CPSC 449 Thesis: Implementation of Mult-ProbCut in Chess

Albert Xin Jiang
52429982

April 19, 2003

Abstract

ProbCut[2] is a selective searching enhancement to the standard alpha-beta algorithm for two-person games. ProbCut and its successor Multi-ProbCut[3] have been very effective in some games like Othello, but there had not been any report of success in the game of chess previously. This paper presents my implementation of ProbCut and Multi-ProbCut (MPC) in the chess engine Crafty. Test results show that the MPC version of Crafty searches faster than the plain version of Crafty. The MPC version beat plain Crafty 41.5 – 22.5 in a 64-game match.

1 Introduction

Computer chess has been a research topic for AI researchers since the invention of the computer, and it has come a long way. Nowadays, the best computer chess programs and the best human grandmasters play at roughly the same level.

Most of the successful chess programs use the so-called brute-force approach, in which the program has limited chess knowledge but rely on a fast search algorithm to find the best move.

There has been much research on searching algorithms for game playing. Michael Buro's ProbCut [2] and Multi-ProbCut [3] are promising approaches. They were first implemented in the game of Othello, and resulted in huge improvements for Buro's program Logistello. In 1997, Logistello defeated the human Othello World Champion Takeshi Murakami by a score of 6-0 [4].

ProbCut and Multi-ProbCut (MPC) are domain-independent, however there were no previous reports of success at implementing them in the game of chess. In this paper I presents my implementations of ProbCut and MPC in a chess program, and some test results on their performance. Section 2 gives some necessary background knowledge. Section 3 discusses my ProbCut implementation and Section 4 discusses my MPC implementation. Finally Section 5 concludes and discusses some ideas for future research.

2 Background

2.1 Minimax and Alpha-Beta

There are a lot of previous research on the field of game searching. I won't attempt to cover them all here, instead I will concentrate on things relevant to ProbCut. For an introduction on game tree searching, a good web-site is http://www.xs4all.nl/~verhelst/chess/search.html.

For two-person zero-sum games like chess, a position can be seen as the root of a tree. Possible moves by both sides are the branches and resulting positions are the nodes. Finding the best move to play means searching through the successors of the position and finding the best successor for the player to move, and at the next level of the tree we are looking for the best successor for the opponent. This is called minimaxing. In practice, computer does not have time to search to the end of the game. Instead, it searches to

a certain depth, and use an heuristic evaluation function to evaluate the leaf nodes statically. For chess the evaluation function is based on the number of pieces on the board and other considerations like king safety.

An important improvement over minimax is alpha-beta pruning[7]. A pseudo-code representation of the algorithm is shown in Figure 1.

```
int AlphaBeta(int alpha, int beta, int depth){
    if (depth == 0) return Evaluate();

    int total_moves = GenerateMoves();
    for (int i=0; i < total_moves; i++){
        MakeMove(i);
        val = -AlphaBeta(-beta, -alpha, depth-1);
        UnMakeMove(i);
        if (val >= beta) return val;
        if (val > alpha) alpha = val;
    }
    return alpha;
}
```

Figure 1: Alpha-Beta Algorithm.

AlphaBeta() returns the correct minimax value (up to a certain depth) if the value is inside the window (alpha, beta). If it returns greater or equal to beta, the return value is a lower bound of the minimax value; if it returns less or equal to alpha, the return value is an upper bound[1].

There have been a number of enhancements to alpha-beta, e.g. transposition tables, iterative deepening, NegaScout, etc ([6], [8]). Armed with these refinements, alpha-beta has become the dominant algorithm for game tree searching [6].

Compared to minimax, alpha-beta is able to prune many subtrees that would not influence the minimax value of the root position. But it still spends most of its time calculating irrelevant branches that human experts would never consider. Researchers have been trying to make the search more selective, while not overlooking important branches. How should we decide whether to search a particular branch or not? One idea is to base this decision

[1] The former is called a "fail-high", the latter a "fail-low".

on the result of a shallower search. Null-Move heuristic [1] and ProbCut are two approaches based on this idea.

2.2 Null-Move Heuristic

A null-move is same as a pass: the player does nothing and let the opponent move. Passing is not allowed in chess, but in chess games it is almost always better to play a move than passing. The null-move heuristic takes advantage of this fact, and does a shallower search on the null-move before searching the regular moves as in alpha-beta. If the search on the null-move returns something greater or equal to beta, then it is very likely that one of the regular moves will also fail-high. In this case we simply return beta after the search on the null-move. This procedure can even be done recursively in the shallower search, as long as no two null-moves are done consecutively. Since the search on the null-move is shallower than the rest, occasionally it will overlook something and mistakenly cut the branch, but the speed-up from cutting these branches allows it to search deeper on more relevant branches. The benefits far outweigh the occasional mistakes. The null-move heuristic is very effective in chess, and most of the strong chess engines use it. But it is domain-dependent: in games like Othello and checkers, it is often better to pass than to move, so the null-move heuristic does not work there.

2.3 ProbCut

ProbCut [2] is based on the idea that the result of a shallow search v' is a rough estimate of the result of a deeper search v. The simplest way to model their relationship is to use a linear model:

$$v = av' + b + e$$

Where e is a normally distributed error variable with mean 0 and standard deviation σ. The parameters a, b and σ can be decided by linear regression on search results of thousands of positions.

If based on the value of v', we are sure with high probability that $v \geq \beta$, where β is the beta-bound for the search on the current subtree, we can prune the subtree and return β. After some manipulation, the above condition becomes $(av' + b - \beta)/\sigma \geq -e/\sigma$. This means that $v \geq \beta$ holds true with probability of at least p iff $(av' + b - \beta)/\sigma \geq \Phi^{-1}(p)$. Here Φ is the Gaussian

distribution function with variance 1. This is equivalent to $v' \geq (\Phi^{-1}(p) \cdot \sigma + \beta - b)/a$. Similarly for $v \leq \alpha$, the condition becomes $v' \leq (-\Phi^{-1}(p) \cdot \sigma + \alpha - b)/a$. This leads to the pseudo-code implementation shown on Figure 2. Note that the search windows for the shallow searches are set to have width 1. These are called null-window searches. Generally the narrower the window is, the faster the search returns. Null-window searches are very efficient when we do not care about the exact minimax value and only want to know whether the value is above or below a certain bound, which is the case here. The shallow depth/deep depth pair and cut threshold are to be determined empirically, by running lots of tests on the performance of the program.

For ProbCut to be successful, v' need to be a good estimator of v, with a fairly small σ. This means the evaluation function need to be stable, i.e. it need to be not too far from what you would get by searching. Evaluation functions for chess are generally not stable, due to opportunities of capturing which cannot be resolved statically. Fortunately most chess programs have so-called quiescence search: at the leafs the game tree where regular search depth reaches zero, instead of calling the evaluation function, a special quiescence search function is called to search only capturing moves, and only calls the evaluation function when there are no profitable capturing moves. Quiescence search returns a much more stable and accurate value.

In summary, null-move heuristic and ProbCut both try to compensate for the lower accuracy of the shallow search by making the shallow search harder to produce a cut. Null-move does this by giving opponent a free move, while ProbCut does it by setting a harder bound for the shallow search.

2.4 Multi-ProbCut

MPC [3] consists of several ways to enhance ProbCut:

- Different regression parameters and cut thresholds for different stages of the game.
- More than one depth pairs. For example, using depth pairs (3,5) and (4,8), if at check depth 8 the 4-ply shallow searches do not produce a cut, then further down the 8-ply subtree we could still cut some 5-ply subtrees using 3-ply searches.
- Iterative deepening shallow searches for one check depth.

```
#define  S   4    /* depth of shallow search */
#define  D   8    /* check depth */
#define  T   1.0  /* cut threshold */
int AlphaBeta(int alpha, int beta, int depth) {
    if (depth == 0) return Evaluate();

    if (depth == D) {
      int bound;

      /* v >= beta likely? */

      bound = round ((T * sigma + beta - b) / a);
      if (AlphaBeta(bound-1, bound, S) >= bound) return beta;

      /* v <= alpha likely? */

      bound = round ((-T * sigma + alpha - b) / a);
      if (AlphaBeta(bound, bound+1, S) <= bound) return alpha;
    }

    /* The rest of AlphaBeta code goes here */
    ...

}
```

Figure 2: ProbCut with depth pair (4,8) and cut threshold 1.0

In the case of Othello, MPC shows significant improvements over ProbCut.

2.5 ProbCut and Chess

There has been no report of success for ProbCut or MPC in chess thus far. There are at least two reasons for this:

1. Null-move is available for chess. Null-move and ProbCut are based on similar ideas, as a result they tend to prune the same type of positions. Part of the reason why ProbCut is so successful in Othello is that

null-move does not work in Othello. But in chess, ProbCut and MPC have to compete with null-move, which is way better than brute-force alpha-beta.

2. Chess searches tend to make more mistakes than Othello searches [5]. This leads to a larger standard deviation in the linear relationship between shallow and deep search results, which makes it harder to get ProbCut cuts.

3 ProbCut Implementation

Before trying MPC, I implemented the simpler ProbCut with one depth pair. My implementation is built on the source code of Crafty (Version 18.15) by Robert Hyatt[2]. Crafty is a state-of-the-art free chess engine. It uses a typical brute-force approach, with a fast evaluation function, NegaScout search and all the standard enhancements: transposition table, iterative deepening, null-move heuristic, etc. Crafty also has quiescence search, so the results of its evaluation function plus quiescence search should be fairly stable.

The philosophy of my approach is to take advantage of the speed-up provided by null-move heuristic whenever possible. The obvious way to combine null-move and ProbCut is to see null-move as part of the brute-force search, and build ProbCut on top of the "alpha-beta plus null-move" search.

First, we need the parameters from linear regression. I let Crafty search a couple thousand positions and record its search results for $1, 2, \ldots 10$ plies. The positions are chosen randomly from some computer chess tournament games and some of Crafty's games against human grandmasters on internet chess servers. Note that Crafty was using null-move heuristic for these searches.

Then I did linear regression fitting on several depth pairs, using the data collected. Results show that there is a linear relationship between shallow search and deep search results, as shown in Figure 3 [3].

However, these are not perfect linear fits. The v' versus v relation has the following characteristics:

[2] Crafty's source code is available at ftp://ftp.cis.uab.edu/pub/hyatt.

[3] Crafty's evaluation function is in units of 1/100 pawns, i.e. a score of 100 means the player to move is one pawn up (or have equivalent positional advantage).

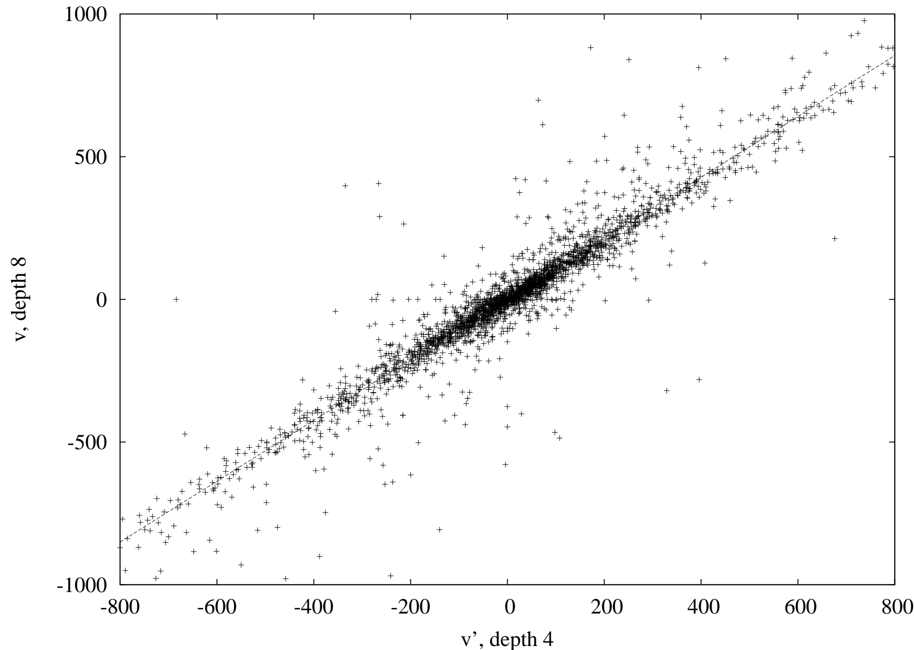

Figure 3: v' versus v for depth pair (4,8)

- The relation is more linear for v' data points near zero, and less linear for data points far from zero (i.e. positions where one side has a big advantage). This can be explained as follows: if say White has a big advantage, then White will likely gain more material advantage after a few more moves. Therefore if the shallow search returns a big advantage, a deeper search will likely return a bigger advantage, and vice versa for disadvantages. I only used v' data points in the range [-300, 300] for the linear regression.

- Occasionally the shallow search would miss a check-mate while the deeper search would find it. For example, in a position White can check-mate in 7 plies. A 4-ply search couldn't find the check-mate while a 8-ply search would find it. A check-mate-in-N-move is represented by a very large integer in Crafty. I excluded these data points from the linear regression, because the evaluation for check-mate is a rather arbitrary large number, there is no proper way to incorporate these data points in the linear regression.

I also did linear regression for data points from different stages of the

game. It turned out that the standard deviation for the fit using only endgame positions is smaller than the standard deviation using only middle-game positions. Table 1 shows some of the results[4].

Pairs	Stage	a	b	sigma	r
(3,5)	midgame	0.998	-7	55.8	0.90
(3,5)	endgame	1.026	-4.1	51.8	0.94
(4,8)	midgame	1.02	2.36	82	0.82
(4,8)	endgame	1.11	1.75	75	0.90

Table 1: Linear regression results.

Adding ProbCut code to Crafty's search function is rather easy. Since I only did linear regression for the interval [-300, 300], I set ProbCut to be active only if beta (or alpha) is inside the interval [-300, 300]. I also used different regression parameters for middle-game and endgame positions.

I did some experiments[5] on my implementation with different depth pairs and cut thresholds. Depth pairs (4,6) and (4,8), and cut thresholds 1.0 and 1.5 were tried. I let the program search about 300 real-game positions, spending 30 seconds on each position, and see how deep it was able to search on average. Results show that

- Versions with depth pairs (4,6) and (4,8) have similar speeds.
- The versions with cut threshold 1.5 are not faster than plain Crafty.
- The versions with cut threshold 1.0 are slightly faster than Crafty: they search 11.6 plies compared to 11.5 plies by Crafty. In some positions, 80-90% of the shallow searches result in cuts, and ProbCut is much faster than plain Crafty. But in some other positions the shallow searches produce cuts less than 60% of the time, and ProbCut is about the same speed or even slower than Crafty. On average, this version of ProbCut produces more cuts than plain Crafty's null-move does at the check depth.

[4] r is the multiple regression correlation coefficient, a measure of how good the data fits the linear model.

[5] All the experiments are run on a PentiumIII-850 under Linux. Crafty's hashtable size is set to 48 MBytes, and pawn hashtable size is set to 6 MBytes.

Since cut threshold 1.5 is no good, I concentrated on the threshold 1.0 for the following experiments.

I ran matches between the ProbCut versions and plain Crafty. Each side has 10 minutes per game. A generic opening book was used. Endgame databases were not used. A conservative statistical test[6] shows that in a 64-game match, a score above 38 points (or 59%) is statistically significant with $p < 0.05$. Here a win count as 1 point and a draw count as 0.5 point.

The match results are not statistically significant. The ProbCut versions seem to be no better nor worse than plain Crafty. For comparison, I ran a 64-game match of ProbCut against Crafty with null-move turned off for both programs. The ProbCut version is significantly better than Crafty here, winning the match 40 – 24.

4 Multi-ProbCut Implementation

ProbCut produces more cuts than plain null-move does, but it seems that the small speed-up provided by ProbCut is not enough to result in better playing strength. This motivates my implementation of MPC. We already have different regression parameters for middle-game and endgame. Now I implemented multiple depth pairs. The implementation was straightforward, much like the Othello implementation in [3]. As in [3], MPC search is not "recursive", in the sense that ProbCut is not applied inside the shallow searches. This is done to avoid the collapsing of search depth.

I chose depth pairs (2,4), (3,5), (4,8), (6,10) for endgames and (3,5), (4,8), (5,9) for middle-games. Cut threshold is set to 1.0. I also tweaked the evaluation function to make king safety more important, in an attempt to compensate for the MPC search's occasional oversights.

I tested the MPC implementation on the same 300+ positions as in Section 3. With 30 seconds per position, it is able to search 12.0 plies on average, which is 0.5 plies deeper than plain crafty.

Finally, I ran a 64-game match of the MPC version against plain Crafty. Each side has 10 minutes per game. Both programs' king safety weight is set

[6]The statistical test is based on the assumption that at least 30% of chess games between these programs are draws, which is a fair estimate. The test is based on Amir Ban's program from his posting on rec.game.chess.computer:
http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&selm=33071608.796A%40msys.co.il

to 125% of the default value. The MPC version defeated plain Crafty 41.5 – 22.5, scoring 65%, with 30 wins, 11 losses and 23 draws.

5 Conclusions and Further Research

Preliminary results show that Multi-ProbCut can be successfully implemented in chess. Due to the existence of null-move heuristic in chess, the improvement provided by MPC in chess is not as huge as in Othello. Still it shows promising potential, especially since I have not yet fine-tuned its parameters. I encourage chess programmers to try MPC in their chess programs.

More experiments need to be done on my MPC implementation, to determine how evaluation function parameters like king safety weight can influence MPC's performance. To further verify the strength of the MPC implementation, we can run matches with longer time controls, not only against Crafty but also other chess programs.

The depth pairs and the cut threshold are probably not yet optimal. One way to optimize them is to run matches between versions with different parameters. But better results against another version of the same program does not necessarily translate to better results against other opponents. An alternative would be to measure the accuracy of search algorithms by a method similar to the one employed in [5], using a deeper search as the "oracle", and look at the difference between the oracle's evaluations on the oracle's best move and the move chosen by the search function we're measuring. Maybe the combination of the above two methods give a better indication of chess strength.

More possible enhancements to the MPC algorithm are worth trying out. For example, using fail-soft alpha-beta[7] will make certain optimizations possible. There are other possible ways to combine MPC and null-move in chess. A successful implementation should make the two enhancements work well together.

[7]Fail-soft alpha-beta can return values outside the window [alpha, beta], while fail-hard alpha-beta (the one Crafty uses) returns alpha for fail-lows and beta for fail-highs.

6 Acknowledgements

I would like to thank David Poole and Michael Buro for their helpful advices, and Bob Hyatt for writing the excellent and very readable Crafty chess engine.

References

[1] Beal, D.F. *A Generalized Quiescence Search Algorithm*, Artificial Intelligence, Vol.43, pp. 85-98, 1990.

[2] Buro, M. *ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm*, ICCA Journal 18(2) 71-76, 1995.

[3] Buro, M. *Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello*, Workshop on game-tree search, NECI, 1997.

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

[5] Junghanns, A., Schaeffer, J., Brockington, M., Bjornsson, Y., Marsland, T. *Dimishing Returns for Additional Search in Chess*, Advances in Computer Chess 8, H. J. van den Herik and J. W. H. M. Uiterwijk (eds.), pp. 53-67, University of Maastricht, ISBN 9-062-16234-7, 1997.

[6] Junghanns, A. *Are there practical alternatives to alpha-beta?* ICCA Journal, 21(1):14-32, 1998.

[7] Knuth, D.E., Moore, R.W. *An Analysis of Alpha-Beta Pruning*, Artificial Intelligence, Vol. 6, pp.293-326, 1975.

[8] Reinefeld, A. *An Improvement of the Scout Tree Search Algorithm*, ICCA Journal 6(4):4-14, 1983.