Reflections RiscIgo Design by Bruce Wilcox

brucewilcox@bigfoot.com

Notes published in the computer Go newsgroup

THE PROBLEM:

In late 1993, I was asked by a Japanese company to build a Go program with the following specifics:

1. It had to play stronger by three stones than the current strongest copy of Goliath in Japan (which was on SuperNintendo).
2. It had to play quickly (Goliath on the SNES was slow).
3. It had to fit in 16K RAM and 162K ROM, with 32K of that ROM not available as code space and accessed only in bytes.
4. It had to run on the ARM RISC Processor as a stand-alone go engine. They would provide the human interface on SNES, and connect via serial channel to the ARM chip.
5. It had to be done in 6 months.

When I got the initial request, my first impulse was to laugh myself silly. What I had to start with was NEMESIS the Go Master, which was arguably 5 or more stones weaker than Goliath, and whose Go engine had previously been trying to stay within 256K ROM and 32K RAM to fit in our own Igo Dojo handheld Go machine. And only six months? Just another project scheduled by marketing. No wonder software is usually delivered late. But the offer was too good to turn down. If there was any chance at all, I had to try.

The first thing I did was see if I could justify it as possible. I mapped how I would spend the RAM budget (described later). I would need a radically different memory design, taking advantage of the 32 bit architecture of the ARM-60 chip. Then I tried to estimate ROM using NEMESIS as a basis. NEMESIS Go code used 162K of code + 15K for patterns + 11K for joseki. But the code was compiled under the Borland C compiler. When compiled for the ARM chip, it was about 15% bigger, or 187K. Since the goal was to fit code in 128K, this was 30% too big. While it was possible to imagine a program fitting in the ROM requirement, it would have to be a complete rewrite both for size, and because it needed to redo all data structures to fit in the RAM limit.

NEMESIS Go engine Code Allocation

Territory tactics	22		group moves	10	
Tactics controller	15		link moves	10	
Board assessment	13		response controller	9	
String + move update	8		game phases moves	8	
Group data		8	string moves		4
Go board/utility	6		moyo moves	4	
Pattern matching	6		edge moves	3	

String tactics	5	territory moves	3
Access data	5	access moves	3
Sector/perimeter data	5	contact fights	1
Link data	4		
Joseki processing	4		
Link tactics	2		
Territory data	2		
Edge data	2		

Total size = 162K (growing to 187K under ARM60 compiler)

Next I considered the debugging environment. The ARM chip development package had a simulator under DOS that ran very slowly. Testing under it was next to impossible. Debugger boards with a built in ARM chip would be available in Japan, but only if I traveled there, and so were generally not available. And the source level debugger wasn't great. So I wouldn't be able to resort much to assembly code to save on size, since I'd never be able to debug it. Instead I would have to write in C as a DOS version, and switch to compiling my C program for the ARM chip at the last minute.

Finally, having argued plausibly that all of the above would work somehow, I had to decide how I was going to get 5 or more stones stronger at the same time and fit that also into the ROM budget. And keep the speed up with limited caching ability. And do it all in 6 months. Right!

I wrote a summary paper to the customer, explaining each of the above points on December 3, 1993. The first thing I wrote was: My analysis of this project says that it can probably be done, but that due to time constraints and the 128K code limit, there is a risk that the final product may only be slightly better than Goliath NES. These words turned out to be completely right, and the project would run to 12 months instead of 6 (they also weren't ready until 12 months had passed, as it turns out).

THE PLAN:

What was the ace up my sleeve? Without one, this project was sheer lunacy. NEMESIS in Version 4 had lots in common with David Fotland's program Many Faces of Go. Lots of special code for doing tactical analysis. In Version 5, I had experimented with replacing string tactics code with pattern matching. In a way, this experiment was a disaster. NEMESIS V5 was slower and weaker than Version 4 (although it had many useful other improvements). My pattern matcher was not incremental, and was way too slow. But, conceptually the pattern matching tactician was a lot stronger than the previous one. Or so I thought.

If there was to be an ace up my sleeve, it was a pattern matcher based on Fotland's, but even faster and better. I pinned my hopes on that. What would it buy me?

1. If it was fast enough, I could substitute patterns for code. Since an average pattern would be just over 16 bytes, one pattern equaled 4 ARM instructions. I expected I could save on code space using patterns, and they could fill up the 32K of non-code ROM.

2. Using patterns would increase reliability, since they couldn't cause

the program to crash or do anything malign other than match or fail to match.

3. With an embedded pattern editor, I could see a mistake, fix it and continue in the same game. It would speed the fix-it loop by making the program interpretive instead of compiled. Writing patterns is much faster than writing code.

4. I could afford any number of special case situations, described by patterns.

RESULTS:

When 6 months were up, I had a go-playing program that could fit on the ARM chip, play fast, and wasn't as strong as Goliath yet, though it was stronger than NEMESIS. In 6 more months, it was somewhat stronger than Goliath. It had played 250 games against a rated 10-kyu and gone from taking 9 handicap stones down to taking Black. Since strength deteriorates against a human as the human gains experience in the foibles of the program, this seemed like a pretty good sign. It took 12,000 lines of go engine source (excluding .h and tools), had 3000 patterns and 30 joseki.

While the program was not as strong as hoped, and not done as quickly as hoped, it was stronger than Goliath, fast, and proved that the pattern based system was a viable starting point for further development.

SIDE PROBLEMS ALONG THE WAY:

During first integration, their serial communications code didn't work right. Non-trivial debugging since they didn't speak much English, nor I much Japanese. Eventually it was solved by slowing down communications. Originally I was to send the entire go board image each turn. But that flooded the communications channel. So instead I had to spend 361 bytes of RAM tracking what had already been sent to them. 361 bytes may seem small, but remember I'm trying to stay within 16K RAM.

Once integration was accomplished, I tried to have the program play a game against itself on the ARM, to see if it played exactly the same as it did under DOS. It did, almost. After a hundred moves all the same, it varied for a few moves before returning to the same moves. After much debugging, I discovered that the ARM version, playing through their interface, used a 5 pt komi, whereas the DOS version defaulted to no komi. And at the point the game departed, one side was close to the boundary of "winning big". With komi, it went over that threshold and started playing more conservatively than its DOS counterpart.

Another problem was the mysterious freezing of the program on the ARM chip, whereas it ran perfectly under DOS. Eventually I discovered the compiler did not consider (((char*) ptr) + 1 + 2) to be the same as (((char*) ptr) + 3). It generated bad addressing code in the former case, so I revised my macros to work around the problem.

Despite these difficulties, the port from DOS to ARM went relatively smoothly.

The most onerous difficulty was created by supplying an extra feature.

Like Goliath and NEMESIS, RiscIgo could show you a shaded map of what it considered to be under each player's control throughout the game. This display is useful for weaker players, but allowed the customer to see into the program. And like the other programs, and like any kyu player, it made mistakes in its assessment. This lead to the customer wanting to change how the program thinks. This has it good and bad parts, and while sometimes it helps to make the program stronger, sometimes it generates friction between customer and programmer. A few examples are in order. First, consider an empty 19x19 board and Black plays the first move. To RiscIgo, territory is defined as those points surrounded by one player's stones and linkages and the edge, and not touching any enemy living stones. So after the first move, Black "owns" the whole board as territory. Of course this territory is easy to invade and never lasts. So I patched it to suppress such behavior on the 19x19 board. But when Black plays the center point on 9x9, he still owns the whole board. What surprised the customer was when White invades at a 3-3 point, Black's center stone was considered dead. Well, to RiscIgo, the Black stone was not in contact with any edge, had little running room, and could see no living friendly group. So it was presumed dead (not that it wouldn't play to try to save it). This didn't sit well with the customer's project manager. I argued that the program was not necessarily wrong, and offered to prove it. I asked him to make his stone live, while I played to kill it. He died. The point here being that many situations are up for grabs, and the program's heuristics for assessment were based on a 19x19 board. But the customer could see them, and thus want them changed. It was a serious problem in customer relations. In all fairness, however, the main problem was that the program wasn't 3 stones or more stronger than Goliath. This was just a symptom they could pick on.

RiscIgo Code Allocation in Kbytes

String tactics	9	group moves	10
Go board/Utility	7	response controller	8
Board assessment	7	link moves	5
Pattern matching	5	semeai moves	4
Tactics controller	4	contact fights	4
String+move data	4	edge moves	3
Territory data	4	territory moves	2
Sector/ Perimeter data	3	moyo moves	2
Link data	3	game phase moves	1
Edge data	2	ko moves	1
Access data	2	string moves	1
Joseki processing	2	access moves	.5
Link tactics	1	---	
Group/chain data	1		41.5
Eye tactics	.2		
---		SNES interface	7
	54.2	Various constants	1.5
		Patterns	56
		Joseki	.5

		Total	~162K

NEMESIS / RiscIgo DIFFERENCES:

Much of what NEMESIS did with code, RiscIgo does with patterns.

There was no room for RiscIgo to keep lists of anything around. So whereas NEMESIS kept strings with lists of stones, liberties, etc., RiscIgo only kept a bit map indicating which points were members of strings, groups, territories, etc., and created a particular data structure as needed by rescanning contiguous points.

The biggest code area in NEMESIS was territory tactics. RiscIgo had no such lookahead ability since I had neither time nor space to write it. However, RiscIgo used lots of pattern knowledge to substitute for some tactics.

The biggest move generator areas in NEMESIS were groups and linkages. The same is true of RiscIgo, reflecting the importance of these concepts in Go play. RiscIgo added additional code to handle semeais and ko.

I simplified and speeded up underlying primitives in strings, links, groups and territory.

NEMESIS had no global evaluation function to assess a board. I created one for RiscIgo that was fast enough to afford. As an additional feature, by tinkering mildly with the evaluation function, I could create a program with multiple personalities. The program had different styles of play the user could select (and which made for great graphics in the user interface).

RiscIgo RAM Allocation

Bytes

4,096 control stack and local scratch data

1,441 Go board maps (361 intersections of 4 bytes each)
Basic data (occ, str, link status, chain status)
7 string id bits
4 link bits (black & white x horizontal & vertical)
2 chain bits (black & white)
2 occ bits (black ,white , none = empty, both = off edge)
1 "been here" bit for scans
- above update automatically incrementally on regular moving

- below assessed data non-incrementally updated by assessment
- refreshes when returning to assessed turn level
4 weak link bits (horizontal & vertical x black & white)
2 group bits (black & white)
2 territory bits (black or white)
5 chain id bits
2 territory header bits
(a unique territory starts here by color)
1 weak link updated bit

1,444 Access map (361 points x 2 bytes x 2 colors)
each color intersection is 2 bytes
9 bits stone nearest of color to here
7 bits distance (only 6 needed)
- updated incrementally

800 Move list history (400 move limit)
- each move is 16 bits

9 bits location
4 bits kill directions (needed to retract moves correctly)
1 bit ko flag (shows suicide or ko depending upon kill bits)
2 bits color of player moving (only 1 bit really needed)

- std game = 250-300 real moves + 100 lookahead moves
- replayability of retracted moves is responsibility of interface

724 copy of current turn top level assessment bits for fast refresh

722 Display marks board (for indicating numbers and letters on location)

362 Tactic board indicates tactical result cache
- if on occupied pt, is on string name describing string
- if on link pt is on sole unique link pt describing link
- updated at top level of real turn only

8 bit tactical value
described why tactics stopped computation (too deep, ko, etc.)
described result of tactic for starting player (win/lose)

361 tile board indicates tactic cache updating requirements
- updated by assessment

8 bit impact rectangle index specifying board part involved

254 String: contiguous stones of same color
- Theoretic max. 228. Practical max 127. Avg. max is 75.
- Updated incrementally
- unit size 2 bytes.

9 bits name (high pt on board 1...361)
2 bits unused
4 bits dame count (15 dame maximum)
1 bit "big" (stones 1) flag

62 group enclosure data (2 bytes per group)
9 bit name of best runnable/enclosable stone of a group
4 bits define open running direction from stone
1 bit means is enclosed
1 bit means can be enclosed in 1 move
1 bit means within foe sector line

62 copy of group enclosure data

62 chains (collection of connected strings using pseudo-linkages)
- Practical max 31. In theory could be in the 30s, maybe 40s.
- Updated by assessment
- unit size 2 bytes

9 bits name (high point of chain)
3 bits safety assessment value
1 Might breakout bit
1 In semeai bit
1 Big group (6 or more stones)
1 Has many liberties

62 copy of chains data

42 21 edge boundary zones (two groups or 1 group + corner or 2 corners)
- Practical limit 20

- Unit size 2 bytes
- updated by assessment
    9 bit 1st line base
    2 bit direction code for along and onto board
    5 bit distance of gap

42 copy of edge boundary zone data from current turn top level

31 group association
- 1 per chain
- updated by assessment
- names chain index of master chain for each chain
(all chains in a group index to master, his is the id of the group)
(groups are chains associated by capture of foe)
- 5 bits chain id
- 3 unused bits

31 copy of group association

The "copy of" data is used to reassert the top level global evaluation of a turn after running a lookahead sequence. Evaluation data is not incremental, and done only at some interesting terminal node.

(Note: sometimes I bring other Go programs I have written into the discussion of RiscIgo. POGO was my first program, written in LISP in the 1970's. NEMESIS was written by me in the 1980s, and I stopped working on it in 1992 and have nothing further to do with that program. Ego is my next generation program after RiscIgo, not yet released.

Some of this write-up needs diagrams, but I'm not willing to spend to time making ASCII diagrams. A real paper will contain proper diagrams.

BASIC INCREMENTAL DATA STRUCTURES:

Basic data is updated incrementally (that is, the bits are always correct after each real or hypothetical move). Some of it can be suppressed during specific kinds of tactical lookahead. For strings, territory, groups and chains, the set of points involved in the unit is computed by scanning points and testing for the presence of the appropriate bit (the ones corresponding to occupation, chain status, or territory status) in the mask for that point.

Strings:

For strings, NEMESIS kept lists of stones, liberties and touching enemy stones, along with the counts of each. RiscIgo keeps just the liberty count, where to find the canonical start (the point of the string most north and east on the board) of the string, and whether the string is "big" (at least 2 stones) or not. Updating is incremental, and simple.

Links:

For every point changing occupancy RiscIgo checks related nearby points to see if the link bit needs changing. Special code is

devoted to this purpose. Setting the link bit is thus incremental, but no data about the link is computed until assessment. In Ego, pattern matching has been sped up enough to duplicate the speed of the special code, so linkage detection is done strictly using patterns.

Chains:

A point is a black chain member (and similarly for white) if:

1. black occupies it
2. black has more touching adjacent stones than white
3. if both have 2 touching stones, a black stone is north of the pt
4. if both have 0-1 touching stones, black has more diagonal stones than white.

Setting the bit is done incrementally, but no data is acquired about the chain until an assessment is needed. Discussion of assessment deferred until later.

Access:

POGO, NEMESIS, and RiscIgo do not use influence. Influence, as it is traditionally implemented, computes the degree of control over a point. If a point has +10 influence, it means black controls it with some measure of 10. But whether this was achieved by distant far black stones and even more distant white stones, or by close black stones and somewhat close white stones is not discriminated by this measure. Access maintains the discrimination of how far away each player's stones are from a point. I incrementally keep track of access. An access map specifies for a point, what point of a color is nearest to this point, and how far away it is (how many moves would it take to join a solid string line between the two points assuming you are not allowed to cross enemy linkages). Whenever a move is played, it generates a list of points that change occupancy, and points that change link status. For friendly access, a wave from this stone propagates to adjacent points as long as the access distance from this stone is less than that already existing on the new point. This is referred to as "spinning". For foe access, access is "ripped" out and then respun. For example, if Black plays a point, then the access for White through that point is retrieved (the white base stone and distance). For all propagated adjacent points with higher access coming from the same white base stone, access is cleared. Each point which is not cleared, is added to a list to respin. Then each is respun once, generating a layer of newly spun points into the cleared zone. Each layer is respun until no new layers are generated.

Access is used to find, among other things,

1. enclosure status (can a group access distant points or friends of another group?).
2. sector lines bounding a group
3. Territory (points which have no access from a living enemy stone).
4. How to handle a moyo (e.g., to defend, find a deep point in the moyo, then retrieve the enemy access stone to that point and play a move to block that stone from moving toward the moyo).
5. Define game phase based on worst access of either player

   9+: early opening 8: mid opening 7: late opening

6 : early midgame 5: mid midgame 4: late midgame
3 : early endgame 2: mid endgame 1: late endgame

6. the closest stone to a point, if you want to crawl toward that point (moyo attack).

PATTERN MATCHING:

Pattern matching is used for almost everything in RiscIgo. It is used in lookeahead, global move generation, and most of group safety evaluation is done using patterns.

In NEMESIS pattern matches were done on request (no caching or incremental updating). Patterns were divided into types, e.g., good shape, linkage defense of a specific type of linkage, real eyes. To request a match you specified the type, and the points used to orient the match. The matcher returned a board point and a value as its answer. Usually the answer was a move suggestion and an associated 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).

Patterns were described as text, and converted into an internal binary decision tree. A simple pattern is the following:

LENS 2 d3 c4 ENDLENS

This says that lens type 2 (diagonal connection defense) given the two endpoints of the diagonal connection, has two recognizable FIELDS (in <>). A field is a collection of points, which if occupied in specific ways, comprises a pattern. Fields were used in POGO, NEMESIS, RISCIGO and EGO.

If D4 is empty and C3 is empty, then return c3/5. If d4 is White and C3 is empty, return c3/33. Any number of points anywhere on the board could be contained in a field. The program to the set of all fields and built a decision tree out of it. In this example, C3-Empty would be at the root of the tree, with two leaves d4-empty and d4-white. The program walked the tree and performed the tests indicated (empty or white) during a pattern match.

The tests that could be specified included:

WHITE, BLACK, EDGE, EMPTY, NOTWHITE, NOTBLACK, NOTEDGE, which were simple occupation tests. Optimizations to avoid naming lots of points included

BLACK8, EMPTY8, WHITE8 (center as designated and the 8 points around it empty)

ANY1, ANY2, ANY3 (the 9x9 field named must be empty or off the board, except for the number of points named can and must be of the count named. E.g. Any2 means the 9x9 field must contain exactly 2 stones of any color. All remaining points must be empty or off the edge).

ADJ0, ADJ1, ADJ2, (the center and immediately touching neighbors must be empty or off the board, except for the number of points named can and must be of the stone count named. E.g, ADJ2 means that 2 of the 5 points designate (the center and 4 touching points), must be occupied by stones).

EMPTY25 (a 5x5 field named must all be empty or off board)

MARK, NOTMARK (the designated center must or must not have a marker bit set on it.) One might mark territory points for an eye pattern match,

for example.)
DAMEG1, DAMEG2, DAMEG3, DAMEL2, DAMEL3, DAMEL4, (liberties must match)
LIVEWHITE, LIVEBLACK (stone must be of a living group)
HYPOCONNECT (the original endpoints would survive an attempt to
disconnect them if this move were played)
HYPOKILL (the original endpoint would be killed if this move were played)
FRIENDEDG (along the edge, a friendly stone is next)
TEST (Perform a test against a global variable. Test can require ==, ,

< 2, dame < 3, dame < 4, dame 1, dame 2, dame 3,
edgeline = 1, edgeline != 1, not on plate of specific group,
friend 3, not on territory point, foe can't safely play,
friend territory, foe territory, foe 5 away, eye cannot be collapsed,
friend 6 away, no foe neighbor in atari

Each pattern has a DONT-MATCH bit. If this bit is on, then when this pattern is matched, it means don't allow this point to be matched (by keeping it on a local list), and continue matching. This allows me to note situations where moves shouldn't match.

Additionally, I can structure collections of patterns into "Globs". A glob is a collection of patterns. If the first pattern in the glob matches, then it skips over remaining patterns in the glob and continues matching. Globs can also be joined together. The effect is if the first pattern matches, then skip the 2nd header and continue matching inside the rest of the glob. If the first header fails, the second is usually a pattern that matches all situations, and all remaining patterns in the glob are skipped. This allows me to set up a precondition board condition for matching patterns in a glob. If the precondition fails to match, then the patterns are ignored. This substitutes rapidly for a hash code. For example, there are 143 patterns for string attack in 7 globs. If the preconditions all failed, then it would take 14 patterns scanned to fail them all (each glob 1st header precondition fails, the 2nd header always matches and skips all the rest of the glob).

All patterns are entered using a GUI pattern editor. Given a type number, 54 patterns from it fit on a page. Click on one, and it magnifies and displays the full detail, ready for editing and testing, after which you can resume

a game. An encoded pattern takes either 12 or 24 bytes (depending on field size and whether embedded tests are used).

There are about 100 pattern types as follows (some generate multiple classes based on linkage type or territory count): openDirection, press, run, enclose, semeai, friend eExtend, foe extend, LadderAttack, Contact attack, contact defend, diagonal contact, External eye, unstable territory boundary, revise planned move, fill a liberty, invade, self-atari reject, split territory into pieces, form links from, attack foe moyo, defend friend moyo, react without thought, big midgame edge moves, obvious follow-up sequence data for lookahead, attack a territory boundary, stop the wriggling of a dead group, desperate group trying to live, do I care about saving this stone, connect through enemy link (watari), is there a link here, Great Wall joseki, eye value for territories from 1-7 points, general eye value for unrecognized small territories, eye value for dead stones, is this link obviously secure, endgame attack, endgame defend, is this a weak contact fight situation, linkage attack move, move here requires updating these linkage points, link defense move, string just reduced to 2 liberties, tactical lookahead reflex moves, seki shapes,

The classes with the most patterns are:
contact attack (162), string tactics attack (143),
string tactics defense (139), react to foe move without thought (135),
single skip link defense (135), obvious link defense sequences (114).

Average pattern size is 18 bytes for the 3,000 patterns in the database. In a sample game playing against itself on a 486 DX 2/50, Ego, using a somewhat faster pattern matcher, can play at a rate of 10 moves a minute. This involves 1410 calls to match a point in some orientation for some class per turn, checking 140 patterns per match. Some 44 million patterns were matched against the board during that game.

GLOBAL EVALUATION:

Global evaluation proceeds using the incremental basic data, without any tactical lookahead. Speed is essential, so tactics are not used. Pattern matching substitutes in most cases for tactics. Sometimes it's wrong.

Step 0: Clear all global assessment bits. These include chain id bits, group bits, territory header bits and weak link bits. The top 16 bits of a point's occupation data area assessment bits.
Step 1: Find the chains. This consists of a sweep over the board looking for contiguous collections of points with the same chain bit on.
Step 2: Find all interesting edge areas. Pairs of consecutive (but not necessary contiguous) edge links involving two distinct chains are interesting.
Step 3: Sweep the board and set the approriate territory bit for each point not reachable by the opponent (using reach map).
Step 4: Adjust territory. To avoid black claiming the whole board with the first move, if a territory point is "too far" from a friendly stone, the territory is ripped back to the nearest linkage boundaries.
Step 5: Find all territories. Find the contiguous points all having the same territory bit on. Choose one point from the territory and turn on the territory header for it. The friendly reach map for

this point should have a base visible to the chain owning this territory. Some points which are territory are not reachable by the friend. An example is a dead enemy group with eyespace, where points of the eyespace are not visible via reach data to the killer.

Step 6: Assess the life status of chains. Pass one does an initial determination for all chains. Each chain is its own group. After this pass, for all chains that are not alive, a conflict assessor tries to decide who dies (including if semeai or seki is involved). This assessor iterates over the groups until no group changes status. As chains get declared dead, the enclosing chains are merged into groups running through the dead stones. Territories are recomputed for the groups that die, so that each pass has the correct territory data.

Step 7: Assess the game value, computing territory, potential territory, and bonuses for each side. Keep track of the distribution of reach values, to decide game phase. Potential territory are points where one side has reach
9) is treated as territory. Potential not close to foe (reach 4) is worth 4 times potential where foe is close. Bonuses come from weak groups which are not yet dead. Bonus is based on life status of group and size of group. Thus making a move which changes the life status of a group up or down a notch is worth a bonus. Bonus is also based on style of computer opponent selected. Influence and territory are combined eventually. If the game phase is early on, 8 potential = 1 territory. Later in the game, 16 potential = 1 territory. Player style choice affects these conversions.

INITIAL CHAIN SAFETY EVALUATION:

For each chain, its reach to the rest of the board is analyzed, in combination with pattern matching, resulting in a determination of ENCLOSED, ENCLOSABLE IN ONE MOVE, WITHIN SECTOR LINE, or OPEN. Each territory it controls is evaluated for the number of eyes it is worth using a variety of pattern matches. Some are on the territory points, and some are on the boundaries to determine their stability.

A chain is ALIVE:
1. two eyes.

A chain is PRESENTLY_ALIVE:
1. enough moyo (points with far reach from foe) and the separate potential for 2 eyes
2. OPEN with enough possible eyes or edge expansions
3. has enough big edge expansions (2 pt skips)
4. has many possible eyes

A chain is LIVABLE:
1. has potential eyes and edge expansions = 2
2. has some eye and some moyo

A chain is PRESENTLY STABLE:
1. if it is outside of any sector line.

A chain is SICKLY:
1. has an eye and edge expansion room
2. is not ENCLOSED and has some edge room
3. just has moyo

A chain is BAD:
1. is just not ENCLOSED

A chain is DEAD WITH AJI:

1. just has some edge expansions
2. has more than one possible eye

A chain is DEAD otherwise.

But, if a chain seems dead, additional edge pattern analysis is then run, and if successful patterns are found, status is changed to SICKLY.

TACTICS

There is a general tactics controller, to provide the lookahead framework. Within that framework, depending upon the search type, it can call modules for string, link, eye, or "obvious" global sequences. Upon arrival at a new board position, all moves RiscIgo might play from here are generated in priority order. Before move generators for the specific search types are called, a generator "obvious reflex" generator is called. Whenever RiscIgo returns to a position, RiscIgo decides whether to reorder or ignore the suggestions, but RiscIgo never generates new moves. Except for the "obvious" global sequences, searches are simple succeed/fail and use no alpha-beta. Obvious sequence searches keep minimax values.

The results of success or failure of tactics are kept in a cache, but the actual move returned is discarded immediately (limited RAM, remember). So if RiscIgo decides that it wants to play a move based on a tactical lookahead, in general it must redo that lookahead a second time to get the answer. The cache keeps a number from 1-32, designating what board area fragment, if played in, would require the search to be redone. A search always fits within some fragment, even if that fragment is the entire board.

STRING TACTICS:

String tactics generates moves against a "virtual string". It takes the designated target string, and determines which other strings are pattern connected to it (provided there is time to keep them joined before getting captured). All liberties of the target complex are then pattern matched to:

1. locate eyes (16 patterns)
2. locate trick plays (16 sacrifice tesuji against the connections)
3. locate ordinary plays (roughly 150 attack and 150 defend patterns)

The search is aware of snapbacks (so even if the string can be captured, if it can snap back to life again, it isn't captured yet). It will terminate on going too deep for the attacker, if it gets two eyes, if the attacker tries to continue a failing ladder, or the liberty count for the defender gets high enough. Additionally, if a target is being attacked as a counterattack from some original target, if the target complex gets more liberties than the original target, the counter attack is stopped. The patterns range from the very general (fill any liberty), to the more specific (fill liberties in this order), to the very specific (play a geta move or some clever tesuji or make a 2nd eye).

LINKAGE TACTICS:

Pattern matching determines the initial moves required to attempt to cut or connect the linkage. When those patterns cease, additional patterns mark the cutting strings to counter attack, and string tactics is called

to try to kill or save them.

OBVIOUS SEQUENCE TACTICS:
Pattern matching is done on the most recent move, and the move before that, to determine the obvious responses to the recent move, and the follow-ups to the previous move that should be tried. When no more moves are generated or the depth limit is reached, a global evaluation is performed and the score returned. 16 lines of C.

EYE TACTICS:
Pattern matching the eye point determines all moves and results. It uses the same 65 patterns that global evaluation uses when judging the eye value of a single point territory. 10 lines of C.

LIFE AND DEATH TACTICS:
0 lines of C. (i.e., not done).

JOSEKI:

RiscIgo has several joseki databases, varying from half board to small corner. It matches without cacheing, 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. Due to space considerations, only 30 joseki are recognized. In addition, RiscIgo uses the pattern matcher to generate moves whenever it wants to play the Great Wall Opening. 7 patterns generate the wall itself, and 28 patterns handle follow-up attacks on corners to make use of the wall.

THE END