REVIVING THE GAME OF CHECKERS

Jonathan Schaeffer
Joseph Culberson
Norman Treloar
Brent Knight
Paul Lu
Duane Szafron

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

ABSTRACT

Chinook is the strongest 8×8 checkers program around today. Its strength is largely a result of brute-force methods. The program is capable of searching to depths that make it a feared tactician. As with chess, knowledge is the Achilles' heel of the program. However, unlike the chess example, endgame databases go a long way to overcoming this limitation. The program has precomputed databases that classify all positions with 6 or less pieces on the board as won, lost or drawn (with 7 pieces under construction). The program came second to the human World Champion in the U.S. National Open, winning the right to play a World Championship match against him. *Chinook* is the first computer program in history to earn the right to play for a human World Championship.

1. Introduction

The game of checkers was reputedly "solved" over 25 years ago. Samuel's famous checkers program [1, 2] was credited with defeating a master and, largely as a result of this one game, efforts stopped on developing a program to play world-class checkers. Unfortunately, this single moment of human oversight in one game was not representative of the relative strengths of the best humans and the best checkers programs. With only a few exceptions (the Duke program being notable [3]), little effort has been devoted to computer checkers since Samuel's pioneering effort.

Why the sudden interest in checkers? There are several reasons, including:

(1) Checkers has a smaller search space than chess, approximately 5×10^{20} positions versus $O(10^{44})$ for chess. The search space is small enough that one could consider solving the game.

(2) The rules of the game are simple, reducing the intrinsic complexity of the core program. In addition, knowledge is easier to represent and manipulate. For example, patterns in checkers can be efficiently represented using 32-bit words.

(3) The knowledge required to play checkers well appears to be considerably less than for chess. This is not meant as a slight to the game. Chess has many more piece types than checkers, greatly multiplying the number of piece interactions, all of which must be learned.

(4) Chess, as a *drosophila* for artificial intelligence, has not been readily amenable to work with learning, knowledge representation, etc.; the domain is too complicated. Checkers provides an easier domain to work with, and provides the same basic research opportunities as does chess.

It is unfortunate that a historical accident deprived checkers of attention from the artificial intelligence research community.

Chinook is a checkers program, developed at the University of Alberta, as the by-product of a research effort into game-playing strategies. The project has two goals:

(1) The short-range objective is to develop a program capable of defeating the human World Champion in a match.

(2) The long-term goal is to solve the game of checkers. In other words, it may be computationally possible to determine the game-theoretic value of checkers. As discussed later, this is an enormous undertaking.

Chinook is the strongest checkers program around today, and is regarded as being of strong master or grandmaster strength. The program won the checkers event at the 1st Computer Olympiad [4] and has scored excellent results against some of the strongest players in the world. In August 1990, the program won the Mississippi State Championship. It followed that up by coming second to the World Champion in the U.S. National Open, winning the right to play a match for the World Championship. It is hoped this match will take place sometime in 1991. *Chinook* is the first computer program to earn the right to play for a human World Championship title.

Chinook has taken the notion of brute-force to an extreme. The program combines deep searches with a database that contains all the positions with 6 or fewer pieces on the board (and some 7-piece results) and classifies them as win, loss or draw. During the game, this perfect information is available to improve the accuracy and effective depth of the search. It is not uncommon for the program to backup a database score to the root of the tree *before move 10* of a game!

This paper gives a brief description of the *Chinook* program, covering its search methods, knowledge, endgame databases, and computer-generated opening book.

2. The Lessons of Computer Chess

Chinook is a typical alpha-beta ($\alpha\beta$) search program, with iterative deepening (2 ply at a time), transposition tables [5] and the history heuristic [6]. Many of the search ideas of computer chess carry over naturally to checkers. However, there are some differences:

(1) Transposition tables are not quite as effective as in chess. In checkers, a man cannot move backwards (only kings can), reducing the likelihood of transpositions. When combined with iterative deepening, the primary benefit of the table is for

move ordering.

(2) Moves are ordered at interior nodes using the transposition table move (if any) and the history heuristic. No application-dependent knowledge is used. The history heuristic appears to be more effective than in chess programs.

(3) Experiments were done with the singular extensions algorithm [7]. Unfortunately, the results were not encouraging. Checkers is not as tactical as chess, and the proportion of forcing moves appears to be less than in chess. The search overhead in finding singular moves is significant and does not appear to be compensated for by the benefits.

(4) The loss of a man in checkers is much more serious than the loss of a pawn in chess. Big reductions in the search tree are possible by taking advantage of this. Lines that appear to lose a man with no chance of retrieving it and no significant positional compensation can be quickly truncated.

(5) Captures in checkers are always forcing, analogous to moving out of check in chess. They can also be treated similarly - every capture could be extended an additional ply of search. Extensive experimentation has led to the conclusion that a more restricted extension heuristic is superior, otherwise too many lines that lose a checker get extended unnecessarily deep. A compromise is to extend capture moves that restore the material balance to that of the root of the tree, or cause the balance to swing from one side to the other.

(6) *Chinook* has no quiescence search. The search continues until a stable position is reached. As a result, it is not uncommon for a search with a 15-ply nominal depth to consider lines 35-ply deep.

In the middlegame, on average an extra 2 ply of search costs a factor of 4 in computation time. In contrast, for chess, a single ply often costs 4- to 8-fold [8].

3. Knowledge

For the game of chess, many of the important pieces of knowledge required to play the game well are documented. Several studies have been done that quantify the value of the knowledge (for example, [9]). In checkers, the issue of knowledge has been less well studied. Samuel gave a description of the knowledge used in his program [1, 2]. As well, Oldbury has made an attempt to define and express mathematically the knowledge of checkers [10]. Unfortunately, these two efforts are only a beginning and it is difficult to know what knowledge is important and just how important it is.

Checkers, like chess, is usually considered to have 3 game stages: the opening, middlegame and endgame. Tactics (winning or threatening to win material) and strategy (positional play) are important in all stages. As seen in other games, tactics are easily uncovered by deep brute-force search. Strategy is of prime importance, especially in the early stages; the result of a game is often determined in the first dozen moves. However, adding effective strategic knowledge to the program is a difficult task.

The checkers knowledge used in the program is a collection of heuristics that were devised from experience in playing with *Chinook* and from general experience in the

programming of other games. The relative importance of each heuristic was adjusted manually, based on data obtained from the checkers literature and from experience playing against the program.

The three stages of the game are treated somewhat differently by the program, and illustrate different aspects of knowledge:

(1) Book Knowledge:Many games programs have extensive opening libraries, and *Chinook* has an option to be guided by such a library of master-play. However, the program (with heuristics appropriately weighted) seems to be able to play openings rather well without an opening book, often playing novel (and apparently sound) moves. Consequently, one of the present options is a *minimal opening book*, containing only moves that appear to be compulsory and which *Chinook* is unable to find under the time constraints of tournament play. This option leaves *Chinook* free to innovate, and the early results have been encouraging.

(2) Heuristics:The majority of the game, usually including the decisive moves, is dependent on the quality of the checkers knowledge in the program. As in other game-playing programs, this is the Achilles' heel of the program. The heuristics used in *Chinook* are briefly summarized in the Appendix.

(3) Perfect Knowledge:The endgame database contains all possible positions with 6 and fewer pieces and some with 7 pieces on the board, with the computed result of best play from each position (win, loss or draw). The positional evaluation is perfect. However, the program may not play optimally once a winning position is reached, nor offer maximum resistance once a lost position is reached. Due to the sheer size of the database, it is not feasible to store the best move for each position. In such cases, heuristic knowledge is used to guide the program towards the completion of the game. Usually the search by itself is adequate; a 25-ply search (not unusual in such positions) is often deep enough find the optimal continuation.

The most difficult stage of the game to tune the heuristics has been the opening and early middlegame, because accurate play here requires subtle positional judgment. In several difficult openings, the consequences of an opening move are revealed only after 30 or 40 ply. It often seemed, in attempting to instill positional judgment in the program, that it was impossible to re-create this degree of subtlety; the heuristics were too crude and simplistic. Nevertheless, we believe that we have succeeded in reproducing reasonably strong opening play using only heuristics.

The middle stages of the game were somewhat easier to cope with, probably because *Chinook*'s search usually brings to light the tactical requirements which start to become more important. In endgame positions not contained in the database, it is sometimes difficult for the program to play as well as a human. Strategic objectives which a human player can see are often well over the brute-force horizon. For example, a human might reason "If I move my man down the board to crown, and then bring it back, I can win this opposing piece which is exposed in the middle of the board". In checkers, such a maneuver may take more than 20 ply. The human understands this piece of knowledge and is capable of *beginning* his search at that point 20 ply in the future; the program cannot.

In combination with *Chinook*'s deep search, the endgame database is an enormous help in playing these later stages of the game. Although the program may not understand how to play the position properly, the database eliminates huge portions of the search tree, allowing greater search depths. In some positions, the program correctly solves problems well beyond its search depth, not because it can see the solution, but because the databases are capable of refuting all the alternatives.

Given that the quality of the evaluation function relies entirely on manual adjustments, the question naturally arises whether there is some way to automate this process. Our evaluation of a position is the weighted sum of 20 heuristic routines, h_i:

$$value = \sum_{i=1}^{80} h_i \times w_i \ .$$

The program breaks the game into 4 phases (see Appendix), each with 20 adjustable weights, for a total of 80 parameters that must be tuned. Given the fixed set of knowledge in the program, the problem becomes one of fine tuning the weights w_i associated with each heuristic to maximize the quality of the program's play on average.

Every time *Chinook*'s knowledge is modified or added to, the weights must be retuned. Eventually, the number of weights that must be tuned becomes too large for manual tuning. With 80 adjustable parameters, this limit has already been surpassed. Therefore, we are experimenting with ways of automating the tuning of *Chinook*'s evaluation function weights.

Tuning requires a quantitative measure for assessing changes in *Chinook*'s performance. One such measure is to see if *Chinook* will play the same move as a human grandmaster in a given position. A database of positions and moves played by grandmasters was created (not to be confused with the endgame databases). *Chinook* was tested to see how often it matched the grandmaster's move based on a shallow search. The goal of the automated tuning is to calculate a new set of weights so that *Chinook* can maximize the number of positions where it makes the same move as the grandmaster.

The initial attempt at tuning involved modifying the program developed by Andreas Nowatzyk for the *Deep Thought* chess program [11]. It computes the weights using linear regression and least-squares-fit. Some of the other proposed methods include using linear discriminants [12], or minimizing a cost function that measures the inaccuracies of the current set of weights [13]. Each of these techniques has its strengths and weaknesses, and it is not known which method suits our needs best.

The current implementation of automated tuning has not been wholly successful. Although *Chinook*'s matching rate against the grandmaster moves increases at the shallow search depths used for tuning, the new sets of weights do not result in better play at the depths of search that occur in real games. Nonetheless, it has been useful in identifying trends. For example, some heuristics consistently get high weights, coinciding with our assessment of the value of the heuristic. Still, there are some potential problems that may prevent automated tuning from ever becoming an unqualified success.

First, the proposed mathematical techniques for tuning assume that each of the evaluation terms are measuring independent features of the checkers position. Ideally, given two positions where the only difference is that the measure of centrality is larger in

one than in the other, all of the other evaluation terms remain the same. However, in practice, increasing the centralization of a checker reduces the mobility of the opponent's checkers towards the center of the board, which increases your own mobility. Therefore, there may be non-linear dependencies between terms, and the conceptualization of the evaluation function as a linear equation is mathematically false. If the evaluation terms in *Chinook* are not independent, the system of equations must be enlarged to include non-linear terms. The problem now becomes one of determining which non-linear terms to include.

Secondly, there is concern that tuning *Chinook*'s evaluation weights so that it tends to play more like a human grandmaster is actually counter-productive. It has been noted many times, in games against strong opponents, that *Chinook* occasionally plays a move that a human would never play, but is as good or better than the human move. Human players, as a group, have certain strengths and preconceptions that cause them to play towards positions where those strengths can be used. Likewise, computers have their own strengths, in particular deep brute-force search. Therefore, tuning *Chinook* to make human-like moves may be playing away from the machine's unique strength. However, the problems caused by the philosophical distinctions between man versus machine play is even more difficult to quantify than the problem of dependent evaluation terms. When the tuning process begins to impede *Chinook* from making innovative moves, a compromise on this point will have to be reached.

If the tuning process can be automated, the designers of *Chinook*'s knowledge can have more freedom to experiment without having to manually tune the evaluation weights. Also, automated tuning may be an important diagnostic tool for detecting serious deficiencies in the knowledge. If *Chinook* consistently plays a known losing move in a set of similar positions, despite tuning, then it is likely that the program is lacking the knowledge to play that position well. For now, this remains an open research problem.

4. Databases

This section provides a brief overview of the methods used to compute the 6-piece databases. Thompson provides an excellent discussion of how to construct chess databases [14].

N-piece endgame database construction involves enumerating all positions with *n* pieces or less on the board and computing whether each is a win, loss or draw. The basic idea is quite simple; what makes the problem difficult is the sheer size of the problem as *n* increases. Table 1 enumerates the number of positions in the databases that have been solved and are accessible to *Chinook*. Roughly 15 billion positions in the 6- and 7-piece databases have been computed. The 4 against 3 databases are currently under construction, and consist of an additional 20 billion positions, as shown in parenthesis in Table 1. Although it appears that 6- and 7-piece endgames are a long way off from the starting 24-piece position, in fact they are not since capture moves are forced. For example, in a 2 minute search of the starting game position, already positions in the database appear in the search tree. Thus, although the 6- and 7-piece databases represent a small part of the overall search space, as Figure 1 shows they represent a significant portion of the solution path of a line of analysis.

White to play	Black to play (pieces) 1	2	3	4	5	6
1	3,488	98,016	1,773,192	23,204,660	233,999,928	1,891,451,952
2	98,016	2,662,932	46,520,744	587,139,846	5,702,475,480	
3	1,773,192	46,520,744	783,806,128	(9,527,629,380)		
4	2,320,4660	587,139,846	(9,527,629,380)			
5	233,999,928	5,702,475,480				
6	1,891,451,952					
Current	2,150,531,236	6,338,897,018	832,100,064	610,344,506	5,936,475,408	1,891,451,952
Soon	-	-	10,359,729,444	10,137,973,886	-	-

Table 1. Endgame databases computed.

The *n*-piece database is computed using a simple iterative algorithm, after the 1, 2, ..., $(n-1)$-piece databases are complete. Initially all *n*-piece positions are viewed as having an "unknown" value. Each iteration scans through all the positions trying to determine whether there is enough information to change a position's value from unknown to win, loss or draw. Given a black-to-move position, its value is determined by the following rules, in order of precedence (reverse the colors if it is white-to-move):

(1) Black wins if there exists a move to a white loss.

(2) The value of black's position remains unknown if there exists a move to a white position whose value is also unknown.

(3) Black draws if there exists a move to a white draw and black is unable to win by (1).

(4) Black loses if all moves lead to a white win.

The program iterates until no more *n*-piece positions can be resolved. At that point, all the remaining unknown positions must be draws.

The program must enumerate all the positions within a database in some order. Each possible board position should map to a unique number and, furthermore, from this number, one must be able to reconstruct the position. These complementary operations are simply a pair of indexing and de-indexing functions. Ideally, the enumeration should contain no gaps (numbers for which there is no corresponding board position), as this will increase the storage requirements.

Each iteration is a loop through all the positions in the database being computed. For each index, the board representation is generated, along with all black's possible moves. Each of the moves is made and the value of the resulting white-to-move board is found in one of the databases. By applying the above 4 rules, the value of the black-to-move position is computed.

Roughly half of the positions in a database are capture positions. Their values can be determined during the first iteration, since all captures lead to positions with $n-1$ or less pieces and whose values are therefore known from a previously computed database. After this has been done, the 1, 2, .., $(n-1)$-piece databases can be freed from memory; no non-capture move will ever play into them. The second and subsequent iterations through the database resolve only non-capture moves. As a result, the values of the positions will be resolved in order of least to most moves required to play into a lesser database. Thus, the algorithm can be viewed as finding all wins in 1 move, then 2, then 3, and so on.

This approach was sufficient to solve the 3k0c:3k0c (3 kings and no checkers versus 3 kings and no checkers) database, consisting of 18,123,840 positions. This database illustrates two problems inherent in the approach presented: space and speed. If this approach were naively scaled up to the entire 3:3 database, 783,806,128 positions would have to be represented, requiring 195 megabytes of memory (at 2 bits per position). On a 20 MIPS (millions of instructions per second) machine, it would require roughly a day per iteration, and hundreds of iterations would be required. The requirements were reduced by breaking the problem into sub-databases based on the material in the positions. Thus, the 3k0c:3kc0 database can be computed by itself and, once done, the 3k0c:2k1c can be tackled, etc. The prerequisite for computing a sub-database is that all the sub-databases that could be played into must already be completed. Even so, the biggest sub-database, 2k1c:1k2c, still requires over 55 megabytes of memory.

The key to solving databases this large is to partition the sub-databases into yet smaller parts. The approach used was to "slice" the positions up based on the rank of the leading (most advanced) checker for each side. A position with the leading checker on rank i stays in the same slice until a checker moves to rank $i+1$. Given that that slice is already solved, we need compute no further. Since each side with a checker can have its leading one on one of 7 possible ranks, the problem can be sub-divided into 49 slices if both sides have a checker. If only one side has a checker, then it can be sub-divided into 7 slices. Of course, all-king endgames cannot be sub-divided using this approach.

There are well-defined dependencies between the slices. The ranks of the leading checkers have to be started from their most advanced position (7th rank). Each slice plays into a preceding one. The largest in-memory image required to compute a "slice" in the 6-piece databases was approximately 20 megabytes. This is a huge improvement on the 195 megabytes necessary in the naive approach.

This still leaves the problem of speed. More than 50% of the nodes in the database exist in cycles of draws. These unresolvable nodes of the game tree may be visited and expanded repeatedly. To alleviate this inefficiency, we keep track of which nodes are resolved in each sweep and, using a reverse move generator, only attempt to resolve the nodes that play into them in the next sweep. This increased the speed of the program enough to allow the entire 3:3 database to be computed in about a week on a 20 MIPS machine with 64 megabytes of RAM.

The last issue is how to save the database. At 2 bits per position, the 3:3 database requires 195 megabytes. We have compressed this to 48 megabytes at the cost of some decompression overhead at program run-time. This smaller file allows us to keep part of

the 3:3 database resident in memory during a game, reducing costly disk i/o. More work is needed on compact representations of the databases.

Finally, it is possible to extrapolate the results of some databases to larger, uncomputed databases. The 6:1 and 5:2 databases are obviously lopsided and it takes an exceptional condition for the weaker side not to lose. An examination of these databases resulted in the discovery of a few heuristics that could determine with 100% accuracy the result of any of these positions. Given positions with more pieces on the dominant side, it seems likely that these heuristics will hold for larger databases. We have extrapolated these heuristics to "solve" the endgames with 7 or more pieces against one and 6 or more pieces against two. Although it cannot be stated with certainty that these heuristics are 100% accurate for these larger problems, we are confident that they are. These databases can be viewed as being solved, not by computation, but by "proof". Figure 2 shows the portion of the checkers search space that we consider solved.

5. Book

In checkers, as in other games, during the opening phase of the game the evaluation function combined with search is often insufficient to find the best strategic moves. Human players have the benefit of over 100 years of analysis to build extensive opening books, and it seems reasonable that this knowledge should be available to the program. On the other hand, *Chinook*'s search often finds play that is sound, but surprising to human players in that it departs from standard opening moves. In consideration of these two observations, we are in the process of developing a two-part opening book for *Chinook*.

In the first part, we are accumulating as many grandmaster games as possible, and storing the move sequences. Of course, many games follow the same initial opening move sequences, and often different openings converge onto the same positions in later parts of the game. Each game is not stored individually, but the set of positions arising from the game and the move played are stored. Further, a record is kept of whether the move led to a win, loss or draw for the player in question.

In one game, a move may lead to a win, but in another the same move may lead to a loss or draw (because of subsequent mistakes). To obtain a reliable value for the move, we have a post-processing program that identifies all of the leaf nodes of our position set, that is the positions from which we have no recorded moves. Then, working backwards, we back up the value for each move that leads to these leaf positions from any other position in the set. After processing the leaves, the backing-up of values continues recursively up the tree of positions, eventually reaching the opening position.

Note that during this process we can identify moves that transpose from any position in the set to any other, even though these moves are not recorded in any game or human analysis. Thus, the program has the potential to improve on human analysis by finding improved lines of play not previously recorded. We also have the capability of entering any game played by *Chinook* into this opening book. This means that *Chinook* is capable, to a limited degree, of rote learning from its mistakes and successes.

Chinook does not follow its opening book blindly, as many game programs do. One of the reasons is partly aesthetic. If we always follow a book line without question,

then *Chinook*'s games would often prove uneventful and dull. Against human players who know their openings, games could be played that always lead to the same result. If we want novel games, then we must allow *Chinook* to find its own moves.

Another reason is that if we follow published play without deviation, the human expert might know of a cook for the line we are following that has not been published, and thus lead us into a trap. Therefore we must always search as a matter of safety. Finally, we want *Chinook* to be able to take advantage of its strength: deep combinatorial searches.

To take advantage of the book, we use it only to bias our search. A bonus is added to a move in the search if it is a book move. If we are in a position in the book with several moves, some may lead to positions from which *Chinook* can win; others may lead to known draws and others only to known losses. For moves leading to winning lines we use a stronger bias than for those leading to draws, while we use a slight negative bias to lead us away from moves that lead to losses. We use only a slight bias for moves to losing lines, because the move was made by a grandmaster, and so may have much to recommend it. The actual error that led to the loss may occur much further on in the line of play. However, if *Chinook*, using its deep search, can find another move nearly as good, then the bias will cause it to take the non-book move. Altogether, this uses the best human information and combines it in a natural way with *Chinook*'s ability to come up with surprising lines of play.

The second part of the book is being generated automatically. In a tournament, the program is allowed an average of two minutes per move of computation. Using the same search heuristics, and using the first part of our book as described above, we use idle machine time to search for ten minutes on each position and record the best move found. We do this for every one of the openings, as well as for all possible moves that follow. In this way, layer by layer, we slowly expand a large tree covering every possible move an opponent might make. The main benefit of this approach is that during a tournament, we can make these moves immediately, without search, since we have already searched far deeper than we could during the game itself.

On the negative side, if we actually processed every node as described, we would spend much time analyzing positions that would never occur during actual play. Because of the exponential growth in the number of positions with the number of moves into the game, we can never hope to expand more than a few ply of the openings (perhaps as many as 20). What we really want is a minimal set of positions so that for every response by the opponent to our best move, we know our next best move. In the automatic book, we are trying to develop such a minimal structure.

Certain problems remain to be explored. Even with the benefit of ten minutes per position, we will eventually find that some of the moves we have chosen lead to bad positions for *Chinook* further on. We must find ways of determining when a particular line is sufficiently bad to abandon it. A difficult problem then arises of determining which of our moves in this bad line should be changed. Determining the best move to examine for change is critical, since we wish to minimize the time spent in searching and analyzing positions which are not going to be included in the final book. We could, of course, use human expert opinions, but once the program is in full operation, possibly using parallel

processing, it is unlikely that a human could keep up with the mass of problems being identified. Thus, we must find a way of doing this automatically.

Other research problems have to do with identifying the set of transpositions that minimize the total set size. Many lines of play can be played into the same positions. Once we expand on a particular position, it makes sense to play into it from other positions when possible, since this can lead to huge time savings as we progress deeper into the game. Clearly, the problem of minimizing the set size will interact with the preceding problem of fixing bad lines.

It is possible that searching for ten minutes per position is not a good idea. The main benefit of such a long search is that it gives us a move that is at least as reliable as any move we could compute under tournament game constraints. (This might not be true if someone decides to loan us a supercomputer). When we determine a method of fixing up bad lines of play, it may prove useful to use shorter search times. The issue is one of a trade-off in the amount of time spent searching versus the time lost when lines are determined to be bad, and thus portions of our structure have to be abandoned. This is another area we intend to spend time researching over the next two years. We note that these methods will likely have applications to other domains, such as chess.

In summary, the program currently has a precomputed book (currently over 4,000 positions) which it follows blindly for the first 5 moves (so far) of the game. When it leaves this portion of the book, there is a second book which consists of processed games and analyses of games (30,000 positions). This book is used to bias the search that *Chinook* makes.

6. Solving the Game of Checkers

There are roughly 5×10^{20} possible "placements" of checkers on the board. Most of them cannot be reached through legal play. Our best estimate is that there are roughly 10^{18} legal positions. To solve this many positions is not practical in the foreseeable future. However, an important observation drastically changes this assessment. The optimal $\alpha\beta$ search tree is roughly the square-root of the number of positions in a minimax tree. Hence, the solution tree for checkers may be of $O(10^9)$ positions - a tantalizingly small number! Thus, in principle, if one were to search the game of checkers and always consider the best move first in every position, one need only consider $O(10^9)$ positions. Given that *Chinook* can examine over 10^9 positions per day on its current hardware, in theory a perfect search could solve the game in one day!

Unfortunately, it is not possible to achieve the minimal tree in practice. The question is how close can one come? Even if the search efficiency is off by a factor of 1000-fold, the solution is still within reach using faster machines and exploiting parallelism. So solving the game may be feasible!

The most important step needed for solving checkers is the construction of endgame databases. With 6 pieces solved, we believe the 7-piece database to be achievable and we are striving in this direction. However, the 8-piece database is essential. We have reason to believe that this database will allow us to solve checkers. Unfortunately, the database comprises $O(10^{11})$ positions, and storage, as well as computation, become serious problems.

Our optimism is fueled by two additional pieces of information:

(1) We have done retrograde analysis on many grandmaster games, taking each position from the game and all the legal moves in those positions, and analyzing them to see if we can "prove" that there is a forced move sequence leading into the databases. If so, then we know the game-theoretic value of that position. Using only the 5-piece databases, we have a position as early as move 9 of a game with 22 men on the board that we can prove is a loss. So far, we have collected roughly 450 proven positions, all within the first 20 moves of the game and all with 8 or more pieces on the board.

(2) In less than two minutes at the start of the game, the program completes a 15-ply search (plus extensions). Already some of the leaf position values are taken from the database. Experiments show that searching the root position to a depth of 21 ply (plus extensions) would result in at least 2.5% of all leaf nodes coming from the 8-piece or less databases. With such a high hit count, it seems likely that the game can be solved. The program can start searching positions 5-10 moves into the game, where we estimate as many as 50% of the leaf nodes will be from the databases. After solving these positions, we can backtrack to the start of the game. This search will utilize the results of the positions 5-10 moves into the game to help reduce the size of tree needed to be searched.

7. Performance

The branching factor in checkers is considerably less than for chess (roughly 8 for a non-capture position and 1 for a capture position in our experience), not only because there are fewer moves, but also because capture moves are forced. Thus, it is possible to search considerably deeper in checkers than in chess. Of course, this also means humans search deeper too! As a point of comparison, the chess program *Phoenix* [15] searches to 7 ply in the middlegame and 8+ ply in the endgame on a 20 MIPS machine on a typical 3 minute move. *Chinook*, on comparable hardware, completes 15-ply at the beginning of the game and as pieces come off, the search depth typically rises to 19 ply. In the endgame, when the database is cutting off large sub-trees, search depths of 25 ply are not uncommon.

With the 6-piece databases, it is often surprising how quickly a database score can back up to the root of the search tree. For example, in an exhibition game against former World Champion Derek Oldbury, *Chinook* found a drawing line leading into the database on move 12. Oldbury didn't believe *Chinook*'s brashness and played an additional 20 moves before conceding the draw.

In August 1989, *Chinook* won the checkers tournament at the 1st Computer Olympiad, going undefeated [4]. In the Mississippi State Championship, August 1990, *Chinook* came an undefeated first. In the U.S. National Open that followed, *Chinook* won 4 and drew 4 of its 8 matches, finishing a clear second to the World Champion, Dr. Marion Tinsley. In these events, the program defeated Richard Hallet, Elbert Lowder, Don Lafferty and Ron King, all players considered in the top 10 in the world. In 4 tournament games against Tinsley, *Chinook* held its own, drawing all 4.

8. Conclusions

Plans are underway for a World Championship match with Dr. Marion Tinsley. Given the current state of the program, we believe that Dr. Tinsley will win this match. However, in two years time the combination of deeper search (through hardware and software enhancements), the 7- and 8-piece databases (if possible), and an extensive computer-generated opening book, will make it very difficult for Dr. Tinsley to defend his title.

The dream of solving the game of checkers may be a long way off. The 8-piece databases are computable with today's technology, but the issue of how to efficiently save and access the roughly 10 gigabytes of data is a difficult problem. This is one area where we will be concentrating our research effort over the next 2 years.

The public perception of checkers is that it is a game for children and old men. This is unfortunate; the game deserves to be more popular than it is. One of the surprising aspects of this work (to us at least) has been discovering the unanticipated subtleties inherent in the game. The game has a beauty all its own, requiring a more delicate touch than does chess and a subtlety of play that rivals Go. Unfortunately, the simplicity of the rules is often misconstrued. In fact, it is this simplicity that enhances the elegance of the game. It is our hope that *Chinook* can help popularize checkers and re-establish it as an intellectual game *par excellence*.

Acknowledgments

Many thanks to Alynn Klassen, Steve Sutphen, Gordon Atwood and Jim Easton for their technical help. Franco Carlacci and Randal Kornelsen did our graphics interface. Patrick Lee assisted with our initial database program. Our group developed the 4- and 6-piece databases, but the 5-piece results came from Ken Thompson.

Financial assistance from the Natural Sciences and Engineering Research Council of Canada and from IBM in Edmonton (Canada), Memphis (United States), and London (England) is gratefully acknowledged.

References

1. A.L. Samuel, Some Studies in Machine Learning Using the Game of Checkers, *IBM Journal of Research and Development 3*, 3 (1959), 210-229. See also (1963), *Computers and Thought*, E.A. Feigenbaum and J. Feldman (eds.), McGraw-Hill, pp. 71-105..
2. A.L. Samuel, Some Studies in Machine Learning Using the Game of Checkers II - Recent Progress, *IBM Journal of Research and Development 11*, 6 (1967), 601-617.
3. T. Truscott, The *Duke* Checkers Program, Duke University report, 1978.
4. D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence*, Ellis Horwood Limited, London, 1989.
5. D.J. Slate and L.R. Atkin, Chess 4.5—The Northwestern University Chess Program, in *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, 1977,

82-118.

6. J. Schaeffer, The History Heuristic and Alpha-Beta Search Enhancements in Practice, *IEEE Transactions on Pattern Analysis and Machine Intelligence 11*, 11 (1989), 1203-1212.

7. T.S. Anantharaman, M.S. Campbell and F-h. Hsu, Singular Extensions: Adding Selectivity to Brute-Force Searching, *AAAI Spring Symposium Proceedings*, 1988, P 8-13. Also published in the *Journal of the International Computer Chess Association 11*, 4 (1988), 135-143 and in *Artificial Intelligence 43*, 1 (1990), 99-110.

8. J.J. Gillogly, Performance Analysis of the Technology Chess Program, Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, 1978.

9. J. Schaeffer and T.A. Marsland, The Utility of Expert Knowledge, *International Joint Conference on Artificial Intelligence*, 1985, 585-587.

10. D. Oldbury, *The Complete Encyclopedia of Checkers*, Draughts & Checker Players' Guild, Torquay, England, 1978. In 6 volumes.

11. F-h. Hsu, M.S. Campbell, T. Anantharaman and A. Nowatzyk, Deep Thought, in *Computers, Chess and Cognition*, T.A. Marsland and J. Schaeffer (ed.), Springer-Verlag, New York, 1990, 55-78. In press.

12. T. Anantharaman, A Statistical Study of Selective Min-Max Search, Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, 1990.

13. T.A. Marsland, Evaluation-Function Factors, *Journal of the International Computer Chess Association 8*, 2 (1985), 47-57.

14. K. Thompson, Retrograde Analysis of Certain Endgames, *Journal of the International Computer Chess Association 9*, 3 (1986), 131-139.

15. J. Schaeffer, Experiments in Search and Knowledge, Ph.D. thesis, Department of Computer Science, University of Waterloo, 1986.

Appendix: Heuristics

Chinook has four game phases (not to be confused with the "stages" of opening, middlegame and endgame). These phases are defined statically by the number of pieces on the board. Currently, phases 1 to 4 are defined to contain 20 - 24, 14 - 19, 10 - 13 and less than 10 pieces, respectively. There are currently 20 adjustable weights per phase, for a total of 80. Heuristic weights can be adjusted for each phase independently.

The heuristics can be grouped together as follows, although some heuristics have characteristics of more than one class. It is interesting to compare these with the heuristics used by Samuel [1, 2].

(a) Material:

- PIECE COUNT: Difference in the number of checkers each side has.
- KING COUNT: Difference in the number of kings each side has.

(b) Relational:

- PIN: One piece holding another at the side of the board. Important in the endgame.

- BALANCE: How equally the pieces are distributed on the left and right sides of the board.

(c) Square Values:

- ANGLE (two heuristics): Encourage some men on the back two ranks to move forward.
- BACKROW: Encourage men to stay behind and keep the backrank covered.
- CENTRALITY: The influence of checkers on the center of the board.
- ADVANCEMENT: The strength of a checkers as it advances up the board.
- SHADOW: A count of the squares that can never again be defended by a man, as a result of the advancement of the men.
- KING CENTRALIZE: Center control for kings.
- CORNERS: Penalties or bonuses to aid king play in double corners.

(d) Forward Planning:

- MOBILITY (three heuristics): The degree to which pieces are free or restricted in their movements.
- RUNAWAY: The ability of a checker to advance unimpeded to become a king.
- FREE KING: A mobile king can harass loose opposing men, and so is given bonus points according to the disposition of back rank defenders.
- TRAPPED KING: Kings that are trapped in corners and unlikely to be freed.
- EXCHANGE: When ahead materially, encourage exchanges that will bring the game closer to its conclusion.
- NUMB KINGS: When ahead prefer positions where the opponent has fewer kings. When behind, prefer positions where you have many kings.

(e) Timing:

- SIDE TO MOVE: In checkers having the move is usually an advantage, but there are many cases where it can be a disadvantage (*zugzwang* in chess).
- THE MOVE: "The move" (or *opposition* in chess) is an endgame term that defines whether one side can force the other to retreat.