VARIABLE DEPTH SEARCH

T.A. Marsland and Y. Björnsson [1]

University of Alberta
Edmonton, Alberta, Canada

Abstract

This chapter provides a brief historical overview of how variable-depth-search methods have evolved in the last half a century of computer chess work. We focus mainly on techniques that have not only withstood the test of time but also embody ideas that are still relevant in contemporary game-playing programs. Pseudo code is provided for a special formulation of the PVS/ZWS alpha-beta search algorithm, as well for an implementation of the method of singular extensions. We provide some data from recent experiments with Abyss'99, an updated Chinese Chess program. We also pinpoint the current research in forward pruning, since this is where the greatest performance improvements are possible. The work closes with a short summary of the impact of computer chess work on Chinese Chess, Shogi and Go.

1. INTRODUCTION

The number of nodes visited by the alpha-beta algorithm grows exponentially with increasing search depth. This obviously limits how search can be used to assess the outcome of a game state. The basic question remains; how can game-playing programs make best use of the available time to find a good move? Although, the basic formulation of the alpha-beta algorithm explores all continuations to the same depth, it has long been evident that this is not the best search strategy. Instead, some continuations should be explored more deeply, while less interesting ones are terminated prematurely. In chess, for example, it is common to resolve forced situations, such as giving check or re-capturing, by searching them more deeply. The search efficiency, and consequently the move-decision quality, of the programs is greatly influenced by the choices of how to vary the search horizon. Therefore, the design of variable-depth search criteria is fundamental to any game-playing program using an alpha-beta minimax method.

---

[1] Department of Computing Science, University of Alberta, Athabasca CSF, Edmonton, AB, Canada T6G 2E8. Email: {tony,yngvi}@cs.ualberta.ca

2. BACKGROUND WORK

In the early 1970s programmers were still pursuing Shannon's proposed strategies. Type-A programs considered every move for a few move sequences (at that time 3 to 5 ply). Type-B programs selected only plausible moves and searched them for a few moves, and followed that with a variable depth quiescence stage until no captures remain. At this time the design of computer chess programs was mostly focussed on move ordering: checks, captures, "killer moves", threats and advanced pawn pushes. By categorizing moves into groups with either tactical (short-term) or strategic (long-term) threats, one can arrange that forcing moves are examined first. This technique leads to cut-offs (reductions) in the search trees of the remaining siblings (unsearched moves in the current position).

It was also during the 1970s that the notion of iterative deepening was refined and tuned. In the interests of completeness and uniformity, the searches were generally done to some fixed depth (or iteratively so until the allotted time ran out). The use of time to control the search is especially important, since it offers the flexibility to confirm that the best move from the previous iteration continues to preserve its status as a safe choice. Meanwhile the selective search approach fell into disuse, although always trying to re-emerge through some kind of forward pruning method. Forward pruning (the discarding of moves after cursory examination) proves to be far more difficult to implement in computers than it appears to be for humans to handle.

Thus it became clear that the reliability that comes from completely searching all necessary continuations provides consistently better results than is possible by the selective discard of some variations on the grounds that they have little potential (are not relevant to the current themes). For example, medium term sacrifices are typically discarded prematurely. Thus the notion of forward pruning fell out of favour during this decade, and with it went Mack Hack's reliance on a plausible move generator (Greenblatt *et al.*, 1967) – though this early chess program was the established leader for about five years. By the end of the decade the benefits of the variable depth quiescence search became apparent and this led to the dominance of the Type-A programs, as they evolved into a two-stage search process by adding a quiescence phase (Figure 1).

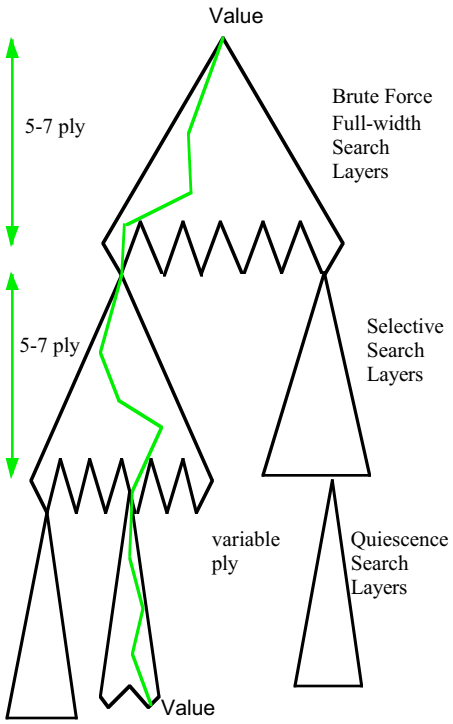

Figure 1: Staged search to variable depth.

3. GROWTH OF THE MICRO

With the steady increase in processing speeds, deeper searches became possible, causing the focus of the 1980s to move to a staged search. Figure 1 illustrates the common three-stage search – several ply of full width search (where every required move is examined), followed by an approximately equal layer of selective search and terminated with a tapered quiescence search of predominantly capturing moves and responses to check. Although somewhat *ad hoc* at the time, this model of variable-depth search proves to be quite robust. Nevertheless, this kind of staged search does not adequately reflect the intuitive notion that extensions to the search depth should be selectively, instead of uniformly, applied along forcing lines. Widely recognized was the need to extend the search by a single ply whenever one side is in check. A related idea was to extend when one side has only a single move, but really what is wanted is an extension whenever one side has only one "sensible" move (e.g., a simple piece exchange). This implies some kind of forward pruning to prematurely stop expansion of identifiably bad moves. Two well-established workable forward pruning ideas were razoring and futility cutoffs. For example: if, just before the horizon, the score is already above the beta bound

for the side to move, prune immediately (razoring). Alternatively, if the current move is below the alpha bound and the available positional factors don't have the potential to raise the score above the alpha bound, discontinue this line (futility cutoff). While seemingly good at reducing the nodes visited, futility cutoffs are often not cost-effective in terms of time. Thus it is a more implementation-dependent method than one would like.

Variable depth methods like these were further refined to ensure that the extensions would be controlled, so that an unbounded search (e.g., during perpetual check) did not arise, or so that search explosion did not occur (e.g., when a pawn is promoted at or near the nominal search horizon). Of primary concern to the programmers of this era was how to deal with the horizon effect (i.e., how to prevent the pushing of material loss beyond the search horizon by the insertion of frivolous moves – often checking moves). Thus from an early time it was clear that some form of selective variable depth search was necessary.

The 1980s was also a period of tremendous technological change, with the price of a small computer falling below that of an automobile. During this time the speed of the central processing unit in a computer doubled every two years or so, and personal computers became common and within reach of recreational programmers. The price of RAM fell dramatically, while secondary storage capacity rose steadily. With the newer processors came a larger address space, so that now three things became possible.

Firstly chess programs quickly migrated from general-purpose multiprogramming mainframe computers to single-user personal machines dedicated to one application at a time. Secondly, move ordering mechanisms continued to be important, but were further refined. For example, the use of "killer moves" was supplemented with the more general idea contained in the history heuristic – a 64×64 table for tracking how often feasible moves had recently been causing cut-offs (Schaeffer, 1983). Finally the additional memory and larger address space made it possible for transposition tables to increase dramatically in size and to be used for more than just storing the results of searches of sub-trees (used to restore a solution, should a previously occurring position occur again through transposition of moves).

Being larger, the transposition table now became the preferred and most powerful move-ordering mechanism, guiding the search from one iteration to the next along the best available path that had been found thus far. By this means move generations are sometimes avoided (e.g., if the table moves causes a cut-off) or delayed until it is certain that some sibling moves must be examined. With increased memory space, renewed interest was also shown in the Best-First strategies like Stockman's SSS* and B* (Berliner, 1979) and combination methods like DUAL* (Marsland *et al.*, 1987) although versions that are computationally efficient were slow to evolve. Nevertheless steady progress was made, eventually culminating in MTD(f) (Plaat *et al.*, 1995) which reformulated SSS* to use zero-width windows and transposition tables, and so obtain efficiency comparable to the alpha-beta methods that are the mainstay of computer chess

search. All state-space methods are limited by their memory requirements, and by their use of expensive methods for determining the best node to expand next.

On the other hand, the alpha-beta search algorithm appears to be concise, especially in Knuth & Moore's original NegaMax framework, but with its many refinements an actual implementation can also be lengthy and involved. Some people find the NegaMax formulation alien, as Figure 2 suggests, and yet its programming elegance and simplicity cannot be denied. Figure 2 illustrates the case when a zero-width window search fails high and this initiates a PVS search (the area in the box). Here PV and ALL nodes represent positions where every successor must be examined, while a CUT node represents the case where only a few successors are examined before a cut-off occurs. In the ZWS phase, CUT represents a node where a cut-off was expected, but now every successor is being expanded (it is being converted into an ALL node). Similarly ALL represents an expected full-width node that is now cutting off, thus indicating that a new PV may be emerging. Although it is true that game trees are made up of only three types of node (PV nodes along the principal variation, and alternating CUT and ALL nodes on the other paths), the true situation is better described with at least five node types (Reinefeld & Marsland, 1987).

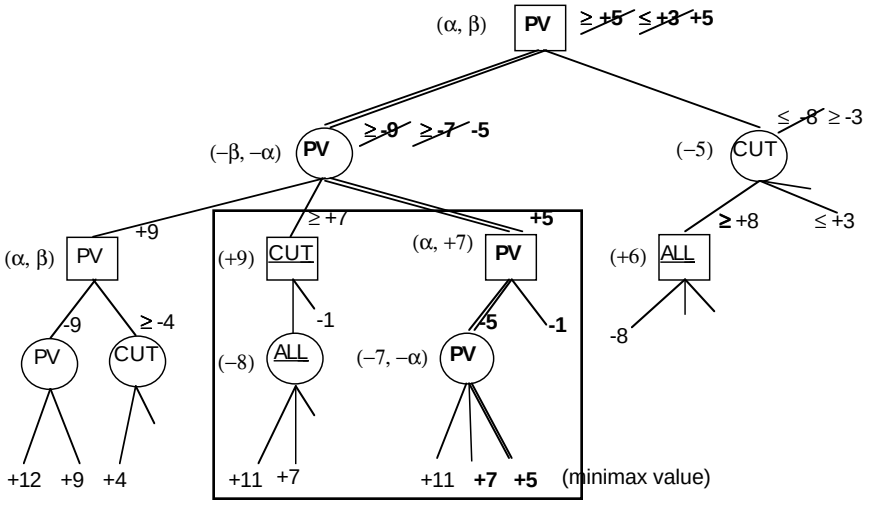

Figure 2: Sample Pruning with the NegaMax Method.

Despite the increased computing power that was brought to bear on the computer chess problem during the 1980s, the best programs were barely contending for Grandmaster status in regular play, though they more than held their own in speed chess against anyone. Nevertheless, the end of the decade saw increased use of dedicated computers, and heavy reliance on various hash-transposition tables not only to provide improved support for iterative deepening, but also to help speed the horizon node quiescence search. This period also saw much work on the

production of 3, 4 and 5-piece endgame databases, and the review of some end games that were thought to be drawn by the 50-move rule (Thompson, 1986). Thus by 1990 all the important elements were in place for computers to play consistent grandmaster chess.

4. ASCENT TO GRANDMASTER

In the 1990s, with good criteria for automatically extending search well understood, attention once again turned to "forward pruning", an idea that had been repeatedly tried in the previous two decades, but with mixed success. Humans are adept at simplification and apparently ignore moves that seem irrelevant to the current themes of play. From the computational standpoint, the size of the game tree can only be significantly reduced through the use of powerful forward pruning techniques.

A generalization of the razoring and futility ideas is use of the null-move in a quiescence search (Beal, 1989). The essence of the third (quiescent) stage of search is to consider only capturing moves, some early checking moves and destabilizing tactical moves like fork threats. The use of a null move (that is, allowing one side to move twice) ensures that a tight bound on the search outcome can be found more quickly. Losing capture sequences are truncated by assuming that one side can "stand pat", and so the search can achieve a merit value equal to that of a non-capturing (quiet) move.

Null-move techniques were the forward pruning method of choice in the early 1990s, and remain so. They also provide the possibility of generating a short list of opponent threats. Thus new criteria for dynamically re-ordering the current player's move list became possible, namely that moves which explicitly counter those immediate threats should be considered first.

The most successful null-move variation, widely incorporated into chess programs during the 1990s, is Null Move Forward Pruning (Goetsch and Campbell, 1990; Beal, 1990; Donninger, 1993). Although a formal definition of the null move was provided many years ago, let us look more closely at what it is trying to do, and consider why and when it is effective as a forward pruning method. Figure 3 shows pseudo code for PVS/ZWS (Principal Variation Search using a Zero-Width Search). The transposition table code is omitted, since it is adequately described elsewhere (Marsland, 1986). This particular formulation is different from that found in NegaScout, but has advantages in parallel processing applications, since it simplifies the work distribution problem. Here the null move heuristic appears as bold font in the ZWS portion of the search. This is the most frequent usage. Here `ReduceSearch()` computes the appropriate search reduction to apply, while `ForwardPrune()` determines whether the pruning condition is met. The code can also be included in the PVS portion, not only to curtail the search but also to raise the alpha bound of a new principal variation, although here the method is more problematic. Given that a null move or "pass" is not legal in chess, it would seem

to be a contradiction to allow one side to move twice. However, by allowing a second move by the same player, albeit to less than the current nominal search depth, one can determine if the situation is probably futile, and so forward prune at that point. The method will fail in Zugzwang situations, where the side to move can only weaken its position. To reduce the chance that Zugzwang will cause a problem, null-move forward pruning is not done in the endgame.

```
/*
*    Given a Principal Variation Search procedure
*       V = PVS (Root, Alpha, Beta, D)
*    that returns V, a value in (Alpha, Beta),
*    by searching the "Root" game-tree to Depth D.
*    PVS draws on ZWS (Zero-width Window Search procedure)
*       Merit = -ZWS (Sibling, -Alpha, D-1, TryNullMove)
*    to determine a bound for the Sibling value.
*    An ZWS search fails-low if Merit < Alpha
*    and fails-high if Merit >= Alpha.
*    TryNullMove enables the Null-move forward pruning
*    heuristic, with depth reduction of R > 1.
*    It is applied recursively.
*    Omitted is any use of a Transposition Table.
*/
PVS (Root, Alpha, Beta, Depth) ➔ ExpectedValue of Root

     // maximum depth or (stale)Mate
  if (Depth <= 0) || (Root == TERMINAL)
     return (Evaluate(Root));

     // generate successors, select first one
  Next = SelectSuccessors (Root);

     // Find expected value of the first variation.
  Best = -PVS (Next, -Beta, -Alpha, Depth-1);
     // Select next move on list
  Next = SelectSibling (Next);

     // Begin zero-width window (ZWS) searches
  while (Next != NIL)
     if (Best >= Beta)
          return(Best);
     Lower = Max (Alpha, Best);          // Raise lower bound
     Merit = -ZWS (Next, -Lower, Depth-1, TRUE);
     If (Merit > Lower)                  // re-search, new PV
          Merit = -PVS (Next, -Beta, -Merit, Depth-1);
     if (Merit > Best)                   // Fail-high
          Best = Merit;
     Next = SelectSibling (Next);

  return (Best);           // A PV-node

ZWS (Root, Bound, Depth, TryNullMove) ➔ EstimateValue of Root
  if (Depth <= 0) || (Root == TERMINAL)
     return (Evaluate(Root));

  R = ReduceSearch (Next, Depth);
              // typically, R is one of {1, 2, 3}
  if (ForwardPrune(Next, TryNullMove, Depth > R))
     Next = SwapSides(Root);         // null-move
     Merit = -ZWS(Next,-Bound+epsilon, Depth—1-R,FALSE);
          // if bound exceeded, treat as CUT-node
     if (Merit >= Bound)
          return(Bound);
```

```
Next = SelectSuccessors (Root);
Estimate = -INFINITY;
   // Loop doing zero-width window searches
while (Next != NIL)
   Merit = -ZWS (Next, -Bound+epsilon, Depth-1, TRUE);
   if (Merit > Estimate)               // Improved bound
        Estimate = Merit;              // Raise lower bound
        if (Estimate >= Bound)
             return(Estimate);         // Fail high, CUT-node
   Next = SelectSibling (Next);
return (Estimate);            // Fail low, ALL-node
```

Figure 3: Null-move forward pruning in PVS/ZWS.

Typical results from this era are given in the work by Ye and Marsland (1992), where a cost-benefit comparison for single and combination extensions for: check evasion, recapturing moves, king threats, evasive moves and strictly forced moves was given. Of these, extensions on check and on a strictly forced move are the most effective in Chinese Chess. Over the years the Chinese Chess program Abyss has evolved to form Abyss'99 and Tables 1A and 1B show some data from the current program. These results are for 5 to 8-ply searches and so complement the earlier ones. The comparison here is against the base program that includes both futility cutoffs and the null move. The full power of the null move is readily apparent in the comparison with the version without that feature (-null in Table 1A). For an 8-ply search a 10-fold improvement in search speed is achieved, without any negative impact on the solution rate (Table 1B). Additional data for the base version with extensions for giving check (+ch) and on a strictly forced move (only one legal reply, +sf) is provided. Both the traditional node-count and the more pertinent time ratios are given. The data in Table 1B illustrates the improved performance of Abyss'99 over the earlier version, not only are 20% more problems now solved with a 5-ply search, compared to Abyy'99, but also slightly fewer nodes are visited. To achieve this, significant improvement was made in the quiescence search (where a new capture-move ordering scheme is used, and all responses to check are examined to a maximum depth of three times the current iterative depth), and the whole program is now more precise and robust. From Table 1A we see the relative cost of the two most effective extension heuristics. Use of the null move in ZWS leads to a significant node count reduction without loss of solutions found, while the check extension and strictly forced move extension provide significantly improved performance at acceptable cost, even at the deeper searches. As a minor statistical note the +ch+sf version of Abyss'99 spent about 3900 seconds processing the 50 problems in the test suite (and solving 86% of them), at an average search rate of 53,000 nodes per second, and using a maximum iterative search depth of 8 ply.

Table 1A: How the total node count and CPU time values increase with different extension heuristics, using a test suite of 50 positions.

	5 ply		6 ply		7 ply		8 ply	
Features	**time ratio**	**node ratio**	**time ratio**	**node ratio**	**time ratio**	**node ratio**	**time ratio**	**node ratio**

-null	1.56	1.14	4.33	2.59	3.85	2.24	10.50	6.27
base	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
+ch	1.25	1.16	1.32	1.18	1.46	1.29	1.39	1.26
+ch+sf	1.58	1.42	1.60	1.39	1.93	1.62	2.03	1.74

Table 1B: The improvement of Abyss'99 over Abyss'92, using a test suite of 50 positions.

	5 ply Abyss'92	**5 ply**	**6 ply**	**7 ply**	**8 ply**
Features	**Solved**	**Solved**	**Solved**	**Solved**	**Solved**
-null	30%	36%	50%	56%	64%
base	**30%**	**38%**	**50%**	**56%**	**64%**
+ch	50%	52%	62%	74%	80%
+ch+sf	50%	60%	68%	80%	86%

At about the time the null-move methods were being described, people began to work on other ways to vary the search distance. It was already well established that responses to check should not count as a move that takes one closer to the search horizon. To this, an automatic extension can be done for every forcing move with but a single response. In the early 1990s, the notion of a singular extension was introduced and tried (Anantharaman *et al.*, 1990). Figure 4 provides pseudo code for our version of this method. It is employed at each node on the PV (but not at the root, because implementation details of this special case are awkward). Moves that are substantially, or singularly, better than any sibling are searched one ply further to reduce the risk of a horizon effect. That effect is particularly troublesome when one side "wins" a major piece and then sacrifices some smaller piece in a futile attempt to prevent the recapture of the major one.

```
/*   ------------------------------------------------
*        PVS with Singular Extensions.
*    ------------------------------------------------
*/
PVS (Root, Alpha, Beta, Depth) ➔ ExpectedValue of Root

  if (Depth <= 0) || (Root == TERMINAL)
     return (evaluate(Root));

     // Generate successors, select first one
  Candidate = SelectSuccessors (Root);

  Best = -PVS (Candidate, -Beta, -Alpha, Depth-1);
  Next = SelectValidSibling (Candidate);

  research:
     // Candidate is the first PV.  Is it singular?
  while (Next != NIL)
     if (Best >= Beta)
          return (Best);
     Lower = Max (Alpha, Best);
     if (Candidate != NIL)
               // determine upper bound on current move
          Merit = -ZWS (Next, -Lower + Margin, Depth-1);
               // Is the current move close to candidate?
          if (Merit > Lower - Margin)
               Candidate = NIL;
     if (Candidate == NIL)
          Merit = -ZWS (Next, -Lower, Depth-1);
```

```
        if (Merit > Lower)
             Merit = -PVS (Next, -Beta, -Lower, Depth-1);
        if (Merit > Best)                    // New PV emerges
             if (Merit > Best + Margin)
                  Candidate = Next;          // new singular
             Best = Merit;
    Next = SelectValidSibling (Next);

  if (Candidate != NIL)
        // does singular move holds up under extension
    SS = -PVS (Candidate, -Best, -Best+Margin, Depth)
        // Candidate may be best, but not singular
    if (SS <= Best - Margin)
        Delete move from "SelectValidSibling List";
        Set Next to first entry in SelectValidSibling;
             // Restart with first move on shorter list
        go to research;
    else return (SS);          // Singular move preserved
  else return (Best);
```

Figure 4: PV Singular Extension

In a similar vein, an implementation of Fail-High singular extensions is given in Figure 5. Despite the strong case that was made for the singular extension method (it was thought to be especially beneficial in human-computer matches), there is little evidence of its effectiveness in computer–computer games.

```
/*   ------------------------------------------------
*        ZWS with Fail-High Singular Extensions.
*    ------------------------------------------------
*/

ZWS (Root, Bound, Depth) ➔ EstimatedValue of Root

  if (Depth <= 0) || (Root == TERMINAL)
    return (Evaluate(Root));

  Next = SelectSuccessors (Root);
  Original = Next;

  Estimate = -INFINITY;
  while (Next != NIL)
    Merit = -ZWS (Next, -Bound + epsilon, Depth-1);
    if (Merit >= Bound)          // A cut-off?
        Singular = Next;
        /*
        *    If Merit exceeds every sibling by more
        *    than FH_Margin, move is singular.
        *    Extend depth, see if the cut-off
        *    preserved.  If so, return.  If not,
        *    keep looking.  Return if not singular
        */
        Tmp = Original;          // Start at previous
        while (Tmp != NIL)
             if (Tmp != Singular)
                  R = ReduceSearch (Next, Depth);
                       // R usually in {1, 2, 3}
                  Value = -ZWS (Tmp, -Bound+FH_Margin,Depth-1-R);
                  if (Value > Bound-FH_Margin)
                       return (Merit);
                  // Candidate not FH_Singular
             Tmp = SelectSibling (Tmp);

        // Currently best move is singular, extend
        // Compare new cut moves to "Candidate"
        Merit = -ZWS (Singular, -Bound+epsilon, Depth);
        if (Merit >= Bound)
             return (Merit);         // valid singular
```

```
        // Merit < Bound, singular did not cut-off
        Original = Singular;        // Note candidate
        if (Merit > Estimate)
            Estimate = Merit;       // Raise best value
   Next = SelectSibling (Next);
Return (Estimate);                  // Fail-low, ALL node
```

Figure 5: Fail-High singular extension

Table 2 shows the outcome of a match between two versions of the same Chinese Chess program, one with and one without singular extensions deployed. The singular extension version does increasingly poorly with increasing time per move available. Abyss'99 was used for this small-scale feasibility study. Note that at 100 seconds per move the tournament mode version of Abyss'99 (null-move version with check and strictly forced move extensions enabled) was searching about 8-ply in the middle game. Clearly SE.Abyss'99 (Abyss with singular extensions enabled) does less well with increasing average search time per move. Despite this discouraging performance the ideas behind singular extensions enabled better forward pruning techniques to emerge (Björnsson and Marsland, 1998). On the other hand it could be argued that this experiment is biased against the Singular extension method. This opens the interesting question of how to estimate the incremental cost of singular extensions. A simple experiment where SE.Abyss'99 was given 10% more time (33 seconds/move) against Abyss'99 (30 seconds/move) was done. The last line in Table 2 provides the outcome from a sample test, and suggests that approximate equality can probably be achieved if Singular Extensions can be run on a machine with a 10% speed advantage. A much more complete experiment is necessary, the pseudo code of Figures 4 and 5 can be used for that purpose, before definitive answers are possible.

Table 2: 24-games (from 12 unique starting positions) matching SE.Abyss'99 against Abyss'99 (both in tournament mode).

	wins	draws	losses	result	SE win percentage
10 secs/move	8	9	7	12.5 - 11.5	52.08
30 secs/move	7	6	11	10 - 14	41.67
100 secs/move	6	6	12	9 - 15	37.5
33 s/m for SE.Abyss	10	5	9	12.5 – 11.5	52.08

Finally, as part of the need for increased variability in the search horizon, more flexible search limits are commonly enabled during the end game phase. By making the search node-count-limited, instead of depth-limited, it is possible to follow much longer sequences of moves at positions with few continuations (hence making it possible to avoid draw by repetition in endings).

5. SOPHISTICATED PRUNING

In the past, forward pruning has been a high-risk method, but one with high potential pay-off. At present the variable-depth search methods are an active research area. Recently, some of the existing pruning methods were greatly

improved (Heinz, 1999). First, the futility-pruning and razoring methods were generalized to allow for pruning further away from the horizon. The generalized methods are called extended-futility-pruning and limited-razoring, respectively. By using a wider "security-margin", the extended pruning methods can be applied relatively safely at pre-frontier nodes. Second, Heinz's experiments with the null-move show that it is relatively safe to use a search-reduction factor of 3 when the remaining search depth in the tree is greater than 6 plies (except when approaching the endgame, when an 8-ply margin is necessary for safety). For shallower sub-trees the null-move searches are shortened by only 2 plies, as is normal practice. This variable scheme is called Adaptive Null-Move pruning, but collectively Heinz refers to the three above methods (Adaptive null-move, Extended futility-cutoff, and Limited razoring) as AEL-pruning.

Multi-cut pruning is another new search reduction method (Björnsson and Marsland, 1998). For a new principal variation to emerge, every expected CUT-node on the path from the root to the horizon must become an ALL-node. At cut nodes, however, it is common that even if the first move does not cause a cut-off, one of the alternative moves will. The observation that expected CUT-nodes –where many moves have a good potential of causing a cut-off – are less likely to become ALL-nodes forms the basis of this method. More specifically, before searching an expected CUT-node to a full depth, the first few children are expanded to a reduced depth. If more than one of the depth-reduced searches causes a cut-off, the search of that subtree is terminated. However, if the pruning condition is not satisfied, the search continues in a normal way. Clearly by basing the pruning decision on a shallower depth, there is some risk of overlooking a tactic that results in the node becoming a part of a new principal continuation. However, it is reasonable to take that risk, since the expectation is that at least one of the moves that caused a cut-off when searched to a reduced depth will cause a genuine cut-off if searched to full depth. This multi-cut scheme can be thought of as the complement of Fail-High singular extensions; the former prunes the tree if there are many viable moves at an expected CUT-node, whereas the latter extends the tree when there is only one viable move there.

Both AEL pruning and Multi-cut pruning have been shown to result in improved game-play, and are being employed by at least some of today's strongest chess programs.

6. SUMMARY: STATE OF THE ART

To conclude, let us consider the situation in other adversary games. Chinese Chess, for example, has complexity comparable to chess, although it seems to lead to longer tactical exchanges. On average slightly more capturing moves are possible, the board is slightly larger, and a draw by repetition of forcing moves is not allowed. Despite that, the tactical exchange and long-term planning ideas from chess carry across. Thus most computer methods developed for chess apply equally well in Chinese Chess.

For Shogi, on the other hand, things are not so clear. First the complexity of Shogi is greater than for Chess and Chinese Chess (Matsubara *et al.*, 1996). Captured pieces in Shogi "change colour" and can be returned to the board at any later time in lieu of a move. Current research on computer Shogi is very active. Second, it would seem that transposition tables are less valuable (since transposition of moves is less common). Many of the latest ideas from chess have been tried, and there is some potential for forward pruning methods to work well, because of the increased complexity that arises from the more uniform search width that must be maintained. In Shogi, the notion of an endgame is also quite different from chess, thus transposition tables are less likely to be as effective for guiding the search in that phase, although it should still be good for recognizing repetition cycles. Other memory functions should be possible, however. Perhaps a good source of information about the difficulties faced by Shogi programmers is to be found in Grimbergen's recent paper (Grimbergen, 1998). One group of Shogi programmers is experimenting with a different type of staged search, one where the alpha-beta algorithm is used is used in the first stage and a Proof Number Search (Allis *et al.*, 1994) for the second.

The next game in increasing complexity is Go, where brute force search techniques are thought to have far lower potential. At first sight Go is simple, but the 19×19 board leads to lengthy move sequences which require long-range planning. Move selection may come down to identifying a few key moves and exploring them to the exclusion of other "provably irrelevant" stone placements. Since search does not yet provide the answer, much of the work continues along classical lines of gathering data about how expert players "see" the game, and how humans learn Go concepts (Yoshikawa, 1998). Some of the recent papers are philosophical in tone (Mueller, 1998). The thrust remains on the need for plausible move generators, like those used by Greenblatt thirty years ago, thus closing our circle. While Computer Go is not thirty years behind in terms of research ideas and activity, the playing strength of Go programs remains at the amateur (good club player) level. Unlike in chess, the Go professional is not yet threatened by, and does not yet need, a Computer Go program as an assistant.

7. REFERENCES

Allis, V., Meulen, M. van der, and Herik, H.J. van den. (1994). Proof Number Search. *Artificial Intelligence*, Vol. 66, No.1, pp. 91-124. ISSN 0004-3702.

Anantharaman, T.S. Campbell, M. and Hsu, F-h. (1988). Singular Extensions: Adding selectivity to brute force searching. *ICCA Journal*, Vol. 11, No. 4, pp. 135-143. ISSN 0920-234X.

Beal, D. (1989). Experiments with the Null Move. *Advances in Computer Chess 5* (ed. D. Beal), pp. 65-79. Elsevier Science Publishers, Amsterdam. ISBN 0-444-

87159-4.

Beal, D. (1990). A Generalized Quiescence Search Algorithm, *Artificial Intelligence*, Vol. 43, pp. 85-98. ISSN 0004-3702.

Berliner, H. (1979). The B*-Tree Search Algorithm – A Best-first proof procedure, *Artificial Intelligence*, Vol. 12, No.1, pp. 23-40. ISSN 0004-3702.

Björnsson, Y. and Marsland, T. (1998). Multi-cut Pruning in Alpha-Beta Search. *Computers and Games* (eds. H.J. van den Herik and H. Iida), pp. 15-25. LNCS #1558, Springer-Verlag. See also Theoretical Computer Science (to appear) for an expanded version.

Donninger, C. (1993). Null Move and Deep Search, Selective-search heuristics for obtuse chess programs. *ICCA Journal*, Vol. 16, No.3, pp. 137-143. ISSN 0920-234X.

Goetsch, G. and Campbell, M. (1990). Experimenting with the Null-Move Heuristic. *Computers, Chess and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 158-168. ISBN 0-387-97415-6

Greenblatt, R.D., Eastlake, D.E. and Crocker, S.D. (1967). The Greenblatt Chess Program. *Proceedings of the Fall Joint Computer Conference*, pp. 801-810. Reprinted in *Computer Chess Compendium* (ed. D. Levy), pp. 56-66. ISBN 0-387-91331-9.

Grimbergen, R. (1998). A Survey on Tsume-Shogi Programs Using Variable-Depth Search. *Computers and Games* (eds. H.J. van den Herik and H. Iida), pp. 300-317. LNCS #1558, Springer-Verlag. ISBN 3-540-65766-5

Heinz, E. (1999). *Scalable Search in Computer Chess*, Ph.D. Thesis, University of Karlsruhe, Germany. Also published by Vieweg. ISBN 3-528-05732-7.

Marsland, T. (1986). A review of game-tree pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3-19. ISSN 0920-234X.

Marsland, T.A., Reinefeld, A. and Schaeffer, J. (1987). Low Overhead Alternatives to SSS*. *Artificial Intelligence*, Vol. 31, pp. 185-199. ISSN 0004-3702.

Matsubara, H. Iida, H and Grimbergen, R. (1996). Natural Developments in Game Research: From chess to Shogi to Go. *ICCA Journal*, Vol. 19, No.2. pp. 103-112. ISSN 0920-234X.

Mueller, M. (1998). Computer Go: A research agenda. *Computers and Games* (eds. H.J. van den Herik and H. Iida), pp. 252-264. LNCS 1558, Springer-Verlag. ISBN 3-540-65766-5

Plaat, A. Schaeffer, J. Pijls W. and de Bruin, A. (1996). "Best First fixed-depth minimax algorithms. *Artificial Intelligence*, Vol. 87, No. 1-2, pp. 255-293. ISSN 0004-3702.

Reinefeld, A. and Marsland, T.A. (1987). A Quantitative Analysis of Minimal Window Search. *Proceedings of the tenth IJCAI Conference* (ed. J. McDonald), pp. 951-954, Milan. ISBN 9-934613-43-5

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16-19. ISSN 0920-234X.

Thompson, K. (1986). Retrograde Analysis of Certain Endgames. *ICCA Journal*, Vol. 9, No. 3, pp. 131-139. ISSN 0920-234X.

Ye, C. and Marsland, T. (1992). Experiments in Forward Pruning with Limited Extensions. *ICCA Journal*, Vol. 15, No. 2, pp. 55-66. ISSN 0920-234X.

Yoshikawa, A., Kojima, T. and Saito, Y. (1998). Relations between skill and the use of terms: An analysis of protocols of the game of Go. *Computers and Games* (eds. H.J. van den Herik and H. Iida), pp. 282-299. LNCS 1558, Springer-Verlag. ISBN 3-540-65766-5.