

1997 Computer Game Developers Conference Paper #5502

Applied AI: Chess is Easy. Go is Hard

Bruce Wilcox brucewilcox@bigfoot.com

Who dares read this paper?

Programmers of strategy games needing artificial opponents.

The 2-bit Rationale

Chess programs have come a long way since the mid 50's. Once considered the *task par excellence* of AI, chess programming was to be the epitome of AI research whose solution would require magical insights into software technology. Now it sits on AI's back burner. Advances in hardware technology allow massive search to make chess programs unbeatable by all but a few humans. The chess program Deep Blue scans billions of moves per turn, evaluates the resulting positions, and steers the game toward the best results found.

Combinatorial explosion combined with an expensive and unreliable evaluation function makes whole-board search unfeasible in Go. Instead, players and computers must piece together disparate local tactical analyses and multi-layered perceptual assessments in an attempt to build an incomplete whole-board view of the game.

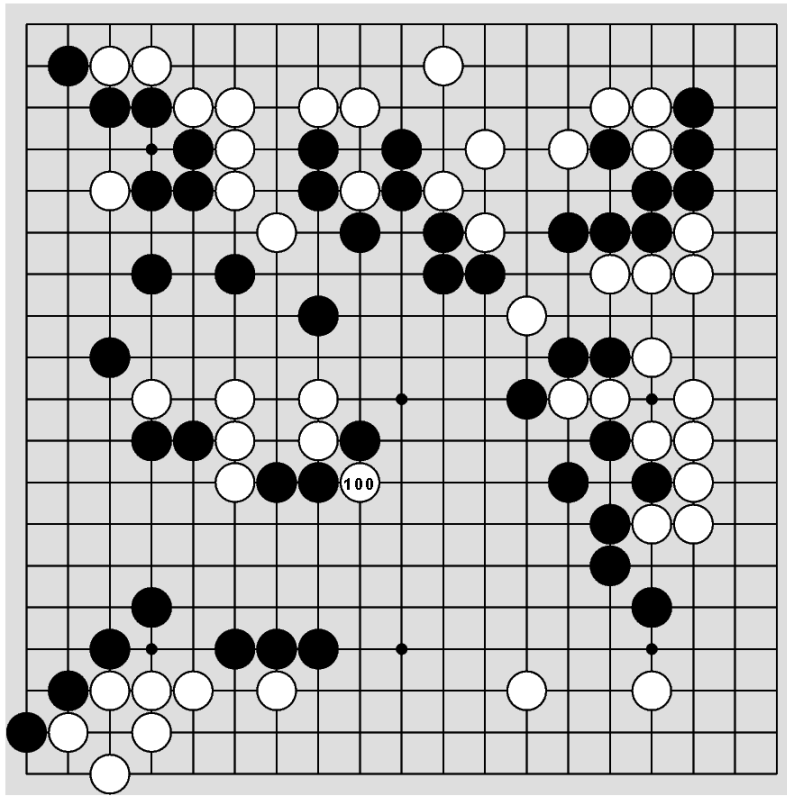
Most computer games have this same task of integrating apples and oranges into a coherent whole. This cannot be done by a single methodology and instead requires compositing various techniques. Since this is my third decade building Go programs under varying constraints of hardware and architectural design, I thought I might say a few words and you might listen.

Lightspeed Go

Two players, Black and White, play on a 19x19 grid. They alternate turns adding a single stone of their color to any empty intersection of the board. Once placed, stones don't move. Over time players build continuous lines of stones to enclose clusters of empty intersections, called *territory*. Each intersection so contained is worth one point. Rules allowing capture of enemy stones assist in defending territory from enemy intrusion and are another way to earn points. Whoever has more points at the end of the game (territory + captured stones) wins.

Stones are captured when all horizontal and vertical intersections adjacent to them are occupied by enemy stones (thus crushing them to death). You can save stones from capture by adding friendly stones on adjacent intersections. The contiguous collection, called a string, acts as a single unit, and the opponent must fill in all touching intersections of all stones of the string to capture it. It is illegal to self-capture a stone (play it where it is already touched on all

sides by enemy stones), and it is thus possible to build configurations that can never be captured.



pro game at turn 100

As befits a four-thousand-year-old game considered one of the ten Zen arts of Japan, there is a lot of written literature and analyses of game records. Nowadays you can even find a fair amount in English.

Chess is PSPACE- hard. Go is PSPACE- harder.

According to Einstein, the number of atoms in the universe is 10^{110} . The number of possible chess positions is 10^{120} , while the number of possible Go positions is 10^{761} . It's an impressive comparison, but I don't play games by looking at all possible positions. Forget those numbers.

Chess is played on a small board (8x8) with a unified theme- maneuverability. The number of legal moves per turn is small, 20 at the start, rising upwards towards 40. The goal is a simple movement one, checkmate the king, and the game stops instantly when this happens.

Go is played on a board the size of six chess boards (19x19) with a complex intertwining of the primary theme, territory, and a secondary theme, capture. The number of legal moves is large, 361 at the start, and after the game ends there are probably some 110 legal moves still left. There is no "instant win." Even after you have achieved enough territory to win, you have to keep playing good moves just to hold onto it until the end of the game. A player who knows he has acquired a sizable advantage is at great risk of losing due to being too conservative.

The opening game of chess uses memorized openings. These can last over 25% of the play and leave the board in a balanced position. In Go the equivalent of a memorized opening is called a *joseki*. Unfortunately its perfection only applies to the corner in which it is played and there are four corners. One may play joseki in each corner and discover that the interaction of locally perfect play on your overall board position yields a disaster. If chess is a battle, Go is a war, and how the local battles interact is as important as who won each one.

Chess books tell you how to “evaluate” the board, summing the value of the pieces (Queen: 9 through Pawn: 1). In top chess programs the sum of all possible positional factors is usually worth no more than a pawn and a half. This makes it easy to evaluate the board and gives simple goals of outcapturing the opponent and lowering his maneuverability (number of legal moves).

In Go, there is no uniform function to merge tactical values with positional ones. Capturing 15 stones may be worth no more than a simple positional move while capturing 1 stone may have huge positional ramifications. Worse, to evaluate the board you first have to decide on the life-status of all stones on the board, often requiring a lot of lookahead dedicated just to that assessment.

For humans playing chess, looking ahead sequences of moves is difficult because each move erodes the present image of the board by removing a piece from one location and adding it to another. After seven or eight such moves (ply), most humans cannot keep track of what the board looks like. Deep Blue looks 13 ply deep. Chess masters only occasionally look ahead 15-20 moves deep.

For humans playing Go, looking ahead sequences of moves is easier because each move augments the current image, making it possible to remember what the board looks like many moves deep. As a consequence typical searches are much deeper in Go, negating the value of being easier to visualize. Even beginners learn to look ahead 60 ply in simple restricted lines of analysis (*ladders*).

While chess has a clearly defined terminus, the checkmate of the king, the end of Go is vague. It ends when both sides pass, a subjective choice on their part. Beginners are often passing and “completing” their game when there remain profitable moves left to play which might change the winner. Or beginners continue to play long past the time any stronger player would have passed. It ain't over 'til it's over and that is a private matter between the two specific players involved. My program once won a game against another program in a tournament only because my program knew when to pass. The other program continued to play, filling in its own territory and losing points with each move.

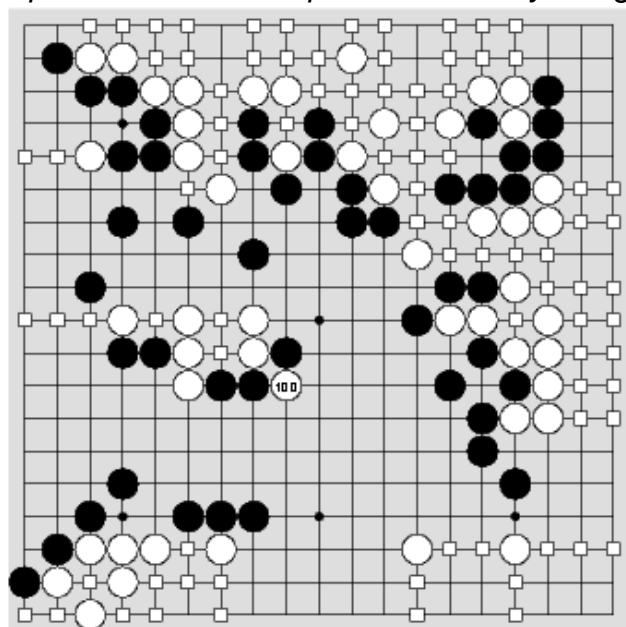
As if the above weren't enough, Go provides the rule of *ko*. Some captures of stones lead to potentially infinite recapture situations called *ko*. In chess, board repetition results in a draw. In Go, it is illegal to recreate a prior board position. This leads to the concept of *ko* threats, moves which may be otherwise foolish but which one plays so as to force the opponent to respond to the threat (creating a new board image) and allow one to then recapture the *ko*. The game may hinge on who wins the *ko*. Threat and counter-threat continue until one side runs out of

sufficiently valuable threats or misjudges either the value of winning the ko, the value of the ko threat being made by the enemy, or the value of the ko threat being chosen by oneself. The concept of finding the “best” move gets mighty confusing during a ko.

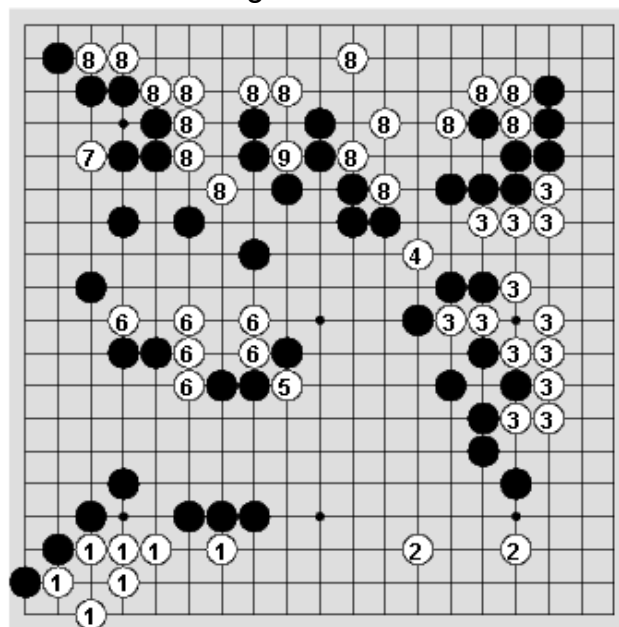
What you see ain't all there

What concretely exists on the board are Black and White stones and empty intersections, but that is not what skilled players see. Instead a skilled player sees perceptual conglomerations of increasing sophistication about which he can make assessments. At the bottom layer are *strings*, contiguous horizontal and vertical collections of stones of a color capturable as a unit if all liberties (touching empty intersections) are filled by enemy stones. Close stones are also used to recognize *links*, potential areas for connecting stones into strings and which delineate regions of the board containing territory or enemy stones.

Strings connected by links form *groups*, which bound territory and whose ultimate life or death hinges upon the ability to make two eyes (particular patterns of stones around an empty intersection that prevent enemy play within). Groups that do not have the ability to form two eyes by the end of the game will be captured during counting, without filling in the liberties of all constituent strings. Groups of the same color that reinforce each other along the edge form a *position* and bound *potential territory* using *sector lines* running between them.



White link points



White groups

Linear collections of stones of one color form *walls* that exert *influence* on nearby intersections, making enemy play in that area possibly unsafe, thus foreshadowing possible potential territory. Also, the edges of the board and particularly the corner where the two edges meet have a huge impact on choice of moves and tactics available.

Building a Go program requires approximating these entities. Key design decisions will revolve around what perceptions are detected and what assessments can be made. Speed is

essential. While chess programs have 3 minutes to make a move, Go programs must average under 30 seconds per turn to play out half of a 250-move game within their allotted hour.

Additional design issues revolve around what kinds of tactical lookahead to support. Tactics can be done on pieces of the board, for string capture, link separation, eye and multiple eye formation, and tactics can be done for global lookahead (albeit with severe limitations on the amount that can be done).

All of the above will involve issues of computation and avoiding recomputation by caching and incremental updating. And all will need pattern matching configurations of stones to provide expertise. And this doesn't even get close to the question of how the program will make decisions about where to play. This is just to get some "simple" information available from which to make decisions.

Decades of Research in a Nutshell

In 1972, the state of the art of computer Go was two 19x19 Go programs, both Ph.D. theses, and both able to lose to someone who had just been taught the rules moments ago. By 1979, working under an NSF grant with Prof. Walter Reitman at the University of Michigan's Mental Health Research Institute, I built a LISP program using megabytes of memory on an IBM mainframe that could give a 9-stone handicap to a raw beginner while playing in seconds per move. That is equivalent to playing a chess game by removing your queen at the start, and still winning.

From 1982 -1989 I wrote NEMESIS Go Master, in C, to run on an IBM PC. The program played in reasonable time (15-30 seconds a move), and in 1984 it was the first program to ever play in a human tournament. In the weaker sections, it achieved a rating of 20 kyu, about 4 stones (ranks) stronger than the LISP program. By 1989 NEMESIS was about 4 stones stronger again. One rank in Go is similar to one rank (100 points) in chess. Each rank is harder to achieve than the previous, so while a human may advance 15 early ranks in a matter of months, later ranks make take a year or more each. The core Go engine fit in 256K of ROM and 64K of RAM, being kept at that size to fit in a hand-held Go machine I was building.

In 1994 I wrote RiscGo, in C, to run on an ARM RISC processor with a mere 162K of ROM and 16K of RAM to fit into an SNES cartridge. It was about 6 stones stronger than NEMESIS. Advancing ranks should take longer to program and representing Go data should take more and more space. RiscGo was an architectural leap forward.

In 1995, I rewrote RiscGo into Ego for the PC and in 1997 I am currently writing SuperEgo.

In each case, design assumptions changed, and my ways of processing changed with them.

The Impossible Dream (aka ID, aka Reitman-Wilcox Go program, aka Interim 2)

In 1972 the Japan Go Association said *it is only a fantastic dream to make the computer play Go in its own way in place of human beings*. Just shows their ignorance. The impossible

dream was done in the 70's as a noncommercial AI research project. LISP was one of the few reasonable language choices available, and we had to buy time on a machine that charged by the CPU second and ignored how much virtual memory one used. Therefore the program was designed to use lots of memory and take as little time as possible. I made extensive use of incremental algorithms for maintaining Go structures. Code took about one megabyte and dynamic data took about two megabytes.

During development I went from never having heard of Go to being a strong 5-Dan player and created a programmable theory for how to play Go as a human or as a computer, including new heuristics and board perceptions. The current revision of that theory is now published in the book EZ-Go, Oriental Strategy in a Nutshell (<http://www.slip.net/~wilcox/book.htm>).

The primary data structure was, not surprisingly, the list. ID incrementally maintained lists of everything. For example, strings had lists of their stones, liberties, and enemy adjacent stones. The classic links were supported, and strings connected by links defined groups. Territories were empty points or dead groups surrounded by links.

Pattern matching was done by setting up camera-like things called *lenses* which monitored particular objects (e.g., links). They had *fields* (lists of intersections) they were watching (with implied purposes based on lens type) and the sequence of play expected within that field. Whenever unexpected stones were placed in a field, the field was deleted. Active fields recommended moves and values which served the purpose of the lens. Usually the answer was a move suggestion and an associated priority value, but the value could also be descriptor bits (e.g., this move cannot be cut) or an index into a sequence database (i.e., play out this sequence). There was no incremental undo code for this, so the lenses were only used at the top-level of a turn and separate code was used during lookahead to detect relevant features. A simple lens definition is the following:

```
LENS 2 d3 c4      < d4 empty c3 empty c3 5      >
                  < d4 foe c3 empty c3 33      >      ENDLENS
```

This says that lens type 2 (diagonal connection defense) given the two endpoints of the diagonal connection, has two recognizable fields (<>). If d4 is empty and c3 is empty, then return c3 value 5. If d4 is foe and c3 is empty, return c3 value 33. Any number of points anywhere on the board could be contained in a field. One could also embed tests within a field, e.g., <d4 foe d4 liberties>4 c3 empty c3 29 >.

Tactical lookahead was used to answer simple questions like "Can this string be captured" or "Can this link be broken". These yes-no questions required only a simplified alpha-beta. However, extensive use was made of failure data returned from deeper ply within the search. Each failure returned why it failed (e.g., string d4 got captured), the set of all points involved in search decisions, and the liberty counts of strings which, if changed sufficiently, would have changed how the search was conducted. Using this data, complex searches were performed with an average branching factor of 1.2. The search returned the initial move for successful answers, or NIL for failure, as well as the data needed to know when to redo the search. Caching for searches was only done across turns of the game. There was no cache of positions arising during a specific search.

Control of the program consisted of two phases. The first was a bottom-up phase in which moves caused low-level data structures to be modified, which passed on messages to modify higher-level data structures. Once all structures were up-to-date, the second phase generated moves for different purposes based on concerns of high-level perceptions. These moves were then analyzed to discover what side-effects they might have and weights applied. A move which captured a stone might accidentally change the safety status of a group, so it would get credit for doing that.

Eventually the University's charging policy changed. Virtual memory became a major factor, the cost of our games of Go skyrocketed to \$2,000 a game, and the project ended.

My NEMESIS

NEMESIS Go Master was to become my personal nemesis, with DOS, Macintosh, Windows and Japanese NEC-9801 versions. I even had to fight (and win) a trademark suit with Konami Coin-Op over the name NEMESIS. Initially NEMESIS dominated the market. Because of its sale in Japan starting in 1986, the Japanese added a Go program to its Fifth Generation AI Initiative and funded it with millions of dollars. Its goal—defeat NEMESIS! They failed. Others, however, eventually succeeded. I spent too much time on commercial interfaces which took away time from keeping NEMESIS the strongest program and I had memory restrictions that others didn't. NEMESIS was a commercial success, but when serious computer Go tournaments eventually arose, NEMESIS rarely won.

When I started on the original IBM-PC, I no longer had lots of memory (no EMM or EMS RAM), I already knew how to build a Go program, and I intended to build a commercial program. Hence I chose C for its portability and availability and aimed to keep the program small. Having already built one Go program, there were several ways to design a new one. One approach was the clean-slate. Throw away your previous design and redesign from scratch. Another was the port approach. Try to reuse everything you can. A third approach was the reversal approach. Decide on the major design choices made previously and consider doing just the opposite. I took that approach.

The major reversal in design I made was to throw out incremental algorithms. Debugging them took forever and they took up too much code and data space. Time, within reason, was free, so I could afford to maintain simple caches and recompute whatever I needed. Go-playing code stayed within 256K of ROM and 64K of RAM (to fit on a hand-held Go-playing machine).

The main data structure in NEMESIS was the array list. Lists were stored as consecutive elements of dynamic arrays, with the 0th element counting how many elements. String data, however, was kept internally as linked lists and transferred to array list notation as needed.

Due to space considerations, it wasn't practical to keep lists of all points involved in a tactical search, so NEMESIS lacked the ability to use failure data passed back by lookahead. Instead the board was divided up into 5x5 chunks and whenever a point was involved in lookahead, the corresponding chunk bit was turned on. The tactical cache thus redid searches somewhat too often, but it didn't take much space to store when to recompute. Early versions of

NEMESIS used feature-detection code during lookahead. The last version replaced this code by using the pattern-matcher, but the matcher was too slow to be effective for this and version 5 of NEMESIS was slower and weaker than version 4.7.

The incrementally updated list-of-fields approach used by ID's pattern matcher was replaced with a corresponding decision tree ordered by most common points first and searched on request. Since pattern matching was going to be used in lookahead, keeping matches cached didn't make sense. Each move in lookahead, being spatially close to the previous move, invalidates all useful cache data. Keeping a cache would take time and provide little benefit.

Risclgo (aka Risky Go)

In late 1993 I was asked by a Japanese company to build a Go program many ranks stronger than NEMESIS, do it in half the ROM and a quarter of the RAM, and do it in half a year. The task was implausible, to say the least. Rewriting everything in assembler might get me to fit in ROM, but it wouldn't help RAM usage and it would take too long. I solved the problem with two design changes followed by a total rewrite. First, I kept only bitmaps, not array-lists, and scanned the board to gather the array-list data when I needed it. Second, I scrapped a lot of special purpose code used to detect features in all parts of the program, and replaced it with a fast compact pattern-matcher and lots of compact patterns. Pattern matching was used for almost everything in Risclgo, including lookahead, global move generation, group safety evaluation, and endgame play. By doing this I compressed the code (an average pattern was the size of 4 ARM instructions), converted the debug cycle from recompilation to interpretation (most mistakes involved patterns and with an embedded pattern editor, I could see a mistake, fix it, and continue in the same game), and increased reliability of the code (patterns couldn't cause the program to crash- just match or fail to match inappropriately).

The primary data structure in Risclgo was the bitmap. Whereas NEMESIS kept strings with lists of stones, liberties, etc., Risclgo only kept a bit map indicating which points were members of strings, groups, territories, etc., and created array lists as needed by re-scanning marked contiguous points. A 361-intersection Go board of longs held data updated incrementally on all move and unmoves, as well as data generated during a whole-board assessment. The incremental update bits were:

- a. 7 string id bits
- b. 4 link bits (black & white x horizontal & vertical)
- c. 2 chain bits (black & white)
- d. 2 occupation bits (black ,white , none = empty, both = off edge)
- e. 1 "been here" bit for scans of contiguous points of same ilk

The string id indexed an array of short ints which described the strings. 9 bits identified the highest point of the string on the board (a canonical name), 4 bits kept the liberty count, and 1 bit indicated if this was a "big" string or not (more than one stone). Access to any other data was recomputed by scanning the board for contiguous points with the same string id.

Special code incrementally updated the link bits.

Groups were broken into two concepts, chains and groups. If a player had more stones touching an intersection, or, failing that, had more stones diagonal to an intersection, or, failing that, occupied the north-most point touching an intersection, then he got the chain bit for that intersection. Collections of stones and chain points formed a chain, which was recognizable independent of any life and death assessment. During assessment, chains were scanned, assigned ids and group bits, and their safety was assessed. Chains which bounded dead enemy chains would have their id's merged into one and were then reassessed after the dead chains were converted into territory.

The whole-board assessment bits were:

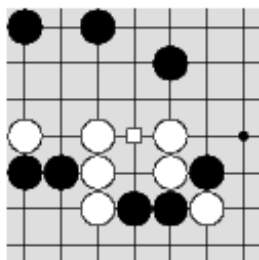
- a. 4 weak link bits (horizontal & vertical - black & white -- can be cut)
- b. 5 chain id bits
- c. 2 group bits (black & white)
- d. 2 territory bits (black or white)
- e. 2 territory header bits (a unique territory starts here by color)

The chain id indexed an array of shorts describing the chain or group. The bits included: 9 bit canonical name, 3 bits safety assessment, 1 bit "might-be-able-to-breakout" bit, 1 bit "involved in death fight", 1 bit "big" (has 6 or more stones), and 1 bit "can fight" (has many liberties).

Everything was tightly packed. Even the move history list was an array of short ints, 1 per turn:

- a. 9 bit location
- b. 1 bit color of player moving
- c. 4 bits kill directions (directions stones were captured for unmove)
- d. 1 bit ko flag (shows suicide or ko depending upon kill bits values)

Pattern matching by testing points of fields one at a time was too slow. I improved on a pattern-matcher originally designed by Fotland. In RiscIgo the field concept of ID (board pattern with move and value) was kept as the set of bits representing a fixed 5x5 or 7x7 view. During matching the occupation bits of the needed area around the focal point were grabbed in correct orientation order and packed into the correct format, flipping the color data as needed to represent friend and foe perspective.



7x7 view around focus

Each pattern point was represented by 4 descriptive bits: not-friend, not-foe, not-empty and not-edge. (Actually only 12 bits were needed to hold the not-edge data, since being on an edge implies that other points would be also). Then each pattern of the type was ANDed with the newly-packed board image fragment which specified what the occupation of the points actually was, and if all resulting bits were 0, the pattern matched. If a bit is on in a pattern, it may not be on in the actual packed board. To require a friend stone at a point, just turn on all

other not-xxx bits in the pattern. If you don't care what state a point is in, don't turn on any bits. Two test conditions could also be stored with the pattern for validation if the board image matched. If the tests passed, the answer move and value was returned. Hundreds of patterns could be matched quickly against the current packed image. Each pattern was 12 or 24 bytes, with an average of 16 bytes. I wrote a pattern editor that enabled them to be quickly written visually.

Patterns were still matched by type for a specific purpose, as was done in ID and NEMESIS, though in Riscigo the number of types extended to 100. These include ones for link attack and defense, contact fights, invasions, potential territory defense, endgame territory attack and defense, and eye estimations for territories of different sizes. There were roughly 3,000 patterns. Since the match was for a specific purpose, careful choice of the focus could limit the number of orientations one had to match in. For example, in tactical analysis of strings, the key patterns revolve around the liberties of the target. By picking a liberty as a focus I could restrict the orientations to only those in which a stone of the string was immediately visible above the focus (usually only 2 orientations out of 8 possible).

Two other enhancements were added to pattern-matching. First, a "don't play this" bit on a move value meant that if a pattern matched and had this bit, it would add the move to a list to ignore and keep matching. Any other patterns recommending this move would be ignored. Second, patterns could be grouped and if the lead pattern failed to match, all remaining patterns in the group would automatically be skipped also. If the lead pattern matched, it didn't provide an answer, it just enabled matching within the group to continue. For example, there were 143 patterns for string attack kept in 7 groups. If the lead patterns all failed then less than 10% of the patterns would have been tested for match.

Riscigo has several joseki databases, varying from half board rectangle to small corner rectangle. It matched them without caching by looking at the move sequence as played in a designated rectangle and following the joseki tree for that rectangle (decoded and rotated into the correct place on the board). If a match is found in a bigger rectangle, then joseki processing for that area stops with the answer(s). Otherwise it switches to the next smaller rectangle and tries again. Keeping different size rectangles allows Riscigo to try to coordinate choice of joseki based on the broadest possible information.

Riscigo simplified the tactical cache of NEMESIS. The recompute data was reduced to 5 bits per search by defining a set of overlapping rectangles of various sizes over the board and keeping the index of the most closest fitting one for the tactical search conducted. All NEMESIS feature analysis code was replaced with pattern matching. Riscigo had no life-and-death tactics, just eye tactics, but string capture now included eye formation and detection.

I created a fast, affordable whole-board evaluation function for Riscigo using the pattern matcher to provide reasonably reliable life and death assessments of groups without using lookahead. Thus Riscigo could perform well-defined whole-board strategic move sequences and then evaluate the result. As an additional feature, by tinkering with the evaluation function, I created multiple personalities. The program had selectable styles of play (which made for great graphics in the user interface). The program would change style late in the game to

accommodate losing or winning big. Riscigo varied moves by selecting randomly among those which were within $\pm 10\%$ of value of each other.

Goliath, a multi-year computer Go champion, was the Go program on SNES that Riscigo was targeted against. Goliath was painfully slow, but in Japan being #1 is the primary thing. Riscigo was a lot faster than Goliath but only marginally stronger, and in 1995 the SNES games market crashed. Riscigo was never released. Sigh.

Ego

In 1995 I needed to distance myself from Riscigo and build a PC product, so I rewrote all the code, keeping the design essentially unchanged. This became Ego. A shareware copy of Ego, called EZ-Go, is supplied on the CGDC CD. It has just the most bizarre two of the personalities available. *Leaper* avoids responding to your moves as much as it can while *Psycho* prefers bizarre but valid openings and hyper-aggressive play.

Ego maintained the tight bit-packed notation of Riscigo as well as all other design innovations. Pattern matching was sped up enough to allow detection of links via patterns, removing that special code. The speedup came from incrementally caching parts of the packed view on move update, so individual points did not have to be packed during pattern matching. They still had to be rotated for some orientations.

A sample game played to conclusion by the computer as both sides takes:

- a. 1.5 million requests for pattern-matching (not counting multiple orientations)
- b. 109 million patterns checked
- c. 11.8 million patterns successfully detected

SuperEgo

My new program under development reverses the assumption of limited memory and is in C for 32-bit Windows.

String data reverts back to the incrementally updated packed list array, while groups and territories stay in the bit-packed notation. String data is cached directly in array list notation.

SuperEgo has a 65Kbyte tactical position cache to be used within each search. Each board position generates a unique hash code (based on one by Zobrist) which is stored in the hash table along with the result and the search id. The hash is incrementally computed by XORing in a unique 64-bit random number assigned for that board location and color whenever a move is played or unplayed there (thus there are two precomputed arrays of 361 64-bit unique random numbers). For all practical purposes, the resulting 64-bit board position hash code is unique. However, the cache uses only the bottom 16 bits as an index. It is not worth the cost of maintaining a collision list in the hash table, so anytime something hashes to the same place in the cache, it will be tested to see if has the same 64-bit hash code and search id. If not, the current entry will be overwritten with the new entry. Thus the cache need not be cleared prior to a search.

SuperEgo speeds up pattern-matching yet again by requiring no rotation of board data at run-time. Most Go programs store pattern data in one orientation, and rotate the board or maintain multiple copies of the board in rotated form to perform a match. SuperEgo keeps pattern data stored for all 8 orientations to minimize update and matching time. What's an extra mega-byte or two, these days?

SuperEgo will reincorporate capabilities of my prior programs that have been omitted as space grew more restricted. E.g., the 1.2 branching factor tactical abilities of ID and the life & death search abilities of NEMESIS.

Whither Computer Go?

There is a pending prize of over \$1.3 million for a program that can beat a human Go professional in a best-of-seven match by the year 2000. With 3 years left and about 15 ranks to be earned-- don't hold your breath. Though Go, with its 361 intersections and multiple tactical searches is well suited to parallel hardware speedups, building *Deeper Blue* is not going to solve the problem. It really will take better AI programming to build a master or grandmaster program. Now, if we only had to beat Kasparov in a game of Go...

To keep up-to-date on computer Go, the obvious place to look is on the Internet. Here are relevant sites.

Ftp://igs.nuri.net:/Go/comp has programs, code, and papers, as well as archives of the computer go mailing list. It includes the paper "Knowledge Representation in The Many Faces of Go (Feb 27, 1993)" by David Fotland.

Computer Go web info: <http://www.usgo.org/computer/index.html> .

Internet Go play: <http://www.well.com/user/mmcadams/igs.howto.html> .

Go contact information and go vendors: <http://www.usgo.org/resources/index.html> .