FAIL-HIGH REDUCTIONS[1]

R. Feldmann[2]

University of Paderborn
Paderborn, Germany

Abstract

Fail-High Reductions (FHR) is a new method to guide the search in game trees in a very selective manner. The main idea is that the search algorithm should not spend too much effort searching subtrees, for which the side to move estimates that the opponent will avoid them to become part of the principal variation. More precisely, a fail-high node is a node v with a search window $[\alpha, \beta]$ at which a static evaluation function e produces a cut-off. The FHR algorithm reduces the search depths at these fail-high nodes thus searching their subtrees with less effort. FHR is domain independent in the sense that it only uses a static evaluation function, which always is assumed to exist, and the search windows to guide the search.

In this paper we describe the incorporation of FHR in our chess program ZUGZWANG. Three different tests are conducted comparing ZUGZWANG with FHR and without FHR. First, we compare the results obtained from running the algorithm on the Bratko-Kopec test set and on the positions of the WinAtChess test suite. Second, we look at the results obtained from three matches of 50 games each. The games were played under tournament conditions between three different versions of ZUGZWANG each with and without FHR. Third, we compare both versions by looking at the results of 100 games played against a chess program developed independently from ZUGZWANG. All

[1] This work was partly supported by the Leibniz Award Fund from the DFG under grant Mo 476/99 and by the DFG research project "Selektive Suchverfahren" under grant Mo 285/12-1.

[2] Department of Mathematics and Computer Science, Fürstenallee 11, University of Paderborn, 33102 Paderborn, Germany. Email: obelix@uni-paderborn.de.

three tests indicate that the FHR version is about 120 to 150 ELO points stronger than the version without FHR. Moreover, from the second test we obtain the result that with statistical significance the FHR version is stronger than the regular version.

1. INTRODUCTION

The problem of evaluating trees appears in many applications. For instance, expert systems, proof systems and various management tools use algorithms similar to the ones used in game-playing programs. The crucial point is that the best-known algorithms for determining the root's minimax value search game trees of depth d and width w in an average running time of $O(w^{d/2})$, i.e., the running time grows exponential with the depth of the game tree. Knuth and Moore (1975) showed that $\Omega(w^{d/2})$ is a lower bound on the running time of any algorithm that evaluates a game tree of width w uniformly to depth d. Hence, it is clear that under real time constraints the look-ahead of programs is very limited.

This is one reason why in many games, such as chess, the game-playing programs are inferior to the best human experts. We note that this is true although human experts are hopelessly inferior in speed – as measured in visited nodes per second – compared to the computers. Besides the superiority of humans in terms of intuition, smartness, and experience, still another reason for the overall inferiority of programs is that humans do a very selective look-ahead. Unlike computers, they consider only reasonable moves in their search.

Several approaches have been studied to mimic a human's selective look-ahead by computers. For instance, Berliner (1979) presented the B* algorithm which computes an upper and a lower bound on the value of an inner node to guide the search into relevant subtrees. The algorithm was investigated in more detail and improved to PB* by Palay (1985).

Later, McAllester (1988) developed the Conspiracy Number Search (CNS). The idea is that a decision at the root of the game tree should not be based on the evaluation of a single leaf node as it may be in the $\alpha\beta$ algorithm. Thus, CNS searches the game tree in a way to guarantee that at least some c leaves ($c > 1$) have to change their value in order to change the decision at the root of the tree. Several variants of CNS has been implemented in game-playing programs (Schaeffer, 1989; Van der Meulen, 1990; Allis, Van der Meulen, and Van den Herik, 1991; Lorenz *et al.*, 1995). Recently, CNS has been used in chess tournaments: ULYSSES CCN, a chess program written by Lorenz and Rottmann, played the 1995 World Computer Chess Championship in Hong Kong using a CNS algorithm.

The most widely-used algorithm in chess programs is the $\alpha\beta$ algorithm or one of its variants like PVS (Campbell and Marsland, 1983) or Negascout (Reinefeld, 1983). Without any selective extensions these algorithms search game trees of width w uniformly to a certain depth d with a running time of $O(w^{d/2})$ in the best case (Knuth and Moore, 1975) and $O(w^d)$ in the worst case. The huge practical importance of the $\alpha\beta$ variants stems from the fact that for most applications the average running time is close to the best-case running time.

Many heuristics have been developed to guide the search of the $\alpha\beta$ algorithm into branches of the game tree which are relevant for the decision at the root of the tree. Many of them are domain dependent, such as check-evasion extensions. Anantharaman (1991) has given an expanded overview together with an empirical comparison for many of these heuristics.

There has been some research on developing domain-independent heuristics to guide the search in game trees. In the late 1980s the Singular Extensions (SE) were developed (Anantharaman, Campbell, and Hsu, 1988) for the chess program DEEP THOUGHT. The idea behind SE is to increase the search depths in subtrees which are reached from the root by making a singular move. Here a move is called *singular* if it is much better than all its siblings. The drawback of this method is that the $\alpha\beta$ algorithm does not provide the search with the information that a move is singular. Extra searches have to be carried out with the risk that extra effort is spent for an uncertain qualification of a node.

Somewhat later, Beal (1989) re-investigated his idea of the Null-Move Heuristic (NMH). NMH is based on the observation that in many games and game-like applications for the side to move there is almost always a better alternative (move) than doing nothing (the null move). We call this the Null-Move Observation (NMO). Based on NMO, at an inner game-tree node first a search after a null move is carried out to a reduced depth. If this search already produces a cut-off then the search at the node is stopped hoping that there is at least one move available that would produce an even better result and therefore a cut-off too. Nowadays, many of the world's best chess programs seem to use the NMH. There are of course no publications of the professional chess programmers about this heuristic, except for Donninger (1993) who described an implementation of the NMH in his program NIMZO presenting also some promising results. The main drawback of the NMH is well-known: it fails in zugzwang positions. In these positions the NMO is not valid, causing chess programs to fail heavily. Therefore, in endgames chess programs usually switch off the NMH or try to verify the NMO by some extra searches.

Another drawback of the NMH is that extra searches are carried out in order to establish the cut-offs by the null move. If the search after a null move fails to produce a cut-off extra effort has been spent for nothing. The effects of various combinations of NMH, SE and some domain-specific heuristics are studied by Beal and Smith (1995).

Schrüfer (1989) proposed a method to achieve selectivity in quiescence searches, while Althöfer (1991) used majority systems to build game trees with only a few successors of the root. Recently, Buro (1995) described an algorithm called ProbCut, which is based on the $\alpha\beta$ algorithm. It uses the results of a shallow $\alpha\beta$ search to decide whether the result of a deep search would yield a value outside the current search window. ProbCut is domain independent and used in an Othello-playing program.

In this paper we describe a new selective-search heuristic, called Fail-High Reductions (FHR). Informally spoken, a node v is a fail-high node if the side to move at v estimates the value of the position so high that it believes that doing nothing at v is sufficient to prevent the opponent from playing into v. The basic idea is to reduce the search depth at these fail-high nodes by unity. We present some exceptions from the above rule later in this paper. In general, applying the above rule recursively all over the game tree leads to a very non-uniform search in chess trees.

Like the NMH, FHR is based on the NMO. However, it is not as rigorous as the NMH in using the NMO: there is still a chance that the reduced search will show that the NMO is not valid at some node, i.e., it fails to produce a cut-off. Instead of re-searching nodes in which the reduced search fails to produce a cut-off, in our implementation a combination of the FHR together with the Negascout algorithm (Reinefeld, 1989) is given, which automatically searches the node to a full depth if it turns out that the value of the node may become relevant for the decision at the root.

In the following we present the basic definitions for FHR. Then the implementation of FHR in ZUGZWANG is described and some implementation details are discussed. In Section 3 we investigate the behaviour of the FHR version of ZUGZWANG in several tests. Conclusions are given in Section 4.

2. FAIL-HIGH REDUCTIONS

In this section we describe the implementation of the FHR in ZUGZWANG, a distributed chess program usually running on parallel hardware. For a detailed description of ZUGZWANG's distributed search algorithm and its development we refer to Feldmann, Monien, and Mysliwietz (1991), Feldmann, Mysliwietz, and Monien (1992), Feldmann (1993), or Feldmann and Mysliwietz (1994). In this paper we only deal with the sequential search algorithm in ZUGZWANG. It is based on the Negascout algorithm as shown in Figure 1.

```
function Negascout(v : node; α,β,d : integer): integer;
var i,w,x,low,val,high : integer;
begin
        if d > 0 then generate all successors v.1, ..., v.w of v
        if v is a leaf then return(e(v));                          /* static evaluation */
        low := α; high := β; val := -∞
        for i := 1 to w do begin
            x := -Negascout(v.i, -high, -low, d - 1;             /* i > 1 : null window search */
            if (x > low) and (x < β) and (i > 1) then
                x := -Negascout(v.i, -β, -x, d - 1);                                 /* re-search */
            low := max(low,x); val := max(val,x)
            if val ≥ β then return(val);                                                /* cutoff */
            high := low + 1;
        end;
        return(val);
end;
```

Figure 1: The Negascout algorithm.

The following notational conventions are used. The function Negascout has four parameters: v, α, β, and d. When called for a node v, the parameters α, β and d have a value. $[\alpha, \beta]$ is called the *window* of v, d the search depth of v.

Note that Negascout may search the same nodes several times with different parameters α and β. The main idea is to search the appropriate successors of a node with an artificial zero-width window (null-window search) and to re-search these successors with their full windows only if the null-window search fails high with respect to the artificial window but not with respect to the full window. Negascout uses a domain-dependent static evaluation function e mapping positions to integers.

ZUGZWANG's regular evaluation function e contains subroutines detecting and scoring several kinds of threats, e.g., whether the side to move is checked or threatened mate[1], whether several high-valued pieces are attacked simultaneously, whether the opponent has passed Pawns etc. While computing $e(v)$ for some position v, a threat evaluation function t is computed ($t(v) \geq 0$). The function t may score the threats in a more pessimistic way than $e(v)$, e.g., we may assume that the function t has value ∞ if the side to move is either

[1] Here only those mate threats are considered, which may statically be checked. Peter Mysliwietz developed the mate-threat-detection mechanism for ZUGZWANG.

checked or threatened mate. However, t only scores the threats already detected by the evaluation function. No domain-specific knowledge is used for computing t that is not used for computing e. Since scoring a threat (simply adding a number) can be done much faster than detecting a threat, $t(v)$ can be computed without any delay while computing $e(v)$.

Definition: Let e be the static evaluation function used by Negascout, let t be a static evaluation function scoring threats against the side to move as roughly described above. A node v searched by Negascout with window $[\alpha, \beta]$ obtains the value $\tilde{e}(v) := e(v) - t(v)$. The node v is called a *fail-high node*, if $\tilde{e}(v) \geq \beta$.

Thus, $\tilde{e}(v)$ produces a value which is more pessimistic than $e(v)$. The amount of pessimism depends on the threat evaluation function t.

By the above definition we intend to mark nodes v in which the side to move estimates v so high ($\tilde{e}(v) \geq \beta$) that the opponent will avoid these nodes to become a part of the principal variation. For these fail-high nodes we will reduce the search depth in order to reduce the search time. The above definition provides a useful heuristic to the state of a node. The rate of errors may depend on the threat evaluation function t. During our work on ZUGZWANG the threat evaluation has been changed owing to Peter Mysliwietz's work on the regular static evaluation function e.

The experiments in Section 3 show that the behaviour of FHR remains unaffected for several different threat evaluation functions t. Early experiments (not reported here) were made with a threat evaluation function t consisting only in detecting whether the side to move is in check. In this case t had the value ∞, in all other cases t had the value 0. Even with this simple threat evaluation FHR worked well.

For the $\alpha\beta$ algorithm the fail-high nodes are nodes v in which the side to move estimates v so good ($\tilde{e}(v) \geq \beta$) that the opponent will avoid reaching v. In the Negascout algorithm this should be reformulated into: the side to move estimates v better than β ($\tilde{e}(v) \geq \beta$ and therefore $e(v) > \beta$), and hence node v can be avoided by the opponent if all assumptions for the artificial windows on the path from the root to v hold.

The main considerations are now as follows. If the NMO holds it is not necessary to search huge subtrees below a fail-high node v only to prove that one of the successors of v really produces a cut-off at v. In this case there is a successor of v which even allows the side to move to improve on $e(v) \geq \tilde{e}(v) \geq \beta$. If the NMO does not hold a search has to be done below v in order to find a less erroneous value for v than $\tilde{e}(v)$. Therefore, the FHR in its basic form works as follows.

FHR: The search depth d of the Negascout algorithm at a fail-high node v is reduced to $d - 1$.

```
function FHR-Negascout(v : node; α,β,d : integer): integer;
var i,w,x,low,val,high,eval,δ,t : integer;
begin
    eval := e(v); t := t(v);                             /* static evaluation */
                                                         /* t is computed together with e */
    δ := d; /* save search depth */
    if eval − t ≥ β and α = β − 1 then δ := δ − 1;
    if δ > 0 then generate all successors v.1, ..., v.w of v;
    if v is a leaf (i.e., w = 0) then return(eval);
    low := α; high := β; val := −∞;
    for i := 1 to w do begin
        x := −FHR-Negascout(v.i, −high, −low, δ − 1);    /* i > 1 : null window search */
        if (x > low) and (x < β) and (i > 1) then
            x := −FHR-Negascout(v.i, −β, −x, d − 1);     /* re-search without FHR at v.i */
        low := max(low,x); val := max(val,x);
        if val ≥ β then return (val);                    /* cutoff */
        high := low + 1;
    end;
    return(val);
end;
```

Figure 2: The Negascout algorithm with FHR.

Figure 2 shows the modified Negascout algorithm with FHR, henceforth denoted by FHR-NS. If a node v has to be searched to a depth d, the algorithm first computes the static evaluation $e(v)$ of v. Together with $e(v)$ a value $t(v)$ is computed and stored into t. Then FHR-NS investigates whether v is a fail-high node which is to be searched with a zero-width window ($\alpha = \beta - 1$). If this is the case, the search depth is reduced by one.

Below, we list some characteristics and implementation details in order to identify our version of FHR. The context of this contribution forces us to refrain from describing the experiments performed to measure the FHR effects. We distinguish eight items.

1. We tested several variants of the FHR. We introduced a margin Δ and investigated whether $eval - t \geq \beta - \Delta$. We explored several margins between 0 and 1 pawn unit. The best margin found so far is $\Delta = 0$.

2. The FHR is based on the NMO. Therefore one could argue that special attention should be given to nodes where the reduced search fails to compute a cut-off. We tested a version of the FHR that re-searched such nodes without depth reduction. This version turned out to be inferior to the original FHR version.

3. The question, whether a better approximation for threats should be used than the static function *t*, e.g., by doing a null-move quiescence search, is addressed. It turned out that the overhead for the extra searches does not pay off, even if these extra searches are restricted to nodes at least 3 ply from the leaves.

4. We tested versions which use FHR only in the deeper levels of the tree. These versions were superior to the version without FHR, but they were clearly beaten by the version that applies FHR recursively.

5. If FHR is used recursively in the full search tree, principal variations may arise which are obtained from searches with reduced depths. However, in Negascout a principal variation always results from a full-window search, i.e., a re-search after a null-window search. Since these re-searches are very rare, we want to use the FHR only when doing a null-window search. If $\alpha \neq \beta - 1$ then FHR-NS is definitely not in a null-window search (but searching the leftmost variation of the tree or doing a re-search) and FHR is excluded. If $\alpha = \beta - 1$ then FHR-NS is doing a null-window search with high probability. In the latter case we allow fail-high reductions. This has the additional effect that principal variations are very unlikely to be reduced in depth even if aspiration search is applied at the root.

6. The artificial windows of the null-window searches may allow many depth reductions. If the result of a null-window search indicates an improvement the re-search may not confirm this result, since many nodes in the subtree are re-searched without depth reductions. This effect may even happen in a regular Negascout algorithm if transposition tables are used. In some implementations the improvement is always taken for real even if the re-search does not verify it. Especially, at the root of the game tree the improving move may be considered best if the search is stopped before the re-search has verified it. In the FHR-NS the improvement may have been obtained by a very shallow search and therefore should not be taken for real until the re-search also indicates an improvement.

7. Every state-of-the-art chess program uses transposition tables to speed up the search. An entry of the table may have the form (*lock*(*v*), *val*, *flag*, *d*) indicating that (*val*, *flag*) was the result of a search below *v* with search

depth d (see, e.g., Marsland and Campbell, 1982). The FHR-NS algorithm may recognize that the search depth has been reduced at v before writing the result for v into the table but in general does not know, how many depths reductions have been done below v. In our implementation transposition-table entries for a node v get the search depth d associated with v, and therefore recognize only the depth reduction at v itself. We did not observe any problems with this implementation, but an implementation as proposed by Breuker, Uiterwijk, and Van den Herik (1994) will avoid any trouble.

8. There is a small time overhead of about 3% caused by calling the evaluation function for inner nodes as well as for leaves.

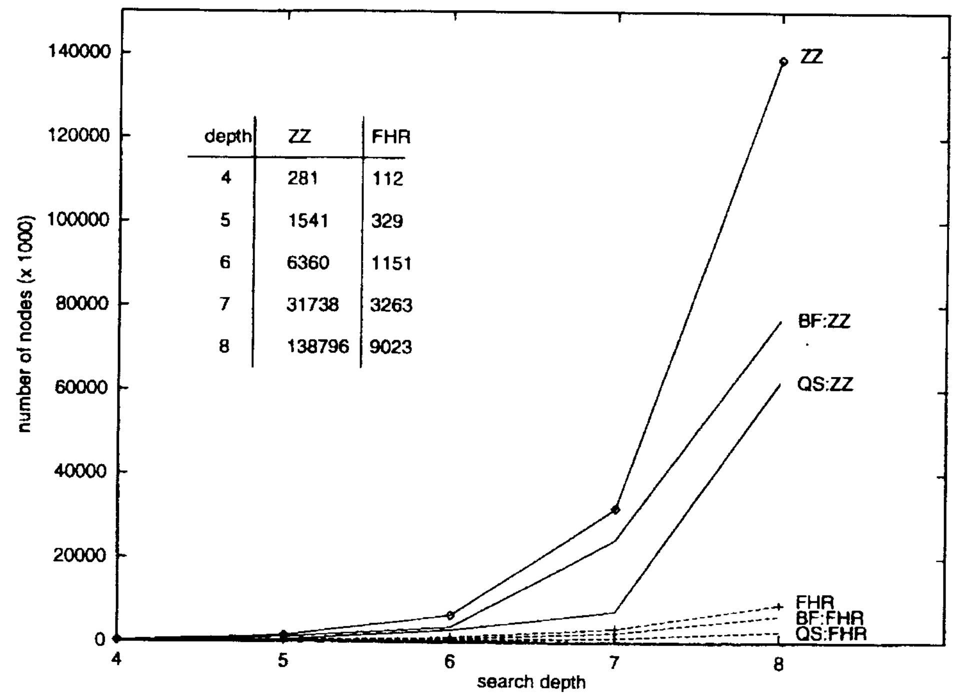
3. EXPERIMENTAL RESULTS

From now on we will refer to two versions of ZUGZWANG: one with FHR (denoted by ZZ_{FHR} or in some tables and figures simply by FHR), and the other one without FHR (denoted by ZZ). ZUGZWANG uses a Negascout algorithm with some chess-specific extensions. It does neither use null-move search nor singular extensions.

ZUGZWANG is a program which changes almost every day. The experiments described below have been made in a range of nearly half a year. In that time we used different versions of ZUGZWANG for the tests. The differences were due to our impatience to fix some bugs and to improve the evaluation function. The search algorithm of all versions has always been the same. After a change had been made we repeated some but not all tests done with previous versions. We note that the results were always very similar. Hence, we believe that our results are still a reliable base for our conclusions.

3.1 Searches to Fixed Depths

The 24 positions of the Bratko-Kopec test set (Kopec and Bratko, 1982) are searched to fixed search depths (4 to 8 ply) using ZZ_{FHR} and ZZ. We counted the number of nodes (total, brute-force tree, and quiescence search) for both versions. For each case the sum of nodes for all 24 positions divided by 1000 is shown in Figure 3. The graph contains six lines (2 versions, 3 numbers per depth). The total number of nodes for the search depths 4 to 8 is given in the table within Figure 3. Because of the search-depth reductions the growth of the trees of the FHR version is much smaller than the growth of the regular trees.



depth	ZZ	FHR
4	281	112
5	1541	329
6	6360	1151
7	31738	3263
8	138796	9023

Figure 3: Nodes searched by ZZ_{FHR} and ZZ for fixed search depths.

Observation 1: If used in an iterative-deepening process, ZZ_{FHR} searches to larger search depths. This is very helpful for the timing heuristic, since there are more check points between two successive iterations when the search can be stopped regularly.

3.2 Running a Suite of Test Positions

In the following we show how FHR performs on the 300 WinAtChess test positions (Reinfeld, 1958).

Timing

In experiments running a set of test positions the timing is either by stopping the program after a fixed amount of time, or by stopping it after a fixed amount of nodes has been searched. The first alternative is bad, if the experiments are run on a multi-user machine (as we do). If many users use the same CPU, stopping by real time gives wrong results. Stopping by CPU cycles gives also erroneous results due to swapping etc. Both alternatives (i.e., time and nodes) do not reflect a performance as occurring in tournament games. In tournaments, a certain amount of time is given and the program typically tries to stop the search just after finishing an iteration of the iterative-deepening process which is close to this deadline. If there are still a few moves to go to the next time control the program may be allowed to run even longer in order to finish the current iteration. We tried to overcome the drawbacks of both traditional alternatives by the following hybrid method.

Let one time unit be equivalent to 2700 nodes searched by ZUGZWANG. This is the number of nodes a very early version of ZUGZWANG searched per second running on a Sparc Classic workstation. We run the program, version ZUGZWANG V 1.20, using the following timing algorithm. The timing is done as if the test position would have been reached at move 20 and ZUGZWANG had 2^i, $i \in \{1, ..., 10\}$ time units average time for the following 20 moves. The timing algorithm tries to stop the search between two successive iterations of the iterative deepening, but sometimes cancels the search if an iteration takes too long.

Results

Table 1 shows the number of positions solved by ZUGZWANG with and without FHR.

	$2^i \times 2700$ nodes									
Version	1	2	3	4	5	6	7	8	9	10
ZZ	189	212	229	241	252	261	268	271	284	288
ZZ_{FHR}	184	184	232	246	258	268	275	282	286	289
ZZ_{FHRX}	190	221	238	249	263	269	276	283	287	289

Table 1: Number of solutions by ZUGZWANG V 1.20 at the WinAtChess set.

ZZ_{FHR} produces more solutions than ZZ if the time is larger than 2^3, i.e., the program searches more than about 21600 nodes per position on the average. If we allow improvements at the root of the tree to be accepted before the re-search confirms the improvement (see item 6 in Section 2) the sequence for ZZ_{FHR} starts with 190, 221, 238, ... as indicated on the line marked ZZ_{FHRX}. ZZ_{FHRX} seems to be slightly better than ZZ_{FHR} but the improvement, if any, is only a minor one and may be due to the class of positions in the test set. Because of the problem mentioned under item 6 ZZ_{FHRX} is not used in games.

It turns out that FHR needs fewer nodes than ZZ to produce more solutions. For instance, ZZ_{FHRX} needs about 163 million nodes to produce 283 solutions, while ZZ searches nearly 206 million nodes for 271 solutions (2^8 time units). While ZZ needs $206 \times 10^6/300 = 2682 \times 2^8$ nodes per problem on the average, ZZ_{FHRX} searches only $163 \times 10^6/300 = 2122 \times 2^8$ nodes per problem. (Note that 2700 nodes correspond to one time unit and therefore the average running time of ZZ is close to the given time limit.) This is due to the effect described in observation 1: FHR grows much smaller trees and thus gives the timing heuristic more possibilities to stop the search between two successive iterations of the iterative-deepening process.

For 2^{10} time units the numbers of solutions nearly coincide. This is due to the fact that for ZUGZWANG V 1.20 ten positions remain unsolved even with much larger computation times.

Observation 2: In the sensitive range from 2^6 *to* 2^8 time units FHR corresponds to a speed-up of 2 to 3 compared to the version without FHR.

3.3 ZUGZWANG in a Self Test

Three versions of ZUGZWANG with FHR played 50 games against ZUGZWANG without FHR. We selected 25 opening positions from a chess opening database each two ply away from the initial position. Starting from these positions (indicated in Table 2) both versions played white and black without opening book. The timing was set to 40 moves in 2 hours and 20 moves in each additional hour. The games were stopped after 100 moves and then adjudicated. The three ZUGZWANG versions were V 1.24, V 1.30, V 1.40. The differences between these three versions are due to some bug fixing (especially in the detection of threefold repetition of positions) as well as to some minor changes to the static evaluation function.

	V 1.24		V 1.30		V 1.40	
Position	FHR = *b*	FHR = *w*	FHR = *b*	FHR = *w*	FHR = *b*	FHR = *w*
e4 e5	1/2	1/2	1-0	1-0	1-0	0-1
e4 e6	0-1	1-0	0-1	0-1	1/2	1/2
e4 c5	1-0	1-0	0-1	1-0	0-1	1-0
e4 c6	1/2	1-0	1-0	1-0	0-1	1-0
e4 d6	1/2	1-0	0-1	0-1	1-0	1-0
e4 g6	0-1	1-0	1-0	1/2	1-0	1-0
e4 Nf6	1/2	1-0	0-1	1-0	1-0	1-0
e4 d5	1/2	1/2	1-0	1-0	1-0	1-0
e4 Nc6	1/2	1/2	1-0	0-1	1-0	0-1
d4 Nf6	1/2	1/2	0-1	1-0	1/2	1-0
d4 d5	1/2	0-1	0-1	1-0	1-0	0-1
d4 f5	0-1	0-1	1/2	1-0	0-1	1-0
c4 c5	0-1	0-1	0-1	0-1	0-1	1-0
c4 e5	1/2	1-0	0-1	1-0	0-1	0-1
c4 c6	1/2	1-0	1-0	1-0	1-0	1-0
c4 f5	1-0	1-0	0-1	1-0	0-1	1-0
c4 e6	1/2	1/2	1/2	1-0	0-1	1-0
c4 Nf6	0-1	1-0	0-1	0-1	0-1	0-1
Nf3 d5	0-1	1-0	0-1	1-0	0-1	1-0
Nf3 Nf6	0-1	1-0	1-0	1-0	1-0	1-0
Nf3 f5	0-1	1/2	0-1	1-0	1-0	1/2
f4 e5	1/2	1-0	0-1	0-1	0-1	1-0
f4 d5	0-1	0-1	0-1	1/2	0-1	0-1
b4 Nf6	0-1	1-0	0-1	1-0	1/2	1-0
b4 e5	1/2	1/2	1/2	0-1	0-1	1/2
Σ	8.5-16.5	17.5-7.5	8.5-16.5	17-8	11.5-13.5	17.5-7.5
	34-16		33.5-16.5		31-19	

Table 2: Results of the games between ZZ_{FHR} and ZZ.

Observation 3: ZZ_{FHR} V 1.24 won the match by 34-16, i.e., it scored 68%. With this result ZZ_{FHR} V 1.24 is better than ZZ V 1.24 at a 5% significance level (Mysliwietz, 1994, pp. 86ff). Almost the same is true for V 1.30. V 1.40 produced a slightly weaker result.

Because of the overall win of 98.5-51.5, i.e., 65.67% for the versions with FHR, the playing strength is estimated to be about 130 ELO points better than the playing strength of the regular versions. This is equivalent to a speed-up of about 2 to 3 (Mysliwietz, 1994) and therefore corresponds to the observations given in Subsection 3.2.

3.4 ZUGZWANG vs. CHEIRON

Table 3 shows the results of 100 games ZUGZWANG V 1.50 played against CHEIRON, a program by Ulf Lorenz. CHEIRON successfully participated in the 1995 World Computer-Chess Championship in Hong Kong and in the 1995 World Microcomputer Chess Championship in Paderborn. ZZ denotes the version without FHR, FHR the one with FHR. The second column contains the result of CHEIRON playing with White against ZUGZWANG without FHR (13-12), the third column the results of CHEIRON playing Black against ZUGZWANG without FHR (8-17), etc. The last line shows the overall result of the two 50-game matches.

	C - ZZ	ZZ - C	C-FHR	FHR-C
Σ	13-12	8-17	13-12	14.5-10.5
Total	C - ZZ: 30-20		C - FHR: 23.5-26.5	

Table 3: Results of the games against CHEIRON.

The games were started on the positions given in Subsection 3.3. No opening books were used. ZUGZWANG had access to Ken Thompson's endgame databases. CHEIRON had not. The timing rules were 40 moves in 2 hours, followed by 20 moves in one hour. Then, CHEIRON continued to play at a rate of 20 moves per hour while ZUGZWANG played the rest of the game in 30 minutes. Note that for this kind of test it is not necessary to have the same timing conditions for the two opponents since we only want to compare ZUGZWANG with a fixed opponent.

Observation 4: ZZ scored 40% against CHEIRON, whereas ZZ_{FHR} won the match by 26.5 to 23.5, i.e., it scored 53%. This allows for an estimation of ZZ_{FHR} being about 130 ELO points stronger than ZZ.

4. CONCLUSIONS AND OPEN QUESTIONS

In this paper we presented a domain-independent method, called FHR, to search selectively game trees. FHR is based on the null-move observation but is less strict than the null-move search and very easy to implement.

In several tests FHR turned out to be superior to the regular Negascout search. For one test, a 150-game match between ZUGZWANG with FHR and without FHR, ZZ_{FHR} turned out to be superior to the regular version with a statistical significance.

A comparison with null-move search as proposed by Donninger (1993) would be interesting. In our first experiments it turned out that FHR was slightly superior to the null-move search. This, however, may also be due to our implementation of null-move searches. Incorporating FHR into a program using a strong null-move search may give further insight.

It remains an open question whether the playing strength of FHR can be further increased, if the implementation of the transposition table takes care about the depth reductions done in the subtrees. An implementation as proposed by Breuker *et al.* (1994) may help.

Another serious question is why our parallel implementation of FHR-NS is not as efficient as the one for Negascout. The reason may be that our distributed algorithm is most efficient in the appropriate subtrees of the game tree, but that these subtrees are considerably smaller as compared to the leftmost subtree when the tree is searched with FHR-NS.

Last but not least it would be interesting to compare the error propagation of the standard $\alpha\beta$ algorithm with the error propagation of the FHR algorithm in a model of game trees, which is not too far away from the trees occurring in practical applications.

5. ACKNOWLEDGEMENTS

ZUGZWANG is programmed by Peter Mysliwietz and myself. I thank Peter for many helpful discussions on the FHR topic and for his work on the threat evaluation function t which definitely improved ZUGZWANG's play even without FHR. Moreover, I would like to thank Burkhard Monien for his long lasting support and Ulf Lorenz for letting his program CHEIRON play 100 games against ZUGZWANG. Finally, I would like to acknowledge the helpful comments of the referees of ACC8.

6. REFERENCES

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1991). αβ Conspiracy-Number Search. *Advances in Computer Chess 6* (ed. D.F. Beal), pp. 73-95. Ellis Horwood Ltd., Chichester, UK. ISBN 0-13-006537-4.

Althöfer, I. (1991). Selective Trees and Majority Systems: Two Experiments with Commercial Chess Computers. *Advances in Computer Chess 6* (ed. D.F. Beal), pp. 37-59. Ellis Horwood Ltd., Chichester, UK. ISBN 0-13-006537-4.

Anantharaman, T.S., Campbell, M., and Hsu, F-h. (1988). Singular Extensions: Adding Selectivity to Brute Force Searching. *AAAI Spring Symposium, Computer Game Playing*, pp. 8-13. Also published in *ICCA Journal*, Vol. 11, No. 4, pp. 135-143. ISSN 0920-234X. Republished (1990) in *Artificial Intelligence*, Vol. 43, No. 1, pp. 99-109. ISSN 0004-3702.

Anantharaman, T.S. (1991). Extension Heuristics. *ICCA Journal*, Vol. 14, No. 2, pp. 47-65. ISSN 0920-234X.

Beal, D.F. (1989). Experiments with the Null Move. *Advances in Computer Chess 5* (ed. D.F. Beal), pp. 65-79. Elsevier Science Publishers, Amsterdam, The Netherlands. ISBN 0-444-87159-4. A revised version is published (1990) under the title A Generalized Quiescence Search Algorithm, *Artificial Intelligence*, Vol. 43, No. 1, pp. 85-98. ISSN 0004-3702.

Beal, D.F. and Smith, M.C. (1995). Quantification of Search-Extension Benefits. *ICCA Journal*, Vol. 18, No. 4, pp. 205-218. ISSN 0920-234X.

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.

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1994). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 4, pp. 183-193. ISSN 0920-234X.

Buro, M. (1995). ProbCut: An Effective Selective Extension of the α–β Algorithm. *ICCA Journal*, Vol. 18, No. 2, pp. 71-76. ISSN 0920-234X.

Campbell, M.S. and Marsland, T.A. (1983). A Comparison of Minimax Tree Search Algorithms. *Artificial Intelligence*, Vol. 20, No. 4, pp. 347-367. ISSN 0004-3702.

Donninger, Chr. (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.

Feldmann, R., Monien, B., and Mysliwietz, P. (1991). A Fully Distributed Chess Program. *Advances in Computer Chess 6* (ed. D.F. Beal), pp. 1-27. Ellis Horwood Ltd., Chichester, UK. ISBN 0-13-006537-4.

Feldmann, R., Mysliwietz, P., and Monien, B. (1992). Experiments with a Fully-Distributed Chess Program. *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad* (eds. H.J. van den Herik and L.V. Allis), pp. 72-87. Ellis Horwood, Chichester, UK. ISBN 0-13-388265-9.

Feldmann, R. (1993). *Spielbaumsuche mit massiv parallelen Systemen* (Engl. Game Tree Search on Massively Parallel Systems). Ph.D. thesis, Universität Gesamthochschule Paderborn, Paderborn, Germany.

Feldmann, R. and Mysliwietz, P. (1994). Studying Overheads in Massively Parallel MIN/MAX-Tree Evaluation. *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 94-103. ACM Press, New York, NY.

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293-326. ISSN 0004-3702.

Kopec, D. and Bratko, I. (1982). The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 57-72. Pergamon Press, Oxford, UK. ISBN 0-08-026898-6.

Lorenz, U., Rottmann, V., Feldmann, R., and Mysliwietz, P. (1995). Controlled Conspiracy-Number Search. *ICCA Journal*, Vol. 18, No. 3, pp. 135-147. ISSN 0920-234X.

Marsland, T.A. and Campbell, M.S. (1982). Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys*, Vol. 14, No. 4, pp. 533-551. ISSN 0360-0300.

McAllester, D.A. (1988). Conspiracy Numbers for Min-Max Searching. *Artificial Intelligence*, Vol. 35, No. 1, pp. 287-310. ISSN 0004-3702.

Meulen, M. van der (1990). Conspiracy-Number Search. *ICCA Journal*, Vol. 13, No. 1, pp. 3-14. ISSN 0920-234X.

Mysliwietz, P. (1994). *Konstruktion und Optimierung von Bewertungsfunktionen beim Schach.* Ph.D. thesis, Universität Gesamthochschule Paderborn, Paderborn, Germany.

Palay, A.J. (1985). *Searching with Probabilities.* Pitman Publishing Inc., Marshfield, MA. Originally (1983) published as Ph.D. thesis, Carnegie-Mellon University.

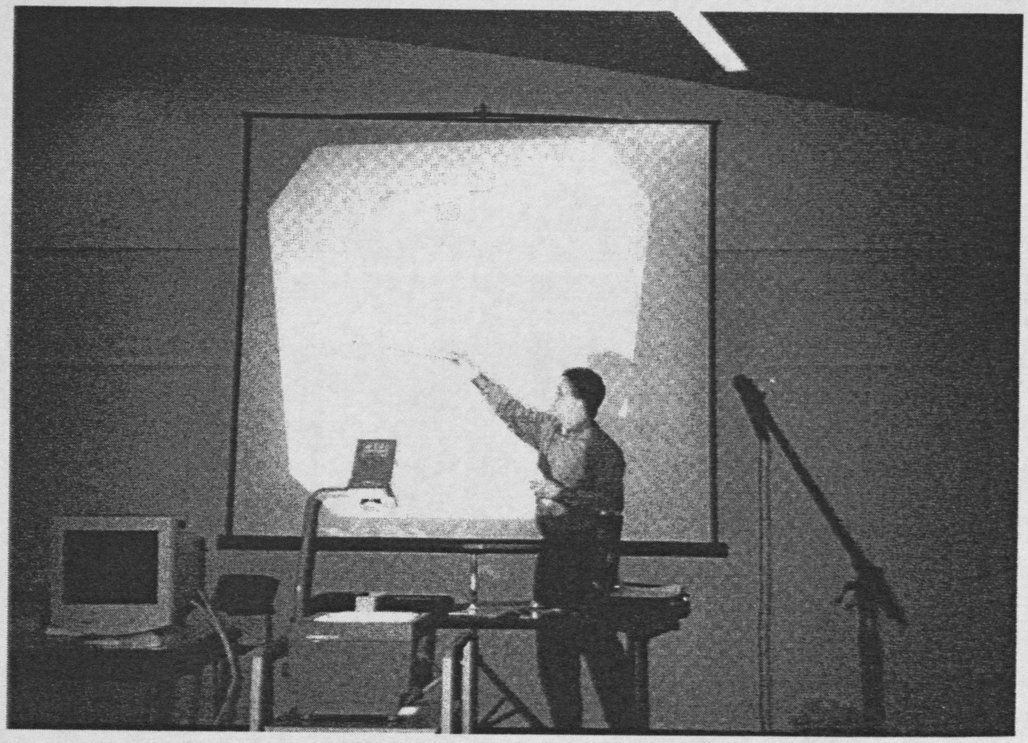
Reinefeld, A. (1983). An Improvement to the Scout Tree Search Algorithm. *ICCA Journal,* Vol. 16, No. 4, pp. 4-14. ISSN 0920-234X.

Reinefeld, A. (1989). Spielbaum Suchverfahren. *Volume Informatik-Fachberichte* 200. Springer-Verlag, Berlin, Germany.

Reinfeld, F. (1958). *Win at Chess.* Dover Publications, Inc., New York, NY. ISBN 0-486-20438-3. Originally published (1945) as *Chess Quiz* by David McKay Company, New York, NY.

Schaeffer, J. (1989). Conspiracy Numbers. *Advances in Computer Chess 5* (ed. D.F. Beal), pp. 199-217. Elsevier Science Publishers, Amsterdam, The Netherlands. ISBN 0-444-87159-4. Also published (1990) in *Artificial Intelligence,* Vol. 43, No. 1, pp. 67-84. ISSN 0004-3702.

Schrüfer, G. (1989). A Strategic Quiescence Search. *ICCA Journal,* Vol. 12, No. 1, pp. 3-9. ISSN 0920-234X.


Ulf Lorenz searching for parallel-controlled conspiracy numbers.

Happy Organizing.