Challenges and Progress on Using Large Lossy Endgame Databases in Chinese Checkers

Nathan R. Sturtevant(✉)

Department of Computer Science, University of Denver, Denver, CO, USA
sturtevant@cs.du.edu

Abstract. A common evaluation function for playing Chinese Checkers with two or more players has been the single-agent distance across the board. This is an abstraction of a perfect heuristic, because it ignores the interactions between the players in the game. Previous work has studied these heuristics for smaller versions of the game, including 6-piece data for a board with 49 locations and 81 locations which have 13.98 million and 324.5 million combinations respectively. The single-agent solution to the full game of Chinese Checkers has 81 locations and 10 pieces per player. This results in 1.88 trillion possible positions and is stored using 500 GB of disk space. In this paper we report results from a preliminary study on how to best use the data to improve the play of a Chinese Checkers program.

1 Introduction

Endgame databases have been a key component of game-playing programs in games like Checkers [23] and Chess [15], where they contain precise information that is easier to pre-compute than it is to summarize in heuristics or machine-tuned evaluation functions. Endgame databases are most useful when a game converges to simpler positions that may take a long time to resolve. In this case, the size of the endgame databases are small relative to the size of the game, and the computation associated with the endgame database would be non-trivial to reproduce at runtime. For instance, work on 7-piece chess endgame databases has discovered a position where 549 moves are required for mate [1]. These lines of play would not be discovered by programs that just attempt to find the best move from the same position, as the depth of search required is far beyond what could be found by normal time controls during a competitive game. Similarly, work on Checkers databases helped prove that a well-studied position from 1800, once assumed to be a win, was actually a draw [22]. Databases have also been successfully built for games like Chinese Chess [7].

Endgame databases are not universally useful, however, particularly when there are many possible endgame positions and when the resolution of each position is simple. In games such as card games, for example, endgame databases are essentially useless. There are $\frac{52!}{44!2^4} = 1.9$ trillion combinations of two cards that 4 players can have at the end of a trick-based card game (assuming a standard 52 card deck), but there are at most 16 ways to play out each possible hand.

T. Cazenave et al. (Eds.): CGW 2015/GIGA 2015, CCIS 614, pp. 3–15, 2016.
DOI: 10.1007/978-3-319-39402-2_1

In this case it will likely be faster to compute the result at runtime than to look the result up in the endgame database.

The game of Chinese Checkers is unique because the game decomposes into a single-agent game as it nears completion. This means that a single-agent heuristic can be used as an endgame database. That is, we can solve the single-agent version of the game to provide perfect distances to the end of the game once players' pieces have separated. But, there is also significant information about piece formation encoded in a full single-agent solution that is helpful for playing the game before it decomposes into a single agent game [25].

Work in the game of Chinese Checkers has often used a smaller board containing 49 locations, instead of the more common 81 location board, in order to facilitate the use of the single-agent solution both as a heuristic during search and as an endgame database [16,20,25]. We recently co-authored a study [18] looking at larger games, in particular the 81 location board with 6 pieces. But, nobody has studied the use of endgame databases in the full game with 81 locations and 10 pieces per player.

We recently built the single-agent solution [24] with 1.88 trillion positions, and requiring 500 GB of disk storage, far more than is found on typical machines. Furthermore, to save space, the data is stored modulo 15, using 4 bits per state. This means that additional work must be done to accurately recover the true distance of any given state. If there are errors in the recovery process, queries to the database may occasionally return incorrect values.

The goal of this paper is to study initial approaches for using this data to improve performance in Chinese Checkers. We use the data to create players based on minimax and MCTS search, comparing performance and uses of the data. We find that accuracy in retrieving values from the database is a primary concern, and that more work is needed to ensure accuracy of the database values. Preliminary experimental results suggest that minimax-based players perform better than MCTS-based players with using large endgame databases.

2 Background

In this section we cover a few important background details for this paper, including information about the game of Chinese Checkers and the algorithms that we will use to play the game.

2.1 Chinese Checkers

Chinese Checkers is a game played by 2–6 players on a star-shaped board shown in Fig. 1(a). In the two-player version of the game the players start with their pieces on the top and bottom of the board. The goal is for the players to get their pieces to the other side of the board. Legal movement is shown in Fig. 1(b). Pieces can either move in steps to one of the 6 adjacent locations, such as from location (c) to (d), or they can move in jumps, as shown by the move from (e) to (f). Jumps can only be taken when a piece is adjacent to another piece, and

the location opposite to the adjacent piece is free; jumps can be taken over any piece regardless of ownership. Jumps can also be chained together, so the piece at (e) can move directly to (g) by taking two jumps as part of the same turn. Note that a variation on these jumping rules is sometimes played where pieces can jump more than one space at a time across the board [28].

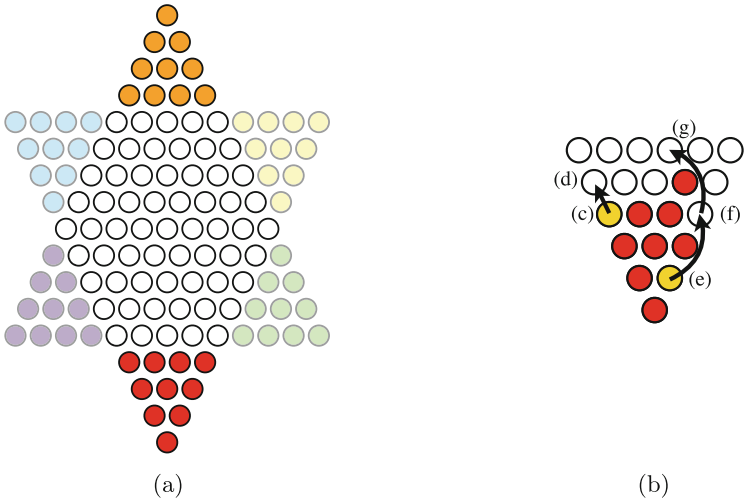


Fig. 1. Chinese Checkers board (a) and legal moves (b).

Most rules that we have seen for the game are ambiguous about some options for play, so we have introduced several conventions that simplify the game for computer play. First, we consider the game to be won by a player if a player's goal area is filled with pieces and at least one of them belongs to that player. This prevents a player from blocking their opponent's goal to prevent a win. Furthermore, if a state in the game is repeated, we consider this a loss by repetition for the player that moved into the repeated state.

Single Agent Chinese Checkers: The single-agent version of Chinese Checkers has $\frac{81!}{10!71!} = 1,878,392,407,320$ possible states. The Chinese Checkers board is symmetric around the x (horizontal) axis, so we can use the same single-agent solution for both players by flipping the board around the horizontally axis. The board is also symmetric around the y (vertical) axis, so the number of states which must be stored can be reduced. We use a quick, but slightly imperfect, scheme for computing symmetry: A state is not stored if (1) the first piece from the top of the board is on the right hand side of the board (excluding the center line) or (2) the first piece is on the center line of the board and the second piece is on the right hand side of the board. Using this scheme the number of stored states can be reduced to 1,072,763,999,648. (A perfect scheme would consider all pieces instead of just the first two; it would be slower to compute but use slightly less space.)

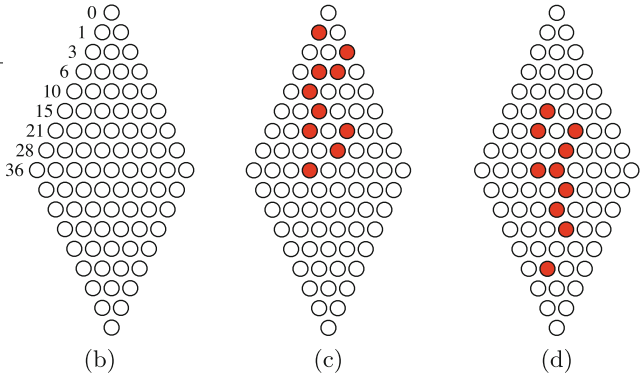
We use a write-minimizing breadth-first search [24] to generate and store the distance from each of these states to the goal state.[1] To save space and make the search more efficient we store the solution using only 4 bits per entry. This results in 16 values per entry, but one value is needed to mark unseen states during the search, so the resulting data stores the depth of each state modulo 15. As a result, states at depth 0 and 15 both have the same value in the database: 0. We discuss the consequences of this compression later in the paper.

In Fig. 2(a) we show, given the location of the first piece on the board, the size of the data needed to store the single-agent distances for all board configurations with a given first piece location. The locations are labeled in Fig. 2(b) - the top of the board is location 0, and the middle row starts with location 36.

The full data, with the first piece in position 0, requires 500 GB to store. The position in Fig. 2(c) has the first piece in position 1; if we only lookup positions with the first piece in location 1 or higher, we will only need 433 GB of storage. The board in Fig. 2(d) has the first piece in position 17. If we want to lookup all positions with the first piece in position 17 or higher, we need to store the data for positions 15 and higher, requiring 61.5 GB. This is due to the symmetry used for lookups: Our symmetry rules prevent us from storing positions where the first piece is in location 20. Instead, a symmetric lookup is used; flipping such a position around the y-axis moves this piece into location 15. Thus, we always store full rows of the single-agent data to ensure that symmetric lookups are always possible.

1st Piece	Size
0	500 GB
1	433 GB
3	337 GB
6	226 GB
10	129 GB
15	61.5 GB
21	22.8 GB
28	6.3 GB
36	1.1 GB

(a)



(b) (c) (d)

Fig. 2. The size of the single-agent data for various points on the board.

2.2 Minimax and Alpha-Beta Pruning

Minimax is the basic algorithm that was used to build expert-level programs for games like Chess [5] and Checkers [22]. It relies on an evaluation function that

[1] Due to symmetry either the start or the goal state can be used, although the choice influences whether subsets of data can be efficiently loaded.

maps states to numerical values which estimate the true value of a state. A large number of enhancements have been proposed to minimax search to improve performance [21].

We build a basic minimax program that is enhanced with iterative deepening, alpha-beta pruning, the history heuristic and transposition tables. The base evaluation function is just the distance of a player's pieces across the board, but we have also used TD(λ) [27] with linear regression to learn an evaluation function. A complete description of the details of our approach is outside the scope of this paper, but it is worth noting that this is not the first use of learning in Chinese Checkers, although it is the first use of TD(λ) that we could find in the literature. Samadi et al. report using learning in their program without further detail [19]. Hutton [11] trained genetic algorithms on training data for mid-game play.

2.3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is an alternate to minimax which has been used successfully in a broad range of other games such as Go [9], Hex [10], Hearts [26], and Amazons [14]. MCTS algorithms build an in-memory tree and then sample the leaves of the tree (e.g. with random playouts to the end of the game) as an alternate to evaluating states with an evaluation function. The tree grows with each playout, and is non-uniformly biased towards better parts of the search space. In many cases a good evaluation function for moves is needed, to ensure that playouts are reasonable and finite. Similarly with minimax, there are numerous enhancements that have been proposed for MCTS [4]. But, unlike minimax enhancements, it is still an open question of which enhancements are redundant relative to each other.

The primary enhancement we use here is forward pruning [14] as well as dynamic early termination [3]. We also use epsilon-greedy playouts [26] and enhanced playout policies to improve play.

We have experimented with slowing the growth of the MCTS tree [18], with initializing states with offline values [8], but current results are inconclusive. Other approaches that would be worth considering in the future include implicit minimax backups [13] and progressive bias [6].

3 Lossy Chinese Checkers Endgame Database

As mentioned previously, the single-agent data for Chinese Checkers is stored modulo 15, but the maximum distance between any two states in the single-agent Chinese Checkers state space is 33 moves [24]. The majority of positions (99.21 %) in the game are between 15 and 29 moves from the goal, but since every game terminates with a position at depth 0, a significant portion of positions seen in practice will be between 0 and 14 moves from the goal.

This creates the need for a classifier that will take a state and the stored depth as input and return the true distance to the goal as output. A simple

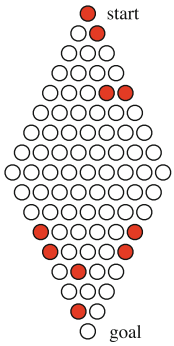


Fig. 3. Chinese Checkers board at depth 18 which must be classified.

lower bound on the number of moves to reach the goal can be computed by (1) the number of pieces that are not yet in the goal area and (2) the number of blank horizontal lines between the goal area and the farthest piece from the goal. This is related to observations made by Bell [2], modified for our purposes here. In particular, it is impossible for the furthest back piece on the board to skip rows when moving towards the goal in the single-agent game, since it either must step through each row between it and the goal, or it will jump over another piece en route to the goal.

Unfortunately, this estimate alone is not enough for a perfect predictor. Other factors such as non-adjacent pieces can be used to improve the classifier, but not with 100 % accuracy. We show a sample board that must be classified in Fig. 3. In this board position there are 8 pieces outside of the goal area. Furthermore, there are 8 empty rows between the goal and the furthest removed piece. Thus, no solution can exist which does not take at least 16 moves to reach the goal. The single-agent solution data says this position is at depth 3. Given the lower bound, the state is either 18 or 33 moves from the goal. In this case, the lower bound is nearly perfect. Our classifier currently mis-classifies this position, thinking that it will take more work to get the pieces split on either side of the board (just outside the goal) into the goal area.

To measure the accuracy of our predictor we sampled every 100,000 states (approximately) for the first 64.112 % of the full state space. We looked up the predicted depth of each state as well as the predicted depth of its neighbors to determine whether the prediction accuracy is correct. If a state and one or more of its neighbors differ by more than 1 step, the predictor must be incorrect. Overall, we looked at approximately 10 million states and their neighbors. When there is an error, on average 3.8 neighbors of the same parent show the error. Overall, 1 in 1800 parents has a child with a heuristic error. This is very small as a percentage, but in a search that expands millions of nodes, it is very likely that the search will encounter nodes with incorrect depth values.

4 Combining with Alpha-Beta Search

The first player we create is a traditional player using minimax with alpha-beta pruning [17]. This player is enhanced using iterative deepening, transposition tables and the history heuristic [21]. Moves are partially randomized to avoid identical play on repeated games. Only forward moves are allowed, as this significantly improves performance over considering all possible moves.

We implement two evaluation functions for this player. In the first each player just attempts to minimize the number of rows between their pieces and the goal. The second evaluation function is trained with self-play using linear regression and TD(λ). This evaluation function learns a weight for each location on the board for each player; it is strictly more expressive than the simple distance evaluation, since the distance evaluation function can be implemented by weighting each location according to its distance to the goal. This approach learns a total of 162 weights, one for each position on the board for each player.

We then integrate the single-agent data as an endgame database into this code base. It is important to note a key difference between endgame databases here and in traditional programs. Because we can use the single-agent data both as a heuristic and as an endgame database, we should not terminate the search immediately once a lookup is available. If the players' pieces are not adequately separated, the single-agent data is only a heuristic. This data gets more accurate the closer we get to the end of the game. Thus, using it at the end of a deeper search will improve performance. In practice, we don't have to modify our program to use the exact values when the game decomposes; it does not hurt the minimax search to lookahead further than the beginning of the endgame database, because the result of the minimax computation stays the same.

4.1 Experimental Results

We experiment primarily with the 1.1 GB database beginning at location 36 and the 61.5 GB database beginning at location 15. We have a 16-core 2.6 GHz AMD Opteron server with 64 GB of RAM, so this is the largest database we can load into memory. At leaves of the tree, if both players are able to use the database lookup, then difference in distances for each player is used as the evaluation function. If only one or neither player is able to use the database lookup, then both players use the regular evaluation function.

We present the results in Table 1. M indicates that the player is using the minimax algorithm to play. We use subscripts to indicate whether the database is being used as part of the evaluation function. The number used as the subscript is the first piece in the loaded database. Superscripts are used to indicate the evaluation function that is used. T for the trained evaluation function, and D for the distance-based evaluation function. So, the player labeled M_{36}^{T} is using minimax and a trained evaluation function in conjunction with the endgame database starting at location 36 on the board.

Each set of players is matched up 200 times - 100 times as the first player and 100 times as the second player. In experimental results we refer to players as Player 1 and Player 2 only for the simplicity of distinguishing player types and win rates, not for indicating which player went first. We record and report the win/loss percentage for each player. One player is better than another with 95 % confidence if it wins 114 games (57 %); 118 games (59 %) are needed for 99 % confidence. Players are allowed 1 s per move. While the exact search depth depends on the player, the average search depth is over 6 ply in practice. All further experiments use this same setup.

Table 1. Minimax win rates.

Player 1	Player 2	P1 Win %	P2 Win %
M^T	M^D	76.0 %	24.0 %
M^T_{36}	M^T	63.0 %	37.0 %
M^D_{36}	M^D	67.5 %	32.5 %
M^T_{36}	M^D_{36}	80.0 %	20.0 %
M^D_{15}	M^D	18.5 %	81.5 %
$M^D_{15}*$	M^D	52.5 %	47.5 %
$M^D_{15}*$	M^D_{36}	62.5 %	37.5 %

In the first half of the table we compare results using the database starting at piece location 36. The trained player soundly beats the regular distance player[2], both players are significantly stronger with the database than without, and the trained player beats the distance player by a significant margin when using the database. However, when the larger database is used (M^D_{15} vs M^D), the performance drops significantly. We hypothesized that this was the result of the errors in computing exact distances in the database: The games that are won by this player exhibit strong play. In games that are lost, however, the player with the database exhibits pathological play: the program moves to a seemingly random configuration of the board and then does its best to preserve that configuration. This is clearly a result of imperfect recovery of distances from the database.

To measure this, we implemented an alternate version of minimax that tracks the database values during play by looking the distance up from disk after every move by each player. Since the change in the optimal number of moves to the goal between a parent and child state (i.e. from applying a single action) cannot be large,[3] we use this to maintain more accurate distance estimates. The single-agent distance of the current board configuration for each player is updated incrementally through the game, starting at the optimal single-agent distance of 27.

[2] In our original implementation the algorithms were indistinguishable. Improving the efficiency of our TD learning, by taking advantage of the binary features, significantly improved the performance of the TD player.

[3] It can change by more than one because we may jump over an opponent's piece, which is not accounted for in the single-agent data.

We assume that the distance from two adjacent states to the goal will never increase by more than 5 or decrease by less than 10.

It is too expensive to perform a disk access for every node in the search tree, but once the start position for each player is within the in-memory database, these distances can also be incrementally maintained within the minimax tree during search. Thus, at the leaves of the tree, accurate distances can be used. The player that uses this improved distance estimation is in the last two line of Table 1 - M_{15}^{D}*. When played against a regular minimax player, this program is not significantly stronger than not using the databases. However, when played against a player using the small endgame databases, the program wins 62.5 % of games. Analysis of the games indicates that there are likely still small errors in the database distance lookups, as we have found positions where the program does not take clearly winning moves. But, this data point clearly indicates that errors in retrieving accurate endgame data are the reason for the poor play, since improving this accuracy significantly improves play.

We are working to build better classifiers that will improve the accuracy in estimating true distances from the database given the modulo distance.

5 Combining with MCTS

Next we integrate the endgame data into a MCTS approach. Our MCTS player is based on UCT [12]. We use the UCB1 rule to select the best node in the tree for exploration. Sampling outside of the tree is done using epsilon-greedy playouts [26]. Playouts are cutoff at depth 10, where a static evaluation function is used to evaluate the state [14]. (Experiments with different cutoff depths indicated that this depth produced robust performance across other parameter settings.) Like in our minimax program, we restrict the program to only taking forward moves. Our playouts are also biased – unless the epsilon-greedy rule takes precedence ($\epsilon = 0.05$), the largest forward move is always taken during each playout step.

We first began by replacing the heuristic evaluation at depth 10 with the value from the database, assuming that both players' lookups were available in the database. Although this basic approach seemed like it should improve performance, we needed to make a few changes to make this successful. First, we immediately cut off a playout once both players' pieces have separated so that a perfect evaluation is possible. Second, we use the database for the evaluation function if it is available for both players, otherwise we use the basic heuristic evaluation. Third, we must be more careful to account for the player to move when performing evaluations, since the player to move at the leaves of the tree is non-uniform. If there is a tie in distances to the goal, the player with the first move will win.

This first rule is particular important. Without it, our program plays poorly despite a perfect evaluation function. It appears that this occurs primarily because the random playouts corrupt the perfect leaf values and because it takes some time for the MCTS tree to converge its playout values to the perfect values after every move is sampled. While the need to immediately use the perfect

values once they are available seems obvious, this is not needed in the minimax program. Similarly, because all leaf nodes are at the same depth, there is no need to account for the player to move in the evaluation function of a minimax player, since the bonus would be added equally to all nodes.

Besides the approach just described, there are several other ways to use the endgame databases in MCTS: Similar to our previous analysis [18], performing random playouts until both players separate to get a more accurate evaluation function did not work well in practice. Also, using the data directly from disk is too slow and results in very poor performance, even though we have the data on a SSD for fast access. We experimented with other playout depths, and distances in the range 10–20 provided the best performance. Slowing the growth of the MCTS tree by growing the tree more slowly as done in our previous work [18] did not have a significant impact on performance.

5.1 Experimental Results

We now compare the performance of MCTS (UCT) implementations using the single-agent database as an evaluation function. Results are in Table 2. U designates a player using UCT. As before, the subscript indicates the first piece in the endgame database that is used by the player during play. In this experiment all players use the distance-based evaluation as the base evaluation function. As before, 1 s is allowed for each move.

Table 2. MCTS/UCT win rates.

Player 1	Player 2	P1 Win %	P2 Win %
U^D_{36}	U^D	49.0 %	51.0 %
U^D_{28}	U^D_{36}	57.0 %	43.0 %
U^D_{15}	U^D	74.5 %	25.5 %
U^D_{15}	U^D_{36}	63.5 %	36.5 %

In the first line we see that adding the endgame database starting at position 36 does not significantly improve or degrade performance over the basic player. But, using the endgame database starting at piece 28 wins over the endgame database starting at piece 36. Similarly, using the endgame database starting at 15 wins significantly over just using the distance evaluation and over the database starting at piece 36. We did not compare the database starting at location 15 with the database starting at location 28 due to memory constraints, although sharing the database between players would allow this comparison.

Our results suggest that MCTS does a better job than minimax of tolerating errors in the single-agent data, as performance improves as we use the larger databases, even without the correction procedures needed for minimax. However, we need to be more careful about when and how we use the endgame data in MCTS.

6 Comparing Minimax and MCTS

Given that we have successfully improved the performance of our two independent approaches using endgame databases, we now compare the performance across techniques. We begin with a baseline comparison without using endgame databases, shown in the top two lines of Table 3.

Table 3. Minimax versus UCT.

Player 1	Player 2	P1 Win %	P2 Win %
U^D	M^D	77.5 %	22.5 %
U^D	M^T	63.0 %	37.0 %
M^D_{36}	U^D_{36}	60.5 %	39.5 %
M^D_{36}	U^D_{15}	90.5 %	9.5 %

Without the endgame databases, UCT beats both the regular and trained minimax player by a significant margin. But, as shown in the bottom half of the table, when minimax is given the database, it outperforms UCT by a significant margin.

Furthermore, the UCT player with the database starting at location 15 does worse than the player with the database at location 36 against minimax. This result suggests that the UCT player does well against other UCT players, but does not necessarily do well against other player types. We expect that improving the accuracy of the UCT endgame estimates could further improve performance.

But, we also note that one of the strengths of UCT is that it gets long-term strategic information about a game from its playouts. Adding endgame databases seems to duplicate this strength. Minimax, on the other hand, does well in local tactics because of the full search, but misses out on longer-term strategies. Thus, endgame databases seem to have the potential to complement the performance of minimax more than UCT.

Further experiments and implementations will be needed to understand this difference in performance more deeply and to clearly isolate the factors that influence performance with endgame databases.

We note that these results are different than in our preliminary study, where minimax performed worse that UCT [18]. There are several notable differences between this work. First, our minimax implementation here is much stronger than in previous work. Second, we are using more pieces on the board, where there is more congestion. Third, we are using better time controls for our experiments, using only time instead of node counts. Finally, we have error in our database lookups in this work, while our previous work had no error.

7 Conclusions and Future Work

This paper describes the first work in using large endgame databases in Chinese Checkers. Several challenges are faced in how to properly integrate the endgame

databases and how to correct errors that occur because of the compression that is used when storing the endgame data. Currently, minimax-based approaches are able to do a better job of using the endgame data than MCTS approaches, but with further study this could still change.

The addition of opening books and further search enhancements could improve either player, particularly since the programs do not take the moves that advance their pieces most quickly across the board at the beginning of the game. (The endgame database can be used for this purpose as an opening book as well.)

Improving our classifier is also an important step to improving performance. Currently our classifier uses relatively simple rules to estimate the true distance to the goal given the modulo distance. Using linear or logistic regression to train a classifier could result in better performance. We should also be able to enhance our UCT player to improve its own estimates during playouts.

Another important step is to measure the robustness of both minimax and MCTS to errors in the evaluation function to see which approach is more tolerant to the type of errors that occur in our evaluation function.

Finally, we need to understand more deeply why our UCT player is able to beat other UCT players by a significant margin, but not minimax players. This is related to the quality of our classifier, but also to a fundamental understanding of the strengths of each approach.

Acknowledgments. This paper benefited from research by a summer student, Evan Boucher, who worked on the problem of determining the true distance of a state from the goal given the modulo distance.

References

1. 8 longest 7-man checkmates. http://tb7.chessok.com/articles/Top8DTM_eng. Accessed 11 May 2015
2. Bell, G.I.: The shortest game of Chinese Checkers and related problems. CoRR abs/0803.1245 (2008). http://arxiv.org/abs/0803.1245
3. Bouzy, B.: Old-fashioned computer Go vs Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence in Games (CIG) (2007). Invited Tutorial
4. Browne, C., Powley, E.J., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
5. Campbell, M., Hoane Jr., A.J., Hsu, F.: Deep blue. Artif. Intell. **134**(1–2), 57–83 (2002)
6. Chaslot, G.M.J.B., Winands, M.H.M., van den Herik, H.J., Uiterwijk, J.W.H.M., Bouzy, B.: Progressive strategies for Monte-Carlo tree search. New Math. Nat. Comput. **4**(3), 343–357 (2008)
7. Fang, H., Hsu, T., Hsu, S.-C.: Construction of Chinese chess endgame databases by retrograde analysis. In: Marsland, T., Frank, I. (eds.) CG 2001. LNCS, vol. 2063, pp. 96–114. Springer, Heidelberg (2002)
8. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Ghahramani, Z. (ed.) Machine Learning. ACM International Conference Proceeding Series, vol. 227, pp. 273–280. ACM, New York (2007)

9. Gelly, S., Silver, D.: Achieving master level play in 9 x 9 computer go. In: Fox, D., Gomes, C.P. (eds.) AAAI, pp. 1537–1540. AAAI Press, Menlo Park (2008)
10. Huang, S.-C., Arneson, B., Hayward, R.B., Müller, M., Pawlewicz, J.: MoHex 2.0: a pattern-based MCTS hex player. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2013. LNCS, vol. 8427, pp. 60–71. Springer, Heidelberg (2014)
11. Hutton, A.: Developing Computer Opponents for Chinese Checkers. Master's thesis, University of Glasgow (2001)
12. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
13. Lanctot, M., Winands, M.H.M., Pepels, T., Sturtevant, N.R.: Monte Carlo tree search with heuristic evaluations using implicit minimax backups. In: 2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, 26–29 August 2014, pp. 341–348. IEEE (2014)
14. Lorentz, R.J.: Amazons discover Monte-Carlo. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 13–24. Springer, Heidelberg (2008)
15. Nalimov, E., Haworth, G.M., Heinz, E.A.: Space-efficient indexing of endgame tables for chess. ICGA J. **23**(3), 148–162 (2000)
16. Nijssen, J.P.A.M., Winands, M.H.M.: Enhancements for multi-player Monte-Carlo tree search. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 238–249. Springer, Heidelberg (2011)
17. Pearl, J.: The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. Commun. ACM **25**(8), 559–564 (1982)
18. Roschke, M., Sturtevant, N.R.: UCT enhancements in Chinese Checkers using an endgame database. In: Cazenave, T., Winands, M.H.M., Iida, H. (eds.) Computer Games (CGW 2013). CCIS, vol. 408, pp. 57–70. Springer International Publishing, Switzerland (2014)
19. Samadi, M., Asr, F.T., Schaeffer, J., Azimifar, Z.: Extending the applicability of pattern and endgame databases. IEEE Trans. Comput. Intell. AI Games **1**(1), 28–38 (2009)
20. Samadi, M., Schaeffer, J., Asr, F.T., Samar, M., Azimifar, Z.: Using abstraction in two-player games. In: ECAI, pp. 545–549 (2008)
21. Schaeffer, J.: The history heuristic and alpha-beta search enhancements in practice. IEEE Trans. Pattern Anal. Mach. Intell. **11**(11), 1203–1212 (1989)
22. Schaeffer, J.: One Jump Ahead - Challenging Human Supremacy in Checkers. Springer, New York (1997)
23. Schaeffer, J., Björnsson, Y., Burch, N., Lake, R., Lu, P., Sutphen, S.: Building the checkers 10-piece endgame databases. Adv. Comput. Games **10**, 193–210 (2003)
24. Sturtevant, N.R., Rutherford, M.J.: Minimizing writes in parallel external memory search. In: International Joint Conference on Artificial Intelligence (IJCAI) (2013)
25. Sturtevant, N.R.: A comparison of algorithms for multi-player games. In: Schaeffer, J., Müller, M., Björnsson, Y. (eds.) CG 2002. LNCS, vol. 2883, pp. 108–122. Springer, Heidelberg (2003)
26. Sturtevant, N.R.: An analysis of UCT in multi-player games. ICGA J. **31**(4), 195–208 (2008)
27. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
28. Tong, K.B.: Intelligent Strategy for Two-person Non-random Perfect Information Zero-sum Game. Master's thesis, Chinese University of Hong Kong (2003)