Applying the Experience of Building a High-Performance Search Engine for One Domain to Another

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

Abstract

This paper describes the high-performance alpha-beta-based search engine used in CHINOOK, the World Man-Machine Checkers Champion. Previous experience in designing a chess program was important in the initial design of the program. As it evolved, however, numerous application-specific modifications had to be made for the algorithms to excel at searching checkers trees. This paper describes the experience of transferring the technology used to develop a chess program to the creation of a high-performance checkers program.

1. Introduction

Developing high-performance game-playing programs has been a part of artificial intelligence research since the dawn of computers. Since the mid-1960s, this effort has been largely focused on developing a chess program capable of defeating the human world champion. In 1997, this long sought-after triumph was achieved by the DEEP BLUE team who edged out World Chess Champion Garry Kasparov in an exhibition match. Although it is not yet obvious whether the best chess program (DEEP BLUE) is in fact a better player than the human World Champion, in many people's mind this is of no relevance; Kasparov lost and a 50-year quest has come to an end. This was a triumph of engineering and artificial intelligence research, and a milestone in the history of computing.

Early on in the development of chess-playing programs, the power of the alpha-beta algorithm was discovered. This simple enhancement to the minimax algorithm allowed one to potentially reduce the search effort to the square root of the minimax search tree (Knuth and Moore, 1975). When alpha-beta was enhanced with transposition tables (Greenblatt, Eastlake, and Crocker, 1967; Slate and Atkin, 1977), it was possible to search fewer nodes than there are in the so-called minimal tree! With the discovery of a strong correlation between the search tree size and program performance (Thompson, 1982), much of the computer-chess research effort shifted to increasing the number of chess positions examined per second.

Over the last two decades, a variety of activities have come under the banner of "computer-chess research." This work can be broken down into the following categories:

1. Computers being used to do research into the mysteries of chess. Many new secrets of the game have been uncovered (for example, the impressive endgame database results). Of course, this increases our understanding of chess, but not of computer science.
2. Computer architecture engineering to build a high-performance (chess positions examined per second) machine. A good example is the building of special-purpose VLSI chips for chess (for example, Hsu, 1999). Although building special-purpose processors is an important topic, it is not clear whether this particular design has any relevance beyond chess.
3. Massively parallel search algorithms. The algorithms have been applied to parallel alpha-beta search (for example: Feldmann, 1993), but there is some hope that these ideas will transfer well to other non-game-playing search-based applications.
4. Sequential search algorithms. This has been the area of greatest activity, resulting in a plethora of alpha-beta enhancements that can significantly reduce the size of the search trees. Some of these ideas have been successfully applied to single-agent search (for example: Korf, 1985; Junghanns and Schaeffer, 2001).
5. Evaluation functions. Clearly, techniques for acquiring expert knowledge and integrating it into an evaluation function are important topics for artificial intelligence research. Unfortunately, this area has been largely ignored. Only recently have we seen some successful attempts at using chess as a testbed for main-stream artificial intelligence research, such as learning (Baxter, Trigell, and Weaver, 1998).
6. Chess as a testbed for AI research. Chess is an ideal domain for exploring many issues in artificial intelligence, such as tutoring systems, knowledge representation, learning and knowledge acquisition.

Sadly, much of the above research has been focused on chess-specific aspects, and not general artificial-intelligence research (Donskoy and Schaeffer, 1990). The research has concentrated on efficient search algorithms (because that is largely understood) and ignored the knowledge engineering aspects (because that is poorly understood). Hence the legacy of the enormous effort that has gone into computer chess may be efficient alpha-beta algorithms, and little else. Although it is easy to criticize some of the chess-related research, one must always keep in mind that it lead to the DEEP BLUE victory. Perhaps the greatest lesson learned from the DEEP BLUE experience is how expensive it is – in man-years, dollars and commitment – to build machines capable of competing with humans.

Now, more than ever, the question has to be asked as to the relevance of the computer-chess research. Were the research results applicable only to chess? If so, then the Deep Blue result has no long-term impact. At the very least, the techniques used to develop a high-performance chess program should be applicable to other games.

This paper describes the search algorithm used in the checkers-playing program CHINOOK (8□8 draughts). Having previously developed a chess program (PHOENIX:

Schaeffer, 1986), the obvious question was how easily the research done for chess could be transferred to checkers, a game with many similar properties. This paper describes how the technology transfer of search and knowledge went from one game-playing program to another. Many of the techniques are only briefly described. Detailed discussions are given for some of the interesting techniques that are either unique to CHINOOK or were pioneered in the CHINOOK project.

2. Background

The CHINOOK project began in 1989 and within two months we had a program that was good enough to win the First Computer Olympiad (Levy and Beal, 1989). One year later, the program was allowed to compete in the U.S. Championship, the biennial event used to determine the next challenger for the human world championship. Our agreement with the organizers covered the possibility of CHINOOK winning prize money and trophies, but there was no mention of what would happen should the program win the right to play for the world championship. In the seventh round of this event, CHINOOK became the first program to play a World Champion in a non-exhibition event, drawing four games with the champion, Dr. Marion Tinsley. The tournament was won by Tinsley, with an undefeated CHINOOK coming in second (winning five and drawing three four-game matches). With this result, CHINOOK became the first program in history to earn the right to play for a human championship.

Marion Tinsley was a remarkable man. He first won the world checkers championship in 1952, but retired in 1958. He returned to checkers in 1970, winning the right to challenge for the world championship, only to retire again when the World Champion (Walter Hellman) became ill. Tinsley returned in 1975 and won the world championship, which he held until he voluntarily relinquished it in 1991.

From 1950 to 1992, a period of 42 years, Tinsley amassed the most incredible record achieved in any competitive game or sport. Over that period, consisting of over 1,000 tournament and match games, Tinsley lost the amazing total of only 3 games. He was justifiably called the “unbeatable Tinsley”. He was as close to perfection in checkers as was humanly possible. The checkers federations did not like the idea of a computer playing for their championship and they refused to sanction the CHINOOK-Tinsley title match. In protest, Tinsley resigned as Champion in 1991, and then immediately signed an agreement to play CHINOOK. The checkers federations were thus forced to compromise, so they created a new title, the World Man-Machine Championship, pitting the best human (Tinsley) against the best computer program (CHINOOK). The first Man-Machine Championship was held in London in August 1992, with Tinsley earning a hard fought victory by a score of four wins to two in the 40-game match. These were the most Tinsley losses in any event since 1950.

The second World Man-Machine Championship was played in Boston in August 1994. After six games (all draws), Tinsley resigned the match and the title to CHINOOK citing health concerns. CHINOOK subsequently defended the title twice against

Grandmaster Don Lafferty. Marion Tinsley died on April 3, 1995, having been diagnosed with cancer only a week after the aborted 1994 match.

CHINOOK has not lost a game since 1994, and has defeated all comers. Its checkers rating is almost 200 points higher than that of the second-best player in the world (Ron King, the human World Champion). With nothing left to prove, the program was retired in 1997. The *Guinness Book of World Records* recognizes CHINOOK as the first computer World Champion.

The CHINOOK story is recounted in the book *One Jump Ahead* (Schaeffer, 1997).

3. Characterizing the Checkers Search Space

Chess and checkers are remarkably similar games. They both have identifiable openings, middle games and endgames. Both require subtle positional play with long-range strategies, yet require razor-sharp tactical awareness. Although checkers is "simpler" than chess in some sense (fewer piece types, 2 versus 6, and less playable squares on the board, 32 versus 64), in no way does this diminish the complexity of the game from the human's point of view. There are roughly $5 \square 10^{20}$ possible piece positions, a vast search space that is difficult to characterize and understand.[1]

The average branching factor in the checkers trees built by CHINOOK is a surprisingly low 2.8: 1.1 in capture positions and 7.8 in non-capture positions (Lu, 1993). In the search trees generated by CHINOOK, capture positions seem to out-number non-capture positions by a ratio of 3 to 1.

From the search perspective, the major difference between chess and checkers is the forced capture rule in checkers: you must play a capture move over a non-capture move. The consequence is that the number of pieces on the board can quickly reduce. Endgames are more quickly reached than in chess.

An important aspect of checkers is the notion of *zugzwang*, being forced to move when a pass would be much better. In chess, zugzwang positions rarely occur and usually only in simplified endgames. In checkers, pieces cannot move backwards (unless they have been promoted to a king). Hence you have two armies that collide; if one side runs out of safe moves, they lose. Thus, zugzwang is an important part of the game, and occurs in virtually every search undertaken by CHINOOK. This has implications for the choice of search extensions/reductions.

4. Searching for the Best Move

CHINOOK uses an alpha-beta-based search, augmented with an assortment of well-known enhancements that have been effective in chess programs.

[1] Of the total possible positions, because of the forced-capture rule in checkers, we have estimated that there are roughly 10^{18} positions that are reachable from the start of the game (Schaeffer and Lake, 1996).

4.1 Alpha-Beta

CHINOOK uses the NegaScout variant (Reinefeld, 1983; Reinefeld, 1989) of the alpha-beta search algorithm. Instead of using a full window (alpha = - , beta = +) the program uses aspiration search (Baudet, 1978; Finkel, Fishburn, and Lawless, 1980), to narrow the search to a likely range of ± 35 points from the expected value of the search (a checker is worth 100 points). The basic search routine's structure is very similar to that used in Phoenix.

A detailed analysis of the efficiency of CHINOOK's search algorithms can be found in (Plaat, 1996). CHINOOK does not use MTD(f) (Plaat *et al*., 1995; Plaat *et al*., 1996b; Plaat, 1996) only because the invention of this algorithm overlapped with the end of CHINOOK's tournament career. Experiments show that for fixed-depth searches, a further 5 percent reduction in search effort can be saved in CHINOOK when using this algorithm.

4.2 Iterative Deepening

CHINOOK uses iterative deepening (Slate and Atkin, 1977), iterating by two ply at a time. The cost of extending the search an additional single ply (with search extensions) is roughly a factor of two. Hence, iterating by two ply means that the cost of an iteration is comparable to the cost of a single-ply chess iteration. Conveniently, this allowed the time control code from Phoenix to be reused in CHINOOK virtually without change.[2]

Before starting the iterative search, the program does a three-ply alpha-beta search with a full window (alpha = - , beta = +) to get a first impression of how good/bad each move is. The result of this search might show that one move appears (at least to three ply) to be significantly better than all the alternatives (35 points). If this is the case, the iterative deepening search uses modified search windows that allow the program to know if the best move is still significantly better than all other moves. If this "obvious" best move retains its status throughout all iterations, the program will cancel the last iteration in an attempt to play the "obvious" move quickly (a time saving matter, but also an important issue in man-machine play; humans do not like to be kept waiting for obvious moves).

The regular search starts at five ply and iterates two ply at a time. The decision to iterate on odd plies was deliberate, because the program tends to be more optimistic (i.e., aggressive) when the leaf node is an odd number of ply from the root.

Most chess programs iterate one ply at a time (Bebe was a notable exception). Simple math dictates that with a small branching factor, iterating by more than one ply makes sense. Assume for simplicity that the average branching factor is 4, and that a perfect alpha-beta search reduces this average to the ideal 4 = 2. The cost of searching *d* ply will be $O(2^d)$. Iterative deepening one ply at a time will have to incur the costs of the previous iterations:

$$\text{iterative deepening overhead} = \sum 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{d-1} = 2^d\text{-}1.$$

[2] Unfortunately, this code also included a bug which cost the program the decisive game in a match with Grandmaster Don Lafferty in 1991. The bug had been present in the code for several years without appearing in any of PHOENIX's games.

Hence, the cost of iterating is equal to the cost of the last ply. In other words, roughly 50 percent of the search effort is iterative deepening overhead. If, instead, you iterate 2 ply at a time, then the overhead of iterative deepening reduces to only 25 percent of the search. At least in the case of CHINOOK, this represents a significant savings.

One CHINOOK innovation in the use of iterative deepening is the idea of restarting the search on a so-called "fail low." CHINOOK's aspiration search uses a search window that is ±35 points from the result of the previous iteration. Occasionally, the result of the search falls outside the window. When the search score exceeds the window's upper bound, good things are happening so there is no need to be concerned. When the search result fails to meet the lower bound, then a critical situation has arisen. The program's expectations for its "best" move have gone seriously awry. The program does not have a good assessment of this move, nor any idea whether it is still the best move.

The normal approach in this situation is to continue searching this move until its true score can be found, and then search the remaining moves looking for something better. However, this can be time consuming and there is a real danger that the program will be forced to move because of time constraints before properly resolving the situation.

The CHINOOK solution is to restart the search at ply five on a fail low. With a transposition table seeded with the results of the previous work, very quickly new moves will come to the fore. The previous best move has its low score from the previous search, and the other moves will have a score from a previous iteration.

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 to 1996 (ending with CHINOOK's retirement), CHINOOK played 228 games against Grandmasters and Masters. In these games, over a total of 2,249 searches (excluding book moves and forced moves), there were 22 non-trivial fail-low search results (Schaeffer, 2000):

8 A positive score for the program (advantageous position) was lower, but there was no change in the choice of move played.

7 A positive score for the program was lowered, resulting in a better move being chosen.

1 Lost position – the result of the search did not matter.

2 The score dropped to a bad negative score, but the re-search failed to change the move choice. Both games ended up being drawn,

4 The score dropped to a bad negative score (the program will have a bad or losing position if the move being searched is made). The fail-low enhancement allowed CHINOOK to find a significantly better move.

The last category is the critical one. In each case the correct move was found within *one second* (at search depth 5 of the restarted search) and its correctness verified by the rest of the search. The quick result is a consequence of the lower score of the formerly best move; the second best score now comes to the forefront. 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 search time expired before the problem could be properly resolved.

4.3 Transposition Table

CHINOOK uses a two-level transposition table (as described in (Breuker, Uiterwijk and van den Herik, 1996; Breuker 1998)), essentially the same code as used in Phoenix. Each entry contains information on two positions: the first is for the position with the greatest search depth that hashes to this location; the second holds the most recent. In effect, the first entry holds the data of greatest accuracy, while the second entry acts as a temporal cache. Experiments show that this table structure reduces the search tree size by 5 to 10 percent (a comparable result was observed in PHOENIX).

Each table entry contains the checkers position. This is represented as three 32-bit vectors (white piece locations, black piece locations, king locations). In contrast, chess programs map a chess position to a sufficiently large random number (typically 64 bits). The view is that the time and storage cost of saving an entire position is too expensive. A checkers position entry requires 16 bytes, 12 for the position and 4 for information such as the position score, search depth, bound information (upper, lower, exact).

Enhanced Transposition table Cutoffs (ETC) were pioneered in CHINOOK. Consider a cutoff node N, with move B causing the cut-off, as in Figure 1. 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 causes the cutoff with the least amount of search effort.

ETC attempts to maximize the use of information available in the transposition table (Plaat *et al*., 1996a; 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.

ETC is easily added to an alpha-beta search program. Before searching the "best" move, ETC plays each move and looks up the resulting position in the transposition table. If one of these table lookups returns a score sufficient to cause a cutoff, then further effort at this node is ended. When implementing ETC, it is important to get two things right: the search depth (the depth of the children is one less than that of the parent) and the table result (since it is one ply further in the tree, the score may have to be negated depending on the alpha-beta variant used). Because of the additional cost at an interior node, ETC should probably not be done within a ply or two of the leaf nodes.

Figure 1. The left-most cut-off move may not be the cheapest (Plaat *et al*., 1996a).

In CHINOOK, ETC results in search trees with 22 percent 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 percent reduction in search tree size for 8-ply searches (Plaat *et al*., 1996; Plaat, 1996).

As a consequence of the deep searches, it is likely that endgame analysis is hindered by the Graph-History Interaction (GHI) problem. A search tree is really a directed acyclic graph, which means that there may be two paths that lead to the same position. The sequence of moves taken to reach that position may influence the assessment of the position (as, for example, happens in chess if a three-fold repetition occurs). Hence a transposition-table entry may reflect a score that is implicitly based on the history of moves that lead to that position. Blindly retrieving a score from the transposition table may yield an incorrect assessment if the path sequence was different. Unfortunately, saving the path history of a move is expensive, both in storage and in reducing the effectiveness of the transposition table.

There is nothing in CHINOOK to address this problem, mainly because the proposed solutions in the literature are expensive to implement. A recent algorithm may yield hope for a practical solution (Breuker, 1998; Breuker *et al*., 2001).

4.4 Search Extensions

Very early on in the project, it became obvious that humans were capable of searching to phenomenal depths, much greater than could normally be reached by a computer using fixed-depth alpha-beta under tournament conditions. For example, in the tenth game of the 1990 CHINOOK-Tinsley match, on the tenth move of the game, Tinsley revealed that he could see to the end of the game and that he knew he would win. In that version of

CHINOOK, it would require a search of over 65 ply to see the forced loss (Schaeffer, 1997). It is unlikely that Tinsley actually performed a search that deep; it is more probable that his profound understanding of the game allowed him to use a relatively shallow search to reach a position that he knew was winning. Clearly, to compete with a player of Tinsley's caliber, CHINOOK needed to overcome this problem. This can be done by appropriate choices of positions for which the search can be extended (or reduced).

Although there is strong documented evidence that appropriately applied search extensions can improve the quality of the search result for the effort expended, most designers of high-performance game-playing programs are very careful about what extensions they use and the context in which they are applied. Search extensions are very difficult to evaluate. The usual method is to perform self-play simulations, pitting the program with a search extension modification against one without. Any conclusions drawn from this experiments have to be seen in the context of similar opponents. Human players may have a different playing style, leading to lines of play that are not seen in self-play games.

CHINOOK contains several search extensions that, if the results of self-play experiments are to be believed, should not be in the program. These extensions are necessary to avoid problems that seem to occur only in play against humans. For example, some types of positional sacrifices rarely occur in CHINOOK's self-play games, but are a favorite strategy of Tinsley's. Extending the search under these circumstances appears to be wasted effort, as judged by self-play, but is critical in human play (and helped CHINOOK to one of its victories over Tinsley in 1992).

Some of the search extensions used in CHINOOK are as follows:

- Since capture moves are forced, CHINOOK will not evaluate a position with a capture move pending. Hence capture moves are played out until a non-capture position is reached for evaluation.
- At a non-capture leaf node, a quick check is done to see if the opponent is threatening a capture. If so, depending on the seriousness of the threat, the search may be extended to resolve this problem.
- At a leaf node, a check for so-called "two-for-one" tactical shots is done (see Section 5.2).
- A checker that has an unobstructed path to becoming a king is called a runaway checker (analogous to a passed pawn in chess). Consider a position with a white checker on a1 and a black checker on h8. It will take a search of 14 ply (7 moves aside) for both checkers to crown. A human, on seeing this position will *start* their search at that point 14 (or more) plies in the future. Using a traditional alpha-beta search, a large tree is built before any position is reached with both checkers crowned. Hence, in this type of position most of the search effort is wasted on what the human considers to be "obvious" play. To help alleviate this problem, once a runaway checker has advanced at least 2 squares, subsequent advances cause the search to be extended one ply.

- When a checker becomes a king, this can cause a major change in the evaluation function. The first king for each side automatically causes a search extension. Subsequent crownings have their search extended only if the program estimates it will have a major impact on the evaluation of this line.
- Many of CHINOOK's search extensions are dynamic in that the conditions for enabling/disabling an extension can change throughout the search (for example, by depending on the distance of the position value from alpha or beta). Hence on one iteration, a line might benefit from an extension, but on a subsequent iteration the conditions may have changed and the extension might not be valid. This can cause problems in the search, since a line with an extension in a d-ply search, may not get the extension in a $(d+2)$-ply search, meaning that that move has only been explored one additional ply, even though the nominal search depth has been increased by two. One solution is never to undo a decision to extend the search, but this can cause a lot of unnecessary searching. The solution used in CHINOOK is to flag (in the transposition table) all positions that appear on any principal variation. An extension that occurs on a principal variation line is always extended in subsequent iterations, regardless of whether the conditions that enabled it change.
- A special type of forced move is a recapture. Here the opponent has captured a man and the program has to recapture. Any recapture moves that bring the material balance of the position close to the search window will be extended.
- The game of checkers has lots of forced moves. The singular extensions algorithm identifies forced moves and extends them an additional ply (Anantharaman, Campbell, and Hsu, 1988). An initial implementation of this algorithm turned out to be a disaster. Forced moves occurred so frequently that the search size increased dramatically. Eventually, a compromise algorithm was developed. The search would try to identify forced moves, flag them in the transposition table, and then postpone the decision to do the extension until the next iteration. By postponing the extension by an iteration, many "one-time" extensions could be filtered from the search. In effect, a forced-move extension will only be used if it can be demonstrated on consecutive iterations that it is needed.
- Unexpected "fail highs" can also cause a search extension. If we have a line of play that is "near" the principal variation where the search score has unexpectedly exceeded the search window, then this line may be the first indication of a change in score for the line. Hence, the search is extended one ply.

Many of these extensions are conditional on the search window. For example, a recapture move that still leaves the program three checkers short of reaching the search window will not be extended.

CHINOOK also does search reductions. In chess, recursive null moves have turned out to be a powerful technique for dramatically reducing search effort (Donninger, 1993). However, this idea fails to work in checkers because of the presence of zugzwang.

Material is extremely important in checkers (more so than in chess). Hence, unless you have sufficient compensation for a lost checker, the game is likely to be over quickly. When CHINOOK comes across a line that is at least one checker down with insufficient positional compensation, the remaining search depth is cut by one third. Lines where the one side is down two or more checkers without compensation are cut by one half. In effect, this is similar in philosophy to the ProbCut algorithm (Buro, 1995). Extensive experimentation has failed to turn up any non-pathological cases where this heuristic fails.

It is interesting to note that none of the search reduction algorithms in the literature appears to generalize beyond one game. Each application seems to require its own technique for reducing the search. Yet many of the techniques used for extending the search do seem to generalize.

How effective are the search extensions? This is a difficult question to answer. A self-play experiment was run where two versions of CHINOOK each had one minute to play each move in a game. The program using search extensions averaged a nominal 17-ply search; the program without extensions averaged 23 ply. As well, the program with extensions defeated the program without extensions by a score of 55-45. This is a surprising result: search extensions cost an amazing six plies of search! Despite the small score improvement (as judged by the self-play results), we believe that the extensions are mandatory. A 17-ply search is deep enough to capture most of the subtleties in a position (and more than enough to beat all but the elite players in the world). Occasionally you need a very deep search to handle deep, forcing lines of play. The search extensions are catered to these type of positions. In practice, this has allowed CHINOOK to produce principal variations that are well over 30 ply in tournament play.

5. Precomputed Databases

Most of the computing resources used to develop CHINOOK have been invested in precomputing large databases of information that can be used at runtime. The three databases are the opening book (used to aid in the selection of the opening moves of a game), the tactical database (used to help identify tactical problems at leaf nodes in a search), and the endgame database (storing perfect information about all positions with 8 or fewer pieces on the board).

5.1. Opening Database

Checkers openings are selected randomly before the start of the game. Since many common lines of play are popular, in the 1930s the “three move ballot” was adopted. This meant that the first three moves (plies) of the game were randomly selected, and then two games were played against an opponent (switching colors between games). As a result, a serious tournament player must study all 144 openings. Some of the openings are lopsided, with one side starting off with a huge advantage. Failure to know the openings analysis for the weak-side positions can quickly prove to be fatal.

Most game-playing programs select their initial moves of the game from a precalculated database of moves. In many games (such as chess and checkers), there is a

vast literature on the openings, identifying the best moves and the known traps. In chess, not knowing the openings can result in the selection of a weak move resulting in an inferior position. In checkers, not knowing the openings can lead to a quick loss. Numerous traps have been identified in the opening, often requiring searches of 30 ply or deeper to see the difficulty. It is important to know of these pitfalls, since in many cases it is difficult to spot them under the real-time constraints of a game.[3]

Our initial attempt to build an opening book consisted of developing a version of CHINOOK that learned through self-analysis which moves were good and bad. Copies of the program would use idle machine cycles to expand dynamically the analysis of key opening lines. After several months, the effort was discontinued. The learning rate was too slow and not focused enough to discover the important features of the openings.

In frustration, we resorted to the tried-and-true method – entering lines from opening books into a database. Over a period of several months, a hand-generated book of over 5,000 positions was created. Each position was verified with a 15-ply (plus extensions) search. The program's evaluation of the position was compared with the human's assessment and if there was a major discrepancy, that position would be reported for further examination. A handful of errors were found in the literature.

Unfortunately, our opening book proved insufficient. As the 1992 CHINOOK-Tinsley match showed, it was relatively easy for Tinsley to get the program out of its book onto its own resources. This strategy resulted in a key win for Tinsley. The match also revealed that 15-ply searches were not enough. CHINOOK played a verified book move, and ended up getting into a difficult position that was eventually lost. Further analysis showed that at least a 19-ply search was needed to uncover the problem. The conclusion was that we needed greater coverage of the openings, and each position had to be verified to a greater depth.

You are not going to beat Marion Tinsley by playing well-known lines of play. Your only chance is to disable his encyclopedic memory. The solution is to prepare so-called *cooks*, prepared analysis that deviates into uncharted waters.[4] Once you get a player using their analytic resources, it increases the chance that they will make an error. Against Tinsley, this is the only hope to play for a win.

In 1994, the three book problems – coverage, verification, and cooks – were addressed. We acquired a copy of Martin Bryant's opening book for his commercial program COLOSSUS. Martin is a strong player, and had been collecting opening analysis for years. This represented a >5-fold increase in our collection of positions. Each position

[3] For example, in the 1990 U.S. Championship, CHINOOK lost a game to Karl Albrecht after falling in to the well-known Dunne's Win. In this position, the obvious move leads to a loss, which is difficult for a computer to search deep enough to find given the time constraints. After losing the game, the opening book was repaired to avoid Dunne's Win and variations on the same idea. Four months later, in the 1990 CHINOOK-Tinsley match, Tinsley deliberately played for a position where there was a chance for CHINOOK to fall into a variation of Dunne's Win. He was sorely disappointed when the program immediately played the right move and avoided the trap (Schaeffer, 1997).

[4] Note that "cook" has a different meaning in chess. It usually refers to a composed problem that has a flawed solution.

was analyzed to 21 ply looking for errors (although as we began to run out of time before the 1994 match, some searches were reduced to 19 ply).

The verification process was instrumented to look for cooks. If in a position P, a move M1 was suggested by the literature, the search would do a wide-window search of M1 (to get its exact value). All other moves would be searched with a narrow window to see if they returned a score that was comparable to or better than M1. If so, then they were re-searched with a wider window to get their exact score. If a move M2 has a score that was "significantly" different from M1, then this position was flagged for additional processing. In addition, moves that appeared comparable in value to those on the main lines of play were also flagged for additional processing.

The verification process resulted in hundreds of corrections to the published literature. Amazingly, the analysis called into question some of the popular main lines of play. The new moves in these lines are now hidden in CHINOOK's opening database, waiting for some unsuspecting opponent to stumble into these lines.

When CHINOOK started out in 1989, it had few book moves and as a result played many outlandish opening moves (many of which turned out to be good). Unfortunately, there were enough errors in our opening play that we had to enlarge the opening book and force the program to play standard book lines. Since one loss against a Tinsley could cost you a match, by 1994 we had built a huge verified book. This forced the program to play tried-and-true human moves, but stifled its creativity.

In 1996, with no Tinsley's looming on the horizon, we turned off using the opening book when playing the strong side of an opening. The results were immediate; CHINOOK played more interesting checkers and won more games. Our only regret is that we did not do this years earlier.

To succeed, CHINOOK required an enormous commitment to opening preparation. In the past, few chess programmers devoted much attention to the opening, short of incorporating lots of published human analysis and previously played games in their databases. Today, most commercial programs make a considerable effort in opening book preparation, usually by hiring a strong chess player to work on the book. To maximize one's chances against the best chess players, opening preparation is essential to success. The DEEP BLUE team did extensive work in this area, although it did not appear to reap benefits in their Kasparov matches.

5.2 Tactical Tables

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 cost of the search. 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. Basically, if the material balance of a leaf position differed from the result obtained from a shallow (3-ply) search, then our program 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 percent of the positions flagged (two additional patterns accounted for another 25 percent of the combinations).

Consider the checkers position in Figure 2. 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 is not enough. Once White wins the piece, we must also make sure that Black cannot immediately recapture it. For example, there cannot be a Black piece on h6 ("□" meaning "not" in the figure); otherwise, Black could regain the lost piece with h6xf4. Similarly, Black should not have a recapture with f6xh4. A number of conditions must hold for this 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 square h4 being empty, square f6 being occupied by a Black piece, and square e7 being occupied.

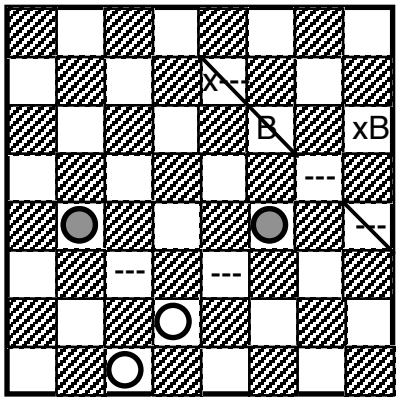

Figure 2. The conditions needed for a two-for-one to work (Schaeffer, 1997).

Translating this information into code was difficult, and it was quickly 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*.[5]

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 tactics 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 these two-for-ones extended the search, 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.

5.3 Endgame Database

Very early on in the CHINOOK project, it was realized that, unlike in chess, endgame databases would be a powerful search enhancement. In chess, endgames with 5 or fewer pieces have been constructed, and a few 6-piece endgames (they are prohibitively large to store). In practice, these databases rarely influence the result of a game since most games are decided well before a simplified endgame is reached. Checkers, however, has the forced capture rule. Consequently, many of the pieces can quickly come off the board, resulting in endgames in major lines of play within the first 20 moves (40 ply) of a game.

Over the span of five years, we undertook an extensive effort to compute all positions with eight or fewer pieces on the board: 444 billion (4.44×10^{11}) positions (Lake, Lu, and Schaeffer, 1994). This was an exhausting effort, requiring the coordination of over 200 computers from around the world. For the 1992 Tinsley match, CHINOOK had access to all the seven-piece databases and a small subset of the eight-piece positions (roughly 42 billion positions). For the 1994 match, the most important part of the eight-piece database, the four pieces versus four pieces subset, was completed (111 billion positions).

The databases have been compressed into roughly six gigabytes of data, designed for easy runtime decompression (Lake *et al.*, 1994). Since the databases are accessed

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

frequently in the search, a surprising consequence is that CHINOOK is I/O bound, as compared to most other game-playing programs which are compute-bound.

The databases improve the program in two significant ways. First, whenever a database position is encountered, CHINOOK can use the exact value for the position (win, loss, draw) from the database (no error), instead of using a heuristic evaluation function (which has error). Second, further search beyond a database node in the tree is unnecessary, allowing large portions of the search tree to be eliminated. The combined effect is a smaller search tree that produces a more accurate result.

An example of the power of deep search and databases is game 37 of the 1992 Tinsley match. On the fifth move of the game, CHINOOK was able to search deep enough to back up a draw score to the root of the search! This kind of prowess has been seen many times subsequently, prompting speculation on how much work would be required to "solve" the game of checkers (Schaeffer and Lake, 1996).

As of the time of this writing (September 2001), a portion of the 9-piece databases have been computed. The five-pieces versus four-pieces subset of the database is roughly five times larger than the complete 8-piece database (roughly $2 \square 10^{12}$ positions). This computation will take a year to complete. With current technology, it is possible to compute the five-piece versus five-piece subset of the 10-piece database (roughly $8.5 \square 10^{12}$ positions).

A small amount of work has been done building a program to solve checkers. This work began in 2001 with the hope of putting it into production use in 2002. In the interim, every endgame database that is computed simplifies the task of building the proof tree.

6. Evaluation Function

Building an evaluation function for a game that was not understood by the programming team was a challenge. In chess, a reasonable evaluation function can be built by studying the numerous articles and theses that have been written on constructing a chess program. They provide a model that can be incrementally improved as deficiencies in the program's play are identified. Unfortunately, there is a paucity of literature on constructing a checkers evaluation function.

Our original source for information on the design of a checkers evaluation function was that used in Samuel's program (Samuel, 1959; 1967). It is interesting to note that many of the evaluation function terms were designed to recognize tactical patterns that could not be discovered given the shallow search depths of his program. With a program searching over 20 ply, like CHINOOK, these terms were not needed. Hence, the Samuel model was of limited benefit for CHINOOK.

The initial foray at a checkers evaluation function was based on chess experience. Many of the terms were borrowed from chess, and used as if this was a chess program. For example, the program valued controlling space, something that is a popular heuristic in chess. In checkers, space is important, but in the opening of the game a space advantage

is often a disadvantage! Too much space means that your pieces may be too far advanced and vulnerable to attack. So much for a chess-players evaluation function.

With the help of Normal Treloar, a checkers expert, we spent three years designing an evaluation function. The result was a linear combination of roughly 25 major evaluation terms, each with different weights for the phases of the game (opening, early middle game, late middle game, endgame). By 1991, we had a strong checkers-playing program, but it was unclear whether the ability was due to deep searches alone, or a combination of search and knowledge.

One interesting evaluation function data point occurred in early 1992. Dissatisfied with CHINOOK's positional play, Norman proposed a more elaborate mobility calculation; one that would be considerably more expensive to compute. Having been reared on the Thompson correlation between program speed and performance, we were initially reluctant to adopt the changes. Once it was implemented, we discovered that the new evaluation function cost the program roughly a factor of two in speed (one ply). In the chess world, this once would have been considered too big a sacrifice (and still may be for some programmers). Nevertheless, the improvements in evaluation quality more than offset the loss in speed. These changes were critical to CHINOOK's future successes.

From 1992 to 1994, the evaluation function structure remained largely unchanged. The lessons learned in building a chess program were invaluable here, especially as it concerned the design for testing and the handling of exceptional conditions. This is an important consideration that can save much grief. Sadly, the lessons learned from the PHOENIX experience were not fully appreciated in the early days of the CHINOOK project, and this was responsible for a number of painful losses.

A serious problem that had to be overcome in the program is the mixing of different evaluation scores in the search. As a result of the forced capture rule in checkers, games can make the transition from the opening to the endgame rather quickly. A single search near the start of the game can mix evaluations from the opening, middle game, endgame and database positions. This poses a problem, since it requires each game phase to have its evaluation function results calibrated with the others. In other words, if one phase's evaluations are more optimistic or pessimistic than the others, it will significantly skew the program's play. To compensate for this effect the evaluation function score was scaled by the number of pieces on the board. The reasoning was that the more pieces on the board, the more complicated the game was and the higher the probability that deep search would be to the program's benefit. This was a major enhancement to the program.

The evaluation function was tuned by hand over a period of four years. Attempts at using tools to automate this process failed (Schaeffer, 1997). Subsequently, experiments with temporal difference learning have shown convincing results, indicating that it would be possible for CHINOOK to learn a strong set of feature weights solely from self-play games (Schaeffer, Hlynka and Jussila, 2001).

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. 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. 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).

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 (1) using expert-specified heuristics (too hard to obtain and code), (2) using the evaluation function to score the drawn position (easy to implement but too simple an assessment of future chances), (3) trying to correlate properties of the search tree with difficulty (for example, selecting the move that maximizes the number of forced opponent moves), and (4) identifying forced moves (with respect to the database) that were not identified as such by the evaluation function (Levy, 1991). Although all these ideas hold 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.

One has to be careful about the interpretation of positive draw scores. Consider a provable draw that returns a draw 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 we lower the positive scores 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.

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 eight-piece databases and draw-differentiation code could play a match against CHINOOK with the four-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, 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.

7. Assessing Search and Knowledge

For tournament play in 1996, CHINOOK's hardware consisted of a Silicon Graphics 4D/480 computer (8 processors) with 380 megabytes of RAM (used mainly for caching endgame database results). The maximum search depth reached in a game was 51 ply. The program averaged completing a nominal search depth between 19 and 21 ply of iterative deepening search. The search extensions meant that the median position for which an evaluation was done was 25 ply into the search.

With access to a 64-processor SGI Origin 3000 computer with 32 gigabytes of RAM, CHINOOK can search to a nominal iterative-deepening search depth of 23 to 25 ply (under tournament conditions). This is achieved using the APHID library for parallel alpha-beta search, that has been used to parallelize chess, checkers and Othello programs (Brockington and Schaeffer, 2000). We have no experience playing tournament games on this hardware.

Figure 3 shows the value of an additional iteration (two ply) of search. The figure shows the results of a 20-game match between CHINOOK searching to depth *d* versus a version searching to depth *d-2* (Junghanns and Schaeffer, 1997) – no opening book, six-piece databases, no search extensions. As can be seen, as the search depth increases, the advantage of additional search effort decreases. Adding the missing enhancements will only further level the match results. Although the program can always benefit in some small way from additional search, it appears as if diminishing returns are taking over. Hence, there is little incentive to work on further enhancing the program's speed.

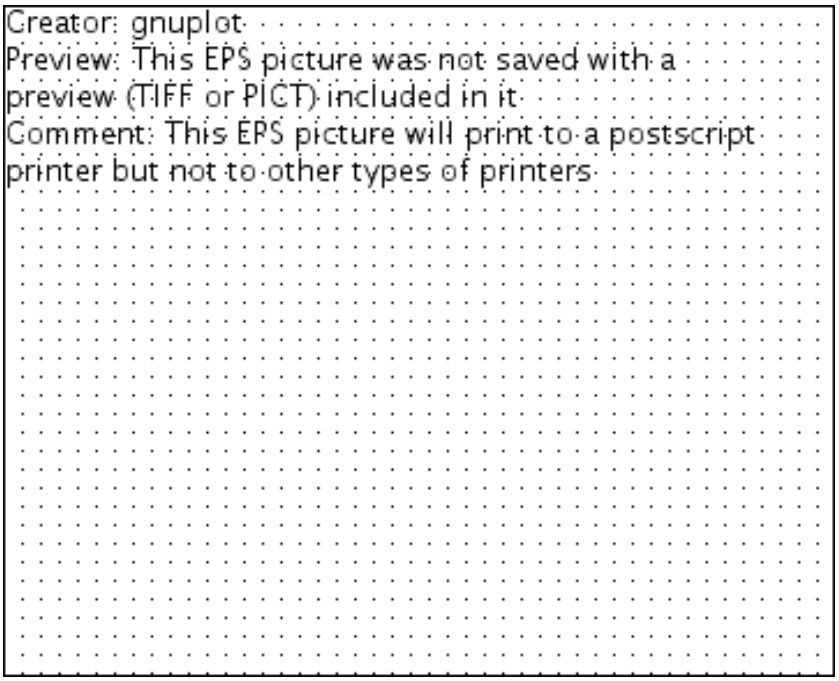

Figure 3. Self-Play Experiments (Junghanns and Schaeffer, 1997).

Figure 4 assesses the quality of the evaluation function. The graph plots the percentage of searches where a one-ply deeper search causes a move change as a function of search depth.[6] While deeper searching continues to result in move changes (even as often as 14 percent of the time at 17 ply), this is misleading. The other lines in the graph show how often a move change results in a significant change in the position's assessment, measured in terms of the value of a checker (100 points). As can be seen, after seven ply is reached, the program rarely changes its move choice and assessment of the position significantly. This is compelling evidence to suggest that the program has a very strong evaluation function.

[6] Usually CHINOOK increments its search depth by two ply at a time. For compatibility with the other games studied in Junghanns and Schaeffer (1997), this graph shows the results of a single-ply increment.

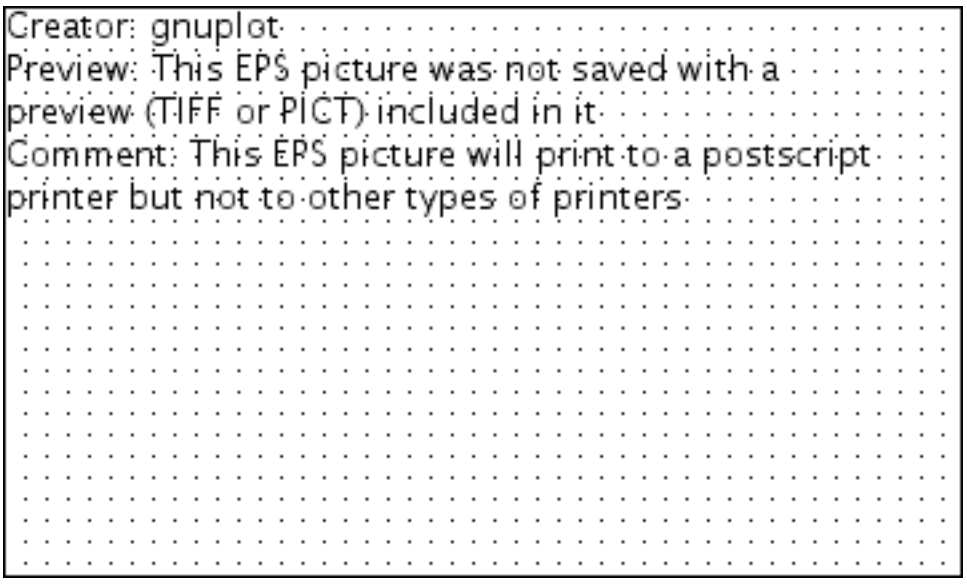

Figure 4. Move Changes From *d* to (*d+1*) Ply (Junghanns and Schaeffer, 1997).

If you think about what CHINOOK can do (search over 20 ply deep, strong evaluation function, large opening book, access all the eight-piece endgame positions), maybe it is not a surprise that a computer is finally the World Champion in checkers. However, the more interesting observation is why did it take so long. The CHINOOK technology would appear to be so overwhelming, that the real surprise is how long the humans were able to withstand the technological onslaught. This is a testament to Tinsley's amazing abilities.

8. Conclusions

Since chess and checkers are similar in many ways, it is interesting to ask the question: how much of the experience in programming chess can be applied to a different game, checkers? After all, this is a test of the generality of the work done in chess.

In summary, experience with chess provided a suite of techniques that one could consider for checkers. Some of the techniques were obviously applicable, some required extensive evaluation before deciding to accept/reject them, while others were clearly inapplicable. Making these decisions was the easy part of designing the search. The hard part, as always, was the application-dependent tuning that goes on to find the "right" combination of features. There is no claim that what we did in CHINOOK is the "best" – indeed someone could improve upon our work – but it met our perceived needs.

When tackling a new search domain, whether single-agent or two-player, there are a plethora of search enhancements available in the literature. The program developer should characterize the search space to be searched, identify its key properties, and choose a selection of enhancements that offer the most promise for benefiting the search algorithm. Ultimately, the choice has to be empirically demonstrated. One thing that is usually unsaid in a project such as this is the countless number of hours that goes into finding the

combination of features that seems to perform best. This is a thankless task which consumes enormous intellectual and computational resources. Literally hundreds of thousands of hours of computer time went into developing CHINOOK, not to mention the thousands of man-hours.

Although the basic ideas can be transferred, there are still many application-dependent enhancements that can have a significant impact on the search effort. That so much application-dependent tuning is required to achieve high performance does not bode well for the expectations from generic black-box search engines (e.g. Multigame (Romein, 2000)). In part, this work helps identify the components where application-dependent knowledge can have a significant impact on performance.

Acknowledgments

Some of this work has appeared in Schaeffer, 2000. The author is grateful to the anonymous referees for their insightful feedback.

This research has been supported by the Natural Sciences and Engineering Research Council of Canada (NSREC) and Alberta's Informatics Circle of Research Excellence (iCORE). I want to thank Hermann Kaindl for encouraging me to write this article. I wrote it in 1999, but it took two years before I could decide what to do with it!

This work would have not been possible without the commitment from the CHINOOK team, including Martin Bryant, Joe Culberson, Brent Knight, Robert Lake, Paul Lu, Duane Szafron and Norman Treloar.

References

More information about CHINOOK, as well as many of the CHINOOK-related references, can be found at http://www.cs.ualberta.ca/~chinook.

Anantharaman, T., Campbell, M., and Hsu, F-h. (1988). Singular Extensions: Adding Selectivity to Brute-Force Searching. *American Association for Artificial Intelligence (AAAI) Spring Symposium Proceedings*, pp. 8-13. Also published in the *ICCA Journal* (1988) Vol. 11, No. 4, pp. 135-143. Also published in *Artificial Intelligence* (1990) Vol. 43, No. 1, pp. 99-110.

Baudet, G. (1978). *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University.

Baxter, J., Trigell, A., and Weaver, L. (1998). Experiments in Parameter Learning Using Temporal Differences. *ICCA Journal*, Vol. 21, No. 2, pp. 84-99.

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1996). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 19, No. 3, pp. 175-180.

Breuker, D.M. (1998). *Memory versus Search in Games*. Ph.D. Thesis, Department of Computer Science, Universiteit Maastricht, The Netherlands.

Breuker, D.M., Herik, ,H.J. van den, Uiterwijk, J.W.H.M., and Allis, V. (2001). A Solution to the GHI Problem for Best-First Search. *Theoretical Computer Science*, Vol. 252, No. 1-2, pp. 121-149.

Brockington, M. and Schaeffer, J. (2000). APHID: Asynchronous Parallel Game-tree Search. *Journal of Parallel and Distributed Computing*, Vol. 60, pp. 247-273.

Buro, M. (1995). ProbCut: An Effective Selective Extension of the Alpha-beta Algorithm. *ICCA Journal*, Vol. 18, No. 2, pp. 71-76.

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

Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137-143.

Donskoy, M. and Schaeffer, J. (1990). Perspectives on Falling from Grace. *Chess, Computers, and Cognition* (eds. T.A. Marsland and J Schaeffer), pp. 259-268. Springer-Verlag. Also appeared in the *ICCA Journal* (1989), Vol. 12, No. 3, pp. 155-163.

Feldmann, R. (1993). *Spielbaumsuche auf Massiv Parallelen Systemen*. Ph.D. Thesis, University of Paderborn, Germany. English translation *Game Tree Search on Massively Parallel Systems* is available at ftp.uni-paderborn.de/doctechreports/Informatik/misc/phdFeldmann.ps.Z.

Finkel, R., Fishburn, J., and Lawless, S. (1980). Parallel Alpha-Beta Search on Arachne. *Proceedings International Conference on Parallel Processing*, pp. 235-243.

Greenblatt, R., Eastlake, D.E., and Crocker, S.D. (1967). The Greenblatt Chess Program. *Proceedings Fall Joint Computer Conference*, pp. 801-810.

Hsu, F-h. (1999). IBM's Deep Blue Chess Grandmaster Chips. *IEEE Micro*, March-April, pp. 70-81.

Junghanns, A. and Schaeffer, J. (1997). Search Versus Knowledge in Game-Playing Programs Revisited. *Proceedings International Joint Conference on Artificial Intelligence*, pp. 692-697.

Junghanns, A. and Schaeffer, J. (2001). Enhancing Single-Agent Search Using Domain Knowledge. *Artificial Intelligence*, Vol. 129, No. 1-2, pp. 219-251.

Korf, R. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, Vol. 27, No. 1, pp. 97-109.

Knuth, D. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, pp. 293-326.

Lake, R., Lu, P., and Schaeffer, J. (1994). Solving Large Retrograde Analysis Problems Using a Network of Workstations *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg and J.W.H.M. Uiterwijk), pp. 135-162. University of Limburg, Maastricht, The Netherlands.

Levy, D. and Beal, D. (eds.) (1989). *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, Ellis-Horwood Limited, England.

Levy, D. (1991). First Among Equals. *ICCA Journal*, Vol. 14, No. 3, p. 142.

Lu, P. (1993). *Parallel Search of Narrow Game Trees*. M.Sc. Thesis, Department of Computing Science, University of Alberta.

Plaat, A., Schaeffer, J. Pijls, W., and Bruin, A. de (1995). Best-First Fixed-Depth Game-Tree Search in Practice. *Proceedings International Joint Conference on Artificial Intelligence* (*IJCAI*), pp. 273-279.

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

Plaat, A., Schaeffer, J., Bruin, A. de, and Pijls, W. (1996b). A Minimax Algorithm Better than SSS*. *Artificial Intelligence*, Vol. 87, No. 1-2, pp. 255-293.

Plaat, A. (1996). *Research Re: Search and Re-Search*. Ph.D. Thesis, Erasmus University, Rotterdam, The Netherlands.

Reinefeld, A. (1983). An Improvement of the Scout Tree Search Algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4-14.

Reinefeld, A. (1989). *Speilbaum-Suchverfahren*, Springer-Verlag, Informatik-Fachberichte 200.

Romein, J. (2000). *Multigame – An Environment for Distributed Game-Tree Search*. Ph.D. Thesis, Computer Science, Vrije Universiteit Amsterdam, The Netherlands.

Samuel, A. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development,* Vol. 3, pp. 210–229.

Samuel, A. (1967). Some Studies in Machine Learning Using the Game of Checkers – Recent Progress. *IBM Journal of Research and Development,* Vol. 11, pp. 601–617.

Schaeffer, J. (1986). *Experiments in Search and Knowledge*. Ph.D. Thesis, Department of Computer Science, University of Waterloo. Also available as technical report TR86-12, Department of Computing Science, University of Alberta.

Schaeffer, J. and Lake, R. (1996). Solving the Game of Checkers. *Games of No Chance* (ed. , R. J. Nowakowski), Cambridge University Press, pp. 119-133.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*, Springer-Verlag, New York.

Schaeffer, J. (2000). Search Ideas in Chinook. *Games in AI Research* (eds. H.J. van den Herik and H. Iida), pp. 19-30, Universiteit of Maastricht, The Netherlands and University of Shizuoka, Japan.

Schaeffer, J., Hlynka M., and Jussila, V. (2001). Temporal Difference Learning Applied to a High-Performance Game-Playing Program. *Proceedings International Joint Conference on Artificial Intelligence*, pp. 529-534.

Slate, D. and Atkin, L. (1977). Chess 4.5 – The Northwestern University Chess Program. *Chess Skill in Man and Machine* (eds. P. Frey), pp. 82-118. Springer-Verlag, New York.

Thompson, K. (1982). Computer Chess Strength. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 55-56. Pergamon Press, London, UK.