Solving Checkers*

J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu and S. Sutphen
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
jonathan@cs.ualberta.ca

Abstract

AI has had notable success in building high-performance game-playing programs to compete against the best human players. However, the availability of fast and plentiful machines with large memories and disks creates the possibility of *solving* a game. This has been done before for simple or relatively small games. In this paper, we present new ideas and algorithms for solving the game of checkers. Checkers is a popular game of skill with a search space of 10^{20} possible positions. This paper reports on our first result. One of the most challenging checkers openings has been solved – the White Doctor opening is a draw. Solving roughly 50 more openings will result in the game-theoretic value of checkers being determined.

1 Introduction

High-performance game-playing programs have been a major success story for AI. In games such as chess, checkers, Othello, and Scrabble, mankind has been humbled by the machine. However, although these programs are very strong, they can still lose a game to a human. They are strong, but not perfect. Perfection requires one to "solve" a game. Allis defines three levels of solving [Allis, 1994]:

1. Ultra-weakly solved. The game-theoretic value for the game has been determined. An ultra-weak solution is mainly of theoretical interest. For example, Hex is a first player win, but no one knows the winning strategy.
2. Weakly solved. The game is ultra-weakly solved and a strategy is known for achieving the game-theoretic value from the opening position, assuming reasonable computing resources. Several well-known games have been weakly solved, including Connect Four [Allis, 1988], Qubic [Allis, 1994], Go Moku [Allis, 1994], and Nine Men's Morris [Gasser, 1996].
3. Strongly solved. For all possible positions, a strategy is known for determining the game-theoretic value for both players, assuming reasonable computing resources. Strongly solved games include 6-piece chess endgames and Awari [Romein and Bal, 2003].

Note the qualification on resources in the above definitions. In some sense, games such as chess are solved since the minimax algorithm can in principle determine the game-theoretic value, given a large enough amount of time. Resource constraints preclude such impractical "solutions".

How difficult is it to solve a game? There are two dimensions to the difficulty [Allis *et al.*, 1991]: *decision complexity*, the difficulty required to make correct decisions, and *space complexity*, the size of the search space. Checkers is considered to have high decision complexity and moderate space complexity. All the games solved thus far have either low decision complexity (Qubic; Go-Moku), low space complexity (Nine Men's Morris, size 10^{11}; Awari, size 10^{12}) or both (Connect-Four, size 10^{14}).

Checkers, or 8×8 draughts, is popular in the British Commonwealth (present and past) and in the United States. The rules are simple (pieces move one square at a time diagonally, and can capture by jumping) and the number of piece types is small (kings and checkers), yet the game is quite challenging. Checkers has high decision complexity (more complex than 9×9 Go and the play of bridge hands, on par with backgammon, but less complex than chess) and moderate space complexity (10^{20} positions versus unsolved games such as backgammon, 10^{19} positions, and chess, 10^{44} positions). The best checkers programs are stronger than the best human players (e.g., CHINOOK won the World Man-Machine Championship [Schaeffer, 1997]).

The number of possible placings of checkers pieces on the board is roughly 5×10^{20} [Chinook, 1995]. This number is misleading since it includes positions that are not legally reachable from the start of the game. For example, although there are 9×10^{19} possible positions with 24 pieces on the board, only a small fraction can be reached in a game (e.g., 12 kings versus 12 kings is not possible).

Our effort to solve the game of checkers began in 1989! After almost 16 years of research and development, and countless thousands of computer hours, we are pleased to report the first major milestone in our quest. The White Doctor opening (shown in Figure 1) has been proven to be a draw.[1] This is a

*The support of NSERC, iCORE and the University of Alberta is acknowledged.

[1] Tournament checkers is played using the so-called "three-move ballot". In competitions, games begin after the first 3 moves (3 ply)

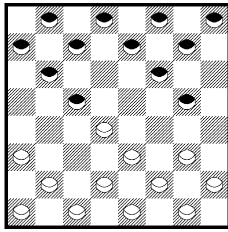

Figure 1: The White Doctor: White to play and only draw

difficult opening, as Black begins the game with a huge positional disadvantage—so large that humans consider Black's best play to be to sacrifice a checker early on to relieve the pressure. Notably, this opening was played in the decisive game of the 1992 Tinsley–CHINOOK World Championship match [Schaeffer, 1997].

This paper has the following contributions:

1. A new hybrid combination of best-first and depth-first search results to the construction of minimax proof trees. Heuristic value iteration is introduced as a means of concentrating search effort where it is most needed.
2. A solution to the White Doctor opening. Achieving this result took years of computing, involving the pre-computation of 10^{13} values (endgame databases), and building the proof tree which took a further roughly 10^{13} positions. This is an important first step towards weakly solving the game of checkers.
3. The largest game-tree proof to date. This was made possible not only by our new hybrid search approach, but also by the integration of several state-of-the-art algorithms and enhancements into one coherent system.

2 Algorithm Overview

A formal proof of a game's value can be done by depth-first alpha-beta ($\alpha\beta$) or best-first proof number search (PNS) [Allis, 1994]. For the large and complex checkers search space, neither turned out to be effective. Instead we propose a hybrid approach of the two: heuristically guided proofs.

The proof procedure has four algorithm/data components:

1. Endgame databases (backward search). Computations from the end of the game backward have resulted in a database of 4×10^{13} positions (≤ 10 pieces on the board) for which the game-theoretic value has been computed (strongly solved).
2. Proof tree manager (search management). This component maintains a tree of the proof in progress, traverses it, and generates positions that need to be explored to further the proof's progress.
3. Proof solver (forward search). Given a position to search, this component uses two programs, $\alpha\beta$ and PNS, to determine the value of the position.

are randomly selected. Each starting position is played twice, with sides switched for the second game.

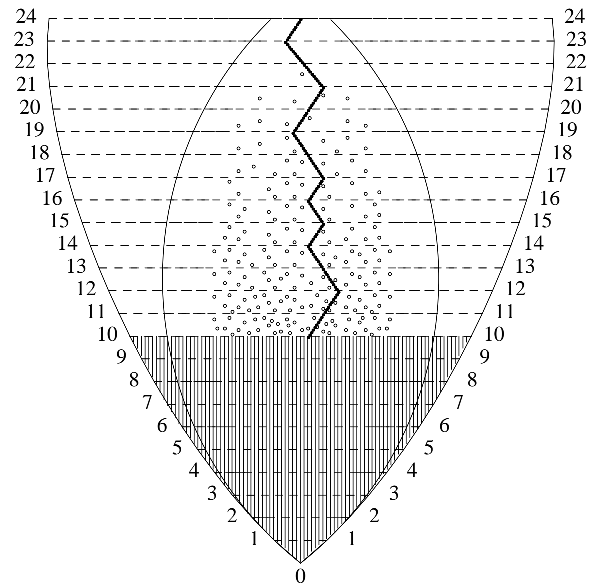

Figure 2: The checkers search space (log scale horizontally)

4. Seeding (expert input). From the human literature, a single line of "best play" is identified. This is fed to the proof tree manager as the initial line to explore.

Figure 2 illustrates this approach. It plots the number of pieces on the board (vertically) versus the logarithm of the number of positions (using data from [Chinook, 1995]; the the widest point is with 23 pieces, 1.3×10^{20} positions). The endgame database phase of the proof is the shaded area, all positions with 10 or fewer pieces.

Seeding the proof process with an initial line of play is critical to the prover's performance; it is shown in Figure 2 as a sequence of solid bold lines. The line leads from the start of the opening into the endgame databases. Seeding is not necessary to the proof—the prover is capable of doing the proof with no expert guidance. However, this single line of play allows the proof process to immediately start doing work that is likely to be relevant to the proof. Without it, we have seen the prover spend considerable effort flailing about while it tries to find the key lines of play.

The inner oval area in Figure 2 illustrates that only a portion of the search space is relevant to the proof. Positions may be irrelevant because they are unreachable in this opening, or are not required for the proof. The small circles illustrate positions with more than 10 pieces for which a value has been proven. Each of the circles represents the root of a search tree which has been solved. For clarity, the solved subtree below that root is not shown in the diagram.

3 Endgame Databases

Endgame databases were pioneered in chess [Thompson, 1986], and were instrumental in the success of the CHINOOK program [Schaeffer, 1997]. Using retrograde analysis, subsets of a game with a small number of pieces on the board can be exhaustively enumerated to compute which positions are

wins, losses or draws. Whenever a search reaches a database position, instead of using an inexact heuristic evaluation function, the exact value of the position can be retrieved.

We have computed all endgame databases up to 10 pieces on the board, for any combination of kings and checkers [van den Herik *et al.*, 2003]. The total database size is 39 *trillion* positions, roughly 4×10^{13}. Although this sounds impressive, it represents a paltry one ten-millionth of the total search space [Chinook, 1995]!

How valuable are the databases? Unfortunately, because of the win/loss/draw construction, a good measure of the difficulty of 10-piece endgames is not available—although it is presumed to be very high. Anecdotal evidence includes:

1. Perfect play databases for the 7-piece endgames have been computed, with the longest winning line being 253 ply [Trice and Dodgen, 2003].
2. [Schaeffer *et al.*, 2003] report that some of their database computations had winning lines in excess of 180 ply—just to force a checker advance or a piece capture.
3. In the 5-king versus 3-king and 2-checker endgame, the longest winning line is 279 ply [Gilbert, 2005].
4. Draws are much harder to prove than wins/losses.

This evidence strongly suggests that the databases are removing hundreds of ply from the search depth of the proof tree.

4 Search Algorithm

The proof search is split into a front-end proof tree manager and a back-end prover. The front-end manager maintains a large persistent proof tree, and selects, stores, and organizes the search results. Proofs of individual positions are done in parallel by a farm of back-end provers.

4.1 Proof Tree Manager (front end)

The proof tree manager starts off with an empty proof tree. The proof is "seeded" by being fed an initial line of play. This line is considered by humans to be best play by both sides. The proof manager starts at the last position in the line and solves it, then backs up one ply, solves that sub-tree, and so on until the root of the line is reached.

The proof is done iteratively, as shown in Figure 3. In minimax search, we have seen iterative algorithms based on search depth (iterative deepening) and confidence in the root value (as in conspiracy numbers [McAllester, 1988]). Our algorithm is novel in that it iterates on the range of relevant heuristic values for the proof.

Classical iterative deepening in game-playing programs is used for several reasons, one of which is to improve search efficiency by using the $\alpha\beta$ tree from iteration i as the basis for starting iteration i+1. The assumption is that increasing the depth does not cause substantial changes to the tree structure; each iteration refines the previous iteration's solution. In contrast, without iterative deepening, the search can go off in the wrong direction and waste lots of effort before stumbling on the best line of play.

Our iterative value solution works for similar reasons. A position may require considerable effort to formally determine its proven value. However, the higher/lower the score of the position, the more likely it is a win/loss. Rather than invest the effort to prove such a position, this work is postponed until it has been shown to be needed. We do this by introducing two new values to the proof tree. To win, loss, draw and unknown, we add *likely win* and *likely loss*.

All searches are done relative to a heuristic threshold. The back-end prover returns an assessment of a position. Any position exceeding the threshold is considered a likely win; any falling below the negative threshold is a likely loss. The proof manager repeatedly traverses the proof tree, identifies nodes to be searched, sends them off for assessment by the prover, and integrates the results. Any node with a score outside the threshold has a likely result and is not expanded further within this threshold; it is effectively treated as if it is proven. The thresholds can be loosely thought of as imposing a symmetric $\alpha\beta$ search window on the best-first proof manager. A value outside the window in $\alpha\beta$ means the result is irrelevant; a value outside the heuristic threshold means that we postpone the proof until we know the result will be relevant.

Once the proof tree is complete for a given heuristic threshold, the threshold is increased and the search restarted to construct a proof for the new threshold. Any position that was previously resolved to a real proven value remains proven, and positions that have a heuristic score that remain outside the new heuristic threshold remain likely wins/losses. Only non-proven positions with heuristic values inside the new threshold are now considered, and they must be re-searched with the new heuristic limit. This iterative process continues until the heuristic threshold exceeds the largest possible heuristic value. At this point all likely wins/losses have been resolved, and the proof tree is complete.

Ideally, after the first likely proof tree is constructed, the work would consist solely of trying to prove that the likely wins/losses are even more likely wins/losses. In practice, the heuristics we use are occasionally wrong (they are, after all, only heuristics), and a proof tree for a position may grow large enough that other previously less desirable positions become a more attractive alternative for expansion.

For the White Doctor, we used settings of MIN = 125 (a checker is worth 100 points) and INC = 5 in Figure 3. These might not be the best choices; we are experimenting with different values on other checkers openings.

The proof manager uses PNS, trying to answer two questions about the value of the proof tree: "is it at least a likely win", and "is it at most a likely loss." A likely win is assigned a proof number of 0 for the is-likely-win question, a likely loss is assigned a proof number of 0 for the is-likely-loss question, and a draw is assigned a disproof number of 0 for both questions. The prover tries to investigate the two questions simultaneously by selecting nodes for consideration that (ideally) contribute to answering both.

The proof tree manager traverses the proof tree and identifies nodes that are needed or likely to be needed in the proof. While a traditional PNS algorithm expands one node at a time, our manager expands many promising nodes (possibly hundreds of them), to have enough work to keep multiple back-end prover processors busy.

When a node is selected for expansion, it is added to a work-to-do file that is used as input to the prover pro-

```
ProofManager( Seeding[] ) {
  // Seeding: seeding line for the proof
  // Seeding[ 1 ] is the opening position
  // MIN: starting proof heuristic threshold
  // WIN: ending proof threshold
  // INC: heuristic increment between likely proofs

  // Iterate on heuristic threshold
  for ( thresh = MIN; thresh <= WIN; thresh += INC )
    // Iterate over the seeded line
    for ( move = Length( Seeding ); move > 0; move -= 1 ) {
      pos = Seeding[ move ];

      // Solve current position to threshold
      repeat {
        worklist = PNSearch( pos, thresh );
        if( NotEmpty( worklist ) ) {
          StartProvers( worklist, results, thresh );
          WaitForResults( worklist, results );
          UpdateProofTree( pos, results, thresh );
        }
      } until ( IsEmpty( worklist ) );
    }
  }
}
```

Figure 3: Proof manager

```
Prover() {
  repeat {
    GetWork( pos, thresh );

    // HeuristicSearch returns alpha-beta heuristic value
    heur = HeuristicSearch( pos, thresh );

    // If value inside threshold range, do PNSearch
    lower = LOSS; upper = WIN;
    if( -thresh*2 < heur && heur < 2*thresh ) {
       // This is a simplified version of the actual code
       // Check for WIN, at most DRAW, or UNKNOWN
       result = DFPNSearch( pos, IS_WIN );
            if( result ==    PROVEN ) { lower = WIN ; }
       else if( result == DISPROVEN ) { upper = DRAW; }

       if( lower != upper && time remaining ) {
         // Check for LOSS, at least DRAW, or UNKNOWN
         result = DFPNSearch( pos, IS_LOSS );
              if( result ==    PROVEN ) { upper = LOSS; }
         else if( result == DISPROVEN ) { lower = DRAW; }
       }
    }
    SaveResult( pos, thresh, heur, lower, upper );
  }
```

Figure 4: Proof solver process

cesses. The back-end is given information for the generation of heuristic proof numbers and any proven information that is known about the position (e.g., it might already be known that it is not a loss).

The proof manager saves all back-end search results so that they can be re-used for searches with a different heuristic threshold (or from a different opening). A persistent B-tree is used to store information on every position examined in every opening, including both the back-end results and the backed-up values computed by the front end.

In the cases where the back-end proof search does not provide enough information to answer the front end's question of interest, the heuristic score from the back end is used to generate heuristic proof and disproof numbers. CHINOOK is used to provide the heuristic scores. If the $\alpha\beta$ search result is small enough to initiate the proof search, then the result is used to help initialize the proof numbers. The proof number for a win is a function of: 1) the difference between the heuristic score and the heuristic threshold (a high score is more likely to be a win than an even score), and 2) a penalty is added for positions where both sides have a king, as these tend to be much harder to prove.

The disproof number for a loss is the same as the proof number for a win, with one exception. We use one additional statistic from CHINOOK. If the principal variation of the $\alpha\beta$ search result depends on a draw from a database lookup, we decrease the disproof number. Proof numbers for a loss and disproof numbers for a win are defined similarly. As a final modification, the numbers that depend on the minimum of their child values (as opposed to the sum) are divided by the average branching factor.

4.2 Proof Solver (back end)

Pseudo-code for the back-end prover is shown in Figure 4. It is a novel combination of a traditional heuristic $\alpha\beta$ search and a proof search. The (cheap) heuristic search value is used to determine whether the current position is relevant to the current proof threshold. The (expensive) proof search attempts to formally prove the result of the current position. The latter is only done if the former indicates it is needed.

Each back-end position is first searched by an $\alpha\beta$ searcher using a heuristic evaluation function and the endgame databases. The search is limited to 25 seconds, reaching a depth of 17 to 23 ply plus search extensions. The heuristic value returned is compared to the search threshold. Values outside the threshold range are considered as likely wins/losses by the proof tree manager. If the heuristic value is more than twice the size of the threshold, then no further processing is done; the formal proof search is postponed until later in the proof when we will have more confidence that we really need this result.

The prover uses Nagai's Df-pn$^+$ algorithm [Nagai, 2002], a depth-first search variant of best-first PNS. Df-pn$^+$ was preferred over PNS, in part, because of memory concerns. The algorithm's space requirements are limited by the size of the transposition table. In contrast, PNS needs to store the entire tree in memory. This gives Df-pn$^+$ an important advantage, since memory is also needed to reduce the frequency of costly disk I/O caused by endgame database accesses.

The time limit for the Df-pn$^+$ component is 100 seconds per search. Most of the search time is spent trying to answer the "easy" question, the opposite result to that of the heuristic search. For example, if the heuristic result indicates that the side to move has a large advantage (a possible win), the prover first tries to prove or disprove the position as a loss. This was done because this was usually the easier question to answer and a disproof of a loss or win may be a sufficient answer for the proof manager given that we expect the value of the proof to be a draw. If a disproof for the question is found in less than the maximum time allotted, another proof search is started to answer the other proof question.

Proving the value of a node implies finding a path to the endgame databases (or a draw by repetition). Clearly, the fewer pieces on the board, the "closer" one is to a database

position and the easier it should be to (dis)prove. In addition, it should be much easier to prove a win or disprove a loss if one side has more pieces than the other side. We combine these two factors to initialize the proof and disproof numbers, giving more attention to fewer pieces and favorable positions. Furthermore, because the standard Df-pn$^+$ algorithm has a fundamental problem of computing (dis)proof numbers when position repetitions are present, the techniques in [Kishimoto and Müller, 2003] are incorporated into the solver.

4.3 Graph History Interaction

A forward-search-based proof cannot be correct unless the Graph History Interaction problem (GHI) is properly addressed. GHI can occur when a search algorithm caches and re-uses a search result that depends on the move history. In checkers, repeated positions are scored as a draw. It is possible that the value of a search is influenced by a draw-by-repetition score, and this result is saved in the proof tree or in a transposition table. If this cached result is later reached by a different move sequence, it might not be correct—we do not know if the draw-by-repetition scores are correct.

In the back-end prover, GHI is correctly handled by utilizing the new technique described in [Kishimoto and Müller, 2004]. GHI is avoided in the boundary between the front-end tree and the back-end searches by not communicating any move history information to the back-end. Finally, in the front-end manager, GHI is avoided by not saving any value in the proof tree that is unsafely influenced by a draw-by-repetition score somewhere in its subtree. This may cause the manager to spend extra time searching the tree, but no extra prover searches need be performed, as the prover results stored in the tree are path independent.

4.4 Implementation Insights

The effort to turn likely wins and likely losses into proven values takes the vast majority of search effort. This holds even for a fairly high initial heuristic threshold, where likely wins/losses are positions that human checkers experts agree are "game over". The major reason for this is postponing moves; a non-winning side always takes the path of maximum resistance. For example, the losing side in a position is able to make terrible moves, such as sacrificing a checker for no apparent gain, to postpone reaching the endgame databases; the unknown aspect of a position with more than 10 pieces on the board (no matter how hopeless the position) is preferable to a known database loss. These lopsided positions must still be chased down to reach a proven conclusion.

The proof process is computationally demanding. Some of the performance issues include:

1. The proof process spends most of its time accessing the endgame databases. The cost can be reduced by locality of disk accesses and caching. Smart organization of the endgame databases and use of a 4GB machine (most of the memory going to I/O caching), means that 99% of all endgame database positions do not require I/O.
2. Despite the above performance optimizations, the proof process is still I/O intensive—keeping the I/O subsystem at maximum capacity. It is rare to see the CPU usage beyond 10%. Many machine architectures do not have the hardware reliability for this sustained I/O demand.
3. The massive I/O means that we can only use local I/O; network I/O is too slow. That limits the machines that we can add to the computation to those that have 300 GB of available local disk.
4. I/O errors. Our programs are instrumented to detect and recover (if possible) from most types of I/O errors.

5 Results

The White Doctor opening was chosen as the first candidate to solve because of its importance in tournament practice to the checkers-playing community. The opening is one of the most difficult of the three-move ballots, with Black starting off with a large disadvantage. Many decades of human analysis determined that Black's desperate situation meant that a checker had to be sacrificed to relieve the pressure. In a typical game, Black is down material for most of the game, barely hanging on to salvage a draw. Is the human analysis correct?

The White Doctor proof began in November 2003 and a heuristic proof for a threshold of 125 was completed one month later. For the next eight months, computer cycles were spent increasing the heuristic threshold to a win and verifying that there were no bugs in the proof.[2] In August 2004, we thought the proof was complete, but an inspection showed that some trivial positions with an advantage of 7 or more checkers had been eliminated from the proof (an accidental, historical error). This was corrected and in January 2005, we completed the last of the unresolved positions.

Is the proof correct? The endgame databases are being verified by Ed Gilbert, the author of the KINGSROW checkers program. He has independently verified that our 9-piece databases and 30% of our 10-piece databases are correct (his computation is ongoing). The front-end White Doctor proof tree is consistent and correct, assuming that all the back-end proofs returned the right answer. Many of the back-end values have been re-verified by a different solver developed as part of this project, increasing our confidence that the proof is indeed correct. The proof is available online at `http://www.cs.ualberta.ca/~chinook`. Several strong players have looked at the proof lines and found no errors.

Some interesting statistics on the proof include:

1. Number of positions given to the back-end to solve (excluding work that was used for experiments or shown to have bugs): 841,810.
2. Number of positions in one minimal proof tree: 223,178. The minimal proof tree is not a minimum proof tree. We did not try to maximize the usage of transpositions or choose the branches leading to the smallest subtree in generating the proof tree.
3. Longest line searched in the front-end proof tree: 67 ply.
4. Longest line in our minimal proof tree: 46 ply. The leaf node in the tree is the result of a 100 second proof number search (to a variable depth, possibly as much as 45

[2] Several minor problems were found, necessitating re-doing some of the calculations.

ply deep), terminating in an endgame database position (possibly worth 100s of plys of search).

5. Number of nodes in a back-end search (25 seconds of clock time for each CHINOOK search and 100 seconds for a proof number search): averaging 13,000,000 positions (slow because of extensive I/O).
6. Average number of processors used: seven (with 1.5 to 4 GB of RAM).
7. Total number of nodes searched in the proof: roughly $841,810 \times 13,000,000 > 1 \times 10^{13}$ positions. The number of unique positions is considerably smaller.

So, how does the computer proof stack up against human analysis? We used Fortman's classic *Basic Checkers* as our guide and compared its analysis to that of the proof [Fortman, 1977]. Of the 428 positions given in the book for this opening, there were no disagreements in the game-theoretic result of a position. This is a bit misleading since some of the positions were not part of the proof, and for others the proof only had a bound (less than or equal to a draw) whereas the book had a result (draw). Nevertheless, this result provides further evidence that the computer proof is correct. More impressive, however, is the quality of the human analysis. The analysis of this opening evolved over 75 years, had many human analysts contributing, requires analysis of lines that are over 50 ply long, and requires extensive insight into the game to assess positions and identify lines to explore.

The checker sacrifice in the White Doctor is correct according to our proof. This line is sufficient to prove that the opening is a draw. We have not invested computer cycles to find out if alternate non-sacrifice defences also lead to a draw.

We have made substantial progress on two more checkers openings. Of interest is that one of these openings appears to be roughly 10 times more difficult than the White Doctor, whereas the other one seems to be half as difficult.

6 Solving Checkers

When will checkers be weakly solved? The White Doctor is solved, and two other proofs are well under way (both appear to be draws). There are 174 three-move checkers openings and, ideally, we would like to solve all of them. However, to solve checkers for the initial starting position, with no moves made, roughly 50 openings need to be computed ($\alpha\beta$ cutoffs eliminate most of the openings). Solving subsequent openings will not take as long as the first one did. The computations will speed up for several reasons:

1. Algorithm efficiency. We will switch from experimental mode (trying different performance parameters) to production mode (fixed parameters).
2. More endgame databases. A new algorithm will allow us to compute parts of the 11-piece database, getting 50% of the benefits for 1% of the computing effort.
3. Data reuse. Openings transpose into each other. Each additional opening increases the chances for reusing the results of one opening for another.

The only obstacle remaining is access to computing resources. Computer cycles are easy to obtain; the bottleneck is large local data storage and I/O speeds. With enough resources, the game could be weakly solved within a year.

References

[Allis *et al.*, 1991] V. Allis, J. van den Herik, and I. Herschberg. Which games will survive. In *Heuristic Programming in Artificial Intelligence 2*, pages 232–243. Ellis Horwood Limited, 1991.

[Allis, 1988] V. Allis. A knowledge-based approach to Connect-Four. The game is solved: White wins. Master's thesis, Vrije Universiteit, Amsterdam, 1988.

[Allis, 1994] V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Department of Computer Science, University of Limburg, 1994.

[Chinook, 1995] Chinook, 1995. http://www.cs.ualberta.ca/~chinook/databases/total.php.

[Fortman, 1977] R. Fortman. Basic checkers, 1977. Privately published, but online at http://home.clara.net/davey/basicche.html.

[Gasser, 1996] R. Gasser. Solving Nine Men's Morris. *Computational Intelligence*, 12:24–41, 1996.

[Gilbert, 2005] E. Gilbert, 2005. http://pages.prodigy.net/eyg/Checkers.

[Kishimoto and Müller, 2003] A. Kishimoto and M. Müller. Df-pn in Go: Application to the one-eye problem. In van den Herik et al. [2003], pages 125–141.

[Kishimoto and Müller, 2004] A. Kishimoto and M. Müller. A general solution to the graph history interaction problem. In *AAAI*, pages 644–649, 2004.

[McAllester, 1988] D. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

[Nagai, 2002] A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Department of Information Science, University of Tokyo, 2002.

[Romein and Bal, 2003] J. Romein and H. Bal. Solving the game of awari using parallel retrograde analysis. *IEEE Computer*, 36(10):26–33, 2003.

[Schaeffer *et al.*, 2003] J. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, and S. Sutphen. Building the checkers 10-piece endgame databases. In van den Herik et al. [2003], pages 193–210.

[Schaeffer, 1997] J. Schaeffer. *One Jump Ahead*. Springer-Verlag, 1997.

[Thompson, 1986] K. Thompson. Retrograde analysis of certain endgames. *Journal of the International Computer Chess Association*, 9(3):131–139, 1986.

[Trice and Dodgen, 2003] E. Trice and G. Dodgen. The 7-piece perfect play lookup database for the game of checkers. In van den Herik et al. [2003], pages 211–230.

[van den Herik *et al.*, 2003] J. van den Herik, H. Iida, and E. Heinz, editors. *Advances in Computer Games 10*. Kluwer Academic Publishers, 2003.