Handling Search Inconsistencies in MTD(f)

Jan-Jaap van Horssen[1]
February 2018

Abstract

Search inconsistencies (or search instability) caused by the use of a transposition table (TT) constitute a well-known problem for efficient algorithms for minimax searching such as MTD(f). This paper presents an effective and efficient solution to the problem of MTD(f) playing occasional blunder moves as a result of search inconsistencies. This makes MTD(f) safe to use, even without clearing the TT.

Introduction

Search inconsistencies caused by the use of a TT constitute a problem for the most efficient algorithms for minimax searching, Aspiration PVS/NegaScout and MTD(f), that is not addressed in the textbook versions of these algorithms. Ways to handle search inconsistencies in Aspiration PVS/NegaScout can be found online [1], and involve re-searches with a widened (α, β) window. However, these solutions do not apply to MTD(f) and to the best of our knowledge no solutions are known for MTD(f). In [2], search inconsistencies are mentioned as an open problem and suggested for future work.

Search Inconsistencies in MTD(f)

MTD(f) [3] was implemented in the author's 10x10 draughts engine MAX. Reproducible situations were found where MTD(f) plays a wrong move, with and without clearing the TT, for instance a drawing move instead of a winning move. These situations can be prevented by

1. clearing the TT before the search, and
2. only allowing the use of TT entries with an exact depth match (not with a greater depth).

Both are needed to avoid all problems, but both are also considered wasteful. In addition, if a program is to be used in the context of machine learning (ultrafast games), clearing the TT in between moves simply takes too much time.

So MTD(f) needs a way to handle these situations. To address the problem of search inconsistencies, we first make the move selection mechanism in MTD(f) explicit, see Algorithm 1. The function *MT*[2] is unrolled for the root node, giving *rootMT*. This function does not use the TT but assigns a value to each searched move. After a cutoff, the remaining moves get a value of −∞.

In the context of MTD(f), a *search inconsistency* arises if a search with window $(\gamma - 1, \gamma)$ returns a value $\geq \gamma$ and a subsequent search with (for instance) window $(\gamma, \gamma + 1)$ returns a value $< \gamma$. Or,

[1] E-mail: janjaapvanhorssen@gmail.com
[2] MT(γ) is equivalent to alphabetaTT($\gamma - 1, \gamma$).

conversely, if a search with window $(\gamma - 1, \gamma)$ returns a value $\leq \gamma - 1$ and a subsequent search with (for instance) window $(\gamma - 2, \gamma - 1)$ returns a value $> \gamma - 1$.

```
*  // post: best move is first in 'moves'
*  int MTDf(Board root, MoveList moves, int f, int d) {
       g = f;
       low = −INF; upp = +INF;
       do {
          gamma = g == low ? g + 1 : g;
*         g = rootMT(root, moves, gamma, d);       // post: moves have bound values
          if (g < gamma) upp = g; else low = g;
*         moves.sort();                            // stable sort descending
       } while (low < upp);
       return g;
   }
```

Algorithm 1. MTD(f) with move selection code (marked by an asterisk and italics).

MTD(f) makes a series of calls to rootMT with an adjusted window for given depth d, yielding a sequence of values $g_1, \ldots, g_k$ $(k \geq 2)$. If there are no search inconsistencies then $g_{k-1} = g_k =$ the minimax value for depth d. The converse is not true: if $g_{k-1} = g_k$ there can still be search inconsistencies. There are two cases:

1. The lass pass failed high. Then all preceding passes must have failed low. We have $g_1 > \cdots > g_{k-1} \leq g_k$.
2. The lass pass failed low. Then all preceding passes must have failed high. We have $g_1 < \cdots < g_{k-1} \geq g_k$.

Case 1. The last pass failed high, so we have a lower bound for the value. If, as expected, $g_{k-1} = g_k$ (and *low = upp*) then it is assumed that we found the minimax value and the best move is first in the list. If, unexpectedly, $g_{k-1} < g_k$ (and *low > upp*: the lower bound overshot the upper bound), then we have a search inconsistency. Let move m_1 be first in the list after sorting. We know from the second-last pass that the values of all moves were $\leq g_{k-1}$, but now we find that the value of m_1 is $\geq g_k > g_{k-1}$. Moreover, the search was cut off (failed high) after searching m_1, so the remaining moves were not searched to find an even better move. In this case it is unlikely that one of the other moves is (much) better than m_1 because it is likely that their values are still roughly $\leq g_{k-1}$. Because we have a lower bound on the value of m_1 (which is better than expected) and we are maximizing, the search inconsistency is not considered harmful in this case.

Case 2. The last pass failed low, so the second-last pass failed high, returning a value of $\geq g_{k-1}$. Let move m_1 be first in the list before the last pass. The search window for the last pass was $(g_{k-1}, g_{k-1} + 1)$ which fails low. MTD(f) expects a value of $g_k = g_{k-1}$ so it can conclude that the minimax value is g_k. Due to search inconsistencies, however, also a value of $g_k < g_{k-1}$ is possible (giving *low > upp*: the upper bound undershot the lower bound). In *either case*, suppose that after sorting the last pass yields a new best move m_2 (see Figure 1). This implies that m_1 failed low with value $< g_{k-1}$, which is a search inconsistency. Move m_2 also failed low, but with value g_k. So m_2 is preferred over m_1, even though we only have *upper bounds* for m_1 and m_2. This does not make sense for a maximizing player. If this happens in the last completed iterative deepening iteration (depth) before we run out of time, *MTD(f) will play a wrong move*. The minimax value of this move can be *any value* $\leq g_k$. So the move can be anything between a good (alternative) move and a blunder. This happens very rarely and is not easy to reproduce, but nevertheless it is the reason why some programmers decide not to use MTD(f) anymore.

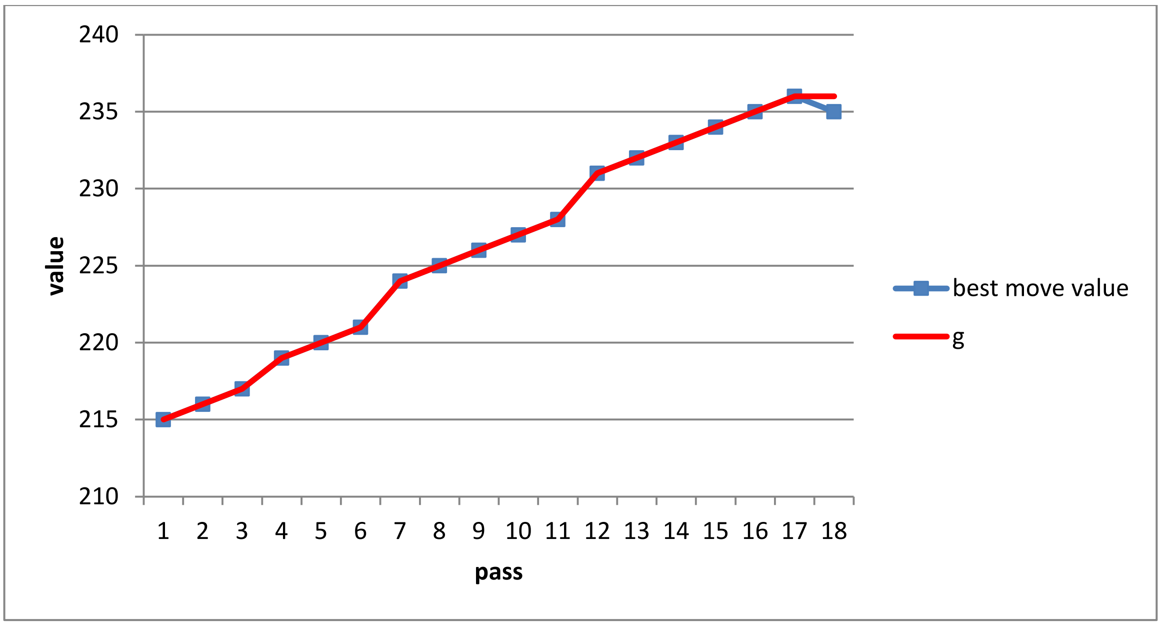

Figure 1. Typical case 2 sequence of values $g_1, \dots, g_k$ in a winning position. Even if $g_{k-1} = g_k$, a search inconsistency can arise if the best move after pass $k-1$ fails low with a value $< g_k$ in pass k.

We propose the following solution. It involves no re-searches and imposes no additional overhead on MTD(f). The above case 2 is easy to detect after the last pass has finished, either with the expected *low = upp* or the search inconsistency *low > upp*. If the last pass failed low and the best move has changed, we do not trust this move and its value (only an *upper bound*), but instead return the best move and value of the previous pass (a *lower bound* after $k-1$ passes), which we saved beforehand. The version of MTD(f) with the proposed adaptation is called *MTDfix*, see Algorithm 2.

Performance

MTDfix was implemented in MAX.[3] It correctly solves (plays the right move in) the reproducible test cases where MTD(f) plays a wrong move. Also it performed 2.7% faster on a benchmark of 49 successive white-to-move positions from a single game, which were searched with iterative deepening to specified depths. This indicates that MTDfix improves move ordering in the root, i.e., the best move is more often first in the list.

To further test the algorithm, a match of 988 games (3-move ballots with both colors) was played between MTDfix and MTD(f), both using 0.1 second per move. Neither program clears the TT before the search.

MAX-MTDFIX vs MAX-MTD(F): 97 wins, 94 losses, 797 draws, score 50.2%

[3] MAX is a developing program, currently single-thread and without the use of forward pruning techniques.

```
    // post: best move is first in 'moves'
    int MTDfix(Board root, MoveList moves, int f, int d) {
      g = f;
      low = −INF; upp = +INF;
      do {
*       bestMove = moves.getFirst();
*       bestValue = g;
        gamma = g == low ? g + 1 : g;
        g = rootMT(root, moves, gamma, d);          // post: moves have bound values
        if (g < gamma) upp = g; else low = g;
        moves.sort();                               // stable sort descending
      } while (low < upp);
*     if (g < gamma && bestMove != moves.getFirst()) {
*       moves.putFirst(bestMove);
*       g = bestValue;
*     }
      return g;
    }
```

Algorithm 2. MTDfix. The lines added to MTD(f) (see Algorithm 1) are marked with an asterisk.

There is no significant increase in playing strength, but we can study the situations of a fail-low in the last pass together with a change of the best move *in the last completed iteration* (depth), i.e., directly affecting the move played. These situations were also detected −but not fixed− in MTD(f). For MTD(f) it occurred 77 times in 70 (out of 988) games, in which MTD(f) scored 70.0%. For MTDfix it occurred 100 times in 70 games, in which it scored 82.9%. In five games the situation occurred in both programs, in which MTDfix scored 60%. The latter five games were analyzed and indeed some blunders by MTD(f) were found. The situation occurred on average in 1 : 627 moves and in 1 : 14 games.

Looking at the score in games where the problem occurs, we see that the score is $\gg$ 50%, even if we don't fix the problem. This indicates that it mainly happens in games where a big advantage is reached, i.e., with increasing search values. The results indicate that MTDfix does better in actually winning most of these games, where MTD(f) may spoil a number of games by playing wrong moves.

To verify this hypothesis, a series of 158-game matchess (2-move ballots with both colors) was played between different versions of MTD(f) with 0.1 second per move and

1. an *equal* opponent: a version of MAX using PVS (without aspiration).
2. a *weaker* opponent: BOASE, participant in the 2011 Dutch Open computer tournament (and improved since then). BOASE was given 1 second per move to raise the level a bit.
3. a *stronger* opponent: SCAN 3.0, winner of the 2017 ICGA Computer Olympiad. SCAN was given 1 second per move to give it an advantage. SCAN was modified to disable time management.

For the (only) version of MTD(f) that clears the TT before every search, the time to clear the TT was not included in the thinking time. For tournament games this time is negligible but for fast games is it not, and we want to study the effect of a clean TT. The tournament results are shown in Table 1.

program	sec		program	sec	opponent	games	win	loss	draw	%	detected
Max-MTDfix	0.1	vs	Max-PVS	0.1	equal	158	18	15	125	**50.9**	29
Max-MTD(f)	0.1	vs	Boase 5.4	1	weaker	158	119	0	39	87.7	83
Max-MTD(f) clearTT	0.1	vs	Boase 5.4	1	weaker	158	132	0	26	91.8	7
Max-MTDfix	0.1	vs	Boase 5.4	1	weaker	158	143	0	15	**95.3**	197
Max-PVS	0.1	vs	Boase 5.4	1	weaker	158	116	0	42	86.7	-
Max-MTD(f)	0.1	vs	Scan 3.0	1	stronger	158	0	90	68	21.5	6
Max-MTD(f) clearTT	0.1	vs	Scan 3.0	1	stronger	158	0	94	64	20.3	4
Max-MTDfix	0.1	vs	Scan 3.0	1	stronger	158	0	86	72	**22.8**	0
Max-PVS	0.1	vs	Scan 3.0	1	stronger	158	0	85	73	23.1	-

Table 1. Tournament results.

The results of 158-game matches should be interpreted with some care as they have a larger margin of error, for instance 95.3±2.3% and 22.8±3.9%, versus 50.2±1.4% for the 988-game match (95% confidence intervals). Nevertheless we can draw some conclusions from the results.

- In the match Max-MTDfix vs Max-PVS the situation was detected 29 times in 15 out of 158 games, with up to five times in one game. MTDfix scored 14 wins and 1 draw in these games. The 15 lost games were analyzed and no blunders were found.
- Not clearing the TT and ignoring the problem increases the probability of a wrong move. Clearing the TT prevents most situations where MTD(f) chooses a wrong move, but not all. In addition, this takes time and useful information is lost.
- MTDfix outperforms MTD(f) and MTD(f)-clearTT, most clearly against a weaker opponent, where there are lots of winning opportunities. Here we see a significant increase in playing strength. Against a stronger opponent the situation occurs far less frequently, and not much is to gain.
- PVS seems to do relatively better against a stronger opponent, i.e., MTD(f) relatively deals less well with decreasing search values. This behavior has been reported before.

Conclusion

We conclude that the problem of MTD(f) playing a wrong move is rare but real. The weaker the opponent, the higher the probability of obtaining a winning position, and the higher the probability of a wrong move. MTDfix is an effective and efficient solution to this problem, which also removes the need to clear the TT before the search. It appears that the problem of occasional blunders by MTD(f) is solved, and it is safe to use the algorithm in tournaments and machine learning, without clearing the TT.

Future Work

Future work will include more experimental support of the conclusions.

References

[1] https://chessprogramming.wikispaces.com/*Search+Instability*

[2] Plaat, A. (1996). *Research re: Search & re-search* (Doctoral dissertation, Thesis Publishers).

[3] https://askeplaat.wordpress.com/534-2/mtdf-algorithm/