Trappy Minimax in chess, and the elusive meaning of "best move"

V. Scott Gordon
CSU Sacramento
gordonvs@ecs.csus.edu

Michael Vollmer
CSU Sacramento
eblivion@gmail.com

Colin Frayn
www.frayn.net
colin@frayn.net

Abstract

Trappy minimax is a modification to minimax that attempts to identify and set traps in two-player strategy games. It defines a trap as a move that looks good at a shallow search depth, but is revealed to be in fact a bad move, when analyzed at a greater search depth. The algorithm utilizes information accumulated during iterative deepening, and then rewards moves that lead to traps. The reward mechanism often causes the computer to select moves that normal minimax considers inferior, because minimax assumes that the opponent always makes the best response, whereas trappy minimax identifies those inferior moves that an opponent might be tempted to play. The algorithm challenges the fundamental notion of "best move", as shown in examples from actual games. Trappy minimax is tested in the game of chess by incorporating it into the Beowulf chess engine. The trappy version of Beowulf earned a significantly higher rating on ICC than did normal Beowulf, against human opponents, but not against other computers. The results illuminate a number of advantages for not always assuming best opponent response.

Introduction

At the core of nearly all computer board game play is the *minimax* algorithm. And at the core of minimax is the assumption that searching for the best available move necessitates assuming that the opponent will also make the best response. While there is no denying the competitive success of this tenet, it carries with it several implications deserving of further consideration.

When humans play, we rarely limit our analysis to the best response. Rather, humans routinely try to trick the opponent into playing weak moves. In many cases we are willing to risk playing a slightly inferior move if we believe there is a good chance that the opponent will make the wrong reply and fall for a trap. Thus, human play includes a good deal of bluff that is inherently absent from minimax. A deeply-laid and difficult to navigate trap will never be set by minimax, if the perfectly-accurate response costs the computer one-tenth of a pawn according to the static evaluation function. In this sense, human play is a richer cornucopia of possibilities that encompasses not only the search for a perfect truth, but also other ways of extracting victory.

An obvious example is when we are behind – i.e., when desperate circumstances dictate desperate measures. If we find ourselves behind by, say, a couple of pawns, we know that defeat is certain unless we get lucky. Knowing that, we humans will try everything in our power to increase the chance for that lucky break. When behind, there is little harm in sacrificing even more material if it sets some landmines for our opponent to navigate. Humans do this naturally, yet minimax will generally discard such moves as inferior, simply assuming that the opponent will not fall for the bait.

Thus, despite the considerable advances in computer game play – to the point of utter dominance in chess – humans still bemoan that their play is still "computer-like". A recent Andy Soltis *Chess Life* column highlights the startling lack of any apparent plan in many computer chess games, such that the very notion of planning in chess may in reality turn out to be a mirage [Sol13]. But there remains a strong sense that the human style of play remains fundamentally different than that of the computer, and that the difference is visible in the games themselves. And while the computer is the undeniable master of opportunity – if a victory is within its horizon, it *will* find it – minimax includes nothing to create such opportunities.

Indeed, so ingrained is our trust in minimax, that we rarely explore the original question: what exactly is the "best move" in a given position? Consider the following scenarios:

(i) Players A and B are approaching the time control, and player A only has a few seconds left on his clock. Player B has two moves to choose from – the first leads to a series of captures with forced recaptures that lead to an even game. The second initiates a complex attack that requires pinpoint accuracy for player A to survive, but if he does, he will have a slightly better pawn structure. Given player A's time pressure, many humans would consider the second plan to be the preferable one. Minimax will always favor the first move.

(ii) Player A is behind by a piece, and player B is trying to steer the game into a decisive endgame. Player A has to choose between two moves: the first leads to an endgame in which he is still behind by a piece, but in a losing bishop-and-pawn endgame. The second leads to the sacrifice of a second piece, but the elimination of all of the pawns, such that player B would be required to execute the difficult bishop-and-knight mate in order to win. Many humans would make player B's life as difficult as possible, and choose the second move. Minimax would always choose the first move because the static score is higher.

(iii) Player A is about to be checkmated in at most 8 moves. He has several moves, all of which result in loss, given best play. One of the options leads to a loss in only 6 moves, but if the opponent doesn't find the correct move, there is a trap and player A can win. Humans would recognize that if they are going to lose anyways, they might as well give the opponent the opportunity to mess up, and choose the 6-move loss because it includes a chance of victory. Minimax always chooses the 8-move loss because it is the slowest, even if it is also the easiest for the opponent to execute.

(iv) Player A is a computer, and is playing against human player B. The two combatants are in an endgame, and the computer (player A) has analyzed every variation to completion. That is, the computer is working with perfect and complete information. Every move leads to a draw with best play. However, one of the moves requires the human to find 6 subtle moves in succession, two of which require the sacrifice of material, and any deviation from this sequence results in a victory for the computer. Humans would no doubt say that this is the best move for the computer to choose. But the computer would consider this move to be no better than any of its other possible moves, and thus would be unlikely to play it – even though it sees the variation that the human would have to find.

Thus, it is argued herein that, while minimax succeeds in its quest for truth and purity, it can hardly be said to produce the "best" move in every case, *even in the presence of an exhaustive search tree!* This latter case gives rise to the observation that, while minimax in its current form fails to exploit options such as those listed above, the algorithm often does in fact traverse those options, and thus need only be modified slightly, into a framework that can reward them.

Background

The *minimax* algorithm is generally attributed to John Von Neumann in 1928 [vonN28], and its applicability for computer chess to Claude Shannon in 1950 [Shan50]. The optimization technique known as *alpha-beta* pruning was described in 1962 by Kotok [Kot62], allowing the algorithm to eliminate certain branches from consideration, speeding up the search without changing the result. Minimax with alpha-beta pruning has been used in most computer implementations of popular board games such as chess and checkers, and forms the basis of the search algorithms used in well-known competitive programs such as Deep Blue (for chess) [Hsu02], Chinook (for checkers) [Sch09], and Logistello (for Othello) [Bur97].

Various improvements to minimax have been developed. Most attempt to alter the ordering of generated moves in order to accelerate alpha-beta pruning [Sch89]. Others use game-dependent knowledge to determine whether certain branches of the tree should be ignored, allowing for deeper search, or whether certain branches should be selected for deeper search.

Iterative Deepening is when a minimax search of depth N is preceded by separate searches at depths 1, 2, etc. That is, a series of progressively deeper searches are performed, enabling minimax to search as deeply as it can in the time allotted, without knowing ahead of time how far ahead it has time to search. The results of the shallower searches can also used to help alpha-beta pruning work more effectively during deeper searches [Sch89]. Trappy minimax also uses iterative deepening to accumulate information on how strong a move may *appear* [GR06].

Efforts at modifying minimax so that it induces an opponent into making an error have focused on identifying an *opponent model*. Pearl examined trap-setting in probabilistic imperfect-information games from a statistical/probabilistic perspective [Pea84], although much of his work in this area centered around steering such search towards positions for which the resulting search trees would contain more perfect (i.e., less uncertain) information. Parker et al. would later extend this work to more direct opponent traps, and dubbed the notion of a perfect opponent the *paranoid assumption*, found on one end of a spectrum ranging from paranoia to overconfidence [Par06]. Although still rooted in imperfect-information games (a perfect opponent is characterized as one who would always choose a move that "minimizes payoff"), the terminology is also suitable for perfect-information games.

Circumstances similar to those described earlier, in which a player may reasonably adopt a technically "inferior" move, were identified and examined by Jansen [Jan90, Jan92].

Specifically, he noted the following three original assumptions made by Shannon, that are "inconsistent with actual human game playing":

- the game-playing situation is symmetric,
- the opponent has the same evaluation function and search strategy, and
- a game is a sequence of moves, in each of which the goal is to find the move that leads to the highest possible value against best play.

Jansen goes on to explore methods for relaxing those assumptions in the presence of an opponent model, leading to "speculative" play capable of increasing the game-theoretic value of a minimax search result. That is, an agent's knowledge about the fallibility of the opponent would enable the agent to evaluate the opponent's probability of error. Jansen also classifies various types of human errors, as well as various types of opponent models. His experiments focused on games in which the tree can be searched to completion, such as found in a complete-information chess endgame database, and small random (probabilistic) games.

Although Jansen's experiments didn't include generalization to full minimax for, say, a complete game of chess, he postulated that therein it might be less frequently admissable to make speculative mistakes, and instead be useful for breaking ties (such as when two potential moves return the same minimax score). He also suggests using how deep the opponent can search as one form of opponent model, and provides an example taken from a world championship (human) candidates match, with analysis by Deep Thought at varying depths [Jan92].

Carmel and Markovitch [CM95] extended Jansen's ideas by implementing a more generalized variation of minimax called M*, in which a game-specific opponent model is incorporated into the search algorithm itself. M* allows minimax to consider whether an opponent is likely to make an inferior move, and to take better advantage of it. The authors also examined using M* to take advantage of opponents that search to a shallower depth, and thus, like the trappy minimax algorithm presented in this paper, M* uses search depth in its traps.

Trappy Minimax

Trappy minimax is a game-independent generalized extension of the minimax adversarial search algorithm that attempts to take advantage of human frailty. It was first described by Gordon and Reda [GR06], and tested in an Othello program named Desdemona. More recently it was incorporated by Fang et al [FCJ13] into the Chinese Chess program Xqwizard. Whereas minimax assumes best play by the opponent, trappy minimax tries to predict when an opponent might make a mistake by comparing the various scores returned through iterative-deepening. Sometimes it chooses a slightly inferior move, if there is an indication that the opponent may fall into a trap, and if the potential profit is sufficiently high. Desdemona achieved a higher rating against human opposition on Yahoo! Games when using the trappy algorithm than when it used standard minimax, despite incorporating no additional Othello knowledge, and in spite of frequently playing moves that, according to minimax, were objectively inferior.

Definition of a Trap

In the trappy minimax algorithm, a *trap* is a move that (1) looks good in the short term, but that (2) has bad consequences in the long term. In minimax terms, it is a move with a high evaluation for the opponent (high negative static score) if assessed at shallow search depths, and a low evaluation for the opponent (high positive static score) when assessed at the maximum search depth. A trap is thus a move with the property that a non-optimal opponent might be tricked into thinking it is good when in fact it is not.

A computer program can *set* a trap by choosing a variation in which one of the opponent's responses is an attractive move that has the above property of being a trap. The opponent may or may not then *fall for* the trap (i.e., play the trap move).

Setting such a trap is only of practical benefit if the result of an opponent *not* falling for the trap is not too much worse than the evaluation of the move recommended by standard minimax. In other words, if the opponent does not fall for the trap, setting the trap which didn't materialize should not yield an overly significant negative cost. A slight negative cost might be an acceptable tradeoff, however, and the degree of negativity permitted can be tuned to achieve a more or less speculative style of play.

Although this definition of a trap does not require game-specific opponent modeling, by Jansen's taxonomy it still relies on a sort of opponent model, in that it is possible to characterize the type(s) of opponent that are susceptible to this sort of trap. An opponent that performs a full-width minimax search to a depth greater than the trappy minimax player would, for example, not fall for any such traps. Humans, however, are always susceptible to such traps, since they in general do not perform full-width search, and do not consider every variation to a uniform depth.

The Trappy Minimax Algorithm

The trappy minimax algorithm can be used both to (a) *identify* traps, and (b) *set* traps. While a system that simply identifies traps could be useful in building an automated game annotator, our goal is to build a system that *sets* traps. Thus we examine both *identifying* and *setting* traps.

Identifying Traps

Recalling that our definition of a trap is based on the differences in a move's evaluation at various depths of search, we can utilize the iterative deepening already present in most minimax implementations to accumulate the data that is needed for those various search depths. Thus, it turns out that accumulating the data needed for evaluating traps is inexpensive, and as we shall see, alpha-beta pruning can still be employed at each ply.

Identifying traps, or more specifically determining whether a particular move contains a trap, is accomplished by accumulating a *vector of evaluations* for that move. The vector contains all of

the minimax evaluations calculated for that move over the course of iterative deepening. That is, the first element of the vector is the evaluation for that move after a 2-ply search, the second element is the evaluation after a 3-ply search, etc. Since our goal is to set traps, our version of trappy minimax creates these vectors for *every possible opponent move*. When the vector contains values that start negative and become positive, the algorithm has identified a trap.

Consider for example the following famous sequence in the Cambridge Springs Defense:

White: human, Black: computer
1. d4 d5, 2. c4 e6, 3. Nc3 Nf6, 4. Bg5

The computer now has several options. For purposes of this discussion, let us consider just the following two: 4…Be7, and 4…Nbd7. Both are likely to have very similar evaluations under standard minimax. However, the latter (4…Nbd7) contains a trap. At first glance, 4…Nbd7 looks like it costs black a pawn, because white can play 5. cxd5 exd5, 6. Nxd5 and black's knight seemingly cannot recapture because it is pinned. However, deeper search reveals that the pin is a mirage, because after 6... Nxd5, 7. Bxd8 Bb4+ white is forced to block the check with the queen, after which he will be not ahead by a pawn, but behind by a knight.

The trap is exposed by trappy minimax in one of the vectors built by iterative deepening:

4… Nbd7, 5.cxd5

Search depth	vector	(comments)
1-ply search (after …exd5)	0.0	evaluation is even
2-ply search (after Nxd5)	-1.0	human ahead by 1 pawn
3-ply search (after …Nxd5)	+2.0	computer ahead by 2 pawns
4-ply search (after Bxd8)	-1.0	human ahead by 1 pawn (6…Nxd5 rejected)
5-ply search (after …Bb4)	-1.0	human ahead by 1 pawn (6…Nxd5 rejected)
6-ply search (after Qd2)	-1.0	human ahead by 1 pawn (6…Nxd5 rejected)
7-ply search (after …Bxd2)	+2.0	computer ahead by 2 pawns

At a depth of 4, 5, or 6 plies, an opposing system would conclude that 6…Nxd5 loses a queen, and would therefore believe that the best black could do is to concede the loss of a pawn. 6…Nxd5 is only revealed as black's best move after a 7-ply search.

The vector values described earlier and listed above are contained in a set of floating point arrays, one per opponent's legal move, indexed by search depth. Any opponent searching 6 fixed plies or less would conclude that 5. cxd5 wins a pawn. In this example, such a system would not yet have completely fallen for the trap, since it is 6. Nxd5 that is the *losing* move. However, an opponent searching 4 plies or less would fall for that too. So more precisely, trappy minimax reveals that 4…Nbd7 *contains* a trap that the opponent might miss, and therefore might select his response in the absence of this important information.

Arguably stronger moves for the opponent (rather than 5. cxd5) are 5. e3 or 5. Nf3, which retain tension in the center. By choosing 4… Nbd7, black tempts white to select cxd5 for the wrong reason, and possibly fall for the even worse trap of subsequently capturing the second d-pawn.

Contrast the above example with the similar line after 4… Be7. With the pin on black's knight removed, there is no similar trap for white to navigate. A variety of moves would be considered by the opponent, including the same ones mentioned previously (5. cxd5, 5. e3, or 5. Nf3 for examples). None of them contain traps, and so even a 4 or 5 ply engine would likely choose amongst these alternatives strictly on the less volatile basis of positional static evaluation. That is, ordinary minimax would not consider the merit of 4… Nbd7 *as a trap*.

In summary, trappy minimax may choose to *set* a trap by playing 4…Nbd7, whereas standard minimax would assume that *any* opponent would reject 5. Nxd5, and would differentiate 4…Nbd7 and 4…Be7 solely on the differences in their positional static evaluation.

<u>Setting Traps</u>

Having identified that 4…Nbd7 contains a trap (and 4…Be7 doesn't), trappy minimax must evaluate the trap to determine whether it is worth setting. A trap is evaluated using three factors:

- How likely is it that the opponent will fall for the trap?
- What is the gain to the computer if the opponent falls for the trap?
- What is the cost to the computer if the opponent doesn't fall for the trap?

Trappy minimax adjusts its move selection criteria by assessing the quality of traps contained within each of its move options, and adding bonus points to their evaluations through a simple (and tunable) risk/benefit analysis. There are a number of ways to analyze the set of vectors in order to assess the quality of the traps. In Desdemona, three different methods were used:

1. calculating the median of the evaluations at all but the maximum depth,
2. using the best evaluation for the opponent among the shallower evaluations, and
3. using only the evaluation at MaxDepth-1.

Methods 1 and 2 were directed towards human opponents, while the method 3 was targetted at computer opponents known to be using the minimax algorithm either at a shallower depth (up to MaxDepth-2), or selective search. (The limit of method 3's applicability to opposing minimax agents searching at most MaxDepth-2 is also mentioned by Jensen [Jen90].)

Two factors are then determined: *trappiness*, and *profitability*. Trappiness is based on the number of plies separating a high negative score (shallow) and the actual positive score (deep). Trappiness thus attempts to measure the likelihood that the opponent could miss the trap. Profitability is the gain to the program if the opponent falls for the trap. Trappiness and profitability are both factored into the evaluation of each possible computer move, along with the standard minimax evaluation. The resulting trappy minimax algorithm is shown in Figure 1.

```
TrappyMinimax(board,maxdepth)
best, rawEval, bestTrapQuality = -∞
{   For each move
    {   make move on board
        for each opponent response
        {   scores[maxdepth] := -Negamax(board,maxdepth)
            if (scores[maxdepth] > rawEval)
                rawEval := scores[maxdepth]
        }
        for each opponent response
            for d := 2 to maxdepth-1
                scores[d] := -Negamax(board,d)
        Tfactor := Trappiness(scores[])
        profit := scores[maxdepth]-rawEval
        trapQuality := profit * Tfactor
        if (trapQuality > bestTrapQuality)
            bestTrapQuality := trapQuality
        adjEval := rawEval + scale(bestTrapQuality)
        if (adjEval > best)
        best:=adjEval
        retract move from board
    }
    return(best)
}
```

Figure 1 — Trappy Minimax in Desdemona

Desdemona was tested both by playing it against itself in various configurations, and by playing against humans in Yahoo! Games (50 Othello games against opponents using standard minimax, and 50 games using trappy minimax). In summary, the results were as follows [GR06]:

- The trappy algorithm was better able to capitalize on weaker computer opposition than standard minimax. In 67% of the cases, using the trappy algorithm enabled Desdemona to achieve a better final score than when standard minimax was used.
- Desdemona achieved a slightly higher rating against human opposition when using the trappy algorithm than when it used standard minimax (1702 versus 1680 Elo).
- Desdemona's occasional deliberate choice of a slightly inferior move, in the interests of setting a trap, caused it to perform slightly worse against strong computer opposition.

Trappy Minimax in Chess

We tested the trappy minimax algorithm as applied to the game of chess, by incorporating trappy minimax into the existing Beowulf chess engine. The resulting program was dubbed *Trappy Beowulf.* We tested Trappy Beowulf by comparing its performance and that of standard Beowulf, with both competing against a variety of rated human opponents on ICC (Internet Chess Club) [ICC14]. Both versions were configured equally in all other respects.

The ChessBrain/Beowulf Engine

The Beowulf chess engine is an open-source International Master strength engine that was originally developed with the aim of providing a strong, well-tested implementation of common chess-playing heuristics and algorithms as source material for research and recreational purposes. It has provided the core for several engines that have built upon the existing code in order to experiment with new or unusual tree-search algorithms. Beowulf has also been adapted as part of the ChessBrain project [Fra06], which gained the world record for the largest-ever networked chess computer. It provides an ideal starting point for the present research in Trappy Minimax, as the time-consuming process of implementing well-known and thoroughly-studied techniques for chess algorithms has already been completed and known to be. Thus we could skip directly to the testing of trappy minimax.

At its core, Beowulf uses a bitboard representation, storing any board position as a series of 64-bit values. This is a widely-used technique, the advantage of which being that it allows for rapid manipulation and examination of the board state via boolean algebra, with a minimum of looping, branching and array manipulation. The core search algorithms implemented in Beowulf follow standard practices, with a core negamax search extended by a variety of safe and unsafe heuristics. Safe heuristics are those that are guaranteed to preserve the theoretical minimax values for any search tree, and prune out branches that are analytically incapable of affecting the result. Unsafe heuristics, such as *futility search* and *razoring* [Hei99] offer search tree reduction at the expense of abandoning theoretical correctness, and relying on the low likelihood of pruning branches that might have had an effect on the final result.

We retained most of the Beowulf engine as-is, simply adding the option for trappy minimax and the resulting periodic bonus points to be added to the score of moves deemed to set traps. We also disabled Beowulf's opening book, to maximize the algorithm's exposure during all phases of game play (i.e., so that trappy minimax is also used during the opening stage).

Alpha-Beta Pruning in Trappy Beowulf

Our adaptation of trappy minimax in Beowulf includes an improved handling of alpha-beta pruning compared with previous trappy minimax implementations.

As described previously, the trappy minimax algorithm accumulates evaluation scores for each move, for each level of depth-limited searches during iterative deepening search. It is possible, however, that in the presence of alpha-beta pruning, a particular move was never actually considered in one or more of the searches, due to standard alpha-beta cutoffs. The trappy minimax algorithm thus would not be able to identify traps involving moves that were cut.

The solution used in previous implementations of trappy minimax (including in Desdemona) was to disable alpha-beta pruning at ply #2, so that all moves on that ply have scores recorded at every depth. This provides a simple mechanism for trappy minimax to consider every possible human response as a potential trap move. However, it incurs a significant performance penalty for the program, because prunes at early plies result in the most time-savings. Furthermore, this approach only allows traps to be identified at the topmost ply (for example, it would miss the Cambridge Springs example described earlier).

Trappy Beowulf uses an improved adaptation of trappy minimax that does not incur these problems. The revised method keeps track of which moves have been searched at each level by marking them during the search, and then takes this information into account when identifying traps. In particular, as iterative deepening proceeds, and scores are returned from iterative deepening, alpha-beta pruning can produce “holes” in the score vectors built by trappy minimax. Trappy Beowulf fills in those holes with results from other searches. Specifically, when considering a certain trap candidate move that was pruned when searching to depth *N*, Trappy Beowulf substitutes in the value of the move’s evaluation when searched to depth *N+1*. Using this pattern, it fills in all missing values before searching the vectors for potential traps.

The rationale for this method is that a value returned from a search of depth *N+1* is more accurate than a value returned from a search of depth *N*. Because these scores are used to try and trick the opponent, using a value from a deeper search is, at worst, overestimating the opponent’s skill. In the worst-case scenario, when a given move has not been considered at any depth except the current depth, that score will be copied to all previous depths, no trap will be identified, and trappy minimax will behave like ordinary minimax.

Thus, Trappy Beowulf’s implementation improves previous versions of trappy minimax in two important ways: (1) if a certain move was evaluated in some searches but skipped in others due to pruning, the move could still be identified as a trap based on the partial information that was recorded; (2) since alpha beta pruning is never disabled, no performance penalty is incurred.

Experimental results

We compared the performance of Trappy Beowulf against that of the original Beowulf program. As expected, when the two programs are pitted directly against each other, the original program is usually able to beat the modified (trappy) version. This was the expected result and is consistent with earlier findings, because trappy minimax relies on an opponent that does not perform a comparable full-width search – an opponent that searches every variation in which the traps appear will not fall for them.

Of greater interest is how each version performs against human opposition or other selective search agents, as that is the "opponent model" specifically targetted by the trappy algorithm. To do this, we entered both the trappy and original versions of Beowulf in ranked blitz matches on the Internet Chess Club (ICC), playing against human players. Both programs played against a variety of human players on ICC over the course of several weeks. To control for any residual discrepancies in runtime performance between the two implementations, both were limited to a maximum search depth of eight plies. Additionally, both programs did not use an opening book.

We expected Trappy Beowulf to perform better against humans on ICC than ordinary Beowulf. This was confirmed by the experiment: after playing 30 ranked blitz matches each against human opponents, regular Beowulf was rated 2262, and Trappy Beowulf was rated 2440.[1]

Data collected for Trappy Beowulf's games revealed that it frequently played moves that it knew to be non-optimal in order to set traps. On average it intentionally played a "non-optimal" move once for every four "optimal" moves that it played. Each trappy minimax search finds hundreds of potential traps, although most are very small traps found in the deeper parts of a search.

Examples of Traps set During Play

Besides the Cambridge Springs example described earlier (which Trappy Beowulf identifies and sets), Trappy Beowulf sets a variety of types of traps over the course of a game. The traps are often subtle or involve the potential win of a small positional gain. They also may involve trap moves several plies deep, and thus may never manifest in an actual occuring board position. However, they also can be tactically dangerous for the opponent, with potential gains for the computer should the opponent misstep.

Consider the following position that occurred during one of Trappy Beowulf's games:

Black (human) to move

[1] As an aside, we were surprised that such ratings could be achieved without benefit of an opening book.

This is a tactically complex position with numerous open lines, pins, and pieces being threatened. Black can either retreat the threatened b4 bishop by playing 1…Ba5, or pin the attacking pawn by 1…Qa4 or 1…Qa6. In fact, Trappy Beowulf has just set a trap, and its analysis of the optimal continuations after each of those three possible moves is:

1. . . .	**Ba5**		1. . . .	**Qa4**		1. . . .	**Qa6**
2. e4	Qa6		2. Ng5	Bd2		2. Ng5	Ba5
3. Qf6	Ba4		3. Be4	h6		3. Be4	Ra7
4. Rb2	Re6		4. Bxa8	hxg5		4. Nxf7	Kxf7
5. Qh4	Bc3		5. Bd5	Bxe3+		5. Rf1+	Bf5
6. Ng5	h5		6. Rxe3	Rxe3		6. Bd5+	Re6
7. Nxe6	Bxb2		7. Bxf7+	Kh7		7. e4	Ke8
8. Rb1	Be8		8. Rf1	Be8		8. Qh8+	Kd7
9. Rxb2	Qxa3		9. Qd2	Bxf7		9. exf5	Re7
	(-64)			(-64)			(-27)

(Negative scores are favorable for the human opponent)

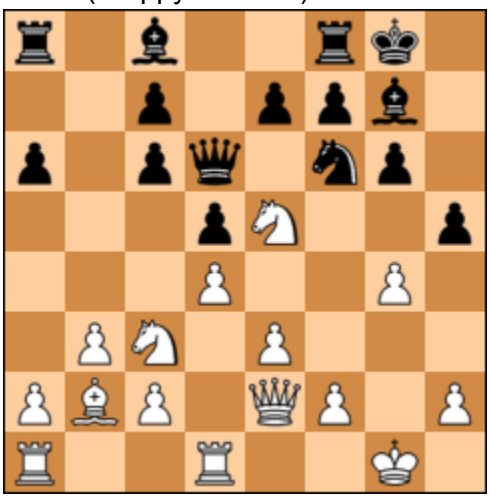
The relevent portion of the iterative deepening vector for **1…Qa6** is as follows:

-72	-53	-53	-53	-51	-27

Trappy Beowulf knows that it is behind, and has created a position in which the human has two options, 1…Ba5 and 1…Qa4, that both lead to excellent positions. However, the move 1…Qa6 appears at shallower search depths to be an even better choice. It is not until six plies deeper that it is revealed to reverse much of the human's advantage. In this case, 1…Qa6 looks good in that it retains the option of B(d7)-a4, it turns out that white's 4. Nxf7 maneuver succeeds after 1…Qa6. If the trap works and the human plays 1…Qa6, Trappy Beowulf is back in the game.

In some cases, Trappy Beowulf incorrectly considers "traps" that humans would be unlikely to fall for. That is, there are some cases in which the vectors appear to indicate a trap, but in reality the trap positions simply aren't appealing, or are byproducts of the horizon effect. For example:

White (Trappy Beowulf) to move:



A typical move selected by standard minimax for the computer (white) is 1. h3. Trappy Beowulf, however, would consider playing 1. g5, because it believes that it sets a trap. Its analysis indicates that the correct human response to 1.g5 is 1…Nd7, but that the opponent might be lured into mistakenly chosing to play 1…c5 instead. In this case, 1…c5 received a reasonable evaluation from the opponent's perspective at a five level deep search, and bad evaluation at a six level deep search. Beowulf's analysis of each option is interesting:

> *If Beowulf were to choose the "correct" option of 1.h3, the predicted optimal play that follows is: 1….Nh7 2. f4 Bxe5 3. dxe5 Qd7 4. f5 Ng5 5. e6 Qd6 6. Qg2 with a score of +1.24 for Beowulf. This is a nearly even score, i.e., only a fraction of a pawn (by Beowulf's numeric scale).*
>
> *If Beowulf instead chooses to set the "trap" by playing 1.g5, the predicted best line of play (if the human doesn't fall for the trap) is 1…Nd7 2. Nf3 Bb7 3. a4 c5 4. dxc5 Qxc5 5. Qd3 Qc6 6. Nd4 with a score of -0.33, also nearly even with Beowulf down by a small fraction of a pawn.*
>
> *If Beowulf sets the "trap" by playing 1.g5, and the human falls for the trap by playing 1…c5, the predicted optimal result that follows is 1…c5 2. gxf6 exf6 3. Nf3 cxd4 4. Rxd4 c6 5. a4 Qc5 6. b4 with a score of +2.96 for Beowulf, a clear edge. Thus, the opponent falling for the trap puts the computer ahead by 2.96, while if the opponent sees through the trap the computer is behind by 0.33. In comparison, if the computer did not set the trap and both the computer and opponent played as ordinary minimax would predict, the computer would be ahead by 1.24.*

Whether the computer would actually choose to set this false trap or not depends on the way it is configured. For example, if the scaling factor was tuned to be more permissive in setting traps, the computer may be more willing to set risky traps. Unlike this trap, most traps identified by Trappy Beowulf are deep and comparatively low profit and low risk.

Of course, in this case, this particular trap is actually a mirage. It is unlikely that any human player would choose 1…c5, simply because the sequence of captures along the a1-h8 diagonal are clearly in white's favor. 1…c5 is not an appealing move, and it is likely that the iterative deepening vectors in this case are suffering from the horizon effect and making 1…c5 seem more appealing to Trappy Beowulf than it actually would be to even an amateur level human.

Other Observations

Most traps set by Trappy Beowulf involve small risk and small gains. Over the course of a typical 8-ply search, it identifies on average about 80 traps that it could potentially set. In the configuration tested on ICC, it chose a "non-optimal" move about 25% of the time. The majority (about 65%) of traps were set during the middle game (after move 10 and before move 35), and very few were set after move 35 (about 3%).

The trappy minimax algorithm allows for different behavior according to tuning and implementation, and the goal of Trappy Beowulf was to implement it in a way that was reasonably conservative but still able to set traps. Alternate implementations of trappy minimax could focus more on risk-taking, or put less emphasis on traps and play more conservatively.

Despite Trappy Beowulf frequently playing moves that it knows to be inferior, and despite the observation that some of the traps it sets are unlikely for humans to fall into, it still performs at a higher level against humans than standard minimax. This suggests that, over the span of a game, humans are sufficiently likely to fall for enough of the traps to result in a net gain.

Conclusion

Unlike computers, when humans play strategy games they do not always assume best play by their opponent. Rather, humans try to steer the game into positions where the opponent might play a bad move. Standard minimax includes no mechanism for such a strategy, and we suggest that this is one of the stylistic differences between human and computer play.

Trappy Minimax is an extension to minimax that extracts information during iterative deepening to identify potential traps that the computer could set, even if they involve some risk. It assumes only a very rudimentary game-independent opponent model; that is, it assumes the opponent is not searching full-width to a greater depth than itself. Thus it is most effective against humans, and against computers employing either shallower search, or selective search.

We tested the algorithm by creating a version of the Beowulf chess engine that incorporates Trappy Minimax, and the resulting Trappy Beowulf significantly outperformed standard Beowulf on ICC against human opposition, despite losing head-to-head against standard Beowulf. The results corroborate those previously observed for Othello. We also improved the handling of alpha-beta pruning, resulting in better performance and better identification of potential traps than in previous implementations of trappy minimax.

Trappy Beowulf successfully utilized non-optimal “speculative” moves about 25% of the time. This contradicts Jensen’s suggestion that in complete games such “mistakes” would not frequently be useful, but supports his idea that considering various search depths is useful.

Trappy Minimax is unlikely to help an elite chess program garner a higher rating in today’s computer-vs.-computer competitions. However, it has potential application in other aspects of two-player games, such as: (1) games in which programs do poorly against humans such as Go, (2) building more human-like computer game-playing programs, and (3) building more insightful game annotation tools.

Finally, the work gives emprical credence to rejecting the long-held notion that “best move” must necessarily be defined under the assumption of best response from the opponent. We described several situations wherein most experts would agree that the best choice would be one based on a non-optimal opponent response. The competitive results of Trappy Beowulf confirms the practicality of strategically relaxing the assumption in adversarial search.

References

[Bur97] Buro, M. (1997). The Othello Match of the Year: Takeshi Murakami vs. Logistello, ICCA Journal 20 (3), pp 189-193.

[CM95] Carmel, D. and Markovitch, S. (1995). *Opponent Modeling in a Multi-Agent System*, Workshop on Adaptation and Learning in Multiagent Systems - IJCAI, Montreal 1995.

[FCG13] Fang, J., Chi, J., and Jian, H.Y. (2013). A Trappy Alpha-Beta Search Algorithm in Chinese Chess Computer Game, Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering, Hangzhou, China, pp 260-263.

[Fra06] Frayn, C., Justiniano, C., and Lew, K. (2006). *ChessBrain II – A Hierarchical Infrastructure for Distributed Inhomogeneous Speed-Critical Computation*, IEEE Symposium on Computational Intelligence and Games (CIG'06), Reno, NV, pp 13-18.

[GR06] Gordon, V. and Reda, A. (2006). *Trappy Minimax - using Iterative Deepening to Identify and Set Traps in Two-Player Games*, IEEE Symposium on Computational Intelligence and Games (CIG'06), Reno, NV, pp 205-210.

[Hei99] Heinz, E. (1999). *Scalable Search in Computer Chess: Algorithmic Enhancements and Experiments at High Search Depths*, GWV-Vieweg © 1999.

[Hsu02] Hsu, F.H. (2002). *Behind Deep Blue*, Princeton University Press © 2002.

[ICC14] The Internet Chess Club. http://www.chessclub.com

[Jan90] Jansen, P. (1990). *Problematic Positions and Speculative Play*, in Computers, Chess, and Cognition, Springer-Verlag, © 1990, Marsland, T. and Schaeffer, J., editors. ([MS90], below)

[Jan92] Jansen, P. (1992). *Using Knowledge about the Opponent in Game-Tree Search*, Ph.D. Thesis, Carnegie Mellon University, 1992.

[Kot62] Kotok, A. (1962). *A Chess Playing Program for the IBM 7090 Computer*, MIT bachelor's thesis in Electrical Engineering, June 1962.

[MS90] Marsland, T. and Schaeffer, J. *Computers, Chess, and Cognition*, Springer-Verlag, © 1990.

[Par06] Parker, A., Nau, D., and Subramanian, V. S., *Paranoia versus Overconfidence in Imperfect-Information Games*, Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06) pp 1045-50.

[Pea84] Pearl, J. (1984). *Heuristics: Intelligence Search Strategies for Computer Problem Solving*, Addison Wesley © 1984

[Sch09] Schaeffer, J. (2009). *One Jump Ahead*, Springer © 2009

[Sch89] Schaeffer, J. (1989). *The History Heuristic and Alpha-Beta Search Enhancements in Practice*, IEEE Trans. on Pattern Analysis and Machine Intelligence, 1989, pp 1203-1212.

[Shan50] Shannon, C.E. (1950). *Programming a Computer for Playing Chess*, in Philosophical Magazine, vol 41 n.7, pp 256-275.

[Sol13] Soltis, A. (2013). *The End of Strategy*, Chess Life, July 2013, pp 14-15.

[vonN28] Von Neumann, J. (1928). *Zur Theorie der Gesellschaftsspiele*, in Mathematische Annalen 100, pp 295–300.