Hypertext/Hypermedia 1991, MAURER, H. (Ed.), Springer Verlag, New York 1991.

[Kierulf/ Nievergelt 85] Computer Go: A Smart Go Board and its Applications. KIERULF, A.; NIEVERGELT, J., In: Go World No.42, pp. 62-64, Winter 1985/86.

[Kierulf/ Nievergelt 89] Swiss Explorer blunders its way into winning the first Computer Go Olympiad. KIERULF, A.; NIEVERGELT, J. In: [Levy/Beal 89], pp. 51-55.

[Kopec/Bratko 82] The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. KOPEC, D., and BRATKO, I. In: [Clarke 82].

[Kraszek 88] Heuristics in the Life and Death Algorithm of a Go-playing Program. KRASZEK, J., In: Computer Go 9, p. 13, 1988.

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

[Levy 88] Computer Games I+II. LEVY, D.N.L. (Editor), Springer Verlag, New York 1988.

[Levy/Beal 89] Heuristic Programming in Artificial Intelligence - The First Computer Olympiad. LEVY, D.N.L.; BEAL, D.F. (Eds.) Ellis Horwood Ltd., Chichester, West Sussex, 1989.

[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), pp. 393-401. (Also: IEEE Symp. on Foundations of Computer Science, (1978), pp. 48-54).

[Marsland/ Computers, Chess, and Cognition. MARSLAND, T. A.; SCHAEFFER, J. (Eds.), Schaeffer 90] Springer Verlag, New York 1990.

[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).

[Moews 93] On Some Combinatorial Games Connected with Go. MOEWS, D.J., Dissertation, University of California at Berkeley, Berkeley 1993.

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

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

[Müller 90] The Smart Game Board as a Tool for Game Programmers. MÜLLER, M. In: Heuristic Programming in Artificial Intelligence 2, LEVY, D.N.L.; BEAL, D.F. (Eds.), London 1990.

[Müller 91a] Measuring the performance of Go programs. MÜLLER, M. In: Proc. International Go Congress, Beijing 1991.

[Müller 91b] Pattern Matching in Explorer. MÜLLER, M. In: Proc. of the Game Playing System Workshop, ICOT, Tokyo 1991.

[Müller 93] Game Theories and Computer Go. MÜLLER, M. In: Proc. of the Go and Computer Science Workshop (GCSW'93), INRIA, Sophia-Antipolis, 1993.

[Müller 94] Explorer User Manual. MÜLLER, M. Internal report, ETH Zürich 1994.

[Müller/Gasser 94] Experiments in Computer Go Endgames. MÜLLER, M., and GASSER, R. To appear in: Proceedings of MSRI Workshop on Combinatorial Games.

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

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

[Nievergelt 93] Experiments in computational heuristics, and their lessons for software and knowledge engineering. NIEVERGELT, J. In: Advances in Computers, Vol 37, pp. 167-205. YOVITS, M. (Ed.), Academic Press, 1993.

[Patashnik 80] Qubic: 4x4x4 Tic-Tac-Toe. PATASHNIK, O., Mathematics Magazine Vol. 53 No. 4, pp. 202-216.

[Peter 93] Erweitertes Pattern Matching in Go. PETER, H.E., Diploma thesis, ETH Zürich 1993.

[Pettersen 94] The Computer Go Ladder. PETTERSEN, E., World Wide Web page, http://cgl.ucsf.edu/go/ladder.html, 1994.

[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, L.V. (Eds.), Ellis Horwood 1992, pp. 157-164.

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

[Ryder 71] Heuristic Analysis of Large Trees as Generated in the Game of Go. RYDER, J. L. Stanford Univ., Ph.D. Thesis, (1971), 1-298 (Microfilm 72-11, 654).

[Schraudolph 94] Temporal Difference Learning of Position Evaluation in the Game of Go. SCHRAUDOLPH, N. In: Neural Information Processing Systems 6, Morgan Kaufmann 1994. Also available by ftp from bsdserver.ucsf.edu.

[Sedgewick 83] Algorithms. SEDGEWICK, R. Addison-Wesley, 1983.

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

[Stoutamire 91] Machine Learning, Game Play, and Go. STOUTAMIRE, D., Reprint of Master's thesis, available by ftp from bsdserver.ucsf.edu, 1991.

[Takagawa 85] How Many Moves is it Possible to Read? TAKAGAWA, S. Go World No.41, pp. 30-33. Ishi Press, Tokyo 1985.

[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, pp. 131-139, 1986.

[Thorp/Walden 64] A Partial Analysis of Go. THORP, E.; WALDEN, W.E. Computer Journal, Vol.7, No.3, pp. 203-207, 1964. Reprinted in: [Levy 88], Vol.II, pp. 143-151.

[Thorp/Walden 72] A Computer Assisted Study of Go on M*N Boards. THORP, E.; WALDEN, W.E. Inf. Sciences, Vol.4 No.1, pp. 1-33, 1972. Reprinted in: [Levy 88], Vol.II, pp. 152-181.

[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, pp. 286-297.

[Wilcox 79] Computer Go - The Reitman-Wilcox Program. WILCOX, B. American Go Journal 14, 5/6 (1979), pp. 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, University of California at Berkeley, Berkeley 1991.

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

[Wolfe 92] WOLFE, D. Personal communication.

[Zobrist 70a] Feature Extraction and Representation for Pattern Recognition and the Game of Go. ZOBRIST, A. L. Ph.D. Thesis, Univ. of Wisconsin, 1970, 1-152. (Microfilm 71-03, 162).

[Zobrist 70b] A New Hashing Method with Application for Game Playing. ZOBRIST, A.L. Tech. rep. 88, Univ. of Wisconsin, April 1970.

Curriculum Vitae

I, Martin Müller, was born on February 3rd, 1965 in Salzburg, Austria. My nationality is Austrian. From 1971 to 1983 I attended primary school and Gymnasium in Salzburg. In 1983 I moved to Graz, Austria, to study technical mathematics at Graz Institute of Technology. I recieved my Dipl.Ing. degree in 1989. Since 1989, I work as assistant and Ph.D. student of Prof. Nievergelt at the Institut für Theoretische Informatik, ETH Zürich.