Blockage Detection in King and Pawn Endings

Omid David Tabibi[1], Ariel Felner[1], and Nathan S. Netanyahu[1,2]

[1] Department of Computer Science, Bar-Ilan University,
Ramat-Gan 52900, Israel
{davoudo,felner,nathan}@cs.biu.ac.il
http://www.cs.biu.ac.il/∼{davoudo,felner}
[2] Center for Automation Research, University of Maryland,
College Park, MD 20742, USA
nathan@cfar.umd.edu

Abstract. The conventional search methods using static evaluation at leaf nodes are liable to missing the fact that no win goal exists in certain positions. Blockage positions, in which neither side can penetrate into the opponent's camp, are a prominent example of such positions. Deep search cannot detect the existence of a blockage, since its judgment is based solely on static evaluation, without taking the goals into consideration. In this paper we introduce a blockage detection method, which manages to detect a large set of blockage positions in pawn endgames, with practically no additional overhead. By examining different aspects of the pawn structure, it checks whether the pawns form a blockage which prevents the king from penetrating into the opponent's camp. It then checks several criteria to find out whether the blockage is permanent.

1 Introduction

In the early days of computer-chess, many were pessimistic about the prospects of creating grandmaster level chess playing programs through the conventional search methods which incorporate chess knowledge only in the evaluation function applied to the leaf nodes. As de Groot [3] mentioned, there are positions which computers have trouble evaluating correctly through exhaustive search, yet humans recognize at a glance.

For many years the pawn endgames remained the toughest stage of the game for computers, as many combinations require very deep searches, infeasible for the computers to conduct. There have been several efforts to improve the pawn endgame play of computers in these positions, notably PEASANT [8], co-ordinate squares [2], and chunking [1]. But since then, most such positions have succumbed to the sheer processing power of computers. Nowadays, tournament playing programs search to depths of tens of plies in pawn endgames, and thus, manage to solve many positions once thought as never solvable by computers. For example, Newborn, the author of PEASANT, estimated that the position shown in Figure 1 (Fine #70 [4]) required 25,000 hours for the computers to solve [8]. Yet today it takes less than a second for most programs to search deep enough to find the correct 1.Kb1 move.

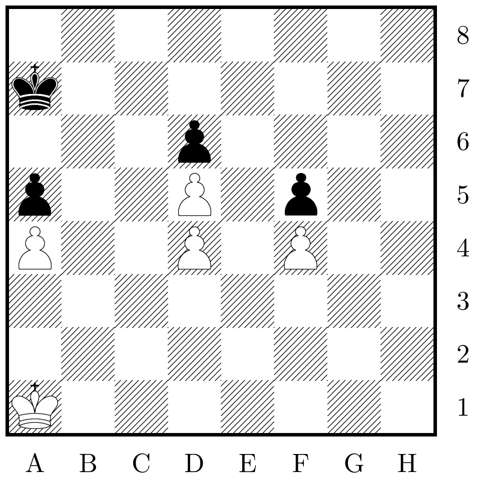

Fig. 1. Fine #70: white to move, 1.Kb1 wins.

However, there are still position types which cannot be evaluated realistically even by the deepest searches. Positions with the element of *blockage* present, are a prominent example of such positions. In these positions there is a fortress of pawns which blocks one side from penetrating into the other's camp. Assuming there is no blockage detection knowledge present in the program, only a 100-ply search will declare the position a draw by fifty-move rule.

In this paper, we focus on pawn endgames featuring the blockage motif. In these position types, the pawns divide the board into two separate disconnected areas. That is, the pawns are blocked, and the king is not able to attack the opponent's pawns in order to clear the way for its own pawns. And so, a draw will be the outcome of the game. Computers using conventional search methods will not be able to detect the draw, because no matter how deep they search, ultimately their judgment of the positions is based on the static evaluation which lacks the knowledge of blockage detection.

Although such positions are not very frequent in practice, whenever they arise, they badly expose this deficiency of computers. For example, while the position shown in Figure 2 is clearly a draw, all top tournament playing programs we tested evaluate the position as a clear win for white (see Appendix). This was also the case in the first game of Deep Fritz vs. Vladimir Kramnik in their Bahrain match [7]. Throughout the game, Deep Fritz maintained an edge over Kramnik, until it found a clear advantage in one of the variations and opted for it. Figure 3 shows the position which Deep Fritz saw in its search, and evaluated it as a clear advantage for white. However, with simple observation a human player can recognize that this resulting position forms a blockage, resulting in no more than a draw (which was indeed the outcome of the game). Had Deep Fritz had the knowledge to detect the true outcome in that blockage position, it would have opted for another variation with greater winning chances.

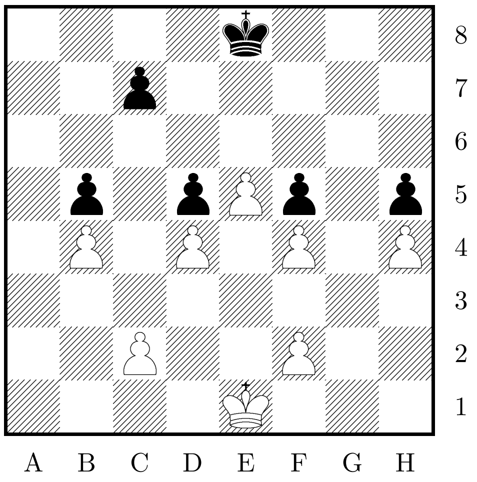

Fig. 2. Despite the material advantage for white, the position is a blockage.

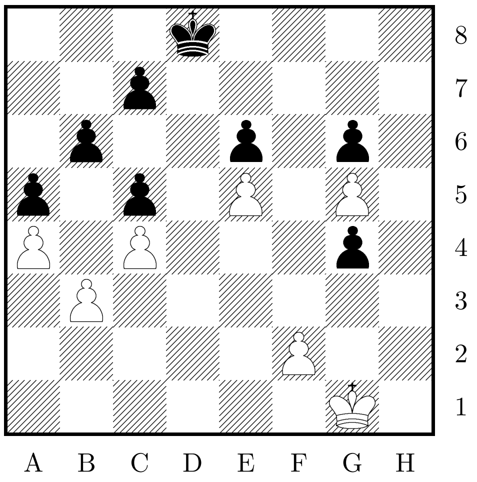

Fig. 3. Deep Fritz vs. Kramnik, Game 1, after white's 28th move.

In this paper we introduce a blockage detection method which manages to detect a large set of such blockage positions in pawn endgames, with practically no additional overhead. The paper is organized as follows: In Section 2 we define blockage, and present a method to detect the presence of blockage on the board. In Section 3 we provide additional criteria to check whether the formed blockage is permanent or can be broken by dynamic pawns. In Section 4 we discuss some practical aspects of blockage detection in search trees. Section 5 contains concluding remarks.

2 Blockage Detection

For the sake of simplicity, throughout this paper we will assume that we are interested in proving that white's upper bound is a draw, i.e., white cannot penetrate the blockage (assuming that black is playing best defense). The process applies similarly for black.

2.1 Applying Blockage Detection within the Search

Figure 4 shows the simplest form of blockage: the pawns form a fence which divides the board into two separate parts, so that white's king cannot attack any black pawns, and there are no mobile pawns on the board. So despite white's material advantage, the game is a draw. A human can easily notice the blockage present. However, the computer's static evaluation will deem the position as better for white, due to the material advantage.

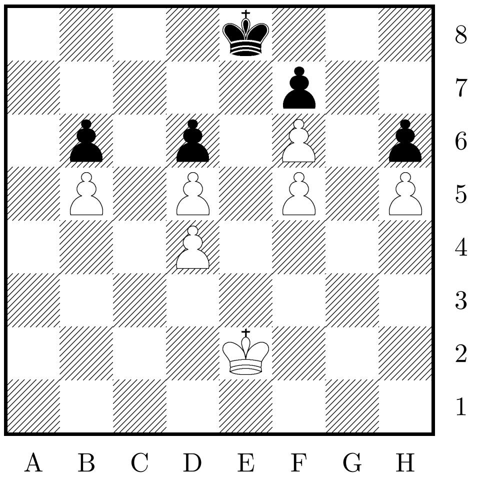

Fig. 4. A simple blockage.

Looking at the position, a human sees that neither side has a mobile pawn, and that the white king has no path to any black pawn. Thus, white cannot achieve more than a draw, so white's upper bound is a draw (i.e., $\beta = DRAW$). Nor does the black king have any path to a white pawn, so black's upper bound is also a draw. Therefore, the outcome is a draw. This thought process could be modeled so that a computer could follow it: If the upper bound of the side to move is greater than the draw score ($\beta > DRAW$), apply blockage detection. In case of blockage, the upper bound will be set to draw ($\beta = DRAW$). In case the lower bound is also greater than or equal to the draw score ($\alpha \geq DRAW$), we cutoff the search at the current node immediately (as $\alpha \geq \beta$). See Figure 5. Blockage detection can also be applied within the search using interior-node recognizers [10, 6].

Thus, blockage detection can be used to prune the search or to narrow the search window. For example, if the program reaches the position shown in Figure 4, it will usually evaluate it as advantageous for white. After applying the blockage detection, it will realize that the true upper bound is a draw, and so will avoid stepping into this position if it can find another truly advantageous position in its search tree.

```
int search(alpha, beta, depth) {
    // blockage detection
    if (beta > DRAW) {
        if (blockage()) {
            if (alpha >= DRAW)
                return DRAW;
            beta = DRAW;
        }
    }
    // continue regular search
    ...
}
```

Fig. 5. Use of blockage detection.

2.2 Marking the Squares

In blockage positions, the most prominent feature is the presence of *fixed pawns* (pawns that cannot move forward or capture a piece) which cannot be attacked. These pawns might form a fence, dividing the board into two disconnected regions (i.e., there is no path from one region to the other). Thus, the fixed pawns in the position should be identified at the first stage of blockage detection.

More formally, we define a *fixed pawn* for white as a pawn which is blocked by a black pawn, or by another white fixed pawn, and cannot capture any piece. A fixed pawn cannot move unless the black pawn which blocks it is captured. We also define a *dynamic pawn* as a non-fixed pawn. For example, in Figure 6 white pawns on a4, c4, e4, and e5 are fixed pawns. Note that even though the g2 pawn cannot currently move, it is not considered fixed according to our definition, and is therefore dynamic.

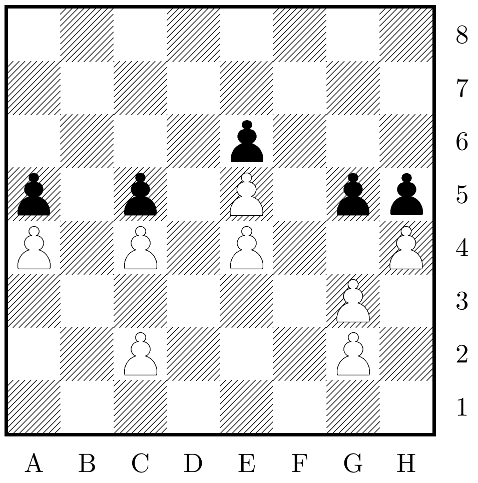

Fig. 6. White's fixed pawns on a4, c4, e4, and e5.

In order to determine whether a blockage is present, we mark the squares on the board which the white king cannot step into, and check whether they form a fence (to be defined below).

These squares can belong to one of the following two classes:

- Squares of white's fixed pawns, such as a4, c3, c4, e3, h3 in Figure 7.
- Squares attacked by black pawns, such as b4, d4, d3, f3, g3, g6 in Figure 7.

Now we have 64 squares, some of which are marked. Next, we check whether the marked squares form a fence. If they do, we have a blockage. Thus, the blockage detection will work as follows:

1. Detect fixed pawns for white and mark them.
2. Mark the squares attacked by black pawns.
3. Apply the fence detection procedure to determine whether the marked squares form a fence.
4. Verify that white cannot penetrate the fence.

2.3 Fence Detection on Marked Squares

We now present a procedure that determines whether the marked squares form a fence. We define a *fence* as a 4-type path [9], i.e., a line of squares, such that each square has a neighboring square either from east, north, west, or south. For

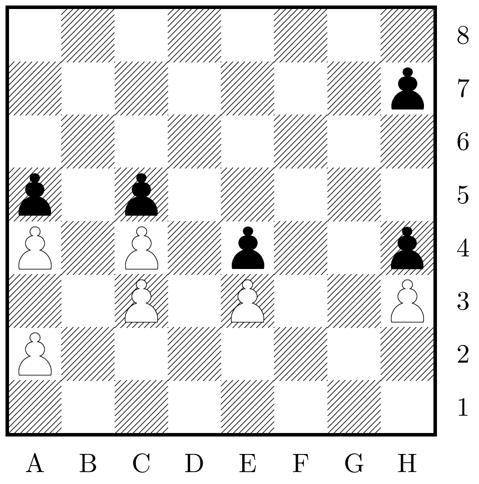

Fig. 7. The marked squares are: a4, c3, c4, e3, h3 (white's fixed pawns), and b4, d4, d3, f3, g3, g6 (attacked by black pawns).

blockage to take place, a necessary condition is that the fence divides the board into two separate regions, such that the kings do not reside in the same region.

The presence of a fence can be detected by means of connected component labeling (see, e.g., Rosenfeld and Kak [9], Haralick and Shapiro [5]). We can view the board as an 8x8 pixel image, where the pixels corresponding to marked squares are taken out of consideration. Connected component labeling will be applied by scanning all the remaining pixels from top to bottom and left to right, and assigning the label of each pixel to the east and south neighboring pixels. If a pixel has no top or left neighbor, then it is assigned a new label. By the end of the process we check whether the pixels corresponding to the squares of the two kings have different labels. If so, then a fence is present. Otherwise, the two kings are in connected regions and so there is no fence present.

However, there exists a more efficient way of detecting the fence by activating a simple depth-first search procedure on the marked squares. We start from the leftmost marked square (which has to be on the A-file). If there are several such marked squares on that file, we choose the topmost square, as we are interested in finding the topmost fence line. Then we consider all its marked 4-neighbors and choose one with the priority of north, east, south. We continue this procedure until reaching the rightmost H-file. If a square in question does not have any neighboring marked squares, we backtrack to the previous marked square. In case we have backtracked to the leftmost file, and there are no marked squares left on that file, this implies that there is no fence present. Figure 8 illustrates this procedure. Note that this procedure can be implemented more efficiently using bitboards, but as the experimental results in Section 4 show, the overhead of the implementation below is already very small.

```
// marked squares have the value "true" and unmarked squares the value "false"
bool marked[8][8];
// squares which are identified as part of the fence, will be set to "true"
bool fence[8][8];
// visited squares will be set to "true"
bool processed[8][8];

// returns "true" if a fence is detected
bool blockage() {
    // find the topmost marked square on the A-file
    for (i = RANK_7; i > RANK_1; i--)
        if (marked[FILE_A][i] && forms_fence(FILE_A, i))
                return true;
    return false;
}

// recursively check whether a fence exists from the current square.
// function arguments are the file and rank of the current square.
bool forms_fence(f, r) {
    processed[f][r]= true;
    // the rightmost file (H-file)
    if (f == FILE_H) {
        fence[f][r]= true;
        return true;
    }
    // check neighbors in the order north, east, south
    // north
    if (marked[f][r+1] && !processed[f][r+1] && forms_fence(f, r+1)) {
        fence[f][r]= true;
        return true;
    }
    // east
    if (marked[f+1][r] && !processed[f+1][r] && forms_fence(f+1, r)) {
        fence[f][r]= true;
        return true;
    }
    // south
    if (marked[f][r-1] && !processed[f][r-1] && forms_fence(f, r-1)) {
        fence[f][r]= true;
        return true;
    }
    return false;
}
```

Fig. 8. DFS fence detection.

After applying the above algorithm, in case the presence of a fence is detected, each square (on file *f*, rank *r*) that belongs to the fence is marked as `fence[f][r]= true`.

In order to ensure that white cannot penetrate the fence, the white king must be below the fence line, and the black king must have at least one vacant square to step into (i.e., black is not in zugzwang). Now, if white has no non-fixed (dynamic) pawns, and black has no pawns below the fence line (that might be captured by the white king), white has no way of penetrating the fence, and a blockage is present. Therefore, the upper bound for white is a draw.

If not all white pawns are fixed, we have to verify that the dynamic pawns cannot break the blockage. This is discussed in the next section.

3 Blockages with Dynamic Pawns

In most blockage positions, in addition to fixed pawns, the side to move has also dynamic pawns. Since as mentioned in the previous section, we are interested only in determining whether one side's upper bound is a draw, we only consider the dynamic pawns of that side (here, we describe the detection methods for white). Assuming that black is not in zugzwang, black's dynamic pawns cannot possibly increase white's upper bound. Additionally, black pawns below the fence might get captured by the white king, and so black cannot rely upon them. Thus, all black pawns below the fence line are removed from the board henceforward (and the white pawns in front of them are considered dynamic). Note that no white pawn should be able to capture a black pawn, as this will change the pawn structure.

Now we will check each dynamic pawn of white to determine whether it can break the blockage. We divide these dynamic pawns into three categories: dynamic pawns above the fence line, dynamic pawns on the fence, and dynamic pawns below the fence line.

3.1 Dynamic Pawns above the Fence Line

A dynamic white pawn above the fence line might be an unstoppable passed pawn (see Figure 9), in which case the result will not be a draw but a win for white. In order to prevent this, we require that the black king be in front of this dynamic pawn (on any square in front of the pawn on that file); see Figure 10. Requiring that the king be in front of the pawn deals with all possible cases of white pawns beyond the fence line. For example, in Figure 11, the black king cannot be on the files B and F at the same time, and so the position will not be deemed a blockage.

3.2 Dynamic Pawns on the Fence Line

Each square in the fence line is either occupied by a fixed white pawn or is attacked by a black pawn (see Section 2). Therefore, if a dynamic white pawn

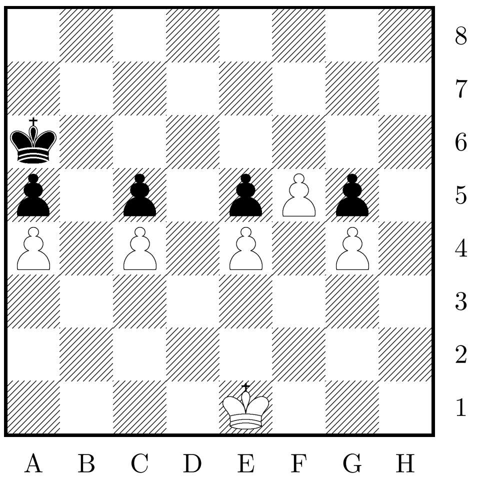

Fig. 9. The fence line is a4-b4-c4-d4-e4-f4-g4-h4. The f5-pawn is above the fence line and cannot be stopped by the black king.

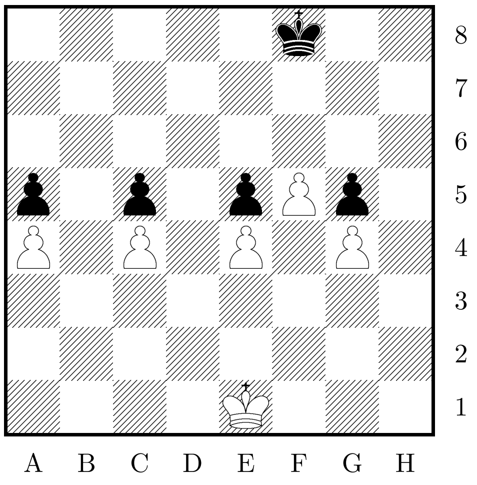

Fig. 10. The fence line is a4-b4-c4-d4-e4-f4-g4-h4. The f5-pawn is stopped by the black king which is in front of the pawn.

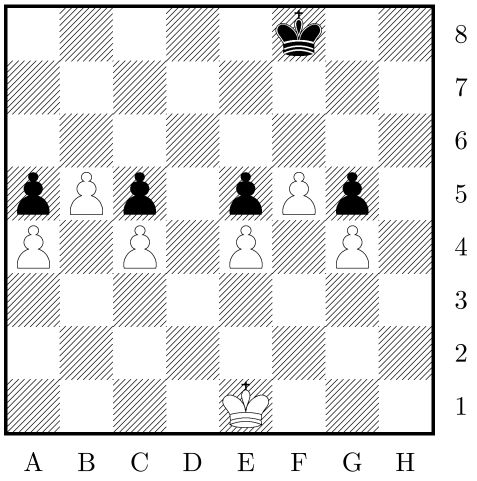

Fig. 11. The fence line is a4-b4-c4-d4-e4-f4-g4-h4. Black king cannot stop both pawns.

is on the fence line, that square is attacked by a black pawn (see Figure 12). In order to prove that this dynamic pawn cannot remove the blockage, we have to show that this dynamic pawn cannot penetrate the fence by capturing the black pawn, nor can it turn into a passed pawn by moving forward.

To satisfy both these points, the presence of a black pawn two squares in front of this dynamic pawn is required. For example, in Figure 12 black has a pawn two squares in front of white's f4-pawn (on f6). This ensures that the blockage will not be removed in any of the following two ways: if the f4-pawn captures a black pawn by either 1.fxg5 or 1.fxe5, black will recapture by 1...fxg5 or 1...fxe5 respectively, and the fence line will remain intact; and if the pawn moves forward by 1.f5, it will turn into a fixed pawn blocked by black's f6 pawn (and the new fence line will pass through e5-f5-g5 in that section).

Additionally, this dynamic pawn should be white's only pawn on that file, so that after an exchange white is not left with another pawn in that file which can move forward and break the blockage; and the square in front of it should not be attacked by a black pawn, so that moving this pawn forward will turn it into a fixed pawn. Figure 13 illustrates a position in which lack of these requirements results in the removal of the blockage.

3.3 Dynamic Pawns below the Fence Line

A dynamic pawn below the fence line can move forward until it either:

- gets blocked by a white pawn (e2-pawn in Figure 14 gets blocked by the e4-pawn),
- engages a black pawn below the fence line (b2-pawn in Figure 14 engages the black c4-pawn after reaching b3), or
- reaches the fence line (g2-pawn in Figure 14 reaches g4).

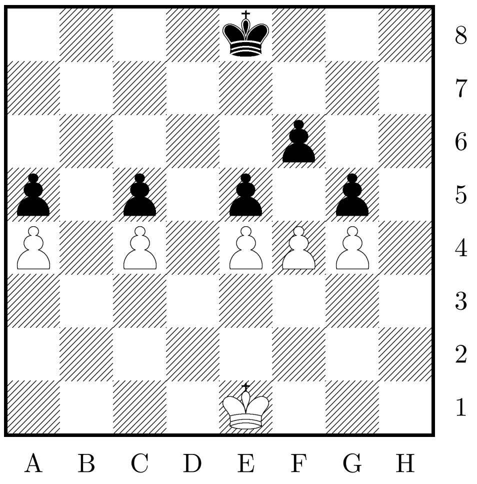

Fig. 12. The fence line is a4-b4-c4-d4-e4-f4-g4-h4. White's f4-pawn is on the fence line.

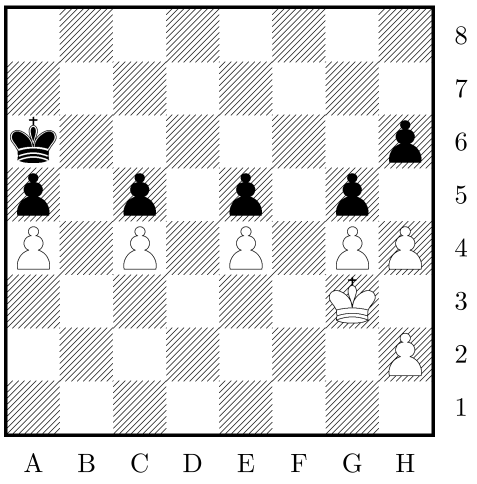

Fig. 13. Since the h4-pawn is not white's only pawn on the H-file, white can break the blockage by 1.hxg5 hxg5 2.h4.

The latter case (dynamic pawn on the fence line) was already discussed in the previous subsection (note that also when a pawn passes the fence line by doing a 2-square move from the second rank, the position still remains blocked, provided that the conditions described in the previous subsection are met). In the first case, it is clear that when the pawn is blocked by a friendly pawn before reaching the fence line, it cannot break the blockage. In the second case, when engaging a black pawn, the pawn structure can change and so the blockage might be removed.

Thus, we start scanning the squares in front of the pawn, until we either reach a white pawn, or a square attacked by a black pawn. In case we stop at a white pawn (e.g., the e2-pawn in Figure 14 which is stopped by the pawn on e4), the pawn will not break the blockage. In case we stop at a square attacked by a black pawn, and this square is below the fence line (e.g., the b2-pawn in Figure 14 which stops at b3), the pawn structure might change as a result of a pawn capture, and so we do not deem the position a blockage (the blockage detection immediately returns "false"). Otherwise, the pawn reaches the fence line (covered in Subsection 3.2).

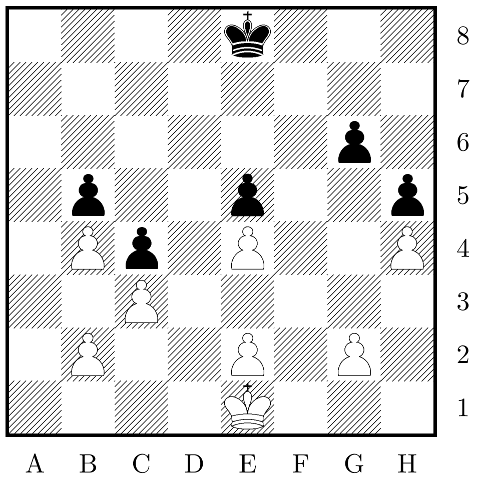

Fig. 14. Different types of dynamic pawns at b2, e2, and g2. The fence line is a4-b4-c4-d4-e4-f4-g4-h4.

3.4 An Exception

A frequent pattern that appears in many blocked positions is illustrated in Figure 15. While not all the criteria mentioned in the previous subsections are satisfied, the position is still a blockage. White's f2-pawn can reach the fence line (at f4) which is attacked by a black pawn. However, there is no black pawn on f6 to defend the fence (as required in Subsection 3.2; see Figure 12). In this

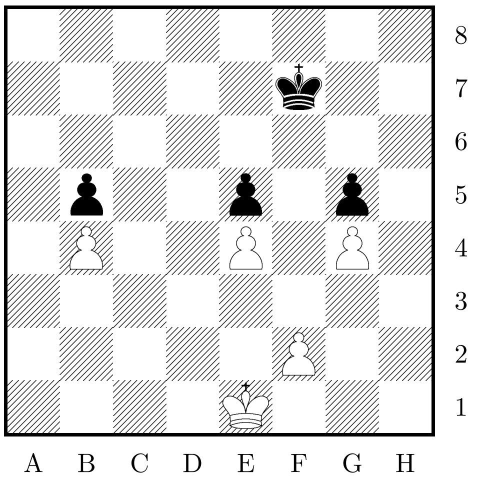

Fig. 15. The fence line is a4-b4-c4-d4-e4-f4-g4-h4. Although white's f2-pawn can reach f4, and a black pawn is not present at f6, white still cannot break the blockage.

pawn structure the blockage will not be broken if the black king is on the dynamic pawn's file, and there is exactly one white pawn on each of the neighboring files, when these pawns are

- fixed pawns,
- on the fence line, and
- both on the same rank, which is below the sixth rank.

In Figure 15 all these conditions hold true:

- black king is on the f2-pawn's file, i.e., the F-file
- exactly one white pawn exists on each of the neighboring files (e4 and g4 pawns on files E and G)
- both the e4 and g4 pawns are fixed pawns
- both the e4 and g4 pawns are on the fence line
- both the e4 and g4 pawns are on the fourth rank, which is below the sixth rank

If all these requirements hold true, even though white might be able to break the blockage by moving the dynamic pawn to the fence line, its upper bound will still remain a draw, since black will capture the pawn and will have a supported passed pawn (e.g., black's f4-pawn after gxf4 in our example), and white's new passed pawn (on g4) will be stopped by the black king.

4 Analysis

In this section we discuss some practical aspects of blockage detection. We have implemented the blockage detection method described in Sections 2 and 3 in the

FALCON engine. FALCON is a fast yet sophisticated grandmaster-strength chess program. Blockage detection was incorporated as described in Figure 5. This was done in all the nodes.

Adding the blockage detection procedure at each node of the search tree might seem costly. However, in practice it incurs almost no overhead. The blockage detection is called only if there are no pieces on the board apart from king and pawns. Additionally, the already visited pawn structures are stored in a pawn hash table. Thus, the costly DFS fence detection is performed once for each pawn structure, and retrieved later from the hash table.

In order to measure the performance of the blockage detection method we used a database of 3118 positions where each side had at least six pawns on the board and no additional pieces, and the game ended in a draw (see Appendix). It is clear that when in addition to pawns there are other pieces on the board, or when there are only a few pawns on the board, blockage detection will either not be triggered at all, or will take a negligible time. So, in order to measure the speed overhead in worst case, we used positions where each side has at least six pawns on the board, since in these positions there is a higher potential for blockage detection. We used two identical versions of FALCON, one employing blockage detection, and the other having it turned off. We let each version analyze each position for 10 seconds. Table 1 provides the results. The results show that blockage detection has only a 3% speed overhead in this extreme worst case. We can further see that 446,824 times in our search, the blockage detection returned "true". From the total of 3118 positions, the search on 895 of them returned the draw score (0.00) when using blockage detection, while only 569 returned that score without blockage detection being employed.

Blockage Det.	Nodes	Blockages	Draws	Speed (NPS)	Overhead
On	7,086,119,946	446,824	895	227,264	3%
Off	7,323,190,952	0	569	234,868	-

Table 1. Total number of nodes searched, blockages detected, positions returning the draw score, nodes per second speed rate, and the speed overhead of blockage detection. Number of positions: 3118. Analysis time: 10 seconds per position.

The results show that applying blockage detection has practically no additional overhead. Whenever there are other pieces in addition to pawns on the board, blockage detection is not triggered at all. In pawn endgames, blockage detection is applied to each pawn structure only once, since the result is hashed, so the cost is negligible. And even when there are many pawns on the board and they can form many pawn formations in our search, the overhead of blockage detection will still be at most about 3%, as our results indicate. Even there, the small cost is negligible in comparison to the benefit coming from blockage detection. As our results show, the version using blockage detection managed to detect 326 draws more than the version without blockage detection.

The blockage detection knowledge contributes to the search in many ways. When the program evaluates a position totally unrealistically, this can wreak havoc to the whole search tree, leading the program into playing a blunder. Figure 16 is an example of a position in which only thanks to blockage detection knowledge, black manages to avoid a loss, and instead draw the game. The only correct move for black is 1...Rxc4, which results in a blockage. While Falcon chooses this move instantly and displays the draw score together with the principal variation of 1...Rxc4 2.dxc4 g6, all top tournament playing programs we have tested choose other moves which result in a loss for black (see Appendix). After 1...Rxc4 2.dxc4 g6 the position is a blockage: the fence line is a4-b4-c4-d4-d5-e5-f5-g5-h5, and none of white's dynamic pawns (b2, e2, and h4) can break the blockage (according to the criteria mentioned in the previous section). Thus, white's upper bound is a draw.

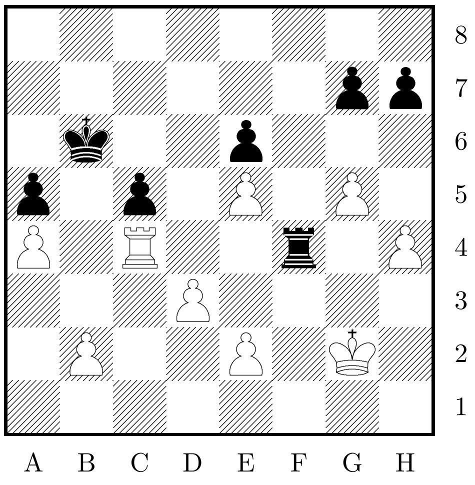

Fig. 16. Black to move. 1...Rxc4 results in a draw, all other moves lose.

In other positions, the lack of blockage detection knowledge might cause the program to trade its advantage for a "greater advantage", which is in fact a mere draw. This was the case in the first game of Deep Fritz vs. Kramnik in their Bahrain match. After the 24th move the position illustrated in Figure 17 arose on the board. Here Deep Fritz played 25.h4, which after 25...hxg4 26.Bg5 Bxg5 27.hxg5 resulted in the position illustrated in Figure 3. After white captures black's g4 pawn the position is evaluated as advantageous for white, but in fact it is a blockage, and so a draw. All the programs we have tested again fail to see the draw, except Falcon which includes the blockage detection knowledge. Had Deep Fritz had the knowledge to detect the blockage, it would have chosen another variation with greater winning chances.

In addition to helping avoid blunders, blockage detection can also serve as a safe pruning method for pawn endgames. Since we apply blockage detection in all

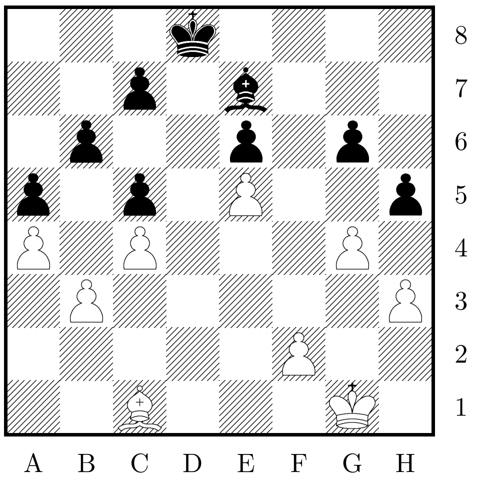

Fig. 17. Deep Fritz vs. Kramnik. Here white played 25.h4 resulting in a draw after 25...hxg4 26.Bg5 Bxg5 27.hxg5.

the nodes, whenever a blockage occurs in the search tree, we know it immediately and so can usually cutoff the search at once. Thus, the program will spend its time searching other moves deeper, avoiding a waste of resources for blockage positions.

5 Conclusion

In this article we have introduced a blockage detection method which manages to detect a large subset of blockage positions. Our method provides just as much knowledge needed for the search to detect the blockage. For example, if in Figure 3 we place the black king on a6 and the white king on g4 (see Figure 18), in our search we will see that the black king manages to reach the f-file just in time to stop a white penetration of the fence with f4-f5. But if we further move the white king to g3 and the white pawn to f4 (see Figure 19), then white will win by 1.f5. That will be noticed in the search, as the black king will not reach the f-file in time to satisfy the conditions mentioned in subsection 3.4.

Lack of blockage detection knowledge has always been a blind spot of even top tournament playing programs. Our presented blockage detection method enables the programs to evaluate such positions realistically, and avoid the blunders which result from a lack of blockage detection knowledge. This method can be incorporated in all the nodes of the search tree at a negligible cost. This enables the program to detect the blockage instantly, and narrow the search window or cutoff the search at the node immediately.

More criteria can be added to the blockage detection method, resulting in detection of more blockage positions. However, a position wrongly labeled as a

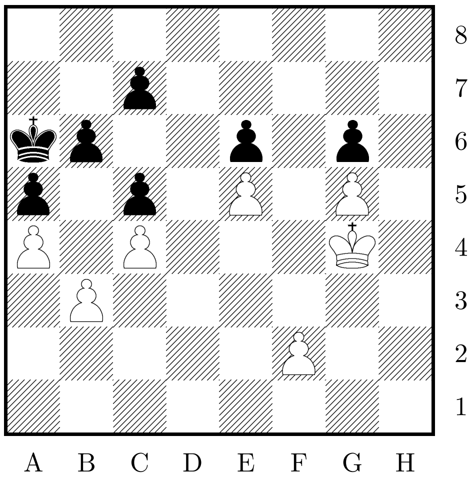

Fig. 18. Black king manages to reach the f-file in time to stop an f4-f5 penetration.

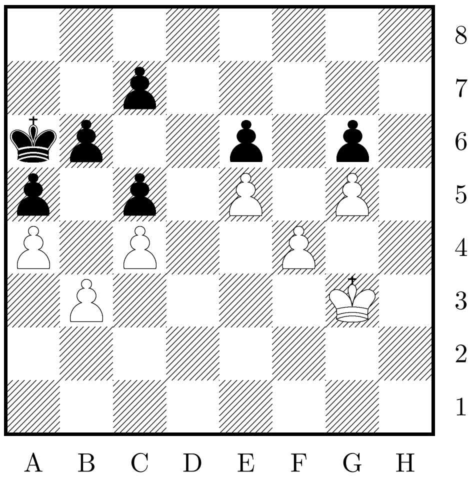

Fig. 19. 1.f5 wins.

blockage might direct the search into an incorrect direction, and so the additional criteria must prove to be correct in all the positions.

Appendix

	Junior 8	Fritz 8	Shredder 7.04	Chess Tiger 15	Hiarcs 8	Falcon
Score	+3.42	+2.91	+5.54	+2.36	+3.82	0.00
Depth	41	28	31	30	25	2

Table 2. Analysis of Figure 2 position. Analysis Time: 30 minutes. Falcon instantly detects the blockage at a depth of 2 plies, and displays the draw score of 0.00. Scores are from white's point of view.

	Junior 8	Fritz 8	Shredder 7.04	Chess Tiger 15	Hiarcs 8	Falcon
Score	+0.35	+0.84	+0.43	+0.66	+0.73	0.00
Depth	41	30	33	29	25	2

Table 3. Analysis of Figure 3 position. Analysis Time: 30 minutes. Falcon instantly detects the blockage at a depth of 2 plies, and displays the draw score of 0.00. Scores are from white's point of view.

	Junior 8	Fritz 8	Shredder 7.04	Chess Tiger 15	Hiarcs 8	Falcon
Move	1...Rf5	1...Rf7	1...Rf5	1...Rf7	1...Rf7	1...Rxc4
Score	+2.42	+2.06	+3.64	+2.46	+2.53	0.00
Depth	24	19	22	21	18	3

Table 4. Analysis of Figure 16 position. Analysis Time: 60 minutes. Falcon instantly detects the correct move leading to the blockage at a depth of 3 plies, and displays the draw score of 0.00. Scores are from white's point of view.

All the programs were run using ChessBase Fritz 8 GUI, with 64MB of hash table, running on a 733 MHz Pentium III system with 256MB RAM, using the Windows XP operating system.

The positions used in Table 1 experiment were extracted by searching a database for games where at some point both sides had at least six pawns on the board and no additional pieces, and the game ended in a draw. From each of the 3118 games found, we chose the first position where both sides had at least six pawns on the board and no additional pieces.

Acknowledgements

We would like to thank Dieter Buerssner for providing the database of games for our empirical studies. We would further like to thank the anonymous referees for their valuable remarks and suggestions.

References

1. Berliner, H. and Campbell, M. (1984). Using chunking to solve chess pawn endgames. *Artificial Intelligence*, Vol. 23, No. 1, pp. 97–120.
2. Church, K.W. (1979). Co-ordinate squares: A solution to many pawn endgames. *IJCAI 1979*, pp. 149–154.
3. de Groot, A.D. (1965). *Thought and Choice in Chess*. Mouton, The Hague.
4. Fine, R. (1941). *Basic Chess Endings*. Random House, 2003, ISBN 0-812-93493-8.
5. Haralick, R.M. and Shapiro, L.G. (1992). *Computer and Robot Vision*. Addison-Wesley, 1992, ISBN 0-201-10877-1.
6. Heinz, E.A. (1998). Efficient interior-node recognition. *ICCA Journal*, Vol. 21, No. 3, pp. 156-167.
7. Muller, K. (2002). The clash of the titans: Kramnik – FRITZ BAHRAIN. *ICGA Journal*, Vol. 25, No. 4, pp. 233–238.
8. Newborn, M. (1977). PEASANT: An endgame program for kings and pawns. *Chess Skill in Man and Machine*, (Ed. P.W. Frey), pp. 119–130. Springer-Verlag, New York, N.Y., 2nd ed. 1983, ISBN 0-387-90790-4/3-540-90790-4.
9. Rosenfeld, A. and Kak, A. (1982). *Digital Picture Processing*. Morgan Kaufmann, 2nd ed. 1982, ISBN 0-125-97301-0.
10. Slate, D.J. (1984). Interior-node score bounds in a brute-force chess program. *ICCA Journal*, Vol. 7, No. 4, pp. 184–192.