FIRST EXPERIMENTAL RESULTS OF PROBCUT APPLIED TO CHESS

Albert Xin Jiang[1] and Michael Buro[2]

[1] *Department of Computer Science, University of British Columbia*
Vancouver V6T 1Z4 Canada
albertjiang@yahoo.com

[2] *Department of Computing Science, University of Alberta*
Edmonton T6J 2E8, Canada
mburo@cs.ualberta.ca

Abstract ProbCut [2] is a selective search enhancement to the standard alpha–beta algorithm for two–person games. ProbCut and its improved variant Multi–ProbCut (MPC) [3] have been shown to be effective in Othello and Shogi, but there had not been any report of success in the game of chess previously. This paper discusses our implementation of ProbCut and MPC in the chess engine Crafty. Initial test results suggest that the MPC version of Crafty is stronger than the original version of Crafty: it searches deeper in promising lines and defeated the original Crafty $+22-10 = 32$ (59.4%) in a 64–game match. Incorporating MPC into Crafty also increased its tournament performance against Yace – another strong chess program: Crafty's speed chess tournament score went up from 51% to 56%.

1. Introduction

Computer chess has been an AI research topic 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 and relies on a fast search algorithm to find the best move. There has been much research on improving the original minimax algorithm for finding moves in two player perfect information games. Enhancements range from sound backward pruning (alpha–beta search), over using transposition tables and iterative deepening, to selective search

heuristics that either extend interesting lines of play or prune uninteresting parts of the search tree.

The ProbCut [2] and Multi–ProbCut (MPC) [3] heuristics fall into the last category. They were first implemented in Othello programs where they resulted in a much better performance compared to full–width alpha–beta search. Utilizing MPC, Logistello defeated the reigning human Othello World Champion Takeshi Murakami by a score of 6–0 in 1997 [4].

ProbCut and MPC do not rely on any game specific properties. However, there were no previous reports of success at implementing them in the game of chess. In this paper we present our first implementations of ProbCut and MPC in a chess program and some experimental results on their performance. Section 2 gives some necessary background knowledge. Section 3 discusses our ProbCut implementation and Section 4 discusses our MPC implementation. Finally, Section 5 concludes and discusses some ideas for future research.

2. Background

2.1 Minimax and Alpha–Beta Search

There has been a lot of previous research in the field of game–tree search. We will not attempt to cover it all here. Instead, we will concentrate on things relevant to ProbCut. For an introduction to game–tree search, a good web–site is `www.xs4all.nl/~verhelst/chess/search.html`.

For two–person zero–sum games like chess, positions can be viewed as nodes in a tree or DAG. In this model, moves are represented by edges which connect nodes. Finding the best move in a given positions then means to search through the successors of the position in order to find the best successor for the player to move after finding the best successor for the opponent in the next level of the tree. This procedure is called minimaxing. In practice, computers do not have time to search to the end of the game. Instead, they search to a certain depth, and use a heuristic evaluation function to evaluate the leaf nodes statically. For chess, the evaluation function is based on material and other considerations such as King safety, mobility, and pawn structure.

An important improvement over minimax search is alpha–beta pruning [10]. An alpha–beta search procedure takes additional parameters alpha and beta, and returns the correct minimax value (up to a certain depth) if the value is inside the window (alpha, beta). A returned value greater or equal to beta is a lower bound on the the minimax value, and a value less or equal to alpha is an upper bound. These cases are called *fail–high* and *fail–low*, respectively. A pseudo–code representa-

```
int AlphaBeta(int alpha, int beta, int height) {
    if (height == 0) return Evaluation();

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

Figure 1. The alpha–beta algorithm (fail–hard version).

tion of one version of the algorithm is shown in Figure 1. The algorithm shown is called "fail–hard" alpha–beta, because it generally returns alpha for fail–lows and beta for fail–highs. There exist "fail–soft" versions of alpha–beta which can return values outside of the alpha–beta window, thus giving better bounds when it fail–high/fail–low.

There have been a number of enhancements to alpha–beta, e.g. transposition tables, iterative deepening, NegaScout, etc. ([7], [11]). Armed with these refinements, alpha–beta has become the dominant algorithm for game tree searching [7].

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 on the result of a shallower search. The null–move heuristic ([1], [5]) and ProbCut are two approaches based on this idea.

2.2 The Null–Move Heuristic

A null–move is equivalent to a pass: the player does nothing and lets 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 (or null–move pruning) takes advantage of this fact, and before searching the regular moves for height-1 plies as in alpha–beta, it does

a shallower search on the null–move for height$-R-1$ plies, where R is usually 2. If the search on the null–move returns a value 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 applied recursively in the shallower search, as long as no two null–moves are played consecutively.

Because 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. However, in chess endgames with few pieces left, *zugzwang* positions are often encountered, in which any move will deteriorate the position. Null–move heuristic fails badly in zugzwang positions. As a result, chess programs turn off null–move heuristic in late endgames.

There have been some research to further fine–tune and improve the null–move heuristic. Adaptive Null–Move Pruning [6]uses $R = 3$ for positions near the root of the tree and $R = 2$ for positions near the leaves of the tree, as a compromise between the too aggressive $R = 3$ and the robust but slower $R = 2$. Verified Null–Move Pruning [13]uses $R = 3$, but whenever the shallow null–move search returns a fail–high, instead of cutting, the search is continued with reduced depth. Verified null–move pruning can detect zugzwang positions, have better tactical strength while searching less nodes than standard $R = 2$.

The null–move heuristic is very effective in chess, and most of the strong chess engines use it. But it depends on the property that the right to move has positive value, so it is not useful to games like Othello and checkers, in which zugzwang positions are common.

2.3 ProbCut

ProbCut is based on the idea that the result v' of a shallow search is a rough estimate of the result v of a deeper search. The simplest way to model this relationship is by means of a linear model:

$$v = a \cdot v' + b + e,$$

where e is a normally distributed error variable with mean 0 and standard deviation σ. The parameters a, b, and σ can be computed by linear regression applied to the search results of thousands of positions.

If based on the value of v', we are certain 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 algebraic manipulations, 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 standard Normal distribution. This inequality 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 earlier 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 depth pair and cut threshold are to be determined empirically, by checking the performance of the program with various parameter settings.

For ProbCut to be successful, v' needs to be a good estimator of v, with a fairly small σ. This means that the evaluation function needs to be a fairly accurate estimator of the search results. Evaluation func-

```
#define  S   4    // depth of shallow search
#define  H   8    // check height
#define  T   1.0  // cut threshold

int AlphaBeta(int alpha, int beta, int height) {
    if (height == 0) return Evaluation();

    if (height == H) {
      int bound;

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

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

    // The rest of alpha-beta code goes here
    ...
}
```

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

tions for chess are generally not very accurate, due to opportunities of capturing which cannot be resolved statically. Fortunately, most chess programs conduct a so–called quiescence search: at the leaves of the game tree where the regular search height reaches zero, instead of calling the evaluation function, a special quiescence search function is called to search only capturing moves, only using the evaluation function's results when there are no profitable capturing moves. Quiescence search returns a much more accurate value.

In summary, the null–move heuristic and ProbCut both try to compensate for the lower accuracy of the shallow search by making it harder for the shallow search to produce a cut. The null–move heuristic does this by giving the opponent a free move, while ProbCut widens the alpha–beta window.

2.4 Multi–ProbCut

MPC enhances ProbCut in several ways:

- Allowing different regression parameters and cut thresholds for different stages of the game.
- Using more than one depth pair. For example, when using depth pairs (3,5) and (4,8), if at check height 8 the 4–ply shallow search does not produce a cut, then further down the 8–ply subtree we could still cut some 5–ply subtrees using 3–ply searches.
- Internal iterative deepening for shallow searches.

Figure 3 shows pseudo–code for a generic implementation of MPC. The MPC search function 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. 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 The null–move heuristic has been successfully applied to 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 the null–move heuristic does not work in Othello because it is a zugzwang game. But in chess, ProbCut and MPC have to compete with null–moves, which already improves upon brute–force alpha–beta search.

```
#define MAX_STAGE   2   // e.g. middle-game, endgame
#define MAX_HEIGHT 10   // max. check height
#define NUM_TRY     2   // max. number of checks

// ProbCut parameter sets for each stage and height

struct Param {
  int d;                // shallow depth
  float t;              // cut threshold
  float a, b, s;        // slope, offset, std.dev.
} param[MAX_STAGE+1][MAX_HEIGHT+1][NUM_TRY];

int MPC(int alpha, int beta, int height) {

  // ProbCut check
  if (height <= MAX_HEIGHT) {
    for (int i=0; i < NUM_TRY; i++) {
      int bound;
      Param &pa = param[stage][height][i];

      // skip if there are no parameters availabe
      if (pa.d < 0) break;

      // is v_height >= beta likely?
      bound = round((pa.t*pa.s+beta-pa.b)/pa.a);
      if (AlphaBeta(bound-1, bound, pa.d) >= bound)
        return beta;

      // is v_height <= alpha likely?
      bound = round((-pa.t*pa.s+alpha-pa.b)/pa.a);
      if (AlphaBeta(bound, bound+1, pa.d) <= bound)
        return alpha;
    }
  }
  // the remainder of the alpha-beta algorithm
  ...
}
```

Figure 3. Multi–ProbCut implementation. AlphaBeta() is the original alpha–beta search function.

2 The probability of a chess search making a serious error is relatively high, probably due to the higher branching factor [9]. This leads to a relatively large standard deviation in the linear relationship between shallow and deep search results, which makes it harder for ProbCut to prune sub–trees.

In the GAMES group at the University of Alberta there had been attempts to make ProbCut work in chess in 1997 [8]. However, the cut–thresholds were chosen too conservatively resulting in a weak performance.

Recently, researchers in Japan have successfully applied ProbCut to Shogi [12]. In Shogi programs forward pruning methods are not widely used, because Shogi endgames are much more volatile than chess endings. Therefore, ProbCut by itself can easily improve search performance compared with plain alpha–beta searchers. As mentioned above, gaining improvements in chess, however, is much harder because of the already very good performance of the null–move heuristic.

3. ProbCut Implementation

Before trying MPC, we implemented the simpler ProbCut heuristic with one depth pair and incorporated it into Crafty (version 18.15) by Robert Hyatt.[1] 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, Adaptive Null–Move heuristic, etc. Crafty also utilizes quiescence search, so the results of its evaluation function plus quiescence search are fairly accurate.

The philosophy of our approach is to take advantage of the speed–up provided by the null–move heuristic whenever possible. One obvious way to combine the null–move and ProbCut heuristics is to view null–move search as part of the brute–force search, and build ProbCut on top of the "alpha–beta plus null–move" search. Applying the necessary changes to Crafty is easy. We put the ProbCut shallow search code in front of the null–move shallow search code. We also implemented the MPC feature that allows different parameters to be used for middle–game and endgame.

Before ProbCut–Crafty could be tested, parameters of the linear ProbCut opinion change model had to be estimated. We let Crafty search (using alpha–beta with null–move heuristic) around 2700 positions and record its search results for $1, 2, \ldots, 10$ plies. The positions were cho-

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

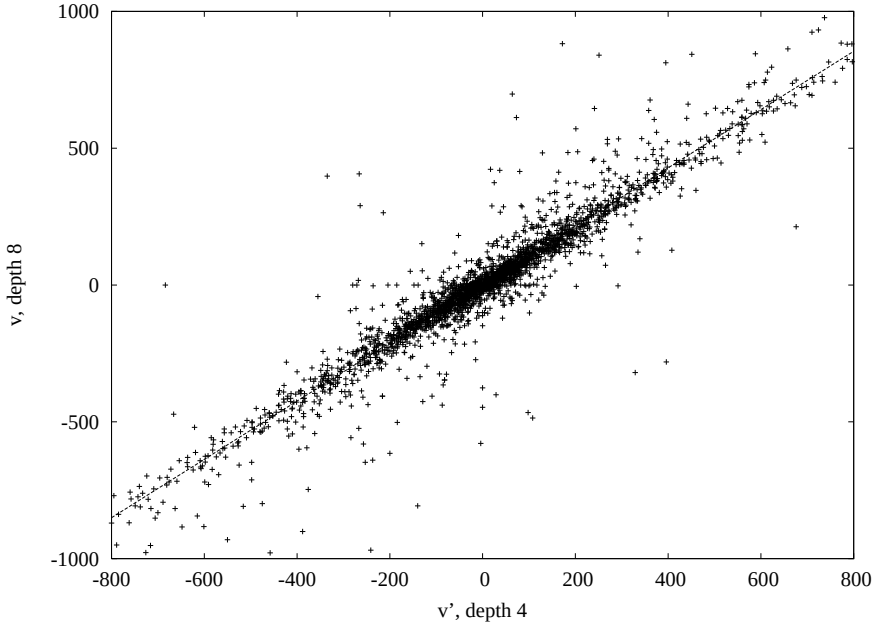

Figure 4. v' versus v for depth pair (4,8) The evaluation function's scale is 100 = one pawn, i.e. a score of 100 means the player to move is one pawn up (or has equivalent positional advantage).

sen 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 the null–move heuristic for these searches.

Then we fitted the linear regression model for several depth pairs and game phases, using the data collected. The results indicate that shallow and deep search results are correlated, as shown in Figure 4. However, the fit is not perfect. The v' versus v relation has the following characteristics:

- The slope is closer to 1.0 and the standard deviation smaller for v' data points closer to zero, For example, for depth pair (4, 8), and v' data points in the range $[-300, 300]$, the slope is 1.07 and the standard deviation is 83; for v' data points in the range $[-1000, 1000]$, the slope is 1.13 and the standard deviation is 103. 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. We only used v' data points in the range $[-300, 300]$ for the linear regression.

Table 1. Linear regression results. The evaluation function's scale is 100 = one pawn. r is the regression correlation coefficient, a measure of how good the data fits the linear model.

Pairs	Stage	a	b	σ	r
(3,5)	middle–game	0.998	−7	55.8	0.90
(3,5)	endgame	1.026	−4.1	51.8	0.94
(4,8)	middle–game	1.02	2.36	82	0.82
(4,8)	endgame	1.11	1.75	75	0.90

- Occasionally the shallow search misses a check–mate while the deeper search finds it. For example, in a position White can check–mate in 7 plies. A 4–ply search cannot find the check–mate while a 8–ply search can find it. For the depth pair (4, 8), and v' data points in the range $[-300, 300]$, this happens roughly once every 1000 positions. A check–mate–in–N–moves is represented by a very large integer in Crafty. We excluded these data points from the linear regression, because the evaluation of check–mate is a rather arbitrary large number, there is no proper way to incorporate these data points in the linear regression.

We also fitted model parameters for different game stages. It turned out that the standard deviation for the fit using only endgame positions[2] is smaller than the standard deviation using only middle–game positions. Table 1 shows some of the results.

We conducted some experiments[3] with different depth pairs and cut thresholds. Depth pairs $(4, 6)$ and $(4, 8)$, and cut thresholds 1.0 and 1.5 were tried. We used two types of tests. First, we test the search speed by running fixed–time searches and look at the depths reached. If a ProbCut version is not faster than the plain null–move version, then the ProbCut version is clearly no good. If a ProbCut version is faster than null–move, it is still not necessarily better. So to test the overall performance, we then run matches between the promising ProbCut versions and the original Crafty.

[2] In Crafty endgame positions are defined as those in which both players have weighted material count less than 15. Here queen is 9, rook is 5, knight/bishop is 3, and pawns don't count.

[3] All initial experiments were run on Pentium–3/850MHz and Athlon–MP/1.66GHz machines under Linux, whereas the later tournaments were all played on Athlon–MP/2GHz machines. Crafty's hash table size was set to 48 MBytes, and the pawn hash table size to 6 MBytes. Opening books and thinking on opponent's time was turned off.

We 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 heuristic does at the check height.

Because the cut threshold 1.5 is no good, we concentrated on the threshold 1.0 for the following experiments. We 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[4] shows that in a 64–game match, a score above 38 points (or 59%) is statistically significant with $p < 0.05$. Here a win counts one point and a draw counts half a point.

The match results are not statistically significant. The ProbCut versions seem to be no better nor worse than plain Crafty. For comparison, we 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 and Results

ProbCut produces more cuts than the plain null–move heuristic does, but it seems that the small speed–up provided by ProbCut is not enough to result in better playing strength. This motivates our implementation of MPC. We already have different regression parameters for middle–game and endgame in our ProbCut implementation. Now we implemented multiple depth pairs. The implementation was straightforward, much like the pseudo–code in Figure 3.

[4]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

Table 2. Endgame threshold optimization results. Reported are the point percentages for MPC–Crafty playing 64–game tournaments against Crafty using different values for the endgame cut thresholds. Game timing was 2 minutes per player per game plus 12 seconds increment on an Athlon–MP 1.67 GHz. The middle–game threshold was fixed at 1.0.

t_{end}	0.5	0.6	0.7	0.8	0.9	1.0	1.05	1.1	1.2	1.3
MPC %	53.9	59.3	53.1	48.5	51.6	57.8	52.3	54.7	51.6	51.6

Table 3. Middle–game threshold optimization results. With the endgame threshold fixed at 1.0 we repeated the 64–game tournaments now using faster hardware (Athlon–MP 2 GHz) that just became available and longer time controls: 10 minutes per player per game plus 60 seconds increment. Each tournament took about eight CPU days.

t_{mid}	0.8	0.9	1.0	1.1	1.2	1.3
MPC %	54.7	59.4	57.8	58.6	59.4	53.1

After initial experiments which showed that the null–move heuristic excels at small heights, we chose depth pairs (2,6), (3,7), (4,8), (3,9), and (4,10) for endgames and middle–games. Another reason for choosing pairs with increasing depth differences is that otherwise the advantage of MPC rapidly diminishes in longer timed games. We tested the speed of the MPC implementation using a cut threshold of 1.0 on the same 300+ positions as in Section 1.3. With 30 seconds per position, it is able to search 12.0 plies on average, which is 0.5 plies deeper than original Crafty.

For optimizing the endgame and middle–game cut thresholds we then ran two sets of 64–game tournaments between MPC–Crafty and the original version. In the first phase we kept the middle–game cut threshold fixed at 1.0 and varied the endgame threshold. The results shown in Table 2 roughly indicate good threshold choices. However, the high fluctuations suggest that we should play more games to get better playing strength estimates. After some more experimentation we fixed the endgame threshold at 1.0 and went on to optimizing the middle–game cut threshold by playing a second set of tournaments, now on faster hardware and longer time controls. Threshold pairs $(1.2, 1.0)$ and $(1.0, 1.0)$ resulted in the highest score (59.4%) against the original Crafty version.

In order to validate the self–play optimization results, we finally let MPC–Crafty play a set of tournaments against Yace — a strong chess program written by Dieter Buerssner which is available for Linux and can be downloaded from http://home1.stofanet.dk/moq/. Table 4 summa-

Table 4. Results of 64–game tournaments played by three Crafty versions against Yace using two different time controls.

Pairing	Crafty % (2min+10sec/move)	Crafty % (8min+20sec/move)
Crafty vs. Yace	42.0%	50.8%
MPC–Crafty (1.2,1.0) vs. Yace	53.1%	56.3%
MPC–Crafty (1.0,1.0) vs. Yace	57.0%	55.5%

rizes the promising results which indicate a moderate playing strength increase even against other chess programs when using MPC.

5. Conclusions and Further Research

Preliminary results show that MPC can be successfully applied to chess. Our MPC implementation shows clear improvement over our ProbCut (plus variable parameters for different stages) implementation. This indicates that the main source of improvement in MPC is the use of multiple depth pairs. Due to the already good performance of the null–move heuristic in chess, the improvement provided by MPC in chess is not as huge as in Othello. However our implementation, which combines MPC and null–move heuristic, shows definite advantage over the plain null–move heuristic in Crafty, as shown by the match results. MPC is relatively easy to implement. We encourage chess programmers to try MPC in their chess programs.

More experiments need to be conducted on our MPC implementation to determine how evaluation function parameters like the King safety weight can influence MPC's performance. To further verify the strength of the MPC implementation, we plan to run matches with even longer time controls.

The depth pairs and the cut threshold can be further fine–tuned. One way to optimize them is to run matches between versions with different parameters. But better results against another version of the same program do not necessarily translate into 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 [9], using a deeper search as the "oracle," and looking at the difference between the oracle's evaluations on the oracle's best move and the move chosen by the search function we are measuring. Maybe the combination of the above two methods gives a better indication of chess strength.

6. Acknowledgments

We would like to thank David Poole for his helpful comments, and Bob Hyatt for making the source code of his excellent and very readable Crafty chess program available to the public.

References

[1] D.F. Beal. A generalized quiescence search algorithm. *Artificial Intelligence*, 43:85–98, 1990.

[2] M. Buro. ProbCut: An effective selective extension of the alpha–beta algorithm. *ICCA Journal*, 18(2):71–76, 1995[1].

[3] M. Buro. Experiments with Multi–ProbCut and a new high–quality evaluation function for Othello. *Workshop on game–tree search, NECI*, 1997[1].

[4] M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *ICCA Journal*, 20(3):189–193, 1997[1].

[5] C. Donninger. Null move and deep search: Selective search heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143, 1993.

[6] E.A. Heinz. Adaptive null-move pruning. *ICCA Journal*, 22(3):123–132, 1999.

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

[8] A. Junghanns and M. Brockington. Personal communication. 2002.

[9] A. Junghanns, J. Schaeffer, M. Brockington, Y. Björnsson, and T. Marsland. Dimishing returns for additional search in chess. In H.J. van den Herik and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 53–67, 1997. ISBN 9-062-16234-7.

[10] D.E. Knuth and R.W. Moore. An analysis of alpha–beta pruning. *Artificial Intelligence Journal*, 6:293–326, 1975.

[11] A. Reinefeld. An improvement of the Scout tree search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[12] K. Shibahara, N. Inui, and Y. Kotani. Effect of ProbCut in Shogi — by changing parameters according to position category. In *Proceedings of the 7th Game Programming Workshop*, Hakone, Japan, 2002.

[1] The author's articles can be downloaded for personal use from http://www.cs.ualberta.ca/~mburo/publications.html

[13] O.D. Tabibi and N.S. Netanyahu. Verified null-move pruning. *ICGA Journal*, 25(3):153–161, 2002.