Mate in 38:
applying proof-number search to chess

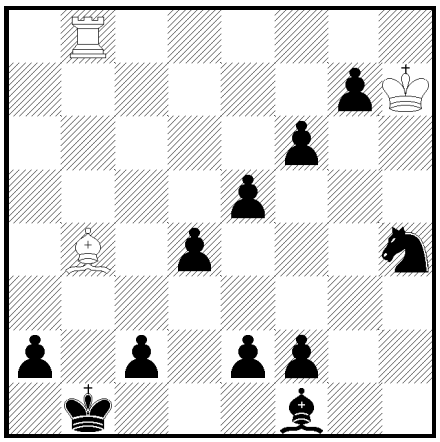
D.M. Breuker
L.V. Allis
H.J. van den Herik

ABSTRACT

Proof-number search (pn-search) has shown its merit in contributing to the solution of Connect-Four, Qubic and Go-Moku.

In this contribution we show that pn-search is a highly capable searcher for mates in chess. Pn-search achieves its results *without* using any heuristic chess knowledge. We present the results of comparing the performances of pn-search and a number of other mate searchers on a large number of problems.

Finally, we discuss how pn-search can be applied to general tactical searches.



Mate in 38, L.Ugren, 1967

1 Background

A major attraction of chess is the number of different skills necessary to play all stages of the game at a high level.

To play the *opening* well, one needs to select an opening repertoire, study the lines in it, understand the positions resulting from it, and constantly keep up to date with enhancements introduced by top-level players. In the *middle game*, a deep understanding of the strategical features of positions, as well as recognition and thorough calculation of the combinatorial opportunities are necessary. The *end-game* again has its own rules. Some end-games are fully analyzed and only require some experience. For others deep analysis, either performed at home or during a game, constantly unveils chess secrets to the chess-playing community.

In short, different parts of the game require different skills. This is partly reflected in the way chess programs are composed. To play the opening, large opening books are installed, and constantly updated with the newest inventions of grandmaster play. For the end-game, databases,

such as Thompson (1991a, Thompson (1991b, Thompson (1991c), are installed. Most chess programs play the middle game using a single analyzing algorithm: α-β search (Knuth and Moore, 1975), although enhanced in several ways. This is somewhat surprisingly, since the middle game requires calculation of tactical variations, as well as assessment of strategical positions.

In this contribution, we propose to use a specialized algorithm (proof-number search, Allis *et al.* (1994b)) to perform the tactical calculations.

First, we show that our pn-search implementation, PROVER, without using heuristic chess knowledge and without transposition tables, is a better mate searcher than alternatives presented in the literature (which do use heuristic evaluation functions and transposition tables). Second, we describe how PROVER may be extended to perform a tactical analysis at each move of a game. Anticipating our conclusions, we believe that a tactical searcher based on pn-search may be an enhancement to most tournament chess programs.

Before continuing, it is important to stress one point. Proof-number search does not favour a shortest mate over longer mates. It will report the first mate found, without checking for the existence of shorter mates. In general the most forcing move sequence will be found, in which the opponent has the least number of choices. Therefore, in a large number of problems presented in this article, PROVER found a different mating sequence than the shortest solution. A discussion on the implications of this property of pn-search can be found in section 5.2.

The course of this article is as follows. In section 2, a short description of pn-search is presented. The architecture of PROVER (pn-search) and DUCK (α-β search) is described in section 3. In section 4 the performance of PROVER and DUCK on a large set of mating problems is compared. Also presented in that section is a comparison of the performance of PROVER with mate searchers from literature. The strengths and weaknesses of PROVER are discussed in section 5. Section 6 presents ideas for future enhancements to PROVER, which should result in a general tactical searcher for tournament chess programs. Conclusions are presented in section 7.

2 Proof-number search

In this section we present a short overview of proof-number search (pn-search). A detailed description of pn-search can be found in Allis *et al.* (1994b) and Allis (1994a).

Proof-number search is a best-first AND/OR-tree search algorithm (Allis, 1994a). It assumes that evaluation of a node returns one of three values: *true*, *false*, or *unknown*. Throughout this contribution we assume that true indicates that the player to move in the root position has achieved his goal, while false indicates that his goal is unreachable. A tree is *solved* as soon as the value of the root has been established. A tree is *proven* if its value has been established to be true, while the tree is *disproven* if its value is determined to be false. Any node evaluating to unknown is an internal node of the search tree, while nodes evaluating to win or loss are leaf nodes. *Temporary terminal nodes* are either unevaluated nodes or unexpanded internal nodes. In our description of pn-search we will assume that there are no unevaluated temporary terminal nodes. Thus, after expanding a node, the newly created children are all instantly evaluated.

Like other best-first search algorithms, pn-search repeatedly selects a temporary terminal node, expands it, evaluates all the children, and updates the tree with the information obtained from the expansion and evaluations. Unlike most other best-first search algorithms, pn-search does not use a heuristic evaluation function to determine a most-promising node. Instead, the shape of the search tree, and the values of the terminal nodes determine which node to select next.

Generally, to prove a tree, a number of temporary terminal nodes of the current search tree need to be proven. A set of temporary terminal nodes, which, if all proven, would prove the tree, is called a *proving set*. Likewise, a set of temporary terminal nodes, which, if all disproven, would disprove the tree, is called a *disproving set*. The size of the smallest proving set of the tree is a lower bound for the number of node expansions necessary to prove the tree, while the size of the smallest disproving set of the tree is a lower bound for the number of node expansions necessary to disprove the tree.

In figure 1 an AND/OR-tree has been depicted. Associated with each node is the proof and

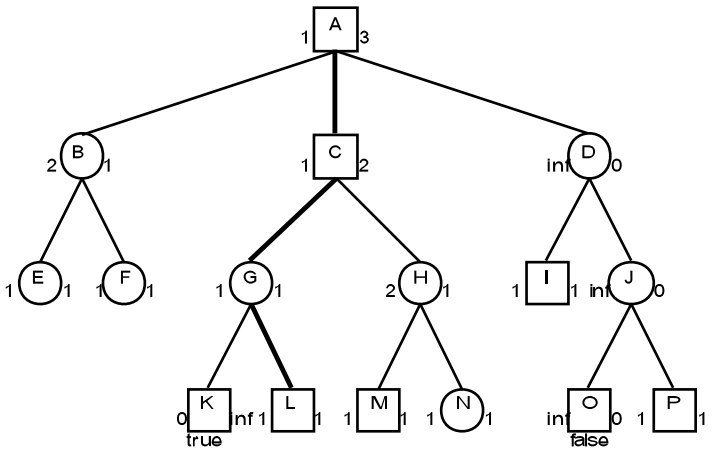


Figure 1: AND/OR-tree with proof and disproof numbers.

disproof number (in that order). Proven nodes (e.g., node K) have proof number 0 and disproof number ∞. This follows from the fact that no expansions are needed to prove the node, as it is already proven, and that no finite number of expansions could disprove the node. Analogously, disproven nodes (e.g., node O) have proof number ∞ and disproof number 0. Temporary terminal nodes have proof and disproof number one, as expanding the node itself may be sufficient to solve the node. Internal AND-nodes (depicted by circles) have as proof number the sum of the proof numbers of their children, since to prove an AND-node, all children must be proven. The disproof number of an AND-node equals the minimum of its childrens' disproof numbers, since only one child needs to be disproven to disprove the AND-node. Analogously, the proof number of an internal OR-node (depicted by a square) equals the minimum of the proof numbers of its children, while its disproof number equals the sum of the disproof numbers of its children.

The main assumption underlying pn-search is that it is generally better to first expand nodes part of smallest proof and/or disproof sets, than selecting nodes which are only part of larger proof and disproof sets. In other words, pn-search concentrates at each step on the potentially least amount of work necessary to solve the tree.

The only remaining question reads, when to select a node from the smallest proof set of the root, and when to select a node from the smallest disproof set. Surprisingly, we can always do both at the same time. In Allis (1994a) it is proven that the intersection of any smallest proof set and any smallest disproof set of the same node, is always non-empty. The nodes which are element of both a smallest proof set and a smallest disproof set of the root are called *most-proving nodes*. Thus, if after expansion of most-proving node J, it obtains value true, the proof number of the root decreases by one, while if J obtains value false, the disproof number of the root decreases by one. If the value of J remains unchanged, the newly created children may have their impact on the proof and/or disproof number of J and its ancestors.

A most-proving node is selected in the tree, by selecting at AND-nodes a child with equal disproof number, and at OR-nodes a child with equal proof number. By traversing from root to a temporary terminal node, a most-proving node is determined.

The full proof-number search algorithm consists of repeatedly selecting a most-proving node J, expanding J, and evaluating all of J's children. Then, traversing the tree backwards, from J to the root, the proof and disproof numbers are adjusted to the new situation. Detailed algorithms, and enhancements to the standard version of pn-search can be found in Allis *et al.* (1994b) and Allis (1994a).

3 prover and duck

In this section we describe two mate-searching programs. PROVER is an implementation of pn-search, while DUCK is a conventional chess-program, created by the first author, based on α-β search.

3.1 prover

PROVER is a pn-search implementation for chess, using the chess-specific routines of DUCK. PROVER currently has as only goal searching for mate. In our description we distinguish between the *attacker* and the *defender*. A position is proven if the defender can be mated, while stalemate, repetition of positions and the 50-move rule are defined as disproven positions. PROVER consists of pn-search, as described in section 2, enhanced with several techniques described in Allis (1994a) to reduce time and memory requirements.

The most important enhancement, also described in Allis (1994a), is the initialization of proof and disproof number at temporary terminal nodes. In the standard algorithm, proof and disproof number are initialized at one each. Let us suppose that after expansion, all the n children evaluate to value unknown. In that case the proof and disproof of the most-proving node are set to 1 and n, for an OR-node, and n and 1 for an AND-node. To distinguish between temporary terminal nodes, before expansion, we set the proof and disproof number of node J to 1 and n (or n and 1, depending on the node type), where n is the number of legal moves in the position represented by J. Experiments presented in Allis (1994a) show that the extra overhead induced by counting the number of legal moves at each node, is more than compensated for by the value of the extra information revealed to the node-selection process.

Worth mentioning is that PROVER does not use any chess-specific heuristic knowledge, such as DUCK's heuristic evaluation function and move sorting algorithm. Thus, moves are not ordered by the move-generator in any way to ensure that the best-looking move is moved forward in the list of moves.

3.2 duck

DUCK is a conventional chess program, created by the first author. It is based on a variation of α-β search, called *Principle Variation Search* (PVS) (Marsland, 1986). Several extension mechanisms are used to postpone the horizon effect (Berliner, 1974). These extensions include singular extensions (Anantharaman *et al.*, 1989), check evasion extensions, and recapture extensions. The search is performed using iterative deepening (Gillogly, 1972). To sort the generated moves the history heuristic is used (Schaeffer, 1983). DUCK uses a transposition table with 1,000,000 entries. Its evaluation function consists of a material part and a positional part. Furthermore, DUCK recognizes mate, and draws by stalemate, repetition of positions and the 50-move rule.

In tournament mode, DUCK, when encountering mate, does not stop searching and tries to find the shortest mate possible. However, when applying the program to the tests described in this article, the search was terminated as soon as a mate was found.

4 Results

To compare the performance of PROVER with those of other mate searchers, we have selected a large number of mate problems. Unfortunately, there is no standard set of mating problems in use for these comparisons. Therefore, we have split our comparisons into two parts. First, we present a comparison between PROVER and DUCK on a large set of mating problems, from various sources (section 4.1). Second, we present a comparison between PROVER and results of other mate searchers reported in literature (section 4.2).

4.1 Comparing prover with duck

For our experiments we have selected mating problems from three sources.

1. *Win at Chess* (Reinfeld, 1958).
   The book contains 300 tactical problems, of which approximately 25% lead to mate. Some of the problems which lead to mate are not identified as such, while in other cases a suggested mate can be staved off by sacrificing material. Therefore, each of the 300 problems has been checked by both PROVER and DUCK. Only problems for which a mating sequence was found by at least one of the programs were included.

2. *Chess Curiosities* (Krabbé, 1985).
   The book contains many curious, funny and remarkable compositions and examples from tournament play. All problems which were listed as mate in 5 or more were selected.

3. *Schachkombinationen* (Colditz, 1983).
   The book contains examples of tactical combinations, with explanations of the patterns involved. All problems which were listed as mate in 5 or more were selected.

For the positions of all three sources, we have performed experiments in which both programs had to solve the positions within 1,000,000 nodes. This limit was selected for two reasons.

- The calculation time (up to 5 minutes), corresponds roughly to tournament conditions.
- The search tree for pn-search must be kept in memory during the calculations. A tree of 1,000,000 nodes is close to the maximum achievable on the hardware used.

4.1.1 Win at Chess

We have applied both PROVER and DUCK to all 300 positions of *Win at Chess*. Both algorithms were given a limit of 1,000,000 nodes per position.

In total, 66 positions were solved by both algorithms, 13 solved only by PROVER and 1 only by DUCK. The problem numbers in *Win at Chess* solved by the algorithms are listed below.

- *Both algorithms.*
  1, 4, 5, 9, 12, 14, 27, 35, 49, 50, 54, 55, 57, 60, 61, 64, 79, 84, 88, 97, 99, 102, 103, 104, 132, 134, 136, 139, 143, 154, 156, 158, 160, 161, 167, 172, 173, 177, 179, 184, 186, 188, 191, 197, 203, 211, 212, 215, 217, 219, 225, 244, 246, 251, 253, 260, 263, 266, 267, 278, 282, 283, 285, 290, 295, 298

- PROVER
  6, 51, 138, 159, 168, 182, 218, 222, 241, 250, 252, 281, 293

- DUCK
  105

In table 1 we present the results of PROVER and DUCK on the problems from *Win at Chess*. In the first line of the table, we have listed the total number of nodes searched on the 66 positions solved by both algorithms. The second line contains the average per position. The third line lists the number of times one of the programs performed better on a position than the other program. For the fourth and fifth line of the table, a position is selected where the factor difference in nodes visited between the algorithms was a high as possible. The fourth line shows PROVER's best performance as compared to DUCK's performance on the same position. The fifth line shows DUCK's best result as compared to PROVER. The sixth line shows the average number of nodes searched by PROVER on the positions not solved within a million nodes by DUCK. The last line shows number of nodes searched by DUCK on position 105, the only position not solved by PROVER.

	PROVER	DUCK
total nodes	696,109	4,461,174
average nodes	10,547	67,593
best algorithm	55	11
PROVER's best	271	331,404
DUCK's best	2,355	311
PROVER solved	305,912	>1,000,000
DUCK solved	>1,000,000	777,182

Table 1: Comparison on *Win at Chess*.

	PROVER	DUCK
total nodes	257,653	736,900
average nodes	36,807	105,271
best algorithm	6	1
PROVER's best	296	244,122
DUCK's best	173,480	158,523
PROVER solved	180,305	>1,000,000
DUCK solved	>1,000,000	930,899

Table 2: Comparison on *Chess Curiosities*.

4.1.2 Chess Curiosities

The book CHESS CURIOSITIES by Tim Krabbé is devoted to oddities in chess. These vary from the latest castling move performed in a tournament game (at move 46), to mating problems with curious themes. We have selected from the book all problems listed as mate in 5 or more moves.

The positions all correspond to diagrams in the book. Below we have listed for each problem the algorithms by which it could be solved within 1,000,000 nodes.

- *Both algorithms*
  35, 38, 197, 211, 212, 261, 317
- PROVER
  61, 192, 194, 196, 198, 199, 206, 207, 208, 214, 215, 216, 218, 219, 333, 334
- DUCK
  60
- *Neither algorithm*
  40, 44, 209, 210, 217, 220

In table 2 we have listed the number of nodes visited by both algorithms on the positions of *Chess Curiosities*. The table is built analogous to table 1.

Although at first sight, PROVER has a smaller advantage over DUCK than it had on the positions of *Win at Chess*, this is not true. PROVER solved 16 positions which were not solved by DUCK, while DUCK solved only one position not solved by PROVER. Some of the problems in *Chess Curiosities* will serve as examples in describing the strengths and weaknesses of PROVER in section 5.

	PROVER	DUCK
total nodes	29,127	1,148,867
average nodes	3,640	143,608
best algorithm	8	0
PROVER's best	490	641,571
DUCK's best	25,461	30,819
PROVER solved	53,106	>1,000,000

Table 3: Comparison on *Schachkombinationen*.

4.1.3 Schachkombinationen

Schachkombinationen contains 120 problems to test your tactical abilities. Among these, 9 mating problems with a distance to mate of 5 or more are listed. We have applied PROVER and DUCK to these 9 problems. All problems were solved by PROVER, while DUCK missed only one of the mates.

- *Both algorithms*
  10, 42, 46, 55, 74, 86, 92, 99
- PROVER
  119

As is shown in table 3, PROVER outperforms DUCK on these problems by a wide margin. The table is built analogous to tables 1 and 2.

In average number of nodes, PROVER outperforms DUCK by a factor of almost 40. On the positions where DUCK loses the least, PROVER gains only about 20%, while on PROVER's best position, it gains a factor of over 1300.

4.2 Comparing prover with the literature

In the literature we have found many articles dealing with tactical analyzers. In most cases, the 300 positions from *Win at Chess* are used as test problems. For the subdomain of mate searchers, fewer articles are available. In most of these, only one or a few positions are shown on which the program under discussion performed well. Only a small number of articles has systematically compared performances of various mate searchers on a large number of positions.

In this section we compare the performance of PROVER with the results reported in the literature. First, we present a comparison with separate sources in the literature. Second, an overview of the individual results is given.

4.2.1 Paradise

In Wilkins (1980), the knowledge-based, best-first tactical searcher PARADISE is described. As an example of PARADISE's achievements a mate in 10 is presented, which PARADISE solves in 109 nodes, in 20 minutes CPU time on a DEC KL-10. One of the reasons the position is difficult to solve for full-width searchers, is the *any* as last defensive move, before black is mated. PROVER finds a winning line of 18 moves in 24,738 nodes, or 7.5 CPU seconds on an IBM RS-6000. Although so vastly different machines are difficult to compare, we may assume that the IBM is roughly 20-30 times faster than the DEC. This means that PROVER's solution is 5-8 times faster than PARADISE's.

4.2.2 Computerschaak

In the Dutch computer-chess magazine *Computerschaak*, three commercial chess-computers have been compared on 10 mating problems (Van Gisteren, 1992). The computers involved were the

	PROVER	SUPER 9	FINAL CC	MM5
1	0.04	16	70	187
2	0.03	5	1	5
3	0.04	3	10	1
4	0.08	19	250	210
5	0.81	525	-	4738
6	0.47	31	2	3
7	0.14	119	31	20
8	16.75	294	480	430
9	0.52	26	104	25
10	14.25	93	27	25
total	33.13	1131	-	5644

Table 4: Comparison with results from *Computerschaak*.

Super 9, 3 MHz, the *Final Chess Card*, 5 MHz, and the *Mephisto MM5*, 5 MHz. To convert the solution times in nodes visited, we remark that PROVER searches approximately 4000 nodes/sec.

In table 4 we have listed for the 10 problems the results for PROVER, and the best result achieved by the commercial programs. For problems 1, 4, 5 and 8 the *Super 9* was best, for problems 2 and 6 the *Final Chess Card* was best, while the *MM5* was quickest on problems 3, 7, 9 and 10.

Table 4 shows that PROVER performs better than the commercial programs, on all 10 problems. The gain factor ranges from 1.75 on problem 10, to almost 650 on problem 5. The gain factor on the total over the 10 problems is 34.

4.2.3 Sam Lloyd's problems

In 1985, three articles have appeared in consecutive issues of the *ICCA Journal* (Grottling, 1985; Wiereyn, 1985b; Wiereyn, 1985a). In the first article, a set of 16 mating problems by Sam Lloyd were presented, and the test results of 18 commercial computer programs. In the second and third article, Wiereyn compares the results of his program with those presented in Grottling (1985). In table 5, we compare the best results of all programs applied to the 16 mating problems to the results of PROVER.

Table 5 shows that PROVER is outperformed 6 times out of 16. On problem 14 PROVER loses by a factor of almost 8, while on problem 7 PROVER does not find the solution within 1,000,000 nodes. The total time count (discarding problem 7) is favourable to PROVER by a factor of over 2.

4.2.4 Touch

The chess program TOUCH, developed by Jos Uiterwijk, applies several kinds of selective extensions. In a description of the 11th Dutch computer-chess championship, Uiterwijk cites five examples of mates found by TOUCH (Uiterwijk, 1991). We have compared PROVER's performance with the reported performance of TOUCH in table 6. Each line specifies the number of nodes searched by each program.

PROVER outperformed TOUCH on all but one position. The gain factor of PROVER over TOUCH ranges from 0.9 to approximately 7.

4.2.5 Bcp

In an article on mating sequences in the quiescence search, Beal reports the result on a mate-in-10 position (Beal, 1984). The position has been checked by BCP with three different parameter settings. BCP's best result and Cray Blitz's performance on the same position have been depicted

	PROVER	*ICCA Journal*
1	0.16	0.15
2	0.07	0.25
3	0.06	0.22
4	0.70	4.35
5	5.81	4.43
6	20	1.02
7	-	3.18
8	52	19
9	0.60	12
10	18	86
11	14	19
12	1.72	134
13	0.10	98
14	84	11
15	2.63	147
16	86	95
total	286	631

Table 5: Comparison with results from the *ICCA Journal*.

	PROVER	TOUCH
1	4,320	4,650
2	3,893	26,540
3	11,500	21,828
4	392	366
5	95,070	500,000
total	115,175	553,384

Table 6: Comparison with the results of TOUCH.

PROVER	BCP	*Cray Blitz*
19,939	382,664	5,653,322

Table 7: Comparison with BCP and Cray Blitz.

	PROVER	BCP
4	0.02	2.60
5	0.01	0.06
9	0.07	2.00
12	0.04	0.40
14	95.27	13.00
27	0.01	0.20
35	0.14	7.00
50	0.05	0.60
54	0.03	0.10
55	8.53	10.00

Table 8: Comparison with BCP on *Win at Chess* positions.

in table 7. The figures indicate the number of nodes visited. On this position, PROVER outperforms BCP by a factor 19, while Cray Blitz loses a factor of more than 280.

In the same article, BCP's performance on the first 10 mating positions of *Win at Chess* is reported. We remark that for some reason positions 1, 6, 49 and 51 have been excluded, while in each of these PROVER has found a forced mate. In table 8 we have compared the performance of BCP and PROVER on the 10 positions reported. In the first column, the problem number from *Win at Chess* is specified. In the other columns, figures depict CPU seconds. Except for problem 14, PROVER outperforms BCP.

4.2.6 Miscellaneous

In Hartmann and Kouwenhoven (1991) a mate in 6 is described, which is found by DAPPET in 24 seconds. PROVER finds the same mate in 3.2 seconds.

In Bij de Weg (1989) it is reported that *The King* announced a mate in 9 moves, at move 73 in its game against *October*. We have been informed that the calculations took approximately 10 minutes. PROVER found the same mate in 2.3 seconds.

In Anantharaman *et al.* (1989) it is reported that *Chiptest* found a mate in 18 moves in problem 213 of *Win at Chess*, in 65 seconds. Given *Chiptest*'s speed, this is the equivalent of some tens of millions positions. Due to hardware restrictions, we have only been able to test PROVER on this position up to 8 million nodes, in which the mate has not been found.

4.2.7 Overview

The comparison of PROVER's performance with the examples in the literature presented in this section, creates a steady image. PROVER generally performs better than other mate searchers, but in some example positions, PROVER is outperformed.

In the next section we will analyze the strengths and weaknesses of PROVER illustrated by the more extreme examples we have encountered here.

5 Strengths and weaknesses

5.1 Strengths

In this section we focus on three aspects of PROVER, which are advantageous compared with other mate searchers. First we look at the use of chess knowledge, second at the complexity of the algorithm, and third at the type of positions PROVER is likely to solve much quicker than conventional mate searchers.

5.1.1 Chess knowledge

As stated in section 3.1, PROVER does not use any heuristic chess knowledge. All that is needed is a move generator, and an evaluation function which is able to recognize mate, stalemate and draws by repetition or the 50-move rule.

We would like to stress that quite unlike α-β search, move ordering has almost no influence on the performance of pn-search. This phenomenon can be easily explained from the way pn-search builds its tree. At each step the child is select with smallest proof or disproof number (depending on the node type). Only if two childs tie for first place, we select a node based on the move ordering. Experiments with the standard move-ordering in DUCK showed that it has virtually no influence on the number of nodes grown, while applying the sorting functions significantly slow down the program.

Not using any chess knowledge has as advantage that within each chess program pn-search can be incorporated, achieving the same results as PROVER, regardless of the evaluation function and other heuristics applied.

5.1.2 Algorithm complexity

The complexity of implementing pn-search in a chess-playing program is comparable to implementing α-β search. Our efforts to create PROVER within the already existing DUCK, took less than half a day of programming and testing.

A detailed algorithmic description of pn-search can be found in Allis *et al.* (1994b, Allis (1994a).

5.1.3 Mobility

Pn-search's strategy may be described as checking first the variations in which the opponent has the least mobility. Instead of applying the calculation of mobility to a single position, an overall mobility for the search tree is calculated. The proof number of the root at any point in time indicates the mobility left to the defender for escaping mate.

PROVER's achievements seem to indicate that during a mate search, mobility is the single most important factor. Clearly, chess heuristics such as material balance, positional advantages lose most of their meaning when trying to force mate.

The depth of a mate is no longer a dominant factor in the size of the search tree to be grown. As long as the mobility of the defender is restricted, pn-search will continue to search a variation, regardless of the depth of the subtree. We present two example positions where this phenomenon leads to the discovery of a deep mate.

The position in figure 2 can be found as diagram 194 in (Krabbé, 1985). Krabbé states:

" **1. ♗a3+ ♔a1 2. ♗b2+ ♔b1 3. ♗xd4+ ♔c1** If White could now play 4. ♗b2+ ♔b1 5. ♗xe5+ etc., that would shorten the procedure enormously, but of course Black would escape: 4. ..., ♔d2. This necessitates the repetition of a seven move operation to bring the zwickmühle around: **4. ♗e3+ ♔d1 5. ♖d8+ ♔e1 6. ♗d2+ ♔d1 7. ♗b4+ ♔c1 8. ♗a3+ ♔b1 9. ♖b8+ ♔a1 10. ♗b2+ ♔b1 11. ♗xe5+ ♔c1 12. ♗f4+ ♔d1 13. ♖d8+ ♔e1 14. ♗d2+ ♔d1 15. ♗b4+ ♔c1 16. ♗a3+ ♔b1 17. ♖b8+ ♔a1 18. ♗b2+ ♔b1 19. ♗xf6+ ♔c1 20. ♗g5+ ♔d1 21. ♖d8+ ♔e1 22. ♗d2+ ♔d1 23. ♗b4+ ♔c1 24. ♗a3+ ♔b1 25. ♖b8+ ♔a1 26. ♗b2+ ♔b1 27. ♗xg7+ ♔c1 28. ♗h6+ ♔d1 29. ♖d8+ ♔e1 30. ♗d2+**

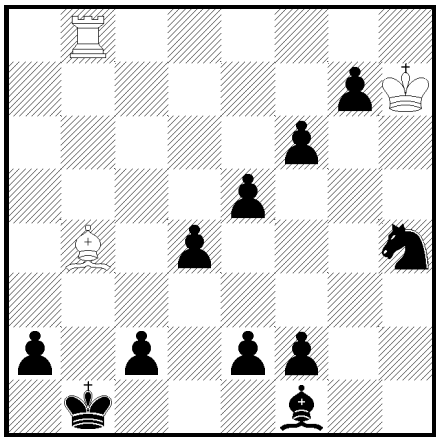


Figure 2: Mate in 38 (L. Ugren).

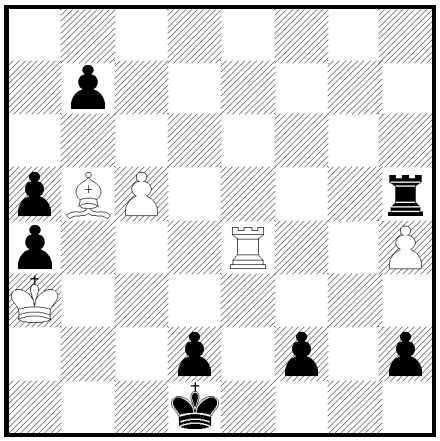


Figure 3: Mate in 25 (J.-L. Seret).

♔**d1 31.** ♗**b4+** ♔**c1 32.** ♗**a3+** ♔**b1 33.** ♖**b8+** ♔**a1 34.** ♗**e7!** and finally the idea is clear: f6 is the only safe safe square to threaten mate; on other squares the ♘h4 or one of the pawns could have thwarted that mate. Pawn d4 and pawn e5 had to go to open the diagonal, pawn f6 to gain access to g7, and pawn g7 to gain access to f6. After 34. ♗e7, mate cannot be staved off for more than a few moves"

The remaining moves are: **34. ..., c1=♕ 35.** ♗**f6+** ♕**b2 36.** ♖**xb2 e1=♕ 37.** ♖**b8+** ♕**e5 38.** ♗**xe5+** mate.

The position in figure 3 can be found as diagram 199 in (Krabbé, 1985). Krabbé states:

"Here, there are also two troublemakers and White disposes of an extended zwickmühle like the one in diagram 194 to silence them. 1. ♔b2 would mean mate in 2 if Black didn't have 1. .., a3+. That pawn can be immediately removed with 1. ♗xa4+, but after 1. ..., ♔c1 2. ♖c4+ ♔b1 3. ♗c2+ ♔a1! (3. ..., ♔c1 4. ♗f5+ allows White to enter the solution at move 14) 4. ♔b3 Black has the nasty 4. ..., b5 5. cxb6 ♖b5+ etc. Therefore, in order to remove the pawn a4, White must first remove the pawn b7. Hence **1.** ♗**e2+** ♔**e1** (1. ..., ♔c2 2. ♖c4+ ♔b1 3. ♗d3+ ♔a1 4. ♖c2 and 5. ♖a2 mate) **2.** ♗**g4+** ♔**f1 3.** ♗**h3+** ♔**g1 4.** ♖**g4+** ♔**h1 5.** ♗**g2+** ♔**g1 6.** ♗**xb7+!** ♔**f1 7.** ♗**a6+** ♔**e1 8.** ♖**e4+** ♔**d1 9.** ♗**e2+** ♔**e1 10.** ♗**b5+!** ♔**d1** and we are back in the diagram, but without the pawn b7 which means the pawn a4 meets its end too. **11.** ♗**xa4+** ♔**c1 12.** ♖**c4+** ♔**d1 13.** ♗**c2+** ♔**c1!** Because if now 13. ..., ♔a1 14. ♔b3! **14.** ♗**f5+** ♔**d1 15.** ♗**g4+** ♔**e1 16.** ♖**e4+** ♔**f1 17.** ♗**h3+** ♔**g1 18.** ♖**g4+** ♔**h1**

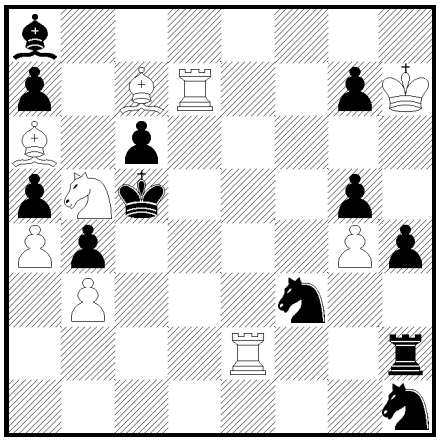


Figure 4: Problem 6 of the *ICCA Journal*.

19. ♗**g2+** ♔**g1 20.** ♗**c6+! ♔f1 21.** ♗**b5+** ♔**e1 22.** ♖**e4+** ♔**d1** and there we are: back in the diagram, but without those inconvenient pawns. **23.** ♔**b2!** ♖**xc5 24.**♗**a4+** ♖**c2+ 25.** ♗**xc2+ mate.**"

In the diagrams of figure 2 and 3, the mate found is also the intended solution to the problem. As we will show in the next section, in some cases the opportunity to play forcing moves may lead to excessively long detours from the optimal solution.

5.2 Weaknesses

5.2.1 Non-forcing moves

In many mating problems, the attacker delivers check on most of his moves, thus restricting the options of the defender. In some cases, however, the attacker makes a non-forcing move, after which almost any move by the defender leads to the same deciding attack. As the mobility of the opponent is enlarged by such a non-forcing move, pn-search prefers to check first the variations in which the defender is restricted. Thus, if the only solution requires a non-forcing move, pn-search will perform relatively worse, than if a more forcing mate exists. This problem is also recognized by Schaeffer (1989) when using conspiracy-number search as a tactical analyzer.

As a measure for the difficulty of a position caused by non-forcing moves, we count the number of different variations within the solution. We present three example positions from section 4.2 where PROVER performed worse than the algorithms it was compared with. In each case, non-forcing moves play a dominant role.

In the position of figure 4 the main line contains two non-forcing moves, to which all replies must be checked, explaining the difficulties experienced by PROVER. In total, the solution consists of 30 variations. In the position of figure 5 problems are even worse. PROVER finds a mate in 5 moves consisting of 697 different variations. Problem 14 of *Win at Chess* is a mate in 4, consisting of 49 variations.

To evaluate this criterion of the number of variations, we look at three problems in which PROVER performed much better than the algorithms it was compared with. On problem 5 of Van Gisteren (1992), PROVER found a mate in 7 consisting of only 3 variations. Problem 13 of Grottling (1985) was solved by an unnecessary long mate in 19, but consisted only of 1 variation, explaining the 0.10 seconds CPU time needed. Finally, the mate in 10 position, solved by Cray Blitz in over 5 million nodes, and by PROVER in less than 20 thousand, consisted of 13 variations. We remark that the mate found by PROVER was 4 moves longer than the optimal solution.

We conclude that in positions where the solution requires non-forcing moves, thus increasing the number of separate variations to consider, pn-search performs worse than in positions where

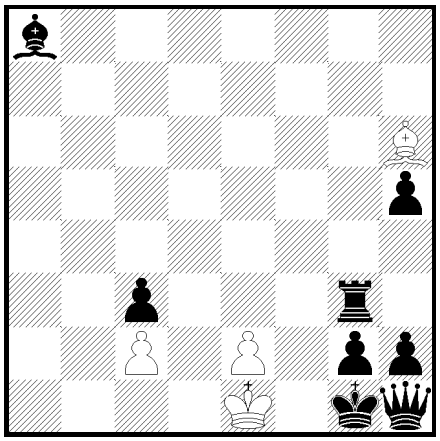


Figure 5: Problem 14 of the *ICCA Journal*.

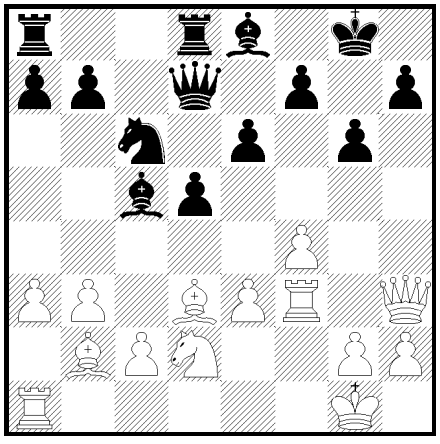


Figure 6: Problem 14 of *Win at Chess*.

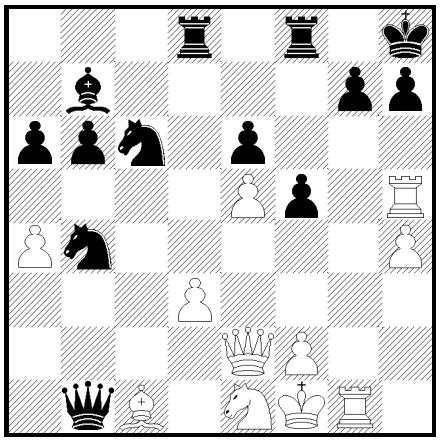


Figure 7: Problem 213 of *Win at Chess*.

only a small number of deep variations is to be found.

5.2.2 Transpositions

Transposition tables help to reduce the search tree in chess programs significantly. Unfortunately, pn-search and transposition tables are difficult to combine. For an in-depth treatment of the subject, see Schijf (1993). We are currently looking for a practical algorithm which allows transpositions to be handled within pn-search.

To see how transpositions slow down pn-search, let us suppose that a single variation occurs as six separate subtrees within one variation. The number of variations to be proven thus increases by a factor 6. The amount of search to be performed increases by a factor of far more than 6, however. Since the subtree is judged to be 6 times as difficult as it really is, alternative subtrees are investigated more deeply, until it is discovered that, despite the factor 6 extra work, the subtree with transpositions is best.

Transpositions are the main reason why PROVER has not been able to solve the position in Anantharaman *et al.* (1989), which we present in the diagram of figure 7.

It can be shown that in the solution tree for this problem, the position of the diagram in figure 8 occurs six times. Since the distance to mate from that positions is still considerable, the transpositions form an obstacle in the search which pn-search is unable to negotiate.

5.2.3 Mate length

In the position of figure 9, the shortest mate possible is mate in 5 moves: 1. ♖h6! ♔xh6 2. ♔xf6 ♔h7 3. g5 ♔h8 4. g6 fxg6 5. ♔xg6+ mate. However, in the pn-search solution, after 1. ♗xf6+ ♔g6 2. ♖h1 ♗h7 3. ♖xh7 ♔xh7 4. ♔xf7, white plays aimlessly with his bishop until he, by accident, stumbles upon the correct configuration. Still, only 408 nodes were grown to find the solution.

As described before, pn-search does not care about the depth of the search, only about the number of options of the defender. In some cases pn-search finds mates in over 100 moves, which actually are mates in less than ten moves. In the position of figure 10 (Howard, 1961), for example, pn-search finds a mate in 114, in 1 second, while a mate in 4 is possible. This solution of 114 moves is entirely forced. White starts with the move 1. e6–e7, after which Black's king plays back and forth between h7 and g8, while white is playing around until he by accident stumbles on a mate. Alternative initializations of proof numbers and disproof numbers, depending on the depth in the search tree, can solve these problems.

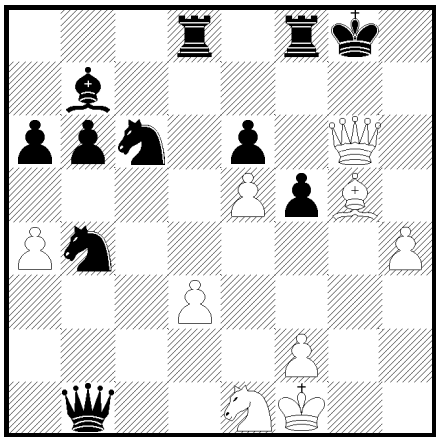


Figure 8: Six-fold transposition in problem 213 of *Win at Chess*.

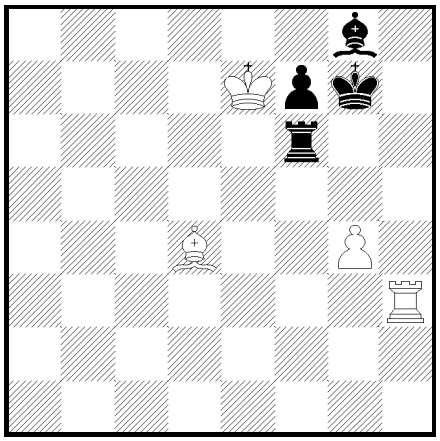


Figure 9: Problem 13 of the *ICCA Journal*.

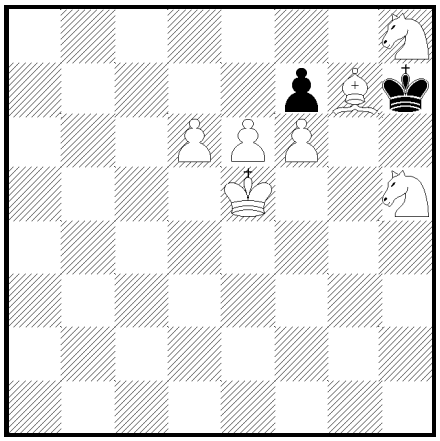


Figure 10: Problem 150 of *The enjoyment of chess problems*.

5.2.4 Memory

Pn-search has to keep the entire search tree in memory, for the node which is chosen to be expanded can be anywhere in the tree. In the pn-search implementation in DUCK, the size of a node is 32 bytes. To grow a tree of 1,000,000 nodes (which takes four to five minutes in DUCK), 32 megabytes of memory is needed.

The large trees are a disadvantage for commercial chess computers, which usually have a very limited amount of RAM. Thus, unfortunately, it will not be feasible for commercial chess programs to use pn-search as a tactical analyzer.

6 Future enhancements

6.1 Mate length

As mentioned in section 5.2.3, pn-search often finds a deeper mate than the shortest mate possible. To force pn-search to find a shallower mate, two solutions come to mind: (1) after pn-search has found a mate, try to find a quicker mate by forbidding pn-search to grow the search tree to a depth greater or equal than the solution found, using the already generated search tree as a guide *or* (2) instead of initializing the proof number and disproof number in a temporary terminal node at 1, they can be initialized at, for instance, the depth of the this node in the tree, thus making it unattractive to search very deep.

6.2 Transpositions

As mentioned in section 5.2.2, pn-search and transposition tables are difficult to combine (Schijf, 1993). Therefore, we hope to solve this problem in the near future by finding a practical solution.

6.3 Tactical analyzer

In this article it has been shown that pn-search is a good mate searcher. It would be interesting to apply pn-search not only to mate searching, but also to other tactical situations in chess. For instance, it can be used as a tactical analyzer to look for a gain in material.

This is more difficult than searching for mate, since the stop condition (recognizing that a node is proven or disproven) is not easy to formulate. One wants to stop searching in a node if the gain value is *stable*, and evaluate that node as a win.

An example definition comes to mind when thinking of the stability of the gain value. The gain value of a node can be defined as stable if the attacker is to move and has gained at least the material which is expected.

6.4 Advisor to α-β search

Since the definition of stable gain value is a heuristic one, it is possible that pn-search unjustly thinks it has found a gain in material. To prevent this, the variation found by pn-search can be checked by an α-β search. The variation can also be used to sort the moves in the α-β search, resulting in deeper searches than a standard full-width search, because of all additional cut-offs.

7 Conclusions

In this paper we presented a new search technique to search for mate in chess: proof-number search. This technique uses no chess-specific knowledge other than mate, stalemate and rules for deciding a draw.

It is shown that this technique almost always outperforms conventional search techniques. The failure in some positions is due to the fact that non-forcing moves are needed in the main variation.

Proof-number search is very easy to implement: we implemented it into an existing chess program in less than half a day.

The results of using proof-number search to search for mate are promising. In the future we will investigate how this technique can be used as a tactical analyzer.

The only real drawback of proof-number search seems to be the lack of finding non-forcing moves which lead to mate. This problem will be investigated in the near future.

References

[1] Allis L.V. (1994a). Games and Artificial Intelligence. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands. To appear. (2, 3, 4, 11)

[2] Allis L.V., Van der Meulen M., and Van den Herik H.J. (1994b). Proof-Number Search. *Artificial Intelligence*. To appear. (2, 3, 11)

[3] Anantharaman T.S., Campbell M.S., and Hsu F.-h. (1989). Singular Extensions: Adding Selectivity to Brute-Force Searching. *Artificial Intelligence*, Vol. 43, No. 1, pp. 99–109. (4, 10, 15)

[4] Beal D.F (1984). Mating Sequences in the Quiescence Search. *ICCA Journal*, Vol. 7, No. 3, pp. 133–137. (8)

[5] Berliner H.J. (1974). Chess as problem solving: The development of a tactics analyzer. Ph.D. Thesis, Carnegy-Mellon University. (4)

[6] Bij de Weg M. (1989). Ed Schröder wint Nederlandse computertitel. *Computerschaak*, Vol. 9, No. 6, pp. 250–255. (10)

[7] Colditz K. (1983). *Lehr-, Übungs- und Testbuch der Schach-kombinationen*. Falken-Verlag, GmbH, Niedernhausen. (5)

[8] Gillogly J.J. (1972). The Technology Chess Program. *Artificial Intelligence*, Vol. 3, pp. 145–163. (4)

[9] Grottling G. (1985). Problem-solving Ability Tested. *ICCA Journal*, Vol. 8, No. 2, pp. 107–110. (8, 13)

[10] Hartmann D. and Kouwenhoven P. (1991). Sundry Computer Chess Topics. *Advances in Computer Chess 6* (ed. D.F. Beal), pp. 61–72. Ellis Horwood, Chichester, England. (10)

[11] Howard K.S. (1961). *The Enjoyment of Chess Problems*. Dover Publications, Inc., New York. (15)

[12] Knuth D.E. and Moore R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. (2)

[13] Krabbé T. (1985). *Chess Curiosities*. George Allen and Unwin, Ltd, London. (5, 11, 12)

[14] Marsland T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19. (4)

[15] Reinfeld F. (1958). *Win at Chess*. Dover Publications, Inc., New York. (5)

[16] Schaeffer J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19. (4)

[17] Schaeffer J. (1989). Conspiracy Numbers. *Advances in Computer Chess 5* (ed. D.F. Beal), pp. 199–217. North-Holland, Amsterdam. (13)

[18] Schijf M. (1993). Proof-Number Search and Transpositions. M.Sc. Thesis, University of Leiden, The Netherlands. (15, 17)

[19] Thompson K. (1991a). Chess Endgames Vol. 1. *ICCA Journal*, Vol. ?, No. ?, pp. ??–?? (2)

[20] Thompson K. (1991b). Chess Endgames Vol. 2. *ICCA Journal*, Vol. ?, No. ?, pp. ??–?? (2)

[21] Thompson K. (1991c). Chess Endgames Vol. 3. *ICCA Journal*, Vol. ?, No. ?, pp. ??–?? (2)

[22] Uiterwijk J. (1991). Extensies: Touch' sterkste wapen op het 11e NK Computerschaak. *Computerschaak*, Vol. 11, No. 6, pp. 253–261. (8)

[23] Van Gisteren F. (1992). Testresultaten van de MM5 en de Final Chesscard 2. *Computerschaak*, Vol. 12, No. 2, pp. 77–81. (7, 13)

[24] Wiereyn P.H. (1985a). Inventive Problem Solving. *ICCA Journal*, Vol. 8, No. 4, pp. 230–234. (8)

[25] Wiereyn P.H. (1985b). Problem-solving Ability Tested II. *ICCA Journal*, Vol. 8, No. 3, pp. 179–180. (8)

[26] Wilkins D.E. (1980). Using Patterns and Plans in Chess. *Artificial Intelligence*, Vol. 14, pp. 165–203. (7)