

Smart game board and Go explorer: a case study in software and knowledge engineering

Report**Author(s):**

Kierulf, Anders; Chen, Ken; Nievergelt, Jürg

Publication date:

1989-05

Permanent link:

<https://doi.org/https://doi.org/10.3929/ethz-a-000505290>

Rights / license:

In Copyright - Non-Commercial Use Permitted

Originally published in:

D-INFK Technical Report 105

83.23.35



Eidgenössische
Technische Hochschule
Zürich

Departement Informatik

Anders Kierulf
Ken Chen
Jürg Nievergelt

**Smart Game Board and
Go Explorer:
A case study in software and
knowledge engineering**

Mai 1989

Eidg. Techn. Hochschule Zürich
Informatikbibliothek
ETH-Zentrum
CH-8092 Zürich

83.23.35

Presented at the Workshop on New Directions in Game-Tree
Search, Edmonton, Canada, May 28 - June 1, 1989.

Address of authors:

Anders Kierulf / J. Nievergelt e-mail: kierulf@inf.ethz.ch
Informatik, ETH nieverge@inf.ethz.ch
CH-8092 Zurich
Switzerland

Ken Chen e-mail: chen@unccvax.uncc.edu
Dept. of Computer Science
University of North Carolina
Charlotte, NC 28223, USA

© 1989 Departement Informatik, ETH Zürich

Smart Game Board and Go Explorer: A case study in software and knowledge engineering

Anders Kierulf, Ken Chen, and Jurg Nievergelt

Abstract

We describe the Smart Game Board, a software workbench dedicated to the development of game-playing programs, as a case study in exploratory software development; and Explorer, a program that plays the Oriental game of Go, as a case study in knowledge engineering. It took four years to build and perfect the Smart Game Board and test it in two applications: An interactive Go calculator that provides basic tactics routines, and a program that plays the game of Othello. By the Spring of 1988 this powerful programming environment and test bed had matured to the stage where a new member of the team, who focused on strategic analysis of board positions, could implement a program to play the full game of Go during a summer of intensive work. In the Fall of '88, in its debugging test, Explorer won 6 games out of 7 in qualifying and competing in the 4th Annual Computer Go tournament in Taiwan.

Although this project is unique in several ways, it is typical of exploratory software development in others. We present it as a case study in the interaction between software engineering and knowledge engineering that illustrates the decisive importance of a powerful software environment.

Keywords: Exploratory software development, expert system, heuristics, search, computer game playing, Go, Othello, chess.

Contents

1. Software engineering and knowledge engineering
 - 1.1 Exploratory software development:
Uncertain resources, open-ended goals
 - 1.2 The unpredictable complexity of knowledge formalization
 - 1.3 A program to play Go: Why? What? How? What can be foreseen?
 - 1.4 The development of computer Go
 2. The Smart Game Board
 - 2.1 What is it? A tool to assist game players
 - 2.2 What games have in common
 - 2.3 Structure of the System
 - 2.4 Implementing Go, Othello, and chess
 3. Explorer
 - 3.1 Characteristics of Go
 - 3.2 What aspects of knowledge engineering are relevant to Go?
 - 3.3 Go knowledge: Design and windfall
 - 3.4 Structure of Explorer
 - 3.5 Experience and current work
- Appendix: Fragments from Explorer Games

1. Software engineering and knowledge engineering

Software engineering is an established discipline that has accumulated and codified more than two decades worth of know-how. Knowledge engineering, on the other hand, is an emerging discipline with lots of issues but, at least so far, little structure. Despite this lack of structure, or perhaps because of it, the practice of knowledge engineering promises to have a noticeable impact on software engineering doctrine. The experimental nature of knowledge engineering goes hand-in-hand with a style of software development best characterized as 'exploratory', which has not been much studied in traditional software engineering.

1.1 Exploratory software development: Uncertain resources, open-ended goals

The vast literature on software engineering discusses, almost exclusively, the production and maintenance of software as an industrial process: Predictable in its outcome, repeatable in its execution, and portable in its independence on particular individuals. Many conditions must be met for the managerial and technical processes advocated in software engineering to work. The principal requirement is that the designer knows fairly well what end product can be achieved (so it can be specified), what resources are required (so the project can be sized up), and how to build the product (so the production process can be planned). The source of all this knowledge is experience based on previous production of similar products. Conventional software engineering know-how works best when producing the (n+1)st version of a compiler, text processor, or other well known software systems component or applications package.

A small but important class of software development projects meets none of the conditions above. For good reasons, such first-of-a kind projects are typically conducted in a manner diametrically opposite to traditional software engineering practice. Typical conditions include:

- The designer has a clear vision of the *direction of achievement*, a "goal at infinity", but has no way to predict how far along this route progress can be pushed.
- For any specified level of achievement, it is impossible to predict the resources needed, perhaps not even their order of magnitude.

- Because nothing can be promised about the outcome, no long-term commitment of resources is made, and the only predictable budget feature is that the project has to be run on a shoestring.
- Software development cannot be planned, as the outcome of each phase and step is unpredictable; it proceeds, mostly by trial and error, along lines of least resistance or greatest promise.
- Last but not least, such a project is intimately tied to the particular individuals involved and reflects their personal characteristics. Removing a key individual will kill the project or transform it beyond recognition; assigning another team to do "the same" project results in a different venture.

The rapidly spreading use of interactive applications has kept software products in a state of flux for more than a decade: Spreadsheets, painting programs, hypertext are just some of the products that lacked practical models a decade ago. All indications point to a continuation of the trend that essentially new types of software will continue to be developed, along with the bread-and-butter tasks of perfecting the next version of existing models. Thus it behooves the software engineering community to recognize that exploratory software development is an activity of long-term importance worth studying. Even a casual acquaintance with several exploratory software projects suggest that they proceed according to rules different than those postulated in the software engineering literature. It is not just a matter of developing a prototype or two before starting on the product, and it is definitely not a matter of going through half a dozen phases of a life-cycle that includes requirements analysis, specification, coding, and testing.

The style of exploratory software development is linked so intimately to the individuals involved that one might be reluctant to search for common traits, expecting only to see talented and dedicated individuals "doing their own thing". The opportunity to observe several such projects by different teams has convinced us, however, that there are significant commonalities. These include:

- A blatant bottom-up approach to systems building: The system evolves around key routines and components that are built early, are perennially modified, and pragmatically combined in different ways in response to feedback. System behavior, far from being specified a priori, is an ever-changing result of these experimental set-ups.
- At all times, have a running system, even if it is only a mini version of what you are striving for. Since specification ahead of time is out,

observation of the program's behavior, and of the users' reactions, is essential at all times.

- Keep your tools sharp! Building an environment useful for rapid prototyping is effort well invested, as code is never "finished", and the turn-around time needed to implement a new version or perform an experiment is critical.
- Strive for a design that leaves as many options open as possible - tomorrow we may know better than today what option to exercise.
- Keep your ears to the ground! A significant project spans years, and during its life time the surrounding circumstances change considerably - a good dose of opportunism is necessary to keep the project relevant and up-to-date.

Software developed according to this philosophy evolves through selection a bit like natural systems do - with one important exception. A single team cannot afford to imitate prolific nature and spawn mutations to be developed concurrently. At any time, the cumulative experience of the entire project must be distilled into one running version. It is worth remarking, though, that in situations where several independent teams work on a similar goal, the analogy with natural selection, and the multitude of related species it creates, is apt - consider, for example, the many versions of UNIX in existence.

1.2 The unpredictable complexity of knowledge formalization

The development of an expert system, or any project that involves the formalization of an open-ended domain of human knowledge, adds specific difficulties associated with knowledge engineering to all the traditional problems of software engineering. The chief difficulty added by knowledge engineering is that hardly anything of importance seems to be predictable, and hindsight becomes the only reliable basis of judgement. This is a drastic departure from the norm of engineering development, which is based on the assumption that we know ahead of time what we will end up with. It forces computer scientists to look at the process of formalization in a new light.

Computer science is the technology of formalization: If anything is to be processed by computer, the rules of processing are expressed in a formal system. This system may be a programming language defined by a collection of hardware and software that includes a computer, an operating system, and a compiler; it may be a programming or

specification language defined abstractly; or it may be a more traditional mathematical system built on axioms and rules of inference. The programmer may or may not think of his/her activity as a task in formalization - the point here is that, once a program has been written, it is a formal system.

One can view computer science education as having two major goals: To teach a sufficiently rich arsenal of *techniques of formalization*, and, in projects and case studies, to impart as much experience as is feasible concerning the *practical application* of these techniques. Thus we expect, for example, that a systems programmer can formalize a communications protocol, and a systems analyst or data base administrator can formalize the structure of payroll information. Both of these examples rely on the fact that the domain of knowledge to be formalized is closed, as opposed to open-ended, and of very limited size and complexity.

The word 'knowledge engineering', on the other hand, is typically used for domains of knowledge much larger and more complex than the examples above. Attempts at formalization capture only a small fraction of the relevant domain. This is true, for example, of standard "microworlds" of artificial intelligence, such as chess. The designer usually cannot even formalize all of his *personal* knowledge of the field, let alone that possessed by the community at large. The literature on expert systems at times presents "interviews" of human experts as the way to capture knowledge, but this is really a technique of *identifying* knowledge, rather than formalizing it.

Given this state of affairs, the knowledge engineer typically starts with an educated guess of what small part of the vast collection of know-how can be formalized and will result in acceptable performance, then repeats the following steps until patience or resources run out: Implement this subset, observe the outcome (it can always be improved!), and tune the knowledge base to achieve a hoped-for effect. This is a basic human mode of operation, but is not what traditional software engineering doctrine has been advocating, and is not what most software environments support.

We illustrate these points with a case study of an exploratory software development project with a large knowledge engineering component: its goal, a program to play the Oriental game of Go, is exotic and needs explaining, including a brief survey of the history and current state of computer Go.

1.3 A program to play Go: Why? What? How? What can be foreseen?

As a case study in exploratory software development and knowledge engineering, we present Explorer, a program that plays the Oriental game of Go. It took four years to build and perfect the Smart Game Board, and test it in two applications: An interactive Go calculator that provides basic tactics routines, and a program that plays Othello. By the Spring of 1988 this powerful programming environment and test bed had matured to the stage where a new member of the team, who focused on the analysis of strategic features of board positions, could implement a program to play the full game of Go during a summer of intensive work. In the Fall of '88, in its first test against other programs, Explorer tied for 2nd among 16 programs that competed in the 4th International Computer Go Congress in Taiwan, the "unofficial world championship" organized by the Ing Foundation.

Although this project is unique in several ways, it is typical of exploratory software development in others. We believe there are lessons to be learned with respect to the interaction between software engineering and knowledge engineering. This case study illustrates the decisive importance of a powerful development environment. A large investment of effort in building the best software engineering environment we were able to devise, out of components that are well understood, made it feasible to experiment extensively in the poorly understood realm of knowledge engineering.

Why work on a Go program?

Work on game-playing programs is always, in part, a hobby. But there must be other justification as well, because game playing programs, when pursued far enough that they can compete among the best, require several man-years of effort. Such justification is readily found, in an academic environment, in terms of education and research.

Education. The idea of robots playing games of strategy has intrigued people since the invention of clockwork, but it remained for electronic computers to turn it into reality. Famous computer pioneers prepared the ground: John von Neumann created the mathematical theory of games, Alan Turing and Claude Shannon devised techniques that allow computers to play chess [Sh 50, Tu 53]. The concepts and techniques developed range from game theory to knowledge representation, and from heuristic search to program optimization. Many of these ideas have become an accepted part of computer science lore, and students should be

exposed to them. The concepts are empirical more than they are theoretical: The typical question is not whether something can be done in principle, but how well it can be done within practical constraints. In computer science, *a meaningful exposure to empirical concepts must be based on working programs*. We have been conducting seminars on heuristic search and computer game playing in which the Smart Game Board serves as a workbench and test-bed that makes it possible for students to write and test a game-playing program of their own design in one semester. It was a seminar held in '86 over North Carolina's state-wide two-way teleclass network that brought the authors together in co-operating on a Go playing program.

Research. A significant part of our knowledge of what computers can and cannot do in artificial intelligence comes from experiments with computer game playing - so much so that computer chess has been called the "drosophila of AI", i.e. the preferred test object for experimentation. This assessment is based on two very useful properties. 1) Games of strategy form "microworlds" of just the right complexity (simple enough to be easily formalized, hard enough to tax computational resources) to develop and test our growing arsenal of artificial intelligence techniques. 2) Progress can be measured accurately in terms of improved playing performance - a chess rating or a Go rank are much more objective measures than exist in most other applications of knowledge engineering.

What to work on?

We took up game-playing programs as a side-line in 1983 based on a combination of factors: personal interest, the view that there was still unexplored land, and the conviction that knowledge engineering must be approached empirically in a microworld where *performance can be measured objectively, accurately, quantitatively*. There was a lot of experience and information about chess playing programs, and computer chess appeared to be in a relatively mature stage where success came from doing the same thing better and faster. Comparatively little was known about computer Go and other games. Go has a reputation for being the most profound game of strategy, thus may be one of the hardest to program, and hence a fertile field for experimentation. We are aware of four Ph.D. theses written on computer Go [Zo 70, Ry 71, Fr 80, Le 81], but in contrast to chess, computer programs that play Go were still in their infancy, playing so poorly that their performance was essentially off the accepted scale for ranking human players. An increased interest in computer Go could be expected; in particular in Japan, where Go is a national hobby, and the government-sponsored

project on Fifth Generation Computer Systems greatly boosted research in artificial intelligence.

How?

The combination of 1) little existing experience, and 2) an anticipation that during the course of the project many competitor projects might emerge, caused us to approach the subject on a broad front, and to design a project that emphasized basics and could be redirected to focus on different goals depending on where progress appeared most promising as the work evolved. Rather than setting specific goals, we decided on a few directions, on "goals at infinity", and embarked on a journey that would lead as far as we could go. These directions included:

1. Design and build a workbench, later called the Smart Game Board, for the development of game-playing programs. What do various games have in common, what components can be shared among programs that play different games? This workbench quickly became a useful teaching tool - a skeleton program to be tailored to specific games in student term projects. Soon it became a useful vehicle to maintain and enhance an Othello program developed independently [Ki 83], and to implement Go-Moku in one semester.
2. The Smart Go Board [KN 85] - a useful teaching and studying tool for the Go player to analyze, annotate, store, and print games. This would ensure a small but dedicated community of users to help in testing and improving the program.
3. Go tactics calculator, expert-guided search. Goal at infinity: Can a person play noticeably better using a Go tactics calculator than when he relies on his own wits?
4. Testbed for experimenting with strategies and techniques for playing a full game of Go.

Section 3 of this paper is devoted to 4), so let us briefly mention the earlier phase 3), where we attacked the seemingly easier problem of developing a "Go calculator", with a function analogous to that of a handheld numerical calculator. Its purpose is not to play Go against an opponent, but to assist a human player in those tasks that are better delegated to a machine because they require speed and accuracy, as opposed to expert judgement. The human player decides on strategy and calls upon the program to calculate tactical sequences of moves to achieve a well-defined purpose. As the program builds and traverses a search tree, the

player directs the search and can interactively modify the board if he so chooses. This approach can be described as using computers as "amplifiers of human intelligence", rather than as autonomous intelligent machines. And rather than building a computerized expert system that tries to capture human knowledge in executable form, we investigated 'expert-guided search', where human experience and intuition guide the great processing power and fast and accurate memory towards fruitful goals. This emphasis on interactive user control investigates computer-aided human decision making. Games of strategy such as Go have always served as simplified models for decision making. In the modern era of decision support systems, it is reasonable to assume that programs that support gaming can serve as test beds for some aspects of computer support for decision making. A reasonably objective test of this game calculator is easy to arrange: does a player with calculator play consistently better than without?

What can be foreseen?

When this project was started in 1984, hardly any guidance was available: A couple of examples served as warning of difficulties to be expected, none provided a model to be followed. With hindsight gained through four years' worth of experience, it is interesting to compare some of the questions and expectations we had at the start of the project, and current answers.

Foreseen: A Smart Game Board is feasible as a useful hypertext medium for storing an annotated collection of games, and browsing through them. It can support a variety of games and run on a low-cost personal computer. Advancing technology quickly solved initial problems such as insufficient main memory and disk, low screen and printer resolution.

Might have been foreseen, but the idea evolved during the project: It is feasible to develop a common user interface that fits entirely into one screen, powerful enough to handle many different games. Interaction with different games takes place using the same windows, and, to a large extent, the same operations. This is possible because the majority of operations provided by the Smart Game Board are defined on the game tree, and thus are game-independent. So far the ones shown in the control panel have proven to be adequate for Go, Othello, Go-Moku, and Chess.

Unforeseen: How hard is it to develop a useful Go tactics calculator? We expected to develop an interactive program capable of analyzing and solving clear-cut tactical tasks identified by an expert amateur user. This

tactics calculator has proven to be much harder than anticipated, and so far is useful for only a very limited range of tactical problems: Mostly, complicated ladders and loose ladders, i.e. forcing sequences in capturing a block that has no more than 2 or 3 liberties at any time. With hindsight, of course, we can see why this problem is hard - the interface between human strategy and machine tactics requires difficult formalization of intuitive, fuzzy concepts. To substantiate this point, we can do no better than quote the operator of Deep Thought (DT), today's strongest chess computer, as it assisted grandmaster Kevin Spraggett in an elimination match for the world championship [Ja 89]: 'I was in the fortunate position of being able to watch and assist the possibly first-time cooperation between an IGM and a computer. ... Other problems were related to "interfacing" DT with Spraggett or his seconds. It is indeed hard to translate a question like "can black get his knight to c5?" or "can white get an attack?" or "what are black's counter chances" into a "program" for DT, since it just doesn't have these concepts. Translating back DT's evaluations and principal variations (PV) is easier, but this information is incomplete most of the time. Indeed, the alpha-beta algorithm generally gives accurate values on the PV only (other lines are cut off), whereas *a human wants to know why alternatives fail*'. Given the weak play of today's Go programs, a useful Go tactics calculator is far off.

Unforeseeable: How strong a Go program to aim at. The lack of experience prevailing in the fifties and sixties triggered useless predictions about the strength of chess programs that ranged from wildly optimistic ("a computer will be world champion within the decade") to overly pessimistic ("computers will never beat human experts"). Similarly, there was no rational basis for predicting how strong, or weak, a Go program we could develop. With hindsight, observing that half a dozen independently developed programs that competed at the most recent computer Go tournament are comparable in strength, one can conclude that a knowledgeable and dedicated small team can develop a 15 kyu Go player (this rating scale is explained in section 3) with an effort of a few man-years.

1.4 The development of computer Go

A brief survey of the history and current state of computer Go is appropriate to place our project in perspective. Compared to other games of strategy, such as checkers, chess, backgammon, and Othello, computer Go started late and, until recently, made very slow progress. Many people feel this is due to the inherent difficulty of Go - a view well ex-

pressed in [Bra 79] as "The Game of Go - The Ultimate Programming Challenge?". In section 3.1 we discuss some reasons for this assessment.

Computer Go started with the dissertations of Zobrist [Zo 70, see also Wi 78] and Ryder [Ry 71, see also Wi 79]. Zobrist relied on pattern recognition, Ryder on tree search. Both of these approaches had already been explored in chess, where the latter, in particular, had proven its value beyond doubt. But although these approaches capture part of what Go is about, neither program played as well as a beginner plays after just a few games of experience. Human players use pattern recognition to a large extent, but their patterns are more complex and abstract than those used by Zobrist. Neither is pure tree search good enough: Without a clear goal to focus the search, the branching factor is far too high.

The Reitman-Wilcox program [RW 79], based on a representation of the board that reflects the way good players perceive the board, was a big step forward. They concentrated on strategy; some tactical analysis was included late in the project. Their approach to model human perception is promising, but even so the program has only reached the level of a novice player. Bruce Wilcox analyzed the strong and weak points of his first program, then rewrote it from scratch [Wi 85].

Since 1985 activity in computer Go has mushroomed. A strong Go-playing program was a goal of the Japanese "Fifth Generation" project, but was dropped (too hard?). Several dozen individuals and small teams are now involved in Go programming, and these efforts are beginning to bear fruit. The annual International Computer Go Congress, organized since '85 by the Ing Foundation of Taiwan, has provided an arena for competition and a yardstick for assessing progress. The best Go programs now play at the level of 12 to 15 kyu, distinctly better than human beginners. But they have a long way to go as compared to chess, where machines have achieved master ratings.

2. Smart Game Board

The Smart Game Board supports two kinds of experiments with game-playing programs. First, as a hypertext medium: Its **user interface** is designed to help serious players study the game in ways that are impossible without a computer. Second, as a workbench for programming games: Its **structure** encourages experimentation with different search algorithms and strategies. Neither of these functions is bound to any one game; originally designed for Go, the Smart Game Board was expanded to include Othello and chess.

2.1 What is it? A tool to assist game players

The Smart Game Board is a tool to assist game players in the many activities they now perform on a wooden board and paper: Playing, teaching, discussing, analyzing, studying, recording and printing. A personal computer should make a better tool for some of these activities, but, as with any new application, it is not a priori clear what functions a game workbench can and should provide. At the very least, it must provide the functions of a wooden board. This is much harder than it seems, because the board is not just used to play a sequence of moves: Players talk while pointing at it and rearranging stones at will. In computer jargon, it is a shared workspace that supports very fast and convenient editing.

A wooden board is more agreeable than a computer screen, and players will only switch to the computer if it offers additional benefits: Easy annotation, ready access to a library of games and openings, help with tactical analysis, or a strong opponent. The Smart Game Board has passed the threshold of usefulness: About a hundred players use it, many say it is their preferred way of studying the game, and it has been used at tournaments to record games and between rounds as an opening library. Its versatility is proven by applications we had not directly designed for: One-on-one lessons by professionals, and taking notes during lectures on Go.

2.2 What games have in common

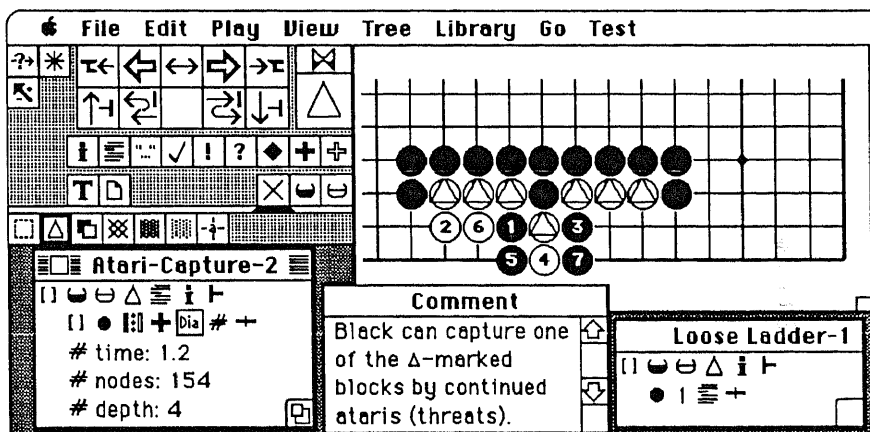
There is no point in trying to make the Smart Game Board general enough to be used for any kind of game - games differ too much to be all brought under one hat. Our choice of Go, Othello, and chess reflects personal and professional interests, but we think they represent a sufficiently diverse sample of an important class of games: Two-player board games with perfect information. How do these characteristics make it easier to build a game-independent user interface?

- **Two players:** Limiting the games to two players, Black and White, reduces the complexity of the interface without unduly restricting the set of games.
- **Board game:** The board is a convenient visual representation of the current state of the game, and makes it easy to enter a move by clicking or dragging the mouse. Throwing dice or drawing cards would make it more complicated to enter a move.
- **Complete information:** If the player always has access to the complete state of the game, there is no need to hide information from one of the players.
- **Sequence of moves:** Most games evolve as a fixed sequence of discrete actions (moves) by the players. Action games like PacMan on the other hand, where moves come any time and timing is critical, do not fit this model well.

These commonalities allow us to implement the functions presented below independently of any particular game. Renju (Go-Moku) was included in an earlier version; other games that could be added without major changes include: Kalah, Nine Men's Morris (Mühle), Checkers, and Connect Four. It would be harder to integrate card games like Blackjack, Bridge, and Poker, dice games like backgammon, or word games like Scrabble. Game-specific differences are discussed in more detail later.

An Editor for Game Trees. The Smart Game Board is an editor specialized for game trees, and its user interface shows similarities with other editors. A game is treated like a document in a text editor, which can be opened, modified, and saved. We concentrate on the aspects that distinguish it from other editors.

The underlying structure of the Smart Game Board is a tree, where each node has a list of properties associated with it. In its simplest form, the tree degenerates into a sequence, and each node has exactly one property, the move. The generality of a tree is needed to let the user experiment with different sequences of moves while keeping the original game intact. The tree is even extended to a directed acyclic graph for opening libraries (different move sequences may lead to the same position). The concept of a list of properties at each node is needed to store comments referring to the move, marks declaring the move to be good or bad, the time left at this point in the game, and so on.



A game is replayed by moving around the tree: Advancing by one node executes the move stored at that node. It is useful to provide various ways of moving about the tree: Depth-first traversal, go to next/previous branch point or terminal node, or search for nodes with specific properties. General tree editing commands allow one to copy properties, nodes, and entire subtrees from one game to another. More powerful operations on game trees include backing up values in min-max fashion, or sorting the branches according to the number of terminal nodes in each subtree. Go, chess, and Othello players intuitively understand the concept of a game tree, as it is the simplest notion powerful enough to capture what they are actually doing when replaying and discussing games. For standard operations, users need not deal with the underlying model; for example, playing a move automatically adds a node to the tree, adds a move property to that node, and executes the move. Simpler interfaces can be designed for programs that offer fewer services, e.g., are designed for just playing against a computer, or for looking up an opening library. In contrast, the Smart Game Board is a general tool that helps serious players in unexpected ways.

Data Base for Games. Data base software dedicated to games, such as ChessBase [Nu 88], is rapidly becoming an essential tool for serious players, as it helps them track large collections of games, standard openings, statistics, etc. Users can order games by openings, play through famous master games, or get a printout of the games their next opponent recently played. Today, games from major tournaments are quickly made available to subscribers in machine-readable form.

The Smart Game Board provides similar functions, but is not yet as good a database system as we aim at. It has been used to organize 1300 Othello games and therefrom create a library of standard Othello openings. Given an opening position on the board, it retrieves all matching games from its database, as shown below.

Game Info

Games: 5 Black: 2 White: 2 Result: B+1

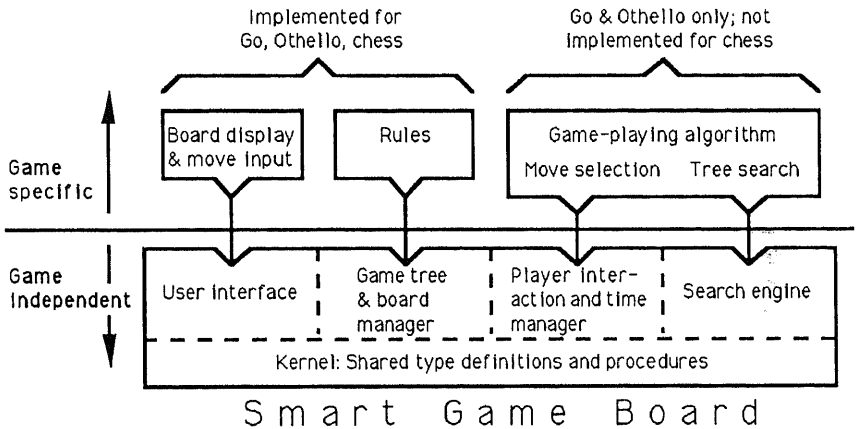
Kanto60 #01	M. Takizawa	D. Shaman	B+10	↑
DK87 #04	K. Tanida	T. Murakami	W+	
DK86 #84	K. Tanida	T. Murakami	B+2	
W88 #S-5	T. Murakami	G. Brightwell	0	
DK86 #33	K. Tanida	T. Murakami	W+6	↓

The board is an 8x8 grid. Black pieces (●) are at (2,3), (2,4), (3,3), (3,4), (3,5), (4,4), (4,5), (5,4), (5,5). White pieces (○) are at (1,3), (1,4), (1,5), (2,2), (2,5), (3,2), (3,6), (4,3), (4,6), (5,3), (5,6). Letters 'c', 'b', and 'a' are at (2,2), (3,2), and (3,6) respectively.

Game Server. Some users working on their own electronic game board or game-playing program want to have access to Smart Game Board's functions from their own program. For example, it can be used as an opponent to test the strength of their programs, it might extract a game from the game library, or it could provide a collection of positions for statistical analysis. For that purpose, the program provides a serial interface where most functions available to the user can be accessed with textual commands. The concept of a game server also helped us test the Go program, because different versions of the program could play each other overnight. However, more work is needed to turn this personal game server into a service people can connect to and play games against.

2.3 Structure of the System

The structure of the Smart Game Board is designed to separate game-specific details from game-independent code. It provides slots for game-specific functions where each game can plug in its routines for the rules (legal move recognition and execution), the user interface (board display, move input, menu functions), and an optional playing algorithm. A search engine provides depth-first search and iterative deepening for different games or tactics that plug in routines for move generation, position evaluation, and time control.



The program is written in SemperSoft™ Modula-2 under MPW (Macintosh Programmer's Workshop). It runs only on the Apple® Macintosh™; portability was not a primary design consideration for a unique program developed with limited resources. The program now consists of a total of 44'000 lines of code (including definition modules and comments, excluding blank lines). The distribution among the components is as follows:

Smart Game Board:	55%	Go:	35%	Othello/Chess:	10%
User interface:	18%	Go rules:	3%	Othello rules:	2%
Tree manager:	8%	Go strategy:	25%	Othello play:	6%
Player & time mgr:	8%	Go tactics:	7%		
Search engine:	6%			Chess rules:	2%
Kernel:	15%				

The Go program compiles to 400 kBytes of object code, the Othello program to 250 kBytes. In addition, at least 150 kBytes are needed for data structures; if more memory is available, a larger hash table and a larger tree are allocated. Game-specific modules make their presence or absence known by installing procedures in some common lower-level module. This structure facilitates experimentation with different search algorithms as well as different games, and it allows us to configure different versions of the program. While handling a command, the program switches back and forth between game-specific and game-independent routines several times. For example, a click on the board intended to enter a move triggers a call to a routine to track the mouse until it is released. This routine is game-specific so as to give feedback appropriate to the game, e.g. in the case of illegal moves. Once the move is recog-

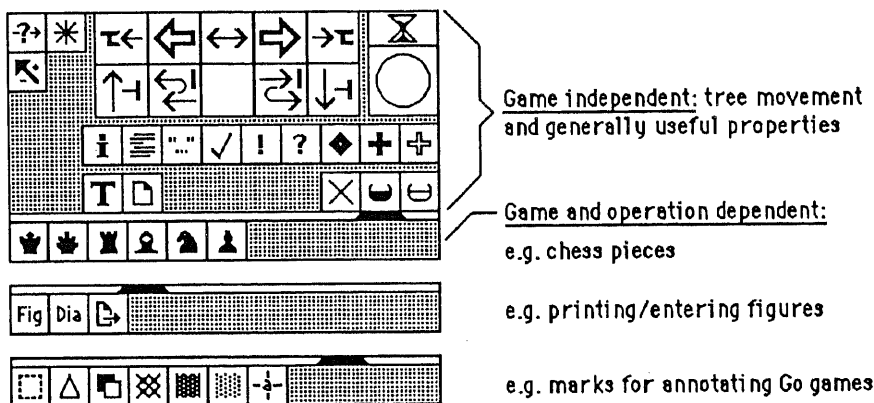
nized as legal, a new node is added to the tree independently of the type of game. Then another game-specific routine is called to execute the move stored in the node.

2.4 Implementing Go, Othello, and chess

So far, Go, Othello, and chess have been implemented in the game workbench. Go came first and determined structure and user interface. Integrating an existing Othello program in the Go framework forced us to restructure. But only a few game-dependent operations were added, because Go and Othello are superficially quite similar: Both have a board with black and white stones, stones are added to the board and never moved, the controlled territory at the end of the game decides the winner. Adding chess forced us to make several operations game-dependent, e.g. move input and move representation (move from-to instead of just a point on the board), and editing the board position (black and white stones suffice for Go and Othello, chess has several kinds of pieces). In addition, such details as specifying castling status when setting up a position are a game-dependent nuisance. Nevertheless, adding chess to the Smart Game Board was a minor task compared with implementing even a rudimentary chess user interface from scratch.

The original Othello algorithm included in the Smart Game Board was "Brand", which placed second at the 1982 North American Computer Othello Championship in Evanston. Recently, a better Othello program, "Peer Gynt", was created by writing a new evaluation function and improving the time control. This work was done in only two man-weeks by using many existing building blocks from Brand and from the Smart Game Board. An early version of Peer Gynt placed third at the 1989 North American Computer Othello Championship in Northridge.

Most of the user interface is the same for each game. Game-dependent operations and status information are provided by a separate menu for each game, and by a part of the panel that is reserved for game-dependent properties and commands, as shown below.



What does it take to add another game to the Smart Game Board? A game-specific module that implements two functions:

- Recognize, execute, and undo legal moves.
- Display the board and track move input.

Go-Moku, for example, is played on the same board as Go, and thus required no change to the board display. If the surface of a new game differs significantly from any of the implemented games, it is probably easier to change some parts of the Smart Game Board to better accommodate this and future games, rather than forcing the new game into the current framework of the Smart Game Board.

3. Explorer

3.1 Characteristics of Go

Assuming the reader knows little or nothing about Go, we attempt to provide some intuition for this game's domain of knowledge, in part by comparing Go to other games. Several excellent introductory books are available from [ISHI]. Go is a two-person game of perfect information in the sense of game theory; at all times, both players know the entire state of the game. The players alternate placing a black and a white stone on some empty intersection of a 19 by 19 grid. Once played, a stone never moves, but it may disappear from the board when captured. A player's objective is to secure more territory than his opponent, counted in terms of grid points. In the process of surrounding territory by laying down a border of stones that must be 'alive', fights erupt that typically lead to some stones being captured ('killed'). Much of the difficulty of Go comes from the fact that during most of the game, few stones are definitively alive or dead. Stones are more or less vital or moribund, and their status can change repeatedly during the course of a game, as long as the surrounding scene can change. Only when the game has ended can all stones be classified definitively as alive or dead. Thus *'life or death', the key concept of Go*, exhibits a split personality. As an operational concept during the game, it is the most important factor in estimating potential territory and for assessing the chances of battle, but is rather fuzzy. It becomes more and more precise as the game progresses, and is a well-defined concept used for counting territory when the game has ended. The game ends when neither player can find a profitable move and all points are classified as one of black, white, or no-man's-land. This situation typically arises after about 200 to 300 moves have been played, with anywhere between 60% and 90% of the 361 grid points occupied. Whereas in chess we count a move (by a white piece) and counter-move (black piece) as a single move, in Go we count the placement of each single stone as a move. Even keeping in mind that a Go move corresponds to half a chess move (a 'ply'), it is evident that a Go game can take a long time.

Professional games last one to two full days. The rich tradition of Go records instances of games that took months. Kawabata Yasunari, winner of the Nobel Prize for Literature in 1968, considered his novel 'The Master of Go' [Ka 51] to be his finest work. It is the chronicle of a single game, played in 14 sessions from August through December of 1938, a contest for supremacy between the heretofore invincible old Master of Go, Honinbo Shusai, and his younger challenger, Kitani Mi-

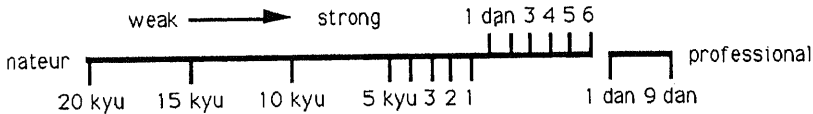
noru. It is a moving tale of the contest between tradition and change, and, ultimately, between life and death.

If chess is a model of a single battle (as fought thousands of years ago), Go is a model of war: Typically, several loosely interacting land-grabbing campaigns and battles proceed concurrently. Go is a great game of **synchronization**: The stronger a player, the better he is able to coordinate his campaigns and disrupt the coordination among enemy forces. Multipurpose moves are the most effective, such as a 'splitting attack' that wedges in between two enemy groups, or a move that threatens to kill an enemy group and extends one's own territory at the same time. Typically one player has the initiative, called 'sente', which enables him to play strong threats that leave the opponent little choice but to respond locally. Thus the player with sente can choose the field of action to suit his goals, whereas his opponent, said to be in 'gote', "follows him around the board". The sente/gote relationship alternates between the players, but among opponents of different skill, the stronger player will manage to keep sente most of the time.

Intuition and experience let a player estimate the numerical value of each move he considers. In the opening, a typical move may be worth about 20 points (of "equivalent territory"). Endgame fights, at the other extreme, may be about a single point, or even 'half a point'. This phenomenon of decreasing value of a typical move as the game progresses causes **timing** to be of the utmost importance. If a move is estimated to be worth x points in a local context, it must be played at just the right moment, when the value of other moves is also about x . If played too early, the opponent may ignore this move and reply with a bigger one elsewhere, perhaps gaining sente. If played too late, when the value of other available moves has diminished, the opponent may prevent it, even at the cost of ending in gote. Players analyzing a game are always debating which move, among several good ones, is 'the largest'.

A **handicap** system makes it possible to balance players of different skill without changing the nature of the game much. The weaker player starts by placing anywhere from 2 to perhaps 13 stones on the board. In Japanese Go, these stones are placed in a fixed pattern on 'handicap points'. In Chinese Go, the weaker player places them anywhere - he starts with x free moves. The handicap system defines a fairly accurate **rating scale**, illustrated in the figure below. Player A is x stones better than B if the chances become equal when B receives x handicap stones. Playing strength of amateurs is measured on a scale where one unit corresponds to a handicap stone. A very weak player may be 20 kyu, implying that he receives 10 handicap stones from a 10 kyu, who in turn

receives 9 handicap stones from a first kyu. A first kyu 'takes Black', i.e. plays first, against a first dan, who receives 5 stones from a 6 dan. That's about as high as the amateur scale goes. Above that, there is a separate scale for professional dan players, from the first (lowest, perhaps equal to an amateur 6 dan) to the 9-th (highest). The professional scale has a finer grating: A difference of 3 levels corresponds to at most one handicap stone.



As computer Go is in its infancy, one would like to build on the mature experience gained with other games, but such experience does not transfer readily. At chess, for example, computers owe their spectacular prowess more to the computer's speed at searching than to their knowledge of chess lore. But the computer chess approach of full board search appears to be impractical in Go, for two reasons:

- **Branching factor.** As Go is played on a 19*19 board, the number of legal moves decreases from 361 at the start to several dozen at the end, creating a tree with an average branching factor of about 200. Compared to a branching factor of about 40 legal moves from a typical chess position, the large fan-out of a Go tree greatly reduces the depth of feasible search.
- **Position evaluation.** Material and mobility, the dominant factors in chess evaluation functions, are easily computed. In Go, possession of territory is the closest equivalent to material possession in chess, but its evaluation is much more subtle: Except at the very end of the game, a player's claim to territory can always be challenged. Go has no clear analog to chess mobility; perhaps 'shape' comes closest, but good and bad shape are very hard to measure.

Whereas success in computer chess was achieved mostly by sidestepping the issue of knowledge engineering, and using fast search to compensate for meager chess knowledge, the analogous recipe is unlikely to lead to comparable success in Go. The insight that *both domain-specific knowledge and search* are of critical importance makes Go a more diverse and balanced test bed for artificial intelligence research than chess is. Computer Go may well become the new "drosophila of AI".

Despite the large branching factor, strong players routinely look 10 to 20 moves ahead when a fight demands it - an activity called 'reading'. This perplexing term suggests that a player, at that moment, is not free to let his imagination roam, but simply has to discover what is given - whether a plausible, perhaps forced, sequence of moves works or doesn't. Go does know the concept of *narrow and deep search*, as does chess, but with a difference. 'Reading' is always limited to a local scene, say a 5 by 6 corner, and never extends to the full board. Even if reading guarantees a win in a local battle, that does not necessarily mean much - the enemy might ignore this battle and get compensation elsewhere. A separate, more intuitive mental act is required to assess the relevance of such a local search to the overall situation. So far, there is no alternative to the vast amount of Go knowledge accumulated by analyzing thousands of professional games each year, compiled and distilled in hundreds of Go books and journals. Explorer is designed to explore the feasibility of a knowledge-based approach to computer Go, augmented by focused tactical lookahead in local fights.

3.2 What aspects of knowledge engineering are relevant to Go?

It is useful to distinguish four major steps in the task of formalizing knowledge: Acquisition, selection, representation, and use. The nature and difficulty of each step differs from application to application. Acquisition and representation are most widely discussed in the literature, but turn out to be non-issues in our project of formalizing Go knowledge. Selection and use, on the other hand, are critical.

Knowledge acquisition. How do you get at the subject matter knowledge that you may want to capture? In contrast to other applications where this knowledge may be scattered and needs to be painstakingly assembled, we have all the Go knowledge at hand necessary at this stage of our project - mostly as the experience of Ken Chen, amateur 6 dan. It is interesting to reflect on the fact that most progress in computer chess is due to amateurs of medium playing strength - strong chess players were evidently not essential to the development of champion chess machines. This reflects the fact that computers can play amazingly powerful chess without much explicit chess knowledge. We expect that explicit representation of game-specific knowledge will be much more important for Go than for chess.

Knowledge selection. Selecting a 'domain of discourse or competence' that matches the program's abilities is the crucial task. At this stage it is hopeless to develop a Go program by trying to capture as much Go knowledge as possible. And if we managed to get a lot of knowledge

third line separated by two empty spaces', and thus can have no explicit rule about them. But Explorer seems to know this fact anyway, as reflected by the relatively high influence on these points, as shown in Fig. b). The influence measure was tuned to produce a reasonable result on such standard configurations. Unlike a collection of explicit rules that can never capture all situations, influence applies to all configurations, and gives reasonable results for most. It is also used to define a territorial claim for each group, shown shaded in Fig. c).

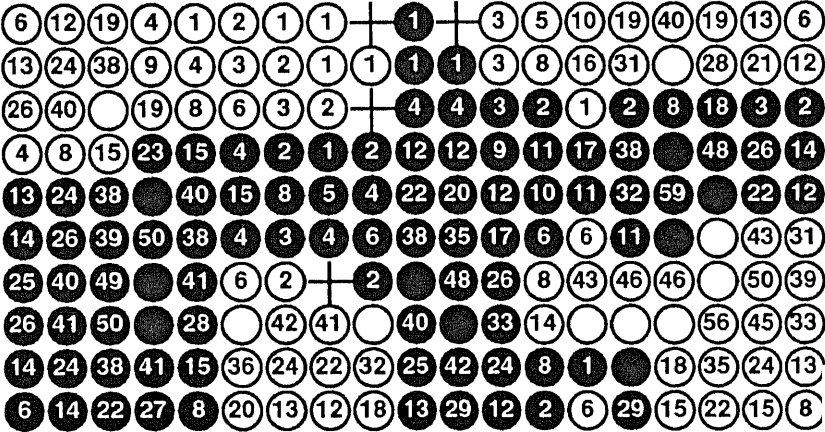


Fig. b: Combined influence of all stones

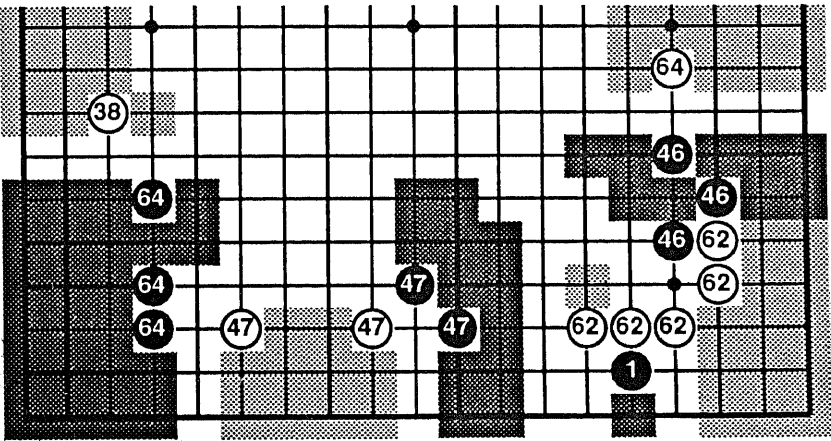


Fig. c: Groups, their safety, and their claim to territory

This analysis of the board is updated after each move. Figure d) shows the effect of a single move in the position of Fig. c). Left: A black invasion lowers the safety of white's 2-stone group from 47 to 13, and enhances the safety of black's group to the right from 47 to 51. Right: A white 'peep' that threatens to break black chain 6 drastically reduces its safety from 46 to 28 and 4.

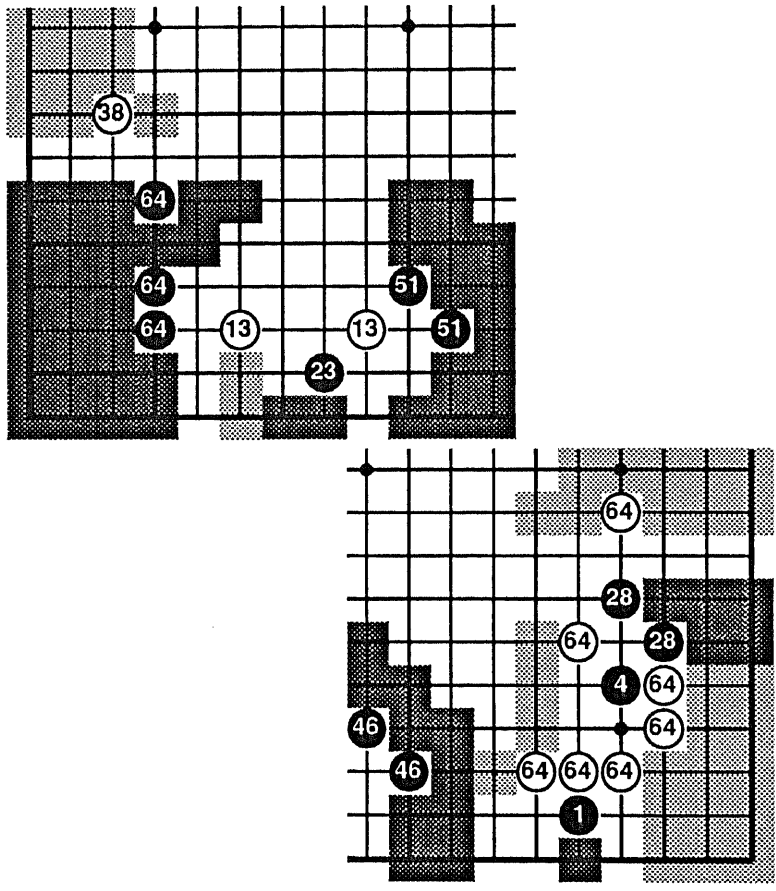


Fig. d: Effect of a black invasion (left) and a white peep (right) on group safety and chains

For Go players, we list the basic concepts we have attempted to design into Explorer:

battle -	collection of transitively adjacent blocks of both colors, with few liberties.
block -	directly connected set of stones of same color.
boshi -	capping attack.
chain -	inseparable collection of blocks of same color.
group -	collection of blocks forming a strategical unit.
hasami -	a pincer or squeeze attack.
hiraki -	an extension usually along third or fourth lines.
influence -	the influence at each empty point from stones in the neighborhood.
joseki -	standard corner opening moves.
kiri -	a cut that separates the opponent's stones into different groups.
ko -	a repetitious situation is avoided by disallowing an immediate recapture.
oshi -	pushing along a line on top of an opponent line of stones.
safety -	security level of a group.
semeai -	a race to capture between two vulnerable groups.
shimari -	a corner enclosure consisting of two stones.
suberi -	a sliding move that slips under the opponent's stones.
territory -	ownership of each grid point, a fuzzy number between -1 and 1.
uchikomi -	an invasion of the opponent's prospective territory.
value -	of a block or a group.
vulnerability -	measures how suitable a group is as a target for attack.
watari -	to connect underneath along the edge of the board.

As there are dependencies among the basic concepts of Go, it is not surprising to see that Explorer's behavior suggests some "knowledge" of other concepts that were not explicitly designed into it. In a way, this knowledge migrated, or piggy-backed, as a side effect of the concepts explicitly built-in. These "windfall" Go concepts include:

atsumi -	thickness; a strong group that cannot be approached closely.
furikawari -	giving up one territory in exchange for another.
keshi -	erasure; a move played to reduce the opponent's area.
kikashi -	a forcing move.
korigatachi -	configuration of stones which is overconcentrated.
moyo -	a large territorial framework which threatens to become secure territory.

- seki - a life and death battle which cannot be won by either side and so remains on the board at the end of the game, all groups involved being stalemated.
- shinogi - moves that give a group good shape for making eyes and securing life.
- tenuki - to play elsewhere, ignoring the opponent's last move.

At this point, Explorer lacks the following Go concepts:

- aji - after-taste; the latent power stones continue to exert even when given up as lost.
- hamete - a basically unsound move which complicates the situation. Often the obvious answer to a hamete is bad and it is difficult to see the right way to play.
- miai - two points related in such a way that if one of them is occupied by one player, his opponent can handle the situation by taking the other.
- sabaki - development of stones in a dangerous situation in a quick, light and flexible way.
- sente - the right to choose where to play next.
- tesuji - a skillful move that enables a player in a local situation to make the most efficient use of his stones and take advantage of the opponent's inefficient use of stones.

A last comment on knowledge selection. Explorer is a weak player playing other weak players who are likely to exhibit typical beginners' shortcomings in their repertoire of techniques. It is tempting to design tricks into it, that is, schemes and techniques that stand a good chance of catching a novice unaware, but are basically unsound and will be refuted by an alert stronger player. Inspired by Bobby Fischer's famous creed: "I don't believe in psychology, I believe in strong moves", we tried to avoid falling into this trap. Although tricks might lead to short-lived success, they are sure to lead into a dead end, as tricks, by definition, contradict general truths, and will only *aggravate a problem inherent in knowledge engineering*: That different rules contradict each other. Any expert player will recognize the Go concepts we aim at as belonging to the classic fundamentals of Go lore.

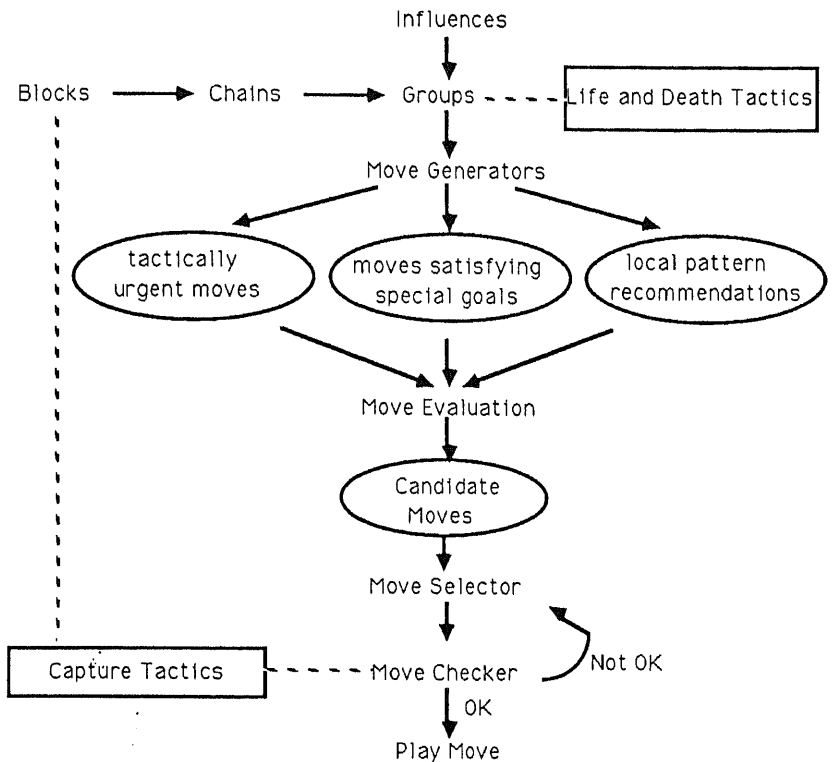


Fig. e: Explorer's move generation and selection process

3.4 Structure of Explorer

Explorer's knowledge is represented internally in four different ways:

1. Relationships among stones on the board are represented by three types of frames: Blocks, chains, and groups. Each has eight to thirty slots filled by attached procedures at the beginning of each move decision cycle.
2. Knowledge concerning local urgent Go patterns is implemented as pattern recognition procedures.
3. Josekis, local urgent moves in a corner opening, are stored separately as a tree on disk.

4. Knowledge concerning moves to achieve specific goals is scattered in two dozen move generators in implicit production rule form.

Explorer's Go knowledge is the major source for its decision making. Note that with the exception of tactical information, Explorer's knowledge is all derived from the present position. The move generation and selection process is shown in the diagram above.

3.5 Experience and current work

Reviewing the games Explorer played in the 1988 International Computer Go Congress tournament in Taiwan last November, and in human tournaments in New Jersey and Oklahoma in early 1989, we observe that our knowledge-based approach to computer Go is feasible for developing a low-rated amateur. Knowledge is power: Its knowledge of group safety and block values alone gave Explorer an advantage over most other Go programs, as reflected in its strong performance in large scale fighting during the mid game stage, in most cases making end game play irrelevant to the win/loss results. Explorer also lived up to its 15 kyu rating in two human tournaments against 7 kyu to 20 kyu opponents. As is the case in chess, a computer's style of playing is different from peoples'. Explorer committed bigger blunders than its human opponents, but often exhibited better positional understanding than human players of equivalent strength.

We learned that how to use an additional source of knowledge to advantage is much harder than making the additional knowledge available to the system. Knowledge may backfire - more knowledge does not necessarily mean better performance. Different sources of knowledge will suggest different urgent moves that conflict with each other. In the only game Explorer lost during the Go congress, the group heuristics move generator detected a 'cut', an aggressive move that splits a chain into two, and pattern matching recognized an urgent local move; one wins the game, the other loses, and unfortunately, our move selection process made the wrong choice. The *coordination of all knowledge sources* in the system is the hardest knowledge engineering task in the Explorer project.

Some knowledge of importance to human problem solvers may be of little value to a machine. For example, human players memorize standard corner opening sequences, called joseki, because they are safe and save a lot of thinking. It is easy to implement such book knowledge, and Explorer has a large joseki library on disk [Is 77]. But it takes considerable strategic understanding to take advantage of sophisticated opening sequences, and today's programs are lacking in this respect (This problem is also well known in chess: What to do at the end of a book sequence?). During the computer Go tournament, we turned Explorer's joseki option off, in order to increase the chances of middle game strategic fighting, where we feel Explorer has an advantage over other programs.

Almost all of Explorer's knowledge is heuristic, and thus imprecise. We believe this knowledge can be refined to reduce conflicts and improve performance, and are working along the following lines.

We continue to investigate additional Go concepts that can profitably be captured and put to good use. As computer chess has proven repeatedly, game playing is an experimental subject where predictions are difficult. But we believe that an expanded pattern library may capture part of the concepts of tesuji and aji, and serve as move generators that focus search on specific goals that would otherwise be lost beyond the horizon. Explorer currently has no concept of one of the most important aspects of Go: Coordination among several battles, as discussed in section 1. Synchronization of campaigns is a major extension beyond Explorer's current structure. As a step in this direction, we have experimented with making use of the concepts of 'sente' and 'gote' in the endgame, where most scenes of action are independent [Ki 87].

Improving Explorer's tactical prowess is an open-ended route to a stronger Go program - no inherent limitation can be seen as yet. Explorer solves many tactical problems when given more time than the one hour per game customary at computer Go tournaments. Strong chess machines have overcome this bottleneck by special-purpose hardware that lets them generate and evaluate 100 to 1000 times more positions per second than a conventional microprocessor could. We are not aware of any work on Go hardware, but that's an obvious approach to investigate. A more immediate attempt to improve the tactical performance is to work on the time allocation for tactical problems: Spend more time on important problems, do more useful things during the opponent's time slot.

In conclusion, computer Go stands today where computer chess was 20 years ago. Among dozens of programs, each one seems to follow its own approach, and the most visible characteristics they share are a very low amateur level of play, and a lot of hope. It was impossible, two decades ago, to predict with any degree of assurance how far and how fast computer chess would progress, and it is impossible today to predict how successful computer Go will be, and what it takes to get there. The only road to insight and progress is hard, trial-and-error experimentation. Explorer is one data point along this road. It tackles a difficult issue head-on: Can a dumb program make good use of classical Go knowledge?

Acknowledgment. Thanks to Bill Hargrove, Ken Thompson, Jim Stein, and unknown referees for helpful comments. This work was supported in part by NSF under grant DCR-8518796. Ken Chen's sabbatical was supported by UNCC.

References

Readers with a deeper interest in computer Go will find the following references useful. [Ki 86/7] is an early version of a bibliography intended to be exhaustive - any additions are welcome. [CG] is a quarterly on computer Go. [AGA] publishes the American Go Journal, with occasional articles on computer Go. [ISHI] is the major publisher of Go literature in English.

- [AGA] American Go Association, P.O. Box 397, Old Chelsea Station, New York, NY 10113.
- [CG] Computer Go. Erbach, D.W. (ed) 71 Brixford Crescent, Winnipeg, Manitoba R2N 1E1, Canada.
- [ISHI] Ishi Press International, 1400 Stierlin Road, Bldg. A7, Mountain View, CA 94043.
- [Bra 79] The Game of Go - The Ultimate Programming Challenge? Bradley, M. B. Creative Computing 5, 3 (March 1979), 89-99.
- [Fr 80] K.J. Friedenbach, Abstraction Hierarchies: A Model of Perception and Cognition in the Game of Go. Ph.D. Thesis; Univ. of California, Santa Cruz, 1980 (Microfilm).

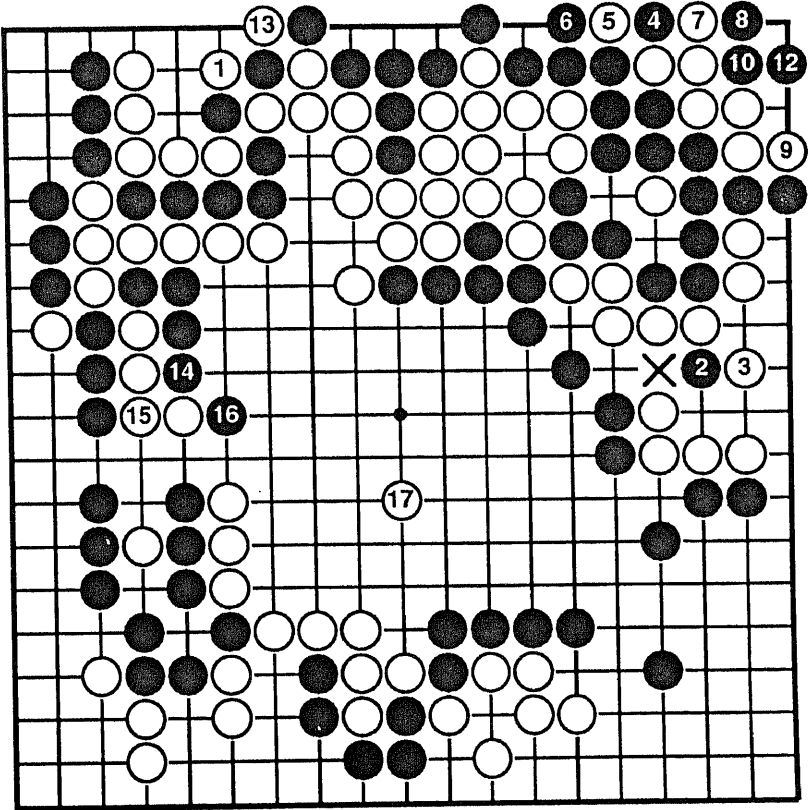
- [Is 77] Y. Ishida, Dictionary of Basic Joseki, Vol. 1, 2, 3, [ISHI].
- [Ja 89] P. Jansen, DT as Spraggett's second in Quebec. Msg on electronic news, 17 Feb 1989.
- [Ka 51] Y. Kawabata: The Master of Go, Perigee Books, NY, 1981. Originally published in Japanese, as 'Meijin', in 1951. Available from [ISHI].
- [Ki 83] A. Kierulf, Brand - an Othello Program. In Bramer, M. A. (ed.) Computer Game-Playing: Theory and Practice, 197-208, Ellis Horwood, Chichester, 1983.
- [Ki 86/7] A. Kierulf, Computer Go Bibliography. Part 1 in [CG] 1, 1 (Winter 1986/87), 17-19; part 2 in [CG] 1, 3 (Summer 1987), 15-19.
- [Ki 87] A. Kierulf, Human-Computer Interaction in the Game of Go. In "Methodologies for Intelligent Systems", Z. W. Ras, M. Zemankova (eds.), North Holland, 1987 (Proc. 2nd International Symp. on Methodologies for Intelligent Systems, Charlotte, North Carolina, October 14-17, 1987), 481-487.
- [KN 85] A. Kierulf and J. Nievergelt, Computer Go: A smart board and its applications. Go World No. 42, Winter 1985/86, 62-65, Ishi Press, Tokyo.
- [Le 81] P.E. Lehner, Planning in Adversity: A Computational Model of Strategic Planning in the Game of Go. Univ. of Michigan, Ph.D. Thesis, (1981), 1-86 (Microfilm 8116280).
- [Le 88] D.N.L. Levy (ed.), Computer Games II. Springer Verlag, New York, 1988. (Includes [Ma 84], [RW 79], [Wi 78], and [Wi 79].)
- [Ma 84] Y. Mano, An Approach to Conquer Difficulties in Developing a Go Playing Program. Journal of Information Processing 7, 2 (1984), 81-88.
- [Nu 88] J. Nunn, Life with ChessBase. ICCA Journal (International Computer Chess Association) 11, 2/3 (June/September), 1988.
- [RW 79] W. Reitman and B. Wilcox, The Structure and Performance of the Interim:2 Go Program. Proc. IJCAI-6 (Tokyo, August 20-23, 1979), 711-719.

- [Ry 71] J.L. Ryder, Heuristic Analysis of Large Trees as Generated in the Game of Go. Ph.D. Thesis, Stanford Univ. (1971), 1-298 (Microfilm 72-11, 654).
- [Sh 50] C.E. Shannon, Programming a computer for playing chess. Philosophical Magazine 41, 314, 256-275, 1950.
- [Sh 89] K. Shirayanagi, A New Approach to Programming Go - Knowledge Representation and Its Refinement. To appear in Proc. of the Workshop on New Directions in Game-Tree Search, Edmonton, Canada, May 28 - June 1, 1989.
- [Tu 53] A.M. Turing, Digital computers applied to games. In 'Faster than Thought: A Symposium on Digital Computing Machines', (B. V. Bowden, ed.), Ch. 25, 286-310, Pitman, London, 1953.
- [Wi 78] B. Wilcox, Zobrist's Program. American Go Journal 13, 4/6 (1978), 48-51.
- [Wi 79] B. Wilcox, Ryder's Program. American Go Journal 14, 1 (1979), 23-28.
- [Wi 85] B. Wilcox, Reflections on Building Two Go Programs. SIGART News 94, 29-43, Oct. 1985.
- [Zo 70] A.L. Zobrist, Feature Extraction and Representation for Pattern Recognition and the Game of Go. Ph.D. Thesis, Univ. of Wisconsin (1970), 1-152 (Microfilm 71-03, 162).

Appendix: Fragments from Explorer Games

Tulsa Midwestern Go Tournament, February 25/26, 1989.

White: Bob Felice, 12 kyu Handicap: 3 stones
 Black: Go Explorer, 15 kyu Result: Black wins by 4 points



This is one of Explorer's better performances. White may not realize that his group in the upper right corner needs an additional move to live. It is surrounded by a big one-eyed Black group, which White proceeds to attack with 1. Explorer realizes that it can only live by capturing some enemy stones and ignores the atari. Black 2 threatens 'x' and the entire White group along the right edge. White defends well at 3. With 4 and several skillful follow-up moves Black kills the White corner (11

is played at 5). Explorer correctly ignores White 13 to capture some important stones.

Black: John Lowe, 15 kyu Komi: 5.5 points
 White: Go Explorer, 15 kyu Result: White wins by 34.5 points

This shows Explorer trading 15-kyu-mistakes with its opponent. Playing optimistically but much more consistently than one would expect from a program that lacks positional look-ahead, it builds territory in a center that is open on three sides. First, it manages to close an upper left corner; second, White 7 closes the previously wide-open upper right boundary. Its human opponent finally tries to penetrate with 12, 14, and 18, but misses his chances.

