UCT Enhancements in Chinese Checkers Using an Endgame Database

Max Roschke and Nathan R. Sturtevant

Department of Computer Science
University of Denver
Denver, CO, USA

max.roschke@gmail.com sturtevant@cs.du.edu

Abstract. The UCT algorithm has gained popularity for use in AI for games, especially in board games. This paper assess the performance of UCT-based AIs and the effectiveness of augmenting them with a lookup table containing evaluations of games states in the game of Chinese Checkers. Our lookup table is only guaranteed to be correct during the endgame, but serves as an accurate heuristic throughout the game. Experiments show that using the lookup table only for its endgames is harmful, while using it for its heuristic values improves the quality of play. This work is performed on a board with 81 locations and 6 pieces, which is larger than previous work on lookup tables in Chinese Checkers. It is a precursor to using the 500GB full-game single-agent data on the full-size board with 81 locations and 10 pieces.

1 Introduction

The UCT algorithm has recently proven to be a powerful tool for running simulations. Similar algorithms have been used to write powerful computer players for Go, a game which had long resisted other tactics. Its strength comes in part from its reliance on simulations, which approximate paths to the end of the game. After simulating to the end of the game, states are easily evaluated as a win or loss, so there is no explicit need for evaluation functions or expert knowledge. Even without these requirements, there are still many modifications that try to improve upon this model. A recent survey [2] lists over thirty potential enhancements across multiple domains, each with varying degrees of success.

Herein, we will test the effects of two enhancements to UCT in Chinese Checkers, each relying on a precomputed lookup table, which contains the distance of a single player's pieces to the goal state [15]. Towards the end of the game, this lookup table is effectively an endgame database and can be used to determine the winner and loser. It has already been demonstrated that opening books can improve the play of the UCT algorithm [1,3], so it seems reasonable to expect that endgame databases can also improve play. By knowing the endgame states, the playout length may be reduced, allowing for more playouts to run. Also, standard UCT playout policies may not follow the perfect endgame strategies. Thus, standard UCT may accept playouts where players make mistakes in

the endgame. Using the lookup table to determine the exact value should eliminate this possibility, making the simulations more accurate. Thus, we propose running simulations only until an endgame state is reached, rather than playing them out to completion.

Even during the mid-game, the lookup table may be used as an evaluation function. The information is not necessarily accurate, as it ignores the positions of the opponents pieces, but it may still favor advantageous board positions. These kinds of lookup tables have been used before in Chinese Checkers and have been found to produce a viable evaluation function [10, 11, 13]. The evaluation function becomes more accurate as the game progresses, as it eventually becomes an endgame database. This heuristic will be compared to a more common evaluation function that uses the average position of the pieces on the board. UCT will also be combined with the lookup table heuristic, to determine whether UCT estimations are a viable tactic for Chinese Checkers.

2 Background

2.1 Minimax Algorithm

The Minimax algorithm is a commonly-used technique for exploring the game tree of a two-player game. It creates a game tree of a certain depth, and then scores each leaf-state using an evaluation function. The evaluations are propagated up the tree with the player choosing the maximum value at his own nodes, and the opponent choosing the minimum value at its nodes. This explores all paths of that depth, and returns the best path for the player.

While simple, this algorithm provides a Nash equilibrium solution, allowing for a player to maximize his score for the given evaluation function. However, there are several drawbacks to this approach.

First, it has an exponential runtime. When searching a game with branching factor b to a depth of d, it has a runtime of $O(b^d)$. Using $\alpha\beta$-pruning, this can be reduced to $O(b^{d/2})$, but this is still gives a constraint on how deep the tree can be searched, especially in games with a large branching factor. There are other methods that attempt to improve on $\alpha\beta$-pruning, but we do not examine those here. Additionally, this approach does not scale well to multiplayer games. The $\max^n$ algorithm, the multiplayer equivalent, can only be pruned to $O(b^{(n-1)d/n})$ for n players [14].

Second, the result of the algorithm is only as accurate as the evaluation function itself. Evaluation functions are often inaccurate, as the middle stages of the game are ambiguous and difficult to rate. Any error in the evaluation function will become apparent in the results of the search.

2.2 UCT Algorithm

The UCT algorithm [5] relies on simulations to gather information about the game tree. It maintains a partial game tree in memory, and traverses it in four distinct stages: selection, expansion, simulation, and propagation.

First, an action is selected from the tree. At each node in the tree, an action is selected to maximize its UCB1 score, which is given by:

$$\overline{x_i} + C\sqrt{\frac{\ln(T)}{T_i}}$$

where $\overline{x_i}$ is the average payoff of an action, C is the exploration constant, T is the number of times the parent of the action has been played, and T_i is the number of times the action has been taken. The second part of the equation determines whether states are explored or exploited. The higher this second term, the less the score of the action depends on its average payoff, and the more it depends on the number of samples. A high C value encourages more exploration. This process is repeated until a leaf node of the tree is reached.

The leaf node may be expanded to add a new node to the tree, depending on the expansion policy of the tree. A common change is to only expand a node after it has been sampled a certain number of times [4]. This prevents the tree from growing rapidly, and tends to expand only nodes which have received better scores.

From this new node, a simulation is run to the end of the game. The simulation may be guided by a playout policy. In many games a random policy is acceptable, but in other games it is not. In fact, for some games, random playouts are completely unfeasible. For example, in Chinese Checkers, pieces may move both forwards and backwards, which means that the games may be infinite in length. Random playouts then would be too long and unrealisitic to provide useful data about the game. For this reason, playout strategies are often imposed, for example, taking only forward moves and ignoring backward moves. If the playout policy matches usual player strategies, then this covers realistic play, and the simulations become more useful.

Finally, a win or loss is observed at the terminal game state, and that value is propagated up the tree. More information may be calculated at this terminal state to give an evaluation as well. For example, to emphasize shorter games, one may include the length of the simulation in its score. This value is then added to the total payout of each node on the path up the tree, updating its UCB1 score.

Many of these simulations are run, and then the best node is chosen from the tree. What constitutes the best node may be difficult to determine, as it should take into account both the sample rate and the average payout of that node.

Over time the game tree created by this algorithm approaches the actual minimax game tree which evaluates states based on wins, losses, or ties. When given infinite time and space to work with, it will eventually converge to that value [5]. However, this relies on all possible paths being traversed. Modifications to the tree, such as changing the playout and expansion policies, that cause some paths to be omitted from the tree eliminate this guarantee from a theoretical perspective.

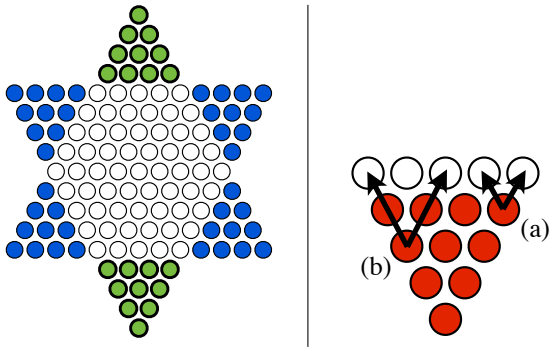

Fig. 1. A Chinese Checkers board.

2.3 Chinese Checkers

Chinese checkers is a board game that can be played by two, three, four, or six players on an isometric board. Each player tries to move his pieces across the board, from his starting corner to the opposite corner. A player wins once all of his pieces have crossed the board, and are in the same configuration in which they started. The board is arranged into a hexagram when there are more than two players, and is simply a diamond when there are only two players. The piece counts may also vary; in smaller games, players have six pieces each, and in larger games, players have ten pieces each.

Pieces may move in two ways. They may move into adjacent empty spaces, or they may jump over an adjacent piece to a free space immediately opposite its original position (see Figure 1 (right)). These jumps may be chained arbitrarily, so a piece may even move most of the way across the board on a single turn.

There is some ambiguity in the end of the game, as one player may leave pieces behind, preventing the opponent from filling that corner. However, to account for this case, we define a player to win once they have at least one piece across the board, and the tiles of that corner are all filled. Thus, if a player leaves pieces behind and the opponent fills all other spaces in that corner, the opponent wins. In this version of the game there can be no ties, as one player must reach the opposing side first. Games may also proceed indefinitely, as there is no restriction on the direction pieces may move. For simplicity, our experiments will be run on a nine-by-nine board with 81 possible locations for pieces – the same size as in the Figure 1, but without the four blue corners – and each player will have only six pieces.

2.4 Endgame Databases

Endgame databases have proven effective in many other games. They contain all the necessary information to complete a game given a certain state. Chess programs make extensive use of these databases to improve play and many resources have been devoted to analyzing and generating these databases (see [16]).

To date, all five-piece endgames have been calculated, giving the computer an immense amount of knowledge and sparing much computational time. These playouts can be especially complex, as many of them must take the fifty-move rule into account. While many databases do not specifically acknowledge the fifty-move rule in their returned solutions, the evaluation still gives the computer a glimpse of the probabilities of winning, losing, and drawing without doing any explicit calculation.

Endgame databases were also heavily used in the Checkers AI Chinook [12]. Move databases were extended to include all ending positions containing ten pieces or less, considerably increasing the accuracy of play.

Endgame databases have also been used before in Chinese Checkers and have been shown to improve the quality of play [10]. This has also been shown to aid UCT playouts as well [7–9], although their experiments were performed with smaller lookup tables than we will be using here.

3 Lookup Table

The lookup table is a simple table containing entries for all of the positions for a single player on the board [15]. Our experiments are run on a nine-by-nine board with each player having six pieces. This board has 81 positions, and is the board seen in Figure 1 without the four blue corners. On this smaller board, there is an entry for each of the $\binom{81}{6} = 324,540,216$ positions. Each entry contains the minimum number of moves it would take that player to move all his pieces to the home area. This kind of database has been used before [7–9], although it was done on a seven-by-seven board, which contains much fewer positions.

Since this table only takes into account one player's pieces, it is not accurate when the players' pieces may interact. Opponent pieces may block the shortest path making the table value too low, or jumps offered by opponent pieces may offer a quicker path, making the table value too high. Table values are completely accurate once the pieces are separated. When there is no interference, the correct path is known, and players may complete the game with perfect play guided by lookups alone, so the lookup table may function as an endgame database.

While the lookup table is quite large, most of its entries are unused during the course of a normal game, so only a fraction of it needs to be loaded into memory. The entries are sorted by the furthest piece from the home area. All of the remaining pieces must be distributed after this starting piece, so there are $\binom{80-n}{5}$ positions for starting position n in general. As this back piece moves across the board, the number of potential configurations drops substantially (see Table 1). In most cases, this piece is not going to stay in the home area for long, as the pieces move quickest when kept in a group. This means that earlier, larger sections of the table can be omitted from memory, greatly reducing the memory requirements of the lookup table.

This is not a problem with the smaller data set, as the six-piece data can easily fit into memory. The ten-piece data set is much larger however, and cannot

Table 1. Lookup Table Size versus Starting Position

	Six Pieces		Ten Pieces	
Start Position	# of Positions	% of Positions	# of Positions	% of Positions
1	324,540,216	100.000	1,878,392,407,320	100.000
5	237,093,780	73.055	1,096,993,404,430	58.401
10	156,238,908	48.142	536,211,932,256	28.546
15	99,795,696	30.750	247,994,680,648	13.202
20	61,474,519	18.942	107,518,933,731	5.724
25	36,288,252	11.181	43,183,019,880	2.299
30	20,358,520	6.273	15,820,024,220	0.842
35	10,737,573	3.309	5,178,066,751	0.276
40	5,245,786	1.616	1,471,442,973	0.078

easily be fit into memory. We have not run experiments on this larger data set yet, but some omission of data will be necessary to make its usage realistic.

4 Proposed Experiments

The experiments will be carried out on a nine-by-nine two player board. Each player will each have six pieces. This smaller board will lessen complexity and enable more trials to be run, but it is larger than boards used in previous experiments [7], so it will offer more information about the effects of board size on the table's effectiveness. Using this board configuration, we shall test two varieties of player. There will be two players using the $\alpha\beta$-pruning approach, and there will be three variations of the UCT algorithm. Each AI will be given a limit of 100,000 node expansions per move, allowing them equal access to resources.

4.1 $\alpha\beta$-Players

Each $\alpha\beta$ player will search to a depth of five ply using only non-backward moves. Backward moves are used much less often than other moves, and, in practice, led to a player that tried to block the opponent instead of moving his own pieces across the board. Removing the backwards moves also reduces the branching factor, leading to quicker play. These settings reached a balance of depth and speed, usually having the player respond in less than one second.

There will be two potential evaluation functions. The first uses the lookup table as a heuristic. The evaluation of a state will be the difference of the two players' values in the table. This will lead a player to block good opponent moves, while trying to move across the board as quickly as possible. The other function will be the difference in average position of the two players. Each player will check how many rows away from the home area their pieces are. This gives an approximation of how close the group is to winning. Again, using the difference should lead players to try and block each other, while moving quickly across the board, if possible.

Table 2. UCT Parameters

Tree Policy	Three variations: Nodes in a tree may contain all moves, all non-backward moves, or all forward moves.
Expansion Policy	Nodes will only added to the tree once they have been visited a minimum number of times.
Playout Policy	Only forward moves are expanded for all playouts to reduce their length and approximate a reasonable strategy.
Random Move %	Some percentage of moves during the playouts are random, while the remaining moves are the farthest move – that which advances a piece the furthest number of rows.
Weight of Playout Length	A small addition to the score which favors shorter wins or longer losses.
C Constant	The exploration versus exploitation value of the tree.
Evaluation	Three variations: a win or loss value is returned once a simulation is completed, a win or loss value is returned once a simulation has reached a known endgame state, or an evaluation is returned once a simulation has run a set length.

The lookup table may have an advantage over the simpler function, as it takes into account jumps and other factors that contribute to a piece's distance from the home area. Whether or not this information is truly more useful remains to be seen. These players will serve as a baseline, as their results do not depend on simulations. Without random simulations, they are guaranteed to give one solution for a given game state, so its results are less varied.

4.2 UCT Players

Basic UCT These players will be constructed with the standard UCT approach without any external knowledge. These players run simulations all the way to the end of the game, and then return a corresponding win or loss value. There are five variables which were tuned for this type of UCT player.

First, the expansion policy was varied. Nodes could only be placed into the tree after a certain number of samples had been run. There were three additional policies on the type of node that could be added to the tree. One policy added all nodes, another added only non-backward nodes, and the last added only forward moves. Experimentally, expanding only forward nodes gave the best results. Nodes were also not added to the tree until they had been sampled a minimum number of times.

Second, the playout policy was varied. All of these players only chose forward moves. However, there is much variation in the forward moves, so another variable was added. A set percentage of the time, the farthest move – one which took a piece the most rows toward its home state – was taken, while the remainder of moves was selected randomly. This attempts to model quick movement strategies, while allowing enough variation to give many sample points.

Third, the return value was augmented with the length of the simulation. Wins and losses accounted for most of the payout, but a small amount was varied based on the simulation length. Quick wins received a better evaluation than long wins, and long losses received a better evaluation than short losses. This steered the playouts to a better options, as quicker wins and longer losses are easier to exploit.

Finally, the C constant was tuned. This takes into account all of the other modifications and found the right amount of exploration versus exploitation for that combination of variables.

Tuning was done using a hill-climbing approach. Three players were created, each with a different category of playout policy, and the other four variables were tuned using the following procedure. All variables initially received a default value. A variable was selected and varied over a range of values near its current value. Six players, identical except for that variable, were created and all assume new values within that range. These six new players were pitted against the original (an equal number of times as first and second player), and the AI with the most victories was chosen as the new AI. The original remained if none of the new players defeated it more than fifty percent of the time. A single pass of tuning did this for each variable. Three passes were run on each policy, shrinking the range each time. The variables depend on each other, and each player was only tuned against variations of itself, so these tunings are in no way guaranteed to be optimal.

These tuned players gained the following configurations (some tuned parameters are omitted):

Name	Playout Policy	Random Move %	Expand Threshold
base_0	Forward	17	18
base_1	Non-backward	14	18
base_2	All	22	20

Heuristic UCT Player This version of the UCT player runs simulations to a fixed depth, and then evaluates them using the difference in distance, just like the $\alpha\beta$ lookup player. It was tuned using the same variables as the general UCT player – playout policy, random move percentage, expand threshold, and C constant– as well as two more: weight of the difference function and the playout depth. The difference function weight is just the linear weight of the difference function, and the playout depth is the number of moves playouts are run before

the difference function is applied. All numeric constants were tuned with the same process used for the general UCT player.

Since this is the same evaluation function as the lookup-based $\alpha\beta$ player, this method will approximate the game tree value at that depth. This AI will evaluate the effectiveness of estimating that tree's value.

The technique of stopping a UCT simulation early has been used before in the game of Amazons [6]. Their AI also did the best after a certain number of random moves had been played – the same tactic that we shall use here.

The tuned players gained the following configurations:

Name	Playout Policy	Random %	Expand Threshold	Playout Depth
heur_0	Forward	43	32	13
heur_1	Non-backward	55	20	13
heur_2	All	32	27	14

Endgame UCT Player This UCT Player runs simulations until the players are separated. Separation is determined using the centerline of the board as a divider. Once the player and their opponent's pieces are on opposite sides of the centerline, the pieces are considered separated. This is not entirely accurate, as it is still possible for a piece to jump to the centerline and then interact with opposing pieces. However, the likelihood of this is small, and it is even less likely that this would benefit or hinder either player.

Once separated, the winner is declared based on each player's distance from his respective goal. All other variables remained the same as the general UCT player.

Name	Moves in Tree	Random %	Expand Threshold
end_0	Forward	17	18
end_1	Non-backward	6	2
end_2	All	21	23

5 Experiment Results and Analysis

5.1 Depth-Based Trials

Table 3 shows the results of the best players in each category. While three versions were created in each category, these were the strongest. The player on the left played as first player, while the player on top played as second player. The percentage shown is the winning percentage of the first player. All results are out of 100 trials, except for the $\alpha\beta$ results, which are not listed, as there is no randomness in their algorithms, hence, no variation. The UCT algorithms were

Table 3. Results for the Best Players Using 100 trials

Player One Wins (First Player Left, Second Player Top)					
	$\alpha\beta$	$\alpha\beta$-Lookup	base_0	heur_0	end_0
$\alpha\beta$	–	–	54.0%	18.0%	67.0%
$\alpha\beta$-Lookup	–	–	75.0%	36.0%	88.0%
base_0	83.0%	65.0%	56.0%	5.0 %	62.0%
heur_0	96.0%	87.0%	96.0%	67.0%	97.0%
end_0	72.0%	27.0%	43.0%	5.0 %	49.0%

given 100,000 node expansions per move. The best tree expansion policy turned out to use only forward moves. In practice, when the players were allowed to use all moves, they tended to play overly defensively, and attempted to block the opponent more than they tried to cross the board. This became especially true when they started to lose, as the evaluation function gave better results for blocking the opponent's advances.

There appears to be a slight bias towards the first player. The general trials showed advantages of approximately 5% more wins for the first player.

The general UCT player tends to do better than the average-based $\alpha\beta$ player, taking an average amount of second player wins (after accounting for the first player advantage), and winning many more first player rounds. It fails to beat the $\alpha\beta$ player that uses the lookup table, however, indicating that the lookup table serves as a decent evaluation function throughout the game.

Further, the lookup table heuristic appears much more accurate than the average distance metric based on the relative performances of the $\alpha\beta$ players. The $\alpha\beta$ player using the lookup table won an additional 18% of the rounds as first player than the $\alpha\beta$ player without that data. It was also able to better defend itself as second player, winning at least 9% more of the overall rounds.

Of the two enhancements to UCT, using the lookup table as a heuristic appears much more effective than using it solely to calculate endgames. While the additional of heuristic values resulted in a stronger player, the addition of endgames lowered the quality of play.

5.2 Sample Based Trials

For this experiment (see Table 4), players were only allowed to complete a certain number of playouts before they made a move. This would remove the benefits of shorter playouts of the heuristic player and the endgame player, as there are a limited number of playouts regardless of depth.

Versus the plain UCT player, the heuristic player's performance suffered. As second player, it lost several times the games that it did in the node expansion experiment. It also did slightly worse as the first player. However, these results

Table 4. Trials with Limited Number of Playouts (200 trials)

Player One Win Rates						
		Number of Playouts per Turn				
p1	p2	1,000	2,000	4,000	8,000	16,000
$base_0$	$heur_0$	13.0%	12.5%	12.5%	17.0%	11.5%
$heur_0$	$base_0$	88.5%	92.5%	94.5%	90.5%	94.0%
$base_0$	end_3	63.0%	61.5%	74.0%	70.0%	80.5%
end_3	$base_0$	49.5%	46.0%	40.5%	37.0%	27.5%
$heur_0$	end_3	97.0%	97.5%	100.0%	98.5%	99.5%
end_3	$heur_0$	6.5 %	4.0 %	3.0 %	1.5%	1.5%

do not seem to vary consistently with the number of playouts. Increasing the number of playouts allowed per turn did not give an advantage to either player.

The plain UCT Player benefitted from more playouts when put against the endgame player. As the number of playouts per turn increased, so did the wins of the plain UCT player as both first and second player.

The heuristic player also did better against the endgame player when allowed more playouts per turn. As first player, the heuristic model won almost all of the time, and as second player it won almost all matches when allowed at least 4,000 playouts per turn.

Overall, both the heuristic and the endgame players suffered some performance penalties. The basic UCT player improved its performance as both the first and second player compared to the two models that shortened playouts. However, the heuristic model remained as the clear winner between these three strategies.

5.3 Time Based Trials

Next, we considered giving an equal amount of time to each player. This would show more of the strengths and weaknesses of each approach. The plain UCT player has no cost for lookups, and never needs to check if a state is in memory, but also then must play each playout game to completion. The heuristic player only plays to a static depth before looking up the end state in memory, which should take little time overall. The endgame player must play out a game until each player is in a state that is in memory. This likely will take the longest time, as it has to check its states most often. This will then vary the number of playouts each player can run in a turn, giving an advantage to the faster players.

The heuristic player remained strong versus the plain UCT player under time trials, while the endgame player did not. The heuristic player won most of its

Table 5. Trials with Limited Time per Turn

Player One Win Rates					
		Time per Turn			
p1	p2	1 sec	2 sec	4 sec	8 sec
$base_0$	$heur_0$	10.6%	8.0%	7.0%	9.0%
$heur_0$	$base_0$	95.0%	93.5%	97.0%	97.0%
$base_0$	end_3	55.6%	62.5%	56.5%	59.0%
end_3	$base_0$	51.9%	49.0%	57.5%	54.0%
$heur_0$	end_3	98.8%	99.0%	96.0%	94.0%
end_3	$heur_0$	8.1%	4.0%	1.5%	3.0%

1 second results from 160 trials
2 and 4 second results from 200 trials
8 second results from 100 trials.

matches, only once having its win rate drop below 90%. This did not seem to vary on the overall time, as the rates remained close as time increased per turn.

The endgame player was able to match the strength of the plain UCT player almost evenly. Each player usually won between 50 and 60 percent of matches when playing as first player. This matches previous indications of a first player bias, so these players seem evenly matched when given equal amounts of time to think.

The endgame player did not play well versus the heuristic player and lost most of the matches it played. This seemed to vary little with time. Although it did the best during the 1 second trials as first player, this could be due to the granularity of the timer used. It also seemed to fare better as the time was increased as second player, as it was able to win six percent of its matches with more time.

When given equal amounts of time, the plain UCT was able to perform 1.5 times as many node expansions as the heuristic player, and twice as many as the endgame player. When playing each other, the endgame and heuristic players expanded roughly the same number of nodes.

6 Conclusions and Further Work

There are several conclusions to be drawn from these results. First, it would seem that the lookup table serves as a good heuristic throughout the game. Not only was the $\alpha\beta$ search able to challenge the UCT players effectively, but the UCT player using this heuristic was able to win more than 85% of its first player games, and more than 60% of its second player games. This leads to the second point, that using the UCT algorithm to approximate the game tree at a certain

depth gives useful results. Since both the UCT algorithm and this $\alpha\beta$ were using the same heuristic, it would seem that the approximation of the heuristic at a depth of fourteen proved more useful than the exact value of that heuristic at depth five.

While the endgame databases caused performance to suffer here, it should not be taken as a general trend. It outperformed the average-based $\alpha\beta$ player, but fell short of defeating the lookup-based $\alpha\beta$ player. This is only based on 100,000 node expansions, however, which is less than one second of calculation per move. Additionally, the tuning procedure was not guaranteed to be optimal. Given more resources, the performance of the endgame player will likely improve.

In general, the heuristic player was shown to give the best results in both time-dependent and time-independent trials. It would seem that the lookup table provides useful, general information about the state of the game. In terms of the larger lookup table, however, this may not be true. In this case, with a lookup table that can completely fit into memory, it is feasible to lookup any state of the board. This strategy requires that option, for it always plays a fixed number of moves ahead. This is less easily accomplished with a larger table, so the performance of this player will likely suffer the most from increases to the lookup table's size.

More work can be done on expanding these results. These results can be scaled up by giving the players more time to make each move. This will give the algorithms more time to converge to a good move, and it will show the cost of each approach. Time trials show that the plain UCT player is much faster than either the heuristic player or the endgame player, and this advantage will only become greater as the size of the lookup table increases. These trials still contain reasonably small tables (approximately 350 Mb at largest). Once the tables can no longer fit into memory, these lookups will be even longer.

The lookup table size will also be scaled up. We have the ten-piece data set, which presents its own challenges. While these players were able to freely query the lookup table for any state, players querying the ten-piece data set will need to confront the massive size of that data set. The effects of loading only portions of the database will be examined, as well as the effects on time. A larger lookup table means that the queries will likely take longer, further reducing the number of trials these players will be able to run.

References

1. Pierre Audouard, Guillaume Chaslot, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Grid coevolution for adaptive simulations: Application to the building of opening books in the game of go. In *Applications of Evolutionary Computing*, pages 323–332. Springer, 2009.
2. Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

3. Guillaume MJ-B Chaslot, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, Olivier Teytaud, and Mark HM Winands. Meta monte-carlo tree search for automatic opening book generation. In *Proc. 21st Int. Joint Conf. Artif. Intell., Pasadena, California*, pages 7–12, 2009.
4. Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2007.
5. Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.
6. Richard J Lorentz. Amazons discover monte-carlo. In *Computers and games*, pages 13–24. Springer, 2008.
7. J Pim AM Nijssen and Mark HM Winands. An overview of search techniques in multi-player games.
8. J Pim AM Nijssen and Mark HM Winands. Enhancements for multi-player monte-carlo tree search. In *Computers and Games*, pages 238–249. Springer, 2011.
9. J Pim AM Nijssen and Mark HM Winands. Playout search for monte-carlo tree search in multi-player games. In *Advances in Computer Games*, pages 72–83. Springer, 2012.
10. Mehdi Samadi, Jonathan Schaeffer, Fatemeh Torabi Asr, Majid Samar, and Zohreh Azimifar. Using abstraction in two-player games. In *ECAI*, pages 545–549, 2008.
11. Maarten P. D. Schadd and Mark H. M. Winands. Best reply search for multiplayer games. *IEEE Trans. Comput. Intellig. and AI in Games*, pages 57–66, 2011.
12. Jonathan Schaeffer, Yngvi Björnsson, Neil Burch, Robert Lake, Paul Lu, and Steve Sutphen. Building the checkers 10-piece endgame databases. *Advances in Computer Games*, 10:193–210, 2003.
13. Nathan R. Sturtevant. A comparison of algorithms for multi-player games. In *Computers and Games*, pages 108–122, 2002.
14. Nathan R. Sturtevant. Last-branch and speculative pruning algorithms for $\max^n$. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 669–678. Morgan Kaufmann, 2003.
15. N.R. Sturtevant and M.J. Rutherford. Minimizing writes in parallel external memory search. *International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.
16. Ken Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9(3):131–139, 1986.