

# Running Fire: Overall Program Architecture

Howard A. Landman

February 12, 2002

## 1 General considerations

1. We want to be able to analyze tactical status of strings. This is the single most important advance of RF over Poka.
2. Combinatorial game theory is the true and accurate way of looking at Go. Therefore all searches should return game-values. We need in general the capability to do arithmetic with (add, subtract, compare) game-values. David Wolfe's code may be useful here, but we almost need to define a new language where games are a superclass of numbers, i.e. are a fundamental data type. Normal game metrics (mean value, left stop, right stop, temperature) should be easily accessible and routinely computed; in fact, it wouldn't be a bad idea to be able to plot thermographs.
3. The combination of (1) and (2) means that traditional search methods (e.g. alpha-beta) are inadequate. They only get the main line (players alternate moves), which is important but not sufficient to e.g. distinguish sente from gote. So we need a broader search technique. Doing this naively (looking at all possible moves) would mean terrible (exponential) increases in run time. So, the question of efficiency is critical - in fact central to the viability of this whole approach - and should be investigated thoroughly and early.
4. Disk space is cheap and RAM abundant. We can assume a computer with 100 GB of disk usable by the program and its databases, and 1 GB or more of RAM. However, virtual memory can only be used up to a limit of 2 GB (or perhaps 4 GB), which is a small fraction of the 100 GB, and also not much larger than the actual RAM size. Therefore vmem will not be a significant help (until we have 64-bit systems), and a large fraction of the total data will be stored in external files.
5. In order to avoid redoing expensive calculations, we will store searches as data structures. Each search must therefore know when it is invalidated, so we can update it just-in-time. We introduce the concept of a *lens*: the set of points which could possibly invalidate an analysis (or advance it a step). Part of the job of doing any search is to compute its lens. Since each subnode of the search is a search in itself, each subnode will have its own lens.
6. The existing Poka/GoDB Region routines may be adequate as a basis for lens calculation.

7. Since there is a very high degree of reconvergence in some searches, our general data structure must be a digraph, not a tree. Even this may not be completely adequate, since (e.g.) when filling in  $N$  liberties of a group there are  $N!$  ways in which to do it and sometimes the order doesn't matter very much. But it's necessary. This also implies that the digraph may be cyclic due to ko fights, so there must be a mechanism for recognizing and analyzing such loops (which can in general have cycles longer than 2).
8. When a move is made on the board, any analysis will either (a) be unaffected and remain valid, (b) be affected in a way that requires the analysis to be redone or extended, or (c) be affected in a way such that the move was anticipated and already analyzed, and the analysis needs to be advanced along that line of play but not recomputed.
9. Testing: there are well-established external tests of Go skill. As many of them as possible should be implemented (e.g., the entire Go Seigen tesuji dictionary) as problems for the program to solve. This implies a problem-presentation mode, external problem files (in some defined format, probably SGF, which we don't currently support, but would need to, implying full SGF-reading and -presenting capability), and an automated way of running tests (probably a scripting language such as TCL) and evaluating test results.

## 2 Game search

We now address the key question of what the game-value search mechanism looks like, how many nodes it must examine, etc.

At a bare minimum we need enough information to calculate the thermograph. This implies that we need the right half of the thermographs of the left subgames, and the left half of the thermographs of the right subgames. More accurately, we need the *most restrictive* of those only; others (corresponding to bad moves) can be discarded. So it appears that some kind of pruning can be achieved, which is good, because otherwise we would need to look at all possible sequences of moves by either player, which is factorially huge.

Thus it appears that we can get away with something roughly 4 times as complex as alpha-beta, in that we need to look at:

1. Black plays first, White ignores, Black gets another move in a row, then White starts answering.
2. Black plays first, White answers, etc.
3. White plays first, Black answers, etc.
4. Black plays first, White ignores, Black gets another move in a row, then White starts answering.

Of course there is a problem here in that we're not tracing out the whole game tree (let's ignore the digraph and loop issues for now), but only part of it. Thus we cannot get the true game values, but

only upper and lower bounds on them. This may be enough. It does imply that the basic concepts of alpha-beta search must be extended to include the possibility that when comparing two nodes the answer may be *fuzzy* (if the subgames are really fuzzy against each other) or *unknown* (if we haven't computed them well enough to be sure). In the unknown case, we may need to extend the search along those subgames until their value is sufficiently resolved. In the fuzzy case, we need to retain both subgames as neither one of them is "best" in all circumstances.

Aside: this assumes we can actually identify terminal nodes and evaluate them with perfect accuracy. Such assumptions are not always warranted, and their violation may introduce further complications. However the game theory does provide some assistance here in that we can often consider a line "played out" when its temperature drops below a certain threshold (often set by the perceived value of moves elsewhere on the board).

At any rate, it appears that we need to review the mathematical definition of "dominated options", and find a way to extend it to the situation of incomplete information, and determine how to use it to guide search. The fundamental questions appear to be:

1. How does one get approximate (correctly bounded) game theory values from an incomplete game tree with the values of some nodes approximated rather than completely explored and computed?
2. How does one do the node approximation in a way which works for the search?

Here's the core algorithm of alpha-beta:

```
int AlphaBeta(int depth, int alpha, int beta)
begin
if (depth == 0)
return Evaluate();
GenerateLegalMoves();
while (MovesLeft()) begin
MakeNextMove();
val = -AlphaBeta(depth - 1, -beta, -alpha);
UnmakeMove();
if (val >= beta)
return beta;
if (val < alpha)
alpha = val;
end
return alpha;
end
```

Now there are several problems with this from a Go point of view:

1. The termination condition is all wrong. In Go, the depth of the search cannot be computed ahead of time. It is absolutely necessary to read ladders. In some cases it is necessary to read

(narrowly) more than a hundred moves deep, and it is frequently necessary to read 30 moves deep. Thus we need to understand the particular goal of a given search, and know when it has been achieved or has failed.

2. Alpha and beta need to be game-values.
3. The comparisons need to be game-value comparisons, and the result can be  $>$ ,  $=$ ,  $<$ , or  $||$ . So we need to specify behavior in the fuzzy case (which is probably the most interesting case). In fact the existence of the fuzzy case probably means that alpha and beta are not single game values, but rather lists of (all the undominated) game values. (It may be possible to create a merged data structure representing all the game values seen, perhaps something like a thermograph, in which case a single data structure would suffice.)
4. And of course, to get a game value, we need to evaluate both the cases of Black and White moving first (and possibly, of taking two moves in a row).

It is not clear at first how to bound game searches in a game context where either side might get 2 or more moves in a row. One approach might be to take all the "edge" moves that we haven't yet evaluated, and give them "extreme" values that can't actually be achieved on a real Go board. To find the max value of a tree we could set all unevaluated black moves to "wins in big sente" and all the unevaluated white moves to "has no effect". Reversing these would give a min value. Setting all to "has no effect" would give a lower bound on the temperature, while setting all to wins in sente (for the player moving) would give an upper bound on temperature. Looking at the difference in the top-level game caused by a change in an edge move evaluation might give us a measure of how important it is to evaluate that move.

One obvious problem with implementing this naively is that the entire game tree would be getting evaluated (4 times!) for each move made. So we clearly need some way to do it incrementally and locally.

### 3 Unit Capture Search

Some of these issues can be illuminated by considering the issue of performing a focused tactical search to determine whether a unit (connected set of stones) can be captured. The key evaluation metric is *liberties*, since if that goes to zero the unit is captured, whereas if it exceeds some bound the unit is considered to have escaped. The unit always has a current liberty count, but this is only a loose estimate of the ultimate status. Also, the liberty count can change radically when the unit's color moves (anywhere from  $-1$  to  $+\infty$ ), but only a little when the opponent moves (from 0 to  $-1$  assuming suicides are forbidden). Most of the time we would expect the liberties to go up around 1 on the unit's move and down 1 on the opponent's move. Anyway, the bound for the two cases should perhaps be different. If we create an ordered pair as the bound, then a normal ladder search could be bounded by (2,1), and a simple net (Japanese *geta*) or loose ladder by (2,2). (Note that these are the upper bounds, for escape; zero is always the lower bound for capture.) Extremely difficult problems like "Iwami Jutarō's Prison Break" might require as much as (5,5).

Let's walk through a simple example to see how this might work.

## 4 Bibliography

1. G. Stockman [1979], A minimax algorithm better than alpha-beta?, *Artificial Intelligence* **12**, pp. 179-196.
2. R. E. Korf [1985], Depth-first iterative deepening: An optimal admissible tree search, *Artificial Intelligence* **27**, pp.97-109.
3. S. Markovitch & Y. Sella [1996], Learning of Resource Allocation Strategies for Game Playing, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp.974-979.
4. M. Mueller [1999], Decomposition Search: A Combinatorial Games Approach to Game Tree Search, *Proceedings of the International Joint Conference on Artificial Intelligence* **1**, pp.578-583.
5. Y. K. Kao [2000], Mean and temperature search for Go endgames, *Information Sciences* **122** 19, pp.77-90.