Chapter 3

The proof-number search algorithm

This chapter is a slightly adapted version of Breuker D.M., Allis L.V., and Herik H.J. van den (1994b). How to Mate: Applying Proof-Number Search. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 251–272. University of Limburg, Maastricht, The Netherlands[1].

The second and third problem statement deal with best-first search. In this chapter we therefore present a relatively new best-first search algorithm, called *proof-number search* (pn search), which will be used in the experiments addressing the second and third problem statement.

The basic ideas behind the pn-search algorithm are presented in Section 3.1. Section 3.2 lists the pseudo-code of the pn-search algorithm for trees. The experimental set-up is given in Section 3.3, and the test set is described in Section 3.4. Section 3.5 provides the experiments, of which the results are discussed in Section 3.6. Finally, Section 3.7 evaluates the experiments.

3.1 An informal description

In this section we present a short overview of pn search, based on Allis (1994). A detailed description of pn search can be found in Allis *et al.* (1994).

Proof-number search is a best-first AND/OR tree-search algorithm, and is inspired by the conspiracy-number algorithm (McAllester, 1988; Schaeffer, 1990). Before starting the search, a search goal is defined (e.g., try to reach at least a draw). The evaluation of a node returns one of three values: *true*, *false*, or *unknown*. The evaluation is seen from the point of view of the player to move in the root position. The value *true* indicates that the player to move in the root position can achieve the

[1] Thanks are due to the Editors of *Advances in Computer Chess 7* for giving permission to use the contents of the article in this chapter.

goal, while *false* indicates that the goal is unreachable. A node is *proved* if its value has been established to be *true*, whereas the node is *disproved* if its value has been determined to be *false*. A node is *solved* as soon as it has been proved or disproved. A tree is solved (proved or disproved) if its root is solved. The goal of pn search is to solve a tree.

Two variants of creating a search tree exist (cf. Allis, 1994).

1. *Immediate evaluation*. Each node in the tree is immediately evaluated after it is generated. The tree is built by first generating (and evaluating) the root. Then at each step a leaf is selected, expanded and all its children are immediately evaluated.

2. *Delayed evaluation*. Each node is only evaluated when it is selected, and not immediately after it is generated. The tree is built by first generating the root (without evaluation). Then, at each step a leaf is selected and evaluated. If the evaluation value is *unknown*, the node is expanded (without evaluating its children).

The advantage of immediate over delayed evaluation is that in the former variant more information is available. However, if the evaluation takes much time, it is better to use the delayed variant, avoiding the evaluation of many nodes that will not be used for solving the tree. Since, in the standard pn-search experiments described in this chapter, the evaluation is fast (only checking whether the position is a win, a loss, or a draw) we use the immediate variant in our further description of pn search.

Like other best-first search algorithms, pn search repeatedly selects a leaf, expands it, evaluates all its children, and updates the tree with the information obtained from the expansions and evaluations. Unlike most other best-first search algorithms, pn search does not use a heuristic evaluation function in order to determine a most-promising node. Instead, the shape of the search tree (the number of children of every internal node) and the values of the leaves determine which node to select next.

In general, to solve a tree, a number of leaves of the current search tree needs to be proved or disproved. A set of leaves, which, if all proved, would prove the tree, is called a *proof set*. Likewise, a set of leaves, which, if all disproved, would disprove the tree, is called a *disproof set*. The size of the smallest proof 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 disproof set of the tree is a lower bound for the number of node expansions necessary to disprove the tree.

In Figure 3.1 an AND/OR tree has been depicted. The numbers to the left of a node denote proof numbers, while the numbers to the right of a node denote disproof numbers. A *proof number* of a node represents the minimum number of leaves which have to be proved in order to prove that node. Analogously, a *disproof number* of a node represents the minimum number of leaves which have to be disproved in order to disprove that node.

Proved nodes (e.g., node K in Figure 3.1) have proof number 0 and disproof number ∞. This follows from the fact that no expansions are needed to prove the

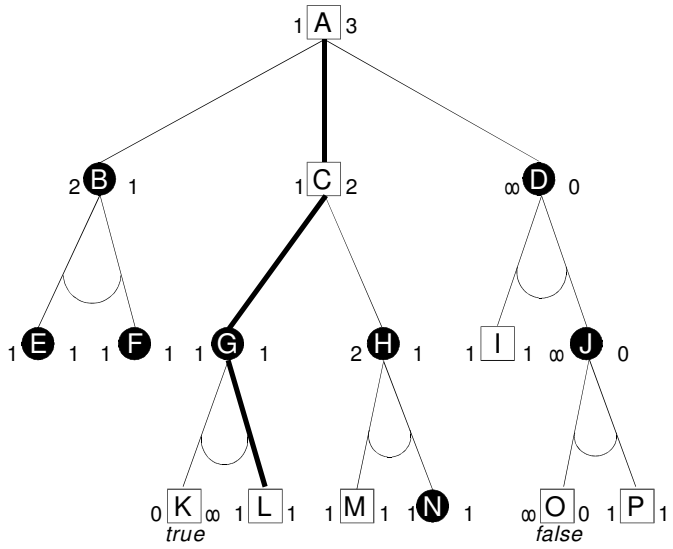

Figure 3.1: An AND/OR tree with proof and disproof numbers.

node, since it is already proved, and that no number of expansions could ever disprove the node. Analogously, disproved nodes (e.g., node O in Figure 3.1) have proof number ∞ and disproof number 0. Unsolved leaves (e.g., nodes E, F, L, M, N, I, and P) have a proof and disproof number of unity, as expanding the node itself may be sufficient to solve the node.

Internal AND nodes have as proof numbers the sum of the proof numbers of their children, since to prove an AND node, all children must be proved. The disproof number of an AND node equals the minimum of its childrens' disproof numbers, since only one child needs to be disproved to disprove the AND node. For instance, the proof number of node H is equal to the sum of the proof numbers of its children M and N ($2 = 1+1$). The disproof number of node H is equal to the minimum of the disproof numbers of its children (1). Analogously, the proof number of an internal OR node equals the minimum of the proof numbers of its children, whereas its disproof number equals the sum of the disproof numbers of its children. For instance, the proof number of node A is equal to the minimum of the proof numbers of its children B, C and D (1). The disproof number of node A is equal to the sum of the disproof numbers of its children ($3 = 1+2+0$).

The root (A) has proof number 1. This means that at least one leaf (in this case node L) should be proved to prove the root. The disproof number of the root is equal to 3. This means that at least three nodes (node E or node F, node L, and node M or node N) have to be disproved to disprove the root.

The main assumption underlying pn search is that it is generally better to expand

those nodes which are in the smallest proof and/or 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 is: when to select a node from the smallest proof set of the root and when to select a node from its smallest disproof set? Surprisingly, we can always do both at the same time. Allis *et al.* (1994) prove 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 elements of both a smallest proof set and a smallest disproof set of the root are called *most-proving nodes*. Thus, if after expansion of a most-proving node P, it obtains the value *true*, the proof number of the root is decremented by unity, while if P obtains the value *false*, the disproof number of the root is decremented by unity. If the value of P remains *unknown*, the newly generated children may have their impact on the proof and/or disproof numbers of P and its ancestors. A most-proving node is determined in the tree by selecting, at AND nodes, a child with disproof number equal to its parent's, and at OR nodes a child with proof number equal to its parent's. By thus traversing the tree from its root to a leaf (e.g., the bold path from A to L in Figure 3.1), it is shown that a most-proving node is found (Allis *et al.*, 1994).

3.2 The pseudo-code of the algorithm

All algorithms given in this section are based on the algorithms given by Allis (1994). The main proof-number search algorithm is given in Figure 3.2. The only parameter of the procedure is root, being the root of the search tree. After execution of the procedure, the root's value can have one of three values: *true*, *false* or *unknown*. First, the root is evaluated and its proof and disproof numbers are initialized. Then, in the main loop, repeatedly a most-proving node is selected, expanded, and all its children are evaluated. Thereafter, traversing the tree backwards to the root, the proof and disproof numbers are adjusted.

The function Evaluate evaluates a position, and returns one of the following three values: *true*, *false*, or *unknown*. The function SetProofAndDisproofNumbers initializes the proof and disproof numbers of a node. The algorithm is given in Figure 3.3. The only parameter of the function is node, being the node to be initialized. Two cases are distinguished. In the first case the node is an internal node (since it is expanded), and the proof and disproof numbers are initialized according to the proof and disproof numbers of its children. In the second case the node is not expanded, but it is evaluated, since immediate evaluation is used. The proof and disproof numbers are initialized according to the evaluation.

The function ResourcesAvailable returns a Boolean value indicating whether sufficient resources are available to continue searching. This is usually dependent on the available memory, but can also depend on a limited amount of time available. The function SelectMostProvingNode finds a most-proving node. The algorithm is given in Figure 3.4. The only parameter of the function is node, being the root of the

```
procedure ProofNumberSearch( root )
    Evaluate( root )
    SetProofAndDisproofNumbers( root )
    root.expanded := false

    while root.proof≠0 and root.disproof≠0 and
            ResourcesAvailable() do begin
        mostProvingNode := SelectMostProvingNode( root )
        ExpandNode( mostProvingNode )
        UpdateAncestors( mostProvingNode, root )
    end

    if root.proof=0 then root.value := true
    elseif root.disproof=0 then root.value := false
    else root.value := unknown /* resources exhausted */
end /* ProofNumberSearch */
```

Figure 3.2: The pn-search algorithm for trees.

(sub)tree where the most-proving node is located. As long as the node is expanded, a child is chosen with proof or disproof number (dependent on the type of node) equal to that of the parent. If a leaf is reached, the algorithm stops, and that node is returned.

The most-proving node found is expanded. This is done by the procedure ExpandNode. The only parameter of this procedure is node, being the node to be expanded. In Figure 3.5 its algorithm is depicted. First, all children are generated. Next, every child is evaluated and its proof and disproof numbers are set according to this evaluation.

After the expansion of the most-proving node, the new information has to be backed up throughout the whole tree. This is done by the procedure UpdateAncestors. The procedure has two parameters. The first parameter (node) is the node to be updated, while the second parameter (root) is the root of the search tree. Its algorithm is shown in Figure 3.6.

3.3 Experimental set-up

Pn search first examines the most forcing variations where the mobility of the opponent is as small as possible. This is explained as follows. The OR player chooses a child with the lowest proof number. By definition, the proof number of this child (an AND node) is equal to the sum of the proof numbers of its children. It follows that the AND child with the lowest proof number has the lowest mobility. Because pn search first examines forcing variations, it is expected that it will work extremely

```
procedure SetProofAndDisproofNumbers( node )
    if node.expanded then /* internal node */
        if node.type=AND then begin /* AND node */
            node.proof := 0
            node.disproof := ∞
            for i:=1 to node.numberOfChildren do begin
                /* Add up proof numbers and minimize disproof numbers */
                node.proof := node.proof + node.children[ i ].proof
                if node.children[ i ].disproof<node.disproof then
                    node.disproof := node.children[ i ].disproof
            end
        end else begin /* OR node */
            node.proof := ∞
            node.disproof := 0
            for i:=1 to node.numberOfChildren do begin
                /* Minimize proof numbers and add up disproof numbers */
                if node.children[ i ].proof<node.proof then
                    node.proof := node.children[ i ].proof
                node.disproof := node.disproof + node.children[ i ].disproof
            end
        end
    else /* leaf */
        case node.value of begin
            false:
                node.proof := ∞
                node.disproof := 0
            true:
                node.proof := 0
                node.disproof := ∞
            unknown:
                node.proof := 1
                node.disproof := 1
        end
end /* SetProofAndDisproofNumbers */
```

Figure 3.3: The proof-and-disproof-numbers-calculation algorithm.

```
function SelectMostProvingNode( node )
    while node.expanded do begin
        i := 1
        if node.type=OR then /* OR node */
            while node.children[ i ].proof≠node.proof do i := i+1
        else /* AND node */
            while node.children[ i ].disproof≠node.disproof do i := i+1
        node := node.children[ i ]
    end

    return node
end /* SelectMostProvingNode */
```

Figure 3.4: The most-proving-node-selection algorithm.

```
procedure ExpandNode( node )
    GenerateAllChildren( node )
    for i:=1 to node.numberOfChildren do begin
        Evaluate( node.children[ i ] )
        SetProofAndDisproofNumbers( node.children[ i ] )
        node.children[ i ].expanded := false
    end
    node.expanded := true
end /* ExpandNode */
```

Figure 3.5: The node-expansion algorithm.

well in cases where the goal can be reached by forcing variations. Therefore, we have chosen to investigate this in the domain of finding checkmates.

3.3.1 The search engine

The proof-number search engine is implemented according to the description in Section 3.1. The most important enhancement of the pn-search implementation, relative to a naïve implementation is in the initialization of proof and disproof numbers at the leaves. In the standard algorithm, proof and disproof numbers are each initialized to unity. Assume that *after* expansion all the n children evaluate to the value unknown. Then the proof and disproof numbers of the most-proving node are set to 1 and n for an OR node, and to n and 1 for an AND node. In our implementation, to distinguish between leaves, *before* expansion, we set the proof and disproof number of node P to 1 and n (or n and 1, depending on the node type), where n is the

```
procedure UpdateAncestors( node, root )
    SetProofAndDisproofNumbers( node )
    while node≠root do begin
        node := node.parent
        SetProofAndDisproofNumbers( node )
    end
end /* UpdateAncestors */
```

Figure 3.6: The ancestor-updating algorithm.

number of legal moves in the position represented by P. Experiments show that the extra overhead introduced by counting the number of legal moves at each node is more than compensated for by the value of the extra information thus revealed to the node-selection process (Allis, 1994).

3.3.2 The move ordering

For pn search the move ordering is of less importance than for $\alpha\beta$ search. For the reproducibility of the experiments we have chosen to order the moves in descending square order (h8, g8, ..., a8, h7, ..., a7, ... , h1 ... a1). The moves are sorted according to their *from* squares. If two moves have identical *from* squares they are sorted according to their *to* squares. If these are also identical, then the moves must be promotion moves, and the moves are sorted according to their promotion pieces (in the order Queen, Rook, Bishop, Knight).

As an example we provide the starting position at the game of chess. The White moves are sorted thus: **h2–h4**, **h2–h3**, **g2–g4**, **g2–g3**, **f2–f4**, **f2–f3**, **e2–e4**, **e2–e3**, **d2–d4**, **d2–d3**, **c2–c4**, **c2–c3**, **b2–b4**, **b2–b3**, **a2–a4**, **a2–a3**, ♘**g1–h3**, ♘**g1–f3**, ♘**b1–c3**, ♘**b1–a3**.

3.4 The test set

For our experiments we used a diverse set of mating problems. They are taken from Krabbé's (1985) *Chess Curiosities* and Reinfeld's (1958) *Win at Chess*. The 35 positions taken from Krabbé (1985) are mating problems in six moves or more. They are indicated by the name Kx, in which x refers to the diagram number in the source and takes the values 8, 35, 37, 38, 40, 44, 60, 61, 78, 192, 194, 195, 196, 197, 198, 199, 206, 207, 208, 209, 210, 211, 212, 214, 215, 216, 217, 218, 219, 220, 261, 284, 317, 333 and 334. The 82 positions taken from Reinfeld (1958) are problems where we know that a forced mate is possible. They are indicated by the name Rx, x again referring to the problem number in the source, this time running over 1, 4, 5, 6, 9, 12, 14, 27, 35, 49, 50, 51, 54, 55, 57, 60, 61, 64, 79, 84, 88, 96, 97, 99, 102, 103, 104, 105, 132, 134, 136, 138, 139, 143, 154, 156, 158, 159, 160, 161, 167,

168, 172, 173, 177, 179, 182, 184, 186, 188, 191, 197, 201, 203, 211, 212, 215, 217, 218, 219, 222, 225, 241, 244, 246, 250, 251, 252, 253, 260, 263, 266, 267, 278, 281, 282, 283, 285, 290, 293, 295 and 298. This results in a test set of 117 positions (see Appendix D).

3.5 Experiments

Pn search always aims at proving or disproving a certain goal. In our experiments the only goal is searching for mate. In our description, we distinguish between the attacker and the defender. The attacker is the player to move in the root position, while the defender is the opponent. A position is proved if the attacker can mate, while draws (by stalemate, by repetition of positions and by the 50-move rule) and mates by the defender are defined to be disproved positions for the attacker. If a position is neither proved nor disproved, it is said not to be solved.

We have compared the the pn-search algorithm to the $\alpha\beta$-search algorithm, implemented in DUCK[2] on the test set described in Section 3.4. We note that it is possible to create a special mate searcher using $\alpha\beta$ search, which will perform better than DUCK. However, pn search does not use any chess-specific knowledge other than recognizing mates, stalemates, and drawn positions. Therefore, we decided to choose DUCK as the $\alpha\beta$ searcher. The search was terminated as soon as any mate was found. The experiments are conducted to investigate how a best-first search algorithm (using much memory, since it stores the whole search tree) compares to the widely used depth-first $\alpha\beta$-search algorithm (using little memory). Furthermore, in the next chapters we concentrate on best-first search, using proof-number search as example and using the same test set.

We have performed experiments in which both programs had to solve each position within 1,000,000 nodes. This limit was selected for two reasons:

1. The calculation time (up to 5 minutes on the hardware used) corresponds roughly to tournament conditions.

2. 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.

3.6 Results

This section contains the results of the experiments described in Section 3.5. The complete results for every test position individually are presented in Appendix E. As

[2] In contrast to ALIBABA, of which the $\alpha\beta$ search engine was designed specifically for the transposition-table experiments, DUCK is a full-blown tournament program, incorporating a detailed evaluation function and several $\alpha\beta$ enhancements such as extension heuristics. For more details, see Breuker *et al.* (1994b).

the measures of performance we use the number of positions solved and the number of nodes investigated.

From the 117 test positions, 106 positions were solved by at least one algorithm: 73 were solved by both algorithms, 30 by pn search only and 3 by $\alpha\beta$ search only. We have stated for each test position the algorithm by which it could be solved within 1,000,000 nodes.

Both algorithms: K35, K38, K197, K211, K212, K261, K317, R1, R4, R5, R9, R12, R14, R27, R35, R49, R50, R54, R55, R57, R60, R61, R64, R79, R84, R88, R97, R99, R102, R103, R104, R132, R134, R136, R139, R143, R154, R156, R158, R160, R161, R167, R172, R173, R177, R179, R184, R186, R188, R191, R197, R203, R211, R212, R215, R217, R219, R225, R244, R246, R251, R253, R260, R263, R266, R267, R278, R282, R283, R285, R290, R295, R298.

Pn search only: K37, K61, K192, K194, K196, K198, K199, K206, K207, K208, K214, K215, K216, K218, K219, K333, K334, R6, R51, R138, R159, R168, R182, R218, R222, R241, R250, R252, R281, R293.

$\alpha\beta$ search only: K60, K284, R105.

Neither algorithm: K8, K40, K44, K78, K195, K209, K210, K217, K220, R96, R201.

In Table 3.1 the results are summarized. In the first row of the table the total number of nodes searched on the 73 positions solved by both algorithms is listed. The second row contains the average number of nodes searched per position. The third row lists the number of times the stated algorithm outperformed the other (by the criterion of the number of nodes searched). In the fourth and fifth row a position is selected where the ratio of nodes visited was lowest for pn search and $\alpha\beta$ search, respectively. The sixth row shows the average number of nodes searched by pn search on the 30 positions not solved within a million nodes by $\alpha\beta$ search. The last row shows the average number of nodes searched by $\alpha\beta$ search on K60, K284 and R105, the only three positions solved by $\alpha\beta$ search but not by pn search.

Comparing the performance of pn search with $\alpha\beta$ search creates a consistent impression of a general superiority of pn search as a mate searcher.

- The total number of nodes investigated by pn search is about 20% of the number of nodes investigated by $\alpha\beta$ search.
- Pn search outperformed $\alpha\beta$ search in some 84% of the cases.
- The ratio of nodes visited from the point of view of pn search was lowest in the case of position R217. The number of investigated nodes by pn search is only a fraction (0.08%) of the number investigated by $\alpha\beta$ search. From the point of view of $\alpha\beta$ search the best position was R253. Here, the number of nodes investigated by $\alpha\beta$ search is about 13% of the number investigated by pn search.

	Pn search	$\alpha\beta$ search
Total number of nodes searched	953,762	5,198,074
Average number of nodes per position	13,065	71,206
Best performer	61	12
Best instance (nodes) of pn search (R217)	271	331,404
Best instance (nodes) of $\alpha\beta$ search (R253)	2,355	311
Pn search (nodes) where $\alpha\beta$ search failed	230,166	>1,000,000
$\alpha\beta$ search (nodes) where pn search failed	>1,000,000	644,058

Table 3.1: Comparing pn search and $\alpha\beta$ search.

- The average number of nodes investigated by pn search on the 30 positions that $\alpha\beta$ search did not solve within 1,000,000 nodes is 230,166. In contrast, the number of nodes investigated by $\alpha\beta$ search on the three positions that pn search did not solve within 1,000,000 nodes is much higher (644,058).

In the next subsection the particular strengths and weaknesses of pn search are discussed.

3.6.1 Strengths of pn search

This subsection discusses two strengths of pn search: (1) the algorithm does not need specific chess knowledge, and (2) the algorithm finds deep, forced mates.

No specific chess knowledge

The pn-search algorithm does not use any specific chess knowledge. All that is needed is a move generator, and an evaluation function able to recognize mate, stalemate and draws by repetition or by the 50-move rule. We would like to stress that, quite unlike $\alpha\beta$ search, move ordering has not much influence on the performance of pn search. This phenomenon is explained by the way pn search builds its tree. At each step, the child with the smallest proof or disproof number (depending on the node type) is selected. Only if two children tie is the selection of a node based on the move ordering. Experiments with changing the move ordering showed that this ordering has little influence on the number of nodes grown[3]. Not using any chess knowledge has the advantage that pn search can be incorporated into any chess program, regardless of the evaluation function and of the heuristics applied.

[3] This is contrary to the results found for conspiracy-number search, as given by Klingbeil and Schaeffer (1990). They show that move ordering does have influence when searching in tactical chess positions.

Finding deep, forced mates

The strategy of pn search may be described as investigating first those variations in which the opponent has the least mobility. Instead of examining the mobility for a single position, pn search examines the mobility of the search tree as a whole. The proof number of the root indicates, at any point in time during the computation, the mobility left to the defender for escaping mate. The achievements seem to indicate that, during a mate search, mobility is the most important factor. Clearly, chess characteristics, such as material balance and positional advantage, lose most of their meaning when trying to force a mate is the unique goal aimed at. Moreover, the distance-to-mate is no longer a dominant factor in the size of the search tree grown. As long as the mobility of the defender is restricted, pn search will continue to explore a variation, regardless of the depth of the subtree explored. We present two sample positions where this characteristic leads to the discovery of a deep mate, which would not be found if the depth of the subtree explored was an important factor (as it is in $\alpha\beta$ search).

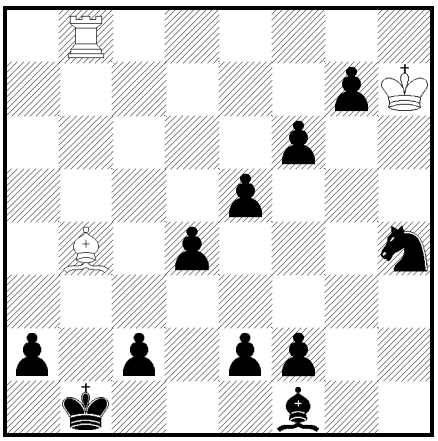

Figure 3.7: Mate in 38 (WTM); (L. Ugren).

The position in Figure 3.7 is taken from Diagram 194 in Krabbé (1985). We note that the square a1 is the left-bottom square, so Black has 4 Pawns ready to promote. For the chess-playing reader we cite the solution as stated by Krabbé: "**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+** ♔**d1 31.** ♗**b4+** ♔**c1 32.** ♗**a3+** ♔**b1 33.** ♖**b8+** ♔**a1 34.** ♗**e7!** and finally the idea is clear: f6 is the only safe square to threaten mate; on other squares the ♘h4 or one of the Pawns could have thwarted that mate. ♙d4 and ♙e5 had to go to open the diagonal, ♙f6 to gain access to g7, and ♙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.

Proof-number search solves this mate in 229,423 nodes, whereas our implementation of $\alpha\beta$ search fails to solve it within 50,000,000 nodes[4].

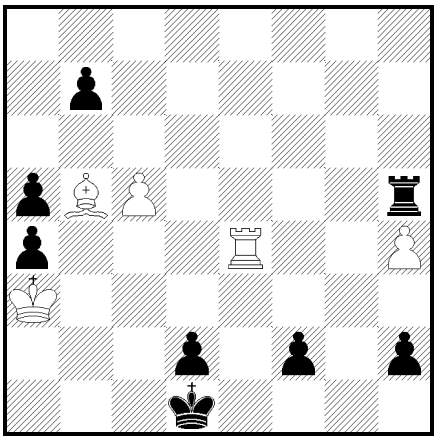

Figure 3.8: Mate in 25 (WTM); (J.-L. Seret).

The position in Figure 3.8 is taken from Diagram 199 in Krabbé (1985). Again, for the chess-playing reader we give the solution as stated by Krabbé: "Here, there are also two troublemakers and White disposes of an extended zwickmühle like the one in diagram 194 [our Figure 3.7] 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 ♔c1 2. ♖c4+ ♔b1 3. ♗c2+ ♔a1! (♔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 ♙a4, White must first remove the ♙b7. Hence **1. ♗e2+** ♔**e1** (♔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**

[4] This result is heavily dependent on the search extensions used. The $\alpha\beta$ program THETURK solves this mate in 3,325,715 nodes when choosing the right extensions (Schaeffer, 1998)

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 ♙b7 which means the ♙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 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**."

Proof-number search solves this mate in 370,016 nodes, whereas our implementation of $\alpha\beta$ search fails to solve it within 50,000,000 nodes.

In Figures 3.7 and 3.8, the mate found by pn search is also the intended solution to the problem. Since the solutions contained many forcing moves (leaving the defender few moves), pn search performed very well. $\alpha\beta$ search performed very poorly because the solutions were very deep (75 and 49 ply, respectively). As we will see in subsection 3.6.2, in some cases, the duty of playing the most-forcing moves imposed by pn search may lead to excessive departures from the optimal solution.

3.6.2 Weaknesses of pn search

This subsection discusses three weaknesses of pn search: (1) the inability to find good, non-forcing moves, (2) the inability to find the shortest mate, and (3) the inability to deal with transpositions.

Non-forcing moves

In many mating problems, the attacker delivers check on most moves, thus restricting the options of the defender. In some cases, however, the attacker plays a non-forcing move, after which almost any move by the defender leads to the same decisive attack. Since the mobility of the opponent is increased by such a non-forcing move, pn search prefers first to investigate those variations in which the defender is most confined.

Hence, if the only solution requires one or more non-forcing moves, pn search will not perform as well as it will when a mate exists with forcing moves only. We note here that its preference is not merely for checking moves (which are forcing moves in human parlance), but it must, by its algorithm, prefer the most-forcing checks. A similar problem is recognized by Schaeffer (1989a, 1990) when using conspiracy-number search as a tactical analyzer.

As a measure of the difficulty of a position for pn search caused by non-forcing moves, we propose considering the number of different variations within the solution. We present a sample position where pn search performed worse than $\alpha\beta$ search. The existence of non-forcing moves proved a significant factor in degrading its performance. Problem 14 of Reinfeld (1958) (Figure 3.9) is a mate in four moves consisting of 49 variations. After **1. ♕xh7+ ♔f8**, the best move is the non-forcing move **2. ♗f6**, threatening the unavoidable **2. ..., ♕g7 mate**. Proof-number search solves this mate in 324,542 nodes, whereas $\alpha\beta$ search only needs 127,519 nodes.

We conclude that in positions where the solution requires non-forcing moves, pn search is at a disadvantage. The three positions not solved by pn search (κ60,

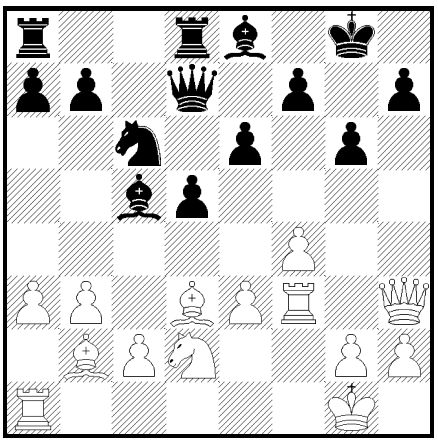

Figure 3.9: Problem 14 of *Win at Chess* (WTM).

K284 and R105) have solutions with non-forcing moves. It is even worse, since the first moves of both solutions are non-forcing, making it impossible for pn search to find the solutions within 1,000,000 nodes.

Mate length

As stated before, pn search is indifferent to the depth of the search, being governed only by the defender's number of options. As a consequence, pn search finds mates in over 100 moves, while optimal ones exist in fewer than ten moves. The position shown in Figure 3.10 is problem 150 of Howard (1961). It shows an example of pn search finding a mate in 114 moves while an optimal mate of four moves exists. The intended solution reads **1.** ♔**e4** and now either **1.** ..., **fxe6 2. f7 e5 3. f8=**♗ ♔**g8 4.** ♘**f6** mate, or **1.** ..., ♔**g8 2. exf7+** ♔**h7 3. f8=**♘**+** ♔**g8 4. f7** mate.

As a solution we suggest initializations of the proof and disproof numbers different from the ones proposed above, specifically with the initial values depending on the depths in the search tree. This may solve the problems of the apparently aimless and certainly long paths to mate.

Transpositions

A third weakness encountered when using pn search is the inability of dealing with transpositions. Assume that an identical subvariation occurs as six separate subtrees within one variation tree. Then, the number of variations to be solved increases by a factor of six. The amount of search to be performed, however, increases by a factor of far more than six. Since, by the rules of combining proof and disproof numbers in

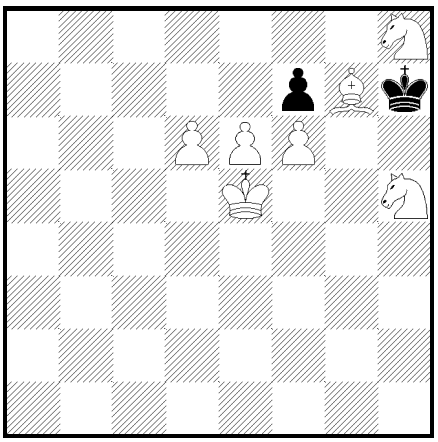

Figure 3.10: Problem 150 of *The Enjoyment of Chess Problems* (WTM).

AND/OR trees, the difficulty of each subtree is propagated upwards sixfold, pn search may well be led to the investigation of other subtrees. If these subtrees fail to deliver a mate pn search will, at long last, arrive at the correct branch in another subtree leading to mate.

As an example we provide problem 213 of Reinfeld (1958) (Figure 3.11). The intended solution starts with the moves **1. ♖xh7+ ♔xh7 2. ♕h5+ ♔g8 3. ♖xg7+ ♔xg7 4. ♗h6+ ♔h7 5. ♗g5+ ♔g7 6. ♕h6+ ♔f7 7. ♕f6+ ♔g8 8. ♕g6+ ♔h8**, reaching the position of Figure 3.12. In the solution tree this position occurs six times, depending on Black's defence at moves 4, 5 and 6. The proof number of this position will be high because the distance-to-mate from that position is still considerable. Upward propagation will expand the proof number six-fold. The resultant high proof number provides an obstacle which pn-search was unable to overcome.

3.7 Chapter conclusions

In this chapter we have described experiments comparing pn search with $\alpha\beta$ search. Pn search has been presented as a best-first search technique easy to implement and uniquely attuned to finding mates in chess. Beyond recognizing mates, stalemates, and drawn positions, no chess-specific knowledge is required. When a mate exists within its horizon, this technique consistently outperforms conventional techniques in terms of nodes visited, except when the solution relies on the presence of non-forcing moves, transpositions, or on providing the shortest mate.

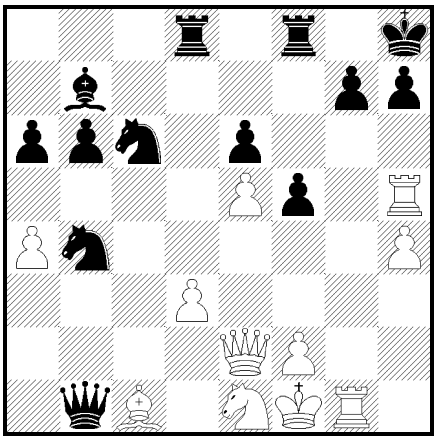

Figure 3.11: Problem 213 of *Win at Chess* (WTM).

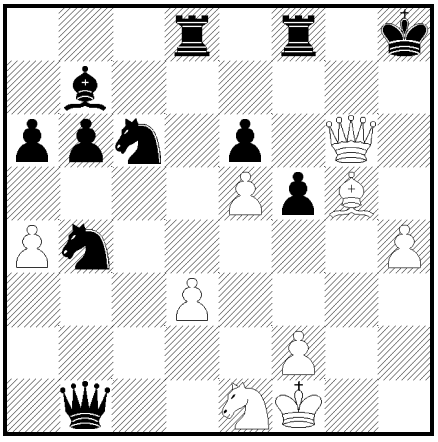

Figure 3.12: Six-fold transposition in problem 213 of *Win at Chess* (WTM).