SEARCH IDEAS IN CHINOOK

Jonathan. Schaeffer[1]

Department of Computing Science,
University of Alberta
Alberta, Edmonton,
Canada T6G 2H1

Abstract

The checkers-playing program CHINOOK is the World Man-Machine Checkers Champion. The program's strength comes from deep alpha-beta searches, combined with a carefully tuned evaluation function and large, powerful endgame databases. These components are not independent of each other. This paper describes a few of the search innovations in CHINOOK and their interactions with the rest of the program.

1. INTRODUCTION

In 1994, the checkers (8×8 draughts) program CHINOOK won the World Man-Machine Championship, becoming the first program to win a world championship from a human opponent in any game. Since then, the program has defended its title twice, and has established a ranking that is well above the best human player alive. With no human opposition on the horizon, CHINOOK was retired in 1997. The full CHINOOK story is detailed in Schaeffer (1997).

CHINOOK's playing strength comes from four components:

(a) *Search*. The low branching factor in checkers (1.1 in capture positions; 7.8 in non-capture positions (Lu, 1994)) allows the program to search quite deeply. The parallel tournament version of CHINOOK using 1994 hardware would typically search a minimum of 21 ply deep, with search extensions often reaching over 40 ply. The median position evaluated is usually around 25 ply.

[1] Email: jonathan@cs.ualberta.ca.

(b) *Knowledge*. The program has a finely tuned evaluation function that has remained stable since 1992. It was developed in consultation with a checkers expert (a class below master level). It is a linear combination of 26 heuristic terms. There are four phases of the game (opening, early middlegame, late middlegame and endgame), with each heuristic term having a different weight for each game phase.

(c) *Endgame databases*. All endgame positions with eight or fewer pieces on the board have been computed (444 billion positions, compressed into 6 gigabytes for real-time decompression (Lake, Lu, and Schaeffer, 1993)). CHINOOK's deep searches mean that the endgame database is accessed throughout every search during a game (including the first move!). Because of the frequency of database accesses and the size of the databases, even on a machine with one gigabyte of RAM, the program quickly becomes I/O bound. This is in sharp contrast to most other game-playing programs; they are usually compute bound and have very little I/O. The database values (win, loss and draw) have no heuristic error, and hence improve the accuracy of the search.

(d) *Opening book*. Extensive work has been done on acquiring and improving the major lines of play in all 144 openings. Unlike most games, where a player can choose an opening to play, in checkers the opening three moves are randomly selected before the game. Hence, one has to study *all* the openings.

This paper discusses a number of the search enhancements in CHINOOK that are likely to be of interest to other search-based game-playing programs. Section 2 introduces an enhancement for the critical case where an iterative-deepening search fails low. Section 3 shows how the transposition table information can be used in new ways. Endgame databases are used throughout the search, resulting in heuristic scores (with errors) being mixed with accurate scores (no error). Section 4 shows that even an accurate score (draw) may need a heuristic assessment. Section 5 introduces tactical tables, a way of pre-computing tactical patterns so that table lookups can be used to replace search effort. Finally, Section 6 provides some concluding remarks.

2. FAILING LOW AT THE ROOT

Most programs use an aspiration window around the expected search value (EV) to improve the efficiency of alpha-beta searching. Instead of searching with a full window $[-\infty, +\infty]$, the window is centered around the expected value [EV-δ, EV+δ], for some small value δ (typically valued at half a pawn in chess, and half a checker in checkers). This works well when the search value falls within the aspiration window (EV–δ < value < EV+δ). When the search *fails high* (EV+δ ≤ value), we have a lower bound on the true score and have to re-search to find the exact value [value, +∞]. This is usually not a problem, since the score is improving, However, the reverse case, *failing low*, is much more serious (value ≤ EV-δ). Here our search expectations have gone seriously awry; we have only an upper bound on the score of the "best" move. The usual method for resolving this is to re-search the move with a wider search window [-∞, value] to find its actual value, and then continue searching the rest of the moves to see if an alternative move yields a better score.

Fail-low results are of critical importance to the program, since the search expectations are changing. Further, it may take a significant amount of time to re-search the fail-low move (possibly finding out that the move is losing) before desperately searching for an alternative. In the meantime, the program may have run out of time allocated to this move, possibly without finding an improvement.

CHINOOK solves this problem be restarting the iterative-deepening search from the beginning when a fail low occurs. Information in the transposition table allows the alternative moves, if better, to show up in the early iterations. If the best move fails at depth *D*, then the other moves have scores in the transposition table for depth *D-1*. By restarting the search, the alternative move's transposition table scores will allow for a better move reordering with a low search depth (such as depth *D = 1*), rather than continuing the expensive *D*-ply iteration. Almost immediately after a fail low, a new best move (if available) will emerge. Figure 1 illustrates the enhancement, with the additional code required highlighted in bold.

Experience with this enhancement in CHINOOK is limited, since the program is good enough that its searches rarely fail low (except at shallow search depths). In the period 1994-1996 (ending with CHINOOK's retirement), CHINOOK played 228 games against Grandmasters and Masters. In that span, there were 22 non-trivial fail-low searches:

1 : lost position – the result of the search did not matter;
6 : a positive score for the program (advantageous position) was lowered, but there was no change in the choice of move played;
1 : a positive score for the program was lowered, resulting in a better move being chosen;
3 : the score dropped to a large negative score (the program will have a bad or losing position if the move being searched is made); in each case, the fail-low enhancement allowed CHINOOK to find a significantly better move.

```
  value = Evaluate( position );
Restart:
  for( depth = 1; depth <= MAX_DEPTH; depth++ ) {
    alpha = value - ASPIRATION_WINDOW;
    beta = value + ASPIRATION_WINDOW;
    value = AlphaBeta( position.moves[1], alpha, beta, depth);
    /* On a fail low, restart the search */
    if( value <= alpha )
       goto Restart;
    /* Rest of iterative deepening */
    . . .
    /* value = MAX score of all the moves */
    if( EffortExceeded() )
       break;
  }
```

Figure 1: Handing a fail low in an iterative-deepening search.

The last category is the critical one. In each case the correct move was found within *one second* and its correctness verified by the rest of the search. Had CHINOOK wasted its time re-searching the previously "best" move, in at least two of the cases the program may have moved into a lost position had the time allocated to the search expired before the problem was resolved.

3. BETTER USE OF TRANSPOSITIONS

Consider a cutoff node N, with move B causing the cutoff, as in Figure 2. In traditional implementations of alpha-beta with transposition tables, all subsequent searches at this node will consider only move B, as long as it continues to cause a cutoff. However, what if move C is legal at node N and it happens to transpose into another part of the search tree? If C is also capable of causing a cutoff at N, then the program should choose the move that achieves the cutoff with the least amount of search effort.

Enhanced Transposition table Cutoffs (ETC) attempt to maximize the use of information available in the transposition table (Plaat *et al.*, 1996; Plaat, 1996). The usual alpha-beta implementation with transposition tables always takes the transposition table move as the "best" move to play in a position. However, search information gathered from elsewhere in the tree may affect this information. If the program blindly plays the same move in the same position, opportunities for reduced search effort may be missed. In particular, new search results may be available that suggest a cheaper way of causing a cutoff.

Figure 3 shows how to integrate ETC into alpha-beta, with the additional ETC coded highlighted in bold. By directing the search towards previously visited

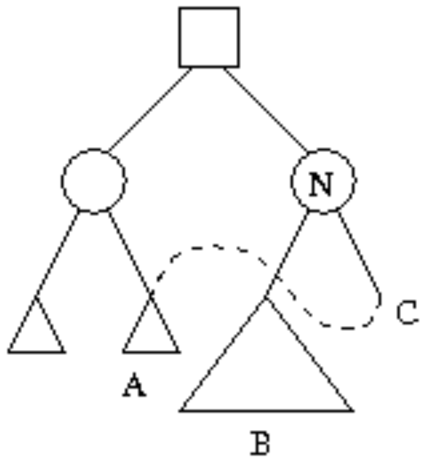

Figure 2: The left-most cutoff move may not be the cheapest.

```
int AlphaBeta( position, alpha, beta, depth ) {
   . . .
   /* Regular transposition table lookup */
   GetTrans( position, &result, &value, &transdepth );
   if( transdepth >= depth ) {
      if( result == LOWER_BND && value > alpha )
        alpha = value;
      if( result == UPPER_BND && value < beta )
        beta = value;
      if( result == ACCURATE || alpha >= beta )
        return( alpha );
   }
   numbmoves = GenerateMoves( position, moves );

   /* Check for an enhanced transposition cutoff */
   for( i = 1; i <= numbmoves; i++ ) {
      GetTrans( position.moves[i], &value, &transdepth );
      /* Since a move has been played, we are one ply */
      /* deeper into the search. Also note that the    */
      /* type of this node (max or min) is now rev-    */
      /* ersed, meaning that in a Negamax framework,   */
      /* the sign on the retrieved transposition       */
      /* table value is reversed.                      */
      if( transdepth < depth-1 )
        continue;
      value = -value;
      if( result == LOWER_BND && value < beta )
        beta = value;
      if( result == UPPER_BND && value > alpha )
        alpha = value;
      if( result == ACCURATE || alpha >= beta )
        return( alpha );
   }
   /* Continue with the normal alpha-beta search */
   . . .
}
```

Figure 3: Enhanced transposition table cutoffs (ETC).

nodes, ETC maximizes information reuse, thereby reducing the number of nodes visited during a search.

In CHINOOK, ETC results in search trees with 22% fewer nodes (17-ply searches). With ETC enabled at all nodes in the tree, the reduction in nodes searched is offset by the increased computation per node. However, the biggest transposition savings occur near the root of the tree. Hence, ETC was disabled for the last two ply of the search, slightly reducing their effectiveness but substantially reducing their execution overhead. Also, note that ETC is even more effective in CHINOOK's parallel search algorithm.

ETC has also been implemented in chess, with a 28% reduction in search tree size for 8-ply searches (Plaat *et al.*, 1996; Plaat, 1996).

4. ALL DRAWS ARE NOT EQUAL

CHINOOK is unique in that the program needs to access endgame databases throughout an entire game. However, mixing accurate evaluations (database lookups) with heuristic evaluations (evaluation function) can result in poor play. In particular, scoring a draw as a 0 (equality), as is done in most game-playing programs, is a mistake (Beal, 1984). The program needs to be able to differentiate between "strong" draws (positions where there is a high probability that the opponent will make a mistake) and weak draws (ones where the opponent plays on hoping for the computer to err). Since CHINOOK often declares the final (drawn) result of a game within 15 moves of the start, it is important that the program continue to play to maximize its chances of inducing a mistake.

The alpha-beta algorithm returns the minimax value of the search and a "best" move that leads to that value. However, is the highest minimax score adequate for defining the "best" move? Consider these events:

- Should a draw be scored as a zero (0)? In game 26 of the 1992 Tinsley-CHINOOK World Man-Machine Championship match, CHINOOK chose a move that preserved a small advantage (small positive minimax score). However, an alternate move led to a provable draw (courtesy of deep search and the endgame databases), but with many opportunities for the opponent to go wrong. Objectively, the alternative move offered better chances for CHINOOK to win the game.

- Should all draws have the same score? In game 37 of the Tinsley-CHINOOK match, CHINOOK's deep searches allowed it to back up a draw to the root of the search. That is not so unusual. What is unusual is that this occured on move 5 of the game! Clearly it is too early to shake hands and move on to the next game of the match. In this situation, CHINOOK must select the drawing move that maximizes the chances for an opponent error.

Various schemes were tried to solve this problem. They involved assigning a non-zero score to a drawn position, a score that attempted to reflect the chances for opponent error. Some of the schemes that we experimented with include using expert-specified heuristics (too hard to obtain and code), using the evaluation function to score the drawn position (easy to implement but too simple an assessment of future chances), trying to correlate properties of the search tree with difficulty (for example, selecting the move that maximizes the number of forced opponent moves), and identifying forced moves (with respect to the database) that were not identified as such by the evaluation function (Levy, 1991). Although all these ideas holds some promise, in practice they did not seem to yield the desired results.

CHINOOK now allows database draw positions found in the search to be assigned positive and negative scores, corresponding to the notion of strong and weak draw. Unlike most programs, CHINOOK will select a drawing line over a non-drawing line if it perceives there is a better chance of winning. When CHINOOK comes across a drawn database position, it does a depth-limited search with the databases turned off. By turning off the databases, we can use the result of a search using the heuristic evaluation function to score this position. So, for example, if a database drawn position leads to a position up a checker, this assessment can be used to score the draw. Clearly, if the program is up a checker, even though the position is drawn—assuming perfect play—there is a real chance that the opponent will find this line difficult to draw.

Assigning a non-zero score for a draw can introduce some subtleties in the search result. What if a drawing line returns a score of –50, while a non-drawing line returns a score of –20. Clearly, alpha-beta will prefer the higher score of –20. But is this correct? The –20 may lead to a dangerous position, but the –50 leads to a draw. Hence the score of the drawing line must be "high" enough that it is preferred over weaker, non-drawing lines. One possibility is to make all draws with negative scores a zero. In this case all weak draws will have the same score (0), and the choice of move will essentially be random. It is better to have a small range of negative draw scores, allowing the program to use alpha-beta to maximize the draw score, allowing it to choose the "best" of the weak draws.

An important point to note is that when CHINOOK backs up a draw score to the root of the search, it is not guaranteed to be a draw. This is a consequence of the error in heuristic evaluations. A node N might have two children, A and B. Move A leads to a database draw (score is zero); move B returns a score of -50. Hence, alpha-beta will prefer A over B. However, if there is a large error in the evaluation of B, because of an overlooked tactic for example, then on the next iteration node N might discover that B has a higher score than A. Hence, one cannot always treat a backed-up draw score as proof that the position is now drawn. Thus, one has to be careful with interpreting negative draw scores.

As well, one has to be cautious about the interpretation of positive draw scores. Consider a provable draw that returns a score of winning a checker. Should this line be preferred over a line that does not win a checker, but is not a provable draw? Since the draw score is only an estimate of the draw difficulty (being a heuristic score, it could have a large error), in CHINOOK the positive draw scores are lowered by a factor of two to reflect our uncertainty. Also, we impose a maximum on the size of the draw score, to prevent moves with high draw scores being preferred over moves with serious winning chances.

These issues arise because all draws are not the same. Because of CHINOOK's extensive endgame databases, heuristic values (evaluation function) are intermingled with accurate values (database results) throughout all searches (even from the start of the game). A heuristic value assesses the likelihood of winning; a higher score indicates a higher winning probability. Database scores, however, do not indicate a probability; they indicate the final result assuming perfect play by both sides. Given a fallible opponent, draw scores need to reflect not only the fact that they lead to a draw, but also the probability of winning (chances of an opponent making an error).

Figure 4 shows a simplified version of CHINOOK's draw scoring code. The actual code to assign scores to favorable and unfavorable draws is slightly more complicated, being a non-linear function. As well, scores from draws are backed up with a special bit being set indicating that the score is for a drawn position. This allows us to identify moves at the root which lead to a draw.

Draw searches are a gamble. At a database draw position in the search, you have an accurate value for the node and can immediately return from the recursive alpha-beta call. Instead, you are going to continue searching, albeit to a reduced search depth, to refine that draw score. This is additional computational effort that negates some of the search effort reductions provided by the databases. The issue then becomes a trade-off: are the benefits of the information uncovered by the search worth the additional search effort?

```
/* Reduce the search depth for draw searches */
#define DRAW_DEPTH_REDUCTION      2
/* Search window for draw searches. Can be improved  */
/* by combining it with the alpha/beta values for    */
/* this position.                                    */
#define DRAW_SEARCH_WINDOW        CHECKER_VALUE
/* Scale back the search score, differentiating bet- */
/* better (positive) and worse (negative) positions. */
#define DRAW_POSITIVE_REDUCTION   2
#define DRAW_NEGATIVE_REDUCTION   10

if( DataBaseResult() == DRAW ) {
    DataBaseUsage = OFF;
    drawdepth = depth / DRAW_DEPTH_REDUCTION;
    alpha = -DRAW_SEARCH_WINDOW;
    beta  =  DRAW_SEARCH_WINDOW;
    value = AlphaBeta( position, alpha, beta, drawdepth );

    /* Score the position differently whether it is    */
    /* advantageous to the program or not.             */
    if( ( ToPlay == Chinook && value >= 0 ) ||
        ( ToPlay != Chinook && value <  0 )  )
         value = value / DRAW_POSITIVE_REDUCTION;
    else value = value / DRAW_NEGATIVE_REDUCTION;

    DataBaseUsage = ON;
    return( value );
}
```

Figure 4: Draw searches in CHINOOK.

It is difficult to quantify the significance of the draw-differentiation enhancement. One could play a series of self-play matches to determine if the draw code is beneficial. For example, CHINOOK with the 8-piece databases and draw-differentiation code could play a match against CHINOOK with the 4-piece databases and no draw-differentiation code. This interesting experiment has not been performed. Instead, we offer some empirical evidence. In the 1996 US Championship, using the draw-differentiation code, CHINOOK won 8 of the 26 games in which it announced a draw. We can contrast this with the 1990 U.S. Championship where CHINOOK did not differentiate between draws. Here only 3 of 23 drawn games were won.

5. TACTICS TABLE

Most game-playing programs attempt to make sure that heuristic evaluations are only done for quiescent positions. Usually the criteria used for defining a quiescent position are based on some simple tactical analysis. In chess, for example, a quiescence search is done to determine whether the leaf node being evaluated contains any hidden tactics that might radically change the material balance. In a typical chess program, the quiescent searches dominate the total search time. It is critical to know whether your heuristics for identifying quiescence are working in practice.

With the aid of Stef Keetman's 10×10 checkers program, Truus, an analysis tool was written to identify the positional features of tactical oversights that occurred at leaf nodes in CHINOOK. Basically, if the material balance of a leaf position differed from the result obtained from a shallow (3-ply) search, then the analysis tool could identify the tactical features that led to the win/loss of material. Several thousand positions were analyzed, and the resulting patterns (if any) were logged to a file. This data was then manually inspected. Analysis of the data showed that one type of tactical combination accounted for almost 50% of the positions flagged (two additional patterns accounted for another 25% of the combinations).

Consider the checkers position in Figure 5. White to move has a "two-for-one": White moves d2-c3, forcing b4xd2 and c1xe3xg5, winning a checker. For this combination of moves to work, several conditions must be in place: there must be White pieces on c1 and d2; there must be Black pieces on b4 and f4; and squares c3, e3, and g5 must be empty (indicated by "---" in the diagram). But that's not enough. Once White wins the piece, we must also make sure that Black can't immediately recapture it. For example, there can't be a Black piece on h6 ("×" meaning "not" in the figure); otherwise, Black could regain the lost piece with h6xf4. Similarly, Black shouldn't have a recapture with f6xh4. A number of conditions must hold for this re-capture to be possible (shown by the line from e7 to h4). The absence of any of those conditions negates the possible recapture. These conditions include h4 being empty, f6 being occupied by a Black piece, and e7 being occupied.

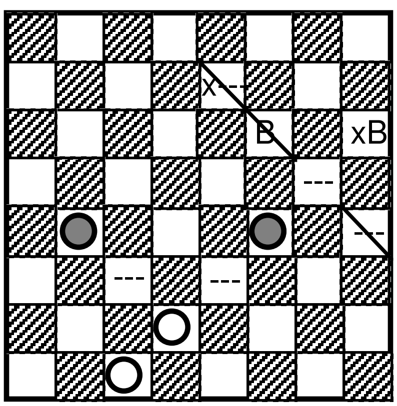

Figure 5: Illustrating the conditions necessary for a "two-for-one" to work.

Translating this information into code was difficult. After a frustrating day of coding *if-then-else* statements, this was abandoned in favor of a more brute-force approach: enumerate all possible patterns and pre-compute the search results. The squares involved in this pattern require consideration of 24 of the 32 squares on the board. By generalizing some of the features, a table of roughly 8,000,000 entries was constructed. Each entry corresponded to a class of positions. A program set up a representative position from each of these classes and did a search to see if the tactical combination was present or not. Each table entry contained one bit indicating whether for that class of positions it was possible to initiate a two-for-one combination to win a checker or not. We call this precomputed result a *tactics table*.[1]

Before a leaf node is evaluated, the tactics table is consulted. The contents of the relevant squares are mapped to a unique index in the table. If the table entry bit indicates a winning combination, the search is extended to play out the combination. CHINOOK knows it is winning at least a checker, however it needs to evaluate the position at the end of the combination. It is possible that the win of the checker(s) could be obtained at the expense of positional considerations. Alternatively, having won the checker, the resulting position may now contain a combination for the opponent to regain the checker.

The tactics table had some pleasant surprises for us. First, one would predict that CHINOOK could now solve some checkers problems (find the right move) with a shallower search depth. On our internal test set, several problems were indeed solved at an earlier iteration than before. Second, since finding a two-for-one results in a search extension, one would expect that on average the program would build larger search trees. Surprisingly, the program ran *faster*. Why? Without the tactics table, the two-for-one combinations were occurring so frequently that they caused search inefficiencies in the iterative deepening (for some nodes, the depth D-1 best move no longer was best at depth D). With the tactics table, CHINOOK found things earlier in the search and ended up avoiding doing a lot of wasteful work.

6. SUMMARY

This paper presents four search ideas that are implemented in the checkers program CHINOOK. All four have been instrumental in CHINOOK'S success. Although none of the ideas is in itself a major contribution to our understanding of game-playing programs, each played a significant role in creating a world-championship-caliber program.

[1] This is an instance of a pattern database (Culberson and Schaeffer, 1998).

7. ACKNOWLEDGEMENTS

I would like to thank Hiroyuki Iida for the invitation to talk about CHINOOK at the IJCAI'97 workshop on computer games. This research has been funded by the Natural Sciences and Engineering Research Council of Canada.

8. REFERENCES

Many of the CHINOOK-related references can be found at:www.cs.ualberta.ca-/~chinook.

Beal, D.F. (1984). Mixing Heuristic and Perfect Evaluations: Nested Minimax. *ICCA Journal*, Vol. 7, No. 1, pp. 10-15. ISSN 0920-234X.

Culberson, J. and Schaeffer, J. (1998). Pattern Databases. *Computational Intelligence*, Vol. 14, No. 3, pp. 318-334.

Lake, R. and Schaeffer, J., and Lu, P. (1993). Solving Large Retrograde-Analysis Problems using a Network of Stations. *Advances in Computer Chess 7* (ed. H.J. van den Herik, I.S. Herschberg and J.W.H.M. Uiterwijk), pp. 135-162. University of Limburg, Maastricht, The Netherlands. ISBN 90 6216 1014.

Levy, D. (1987). Improving the performance of Endgame Databases. *ICCA Journal*, Vol. 10, No. 4, pp. 191-192. ISSN 0920-234X.

Levy, D. (1991). First Among Equals. *ICCA Journal*, Vol. 14, No. 3, p. 142. ISSN 0920-234X.

Lu, P. (1994). *Parallel Search of Narrow Search Trees.* M.Sc. thesis, Department of Computing Science, University of Alberta, Edmonton, Canada.

Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de (1996). Exploiting Graph Properties of Game Trees. *American Association for Artificial Intelligence* (*AAAI*) National Conference, pp. 234-239.

Plaat, A. (1996). Research Re: Search & Re-Search. Ph.D. thesis, Erasmus University, Rotterdam, The Netherlands.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy at Checkers*. Springer-Verlag, New York, N.Y. ISBN 0-387-94930-5.