AN ANALYSIS OF UCT IN MULTI-PLAYER GAMES

Nathan Sturtevant[1]

Department of Computing Science
University of Alberta

ABSTRACT

The UCT algorithm has been exceedingly popular for Go, a two-player game, significantly increasing the playing strength of Go programs in a very short time. This paper provides an analysis of the UCT algorithm in multi-player games, showing that UCT is computing a mixed-strategy equilibrium, as opposed to max^n, which computes a pure-strategy equilibrium. We analyze the performance of UCT in several known domains and show that it performs as well or better than existing algorithms. We also test several well-known UCT learning and playout rules. We suggest two criteria, branching factor and n-ply state variance. The experiments show that they are correlated with the success of the techniques.

1. INTRODUCTION

Monte-Carlo methods have become popular in the game of Go over the last few years, and even more so with the introduction of the UCT algorithm (Kocsis and Szepesvári, 2006). Go is probably the best-known two-player game in which computer players are still significantly weaker than humans. UCT works particularly well in Go for several reasons. We mention two of them. First, in Go it is difficult to evaluate states in the middle of a game, but UCT only evaluates endgames states, which is relatively easy. Second, the game of Go converges for random play, meaning that it is not very difficult to arrive at an end-game state.

Multi-player games are also difficult for computers to play well. First, it is more difficult to prune in multi-player games, meaning that normal search algorithms are less effective at obtaining deep lookahead. While alpha-beta pruning reduces the size of a game tree from $O(b^d)$ to $O(b^{d/2})$, the best techniques in multi-player games only reduce the size of the game tree to $O(b^{\frac{n-1}{n}d})$, where n is the number of players in the game (Sturtevant and Korf, 2000; Sturtevant, 2003a). A second reason why multi-player games are difficult is because of opponent modelling. In two-player zero-sum games opponent modelling has never been shown to be necessary for high-quality play, while in multi-player games, opponent modelling is a necessity for robust play versus unknown opponents in some domains (Sturtevant and Bowling, 2006).

As a result, it is worth investigating UCT to see how it performs in multi-player games. We first present a theoretical analysis, where we show that UCT may compute a mixed-strategy equilibrium in multi-player games and discuss the implications. Then, we analyze UCT's performance in a variety of domains, showing that it performs as well or better as the best previous approaches. This is an extended version of Sturtevant (2008) with additional results and analysis. An preliminary analysis of UCT for multi-player Go can be found in Cazenave (2008) .

2. BACKGROUND

The max^n algorithm (Luckhardt and Irani, 1986) was developed to play multi-player games. Max^n searches a game tree and finds a strategy which is in equilibrium. That is, if all players were to use this strategy, no player

[1]University of Alberta, Dept. of Computing Science, Edmonton, Alberta, email:nathanst@cs.ualberta.ca

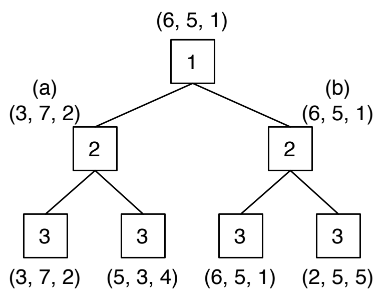

Figure 1: A sample max^n tree.

could unilaterally gain by changing their strategy. In every perfect information extensive form game (e.g., tree search) there is guaranteed to be at least one pure-strategy equilibrium, that is, one that does not require the use of mixed or randomized strategies.

We demonstrate the max^n algorithm in Figure 1, a portion of a 3-player max^n tree. Nodes in the tree are marked with the player to move at that node. At the leaves of the tree each players' payouts are in a n-tuple, where the ith value is the payoff for the ith player. At internal nodes in a max^n tree the player to play selects the move that leads to the maximum payoff. So, at the node marked (a), Player 2 chooses (3, 7, 2) to get a payoff of 7 instead of (5, 3, 4) which leads to a payoff of 3. At the node marked (b) Player 2 can choose either move, because they both lead to the same payoff. In this case Player 1 chooses the leftmost value and returns (6, 5, 1). At the root of the tree Player 1 chooses the move which leads to the maximum payoff, (6, 5, 1).

While the max^n algorithm is simple, there are several complications. In real games players rarely communicate explicitly before the beginning of a game. This means that they are not guaranteed to be playing the same equilibrium strategy, and, unlike in two-player games, max^n does not provide a lower bound on the final payoff in the game when this occurs (Sturtevant, 2004). In practice it is also not always clear which payoffs should be used at leaf nodes. The values at the leaves of a tree may be scores, but can also be the utility of each score; the opponents' utility function may not be known *a priori*. For instance, one player might play a riskier strategy to increase the chances of winning the game, while a different player may be content to take second place instead of risking losing for the chance of a win. While the first approach may be better from a tournament perspective (Billings, 2006), you cannot guarantee that your opponents will play the best strategies possible.

One may look at the payoffs in Figure 1 and argue that a preference does exist at node (b). That is, Player 2 should prefer the node with the highest relative payoff in comparison to other players. This makes sense if the game tree is a winner-take-all tournament. But, if the payoffs are cash prizes, Player 2 may only be interesting in achieving a maximal score. And, even if the payoffs are converted into ranks, there can still exist cases where Player 2 is a kingmaker – Player 2 may have the decision of who will win between Player 1 and Player 3, which will likely give an advantage to the player that has a better model of Player 2. In this way, the payoffs at the leaves of the trees should be seen as the players' utilities and not the actual game payoffs. But the question of how to model opponents still remains.

Thus, there is an additional strategic complexity in multi-player games beyond what exists in two-player games. Although two-player tournaments can exhibit these characteristics, they are present at a deeper level in multi-player games as compared to two-player games. We will not address these issues in detail here, but the context of these issues is important for understanding some of the results in the paper, particularly those in the game of Spades.

The max^n algorithm assumes a fixed model of the opponents in the game, as defined by the utility function. This can be purely competitive, cooperative, or any mix of the two. The paranoid algorithm (Sturtevant and Korf, 2000) is a special case of max^n where the opponents are purely cooperative with each other and competitive with the player doing planning. Two algorithms have been introduced that attempt to deal with imperfect opponent models. The soft-max^n algorithm (Sturtevant and Bowling, 2006) uses a partial ordering over game outcomes to analyse games. It returns sets of max^n values at each node, with each max^n value in the set corresponding to a strategy that the opponents might play in a subtree. The prob-max^n algorithm (Sturtevant, Zinkevich, and Bowling, 2006) uses sets of opponent models and probabilistic weights which are used for back-up at each node according to current opponent models. Both algorithms have learning mechanisms for updating opponent

```
   SampleUCTTree(state $s$, player $p$)
1  if children of $s$ not in UCT tree
2      Expand($s$)
3      $val \leftarrow$ RandomSample($s$)
4  else
5      $best \leftarrow \text{argmax}_i( \bar{X}_i[p] + C\sqrt{\frac{\ln T}{T_i}})$
6      $val \leftarrow$ SampleUCTTree($c_{best}$, next $p$)
7  UpdateAverages($s$, $val$)
8  return val
```

Figure 2: Multi-Player UCT Pseudo-Code.

models during play. In the game of Spades, the approaches were able to mitigate the problems associated with an unknown opponent. We will compare the results more in detail in our experimental section.

2.1 UCT

UCT (Kocsis and Szepesvári, 2006) is one of several recent Monte-Carlo-like algorithms proposed for game-playing. The algorithm analyzes games in two stages. In the first stage a UCT tree is built and explored. The second stage begins when the end of the UCT tree is reached. At this point a leaf in the tree is expanded and then the rest of the game is played out according to a random policy. The UCT tree is built and played out according to a greedy policy. At each node in the UCT tree, UCT selects and follows the move i for which

$$\bar{X}_i + C\sqrt{\frac{\ln T}{T_i}}$$

is maximal, where $\bar{X}_i$ is the average payoff of move i, T is the number of times the parent of i has been visited and T_i is the number of times i has been sampled. C is a tuning constant used to trade off exploration and exploitation. Larger values of C result in more exploration. In two-player games the move and value returned by UCT converges on the same result as minimax.

3. MULTI-PLAYER UCT

Multi-player UCT is nearly identical to regular UCT. At the highest level of the algorithm, the tree is repeatedly sampled until it is time to make an action. The sampling process is illustrated in Figure 2. The only difference between this code and a two-player implementation is that in line 5 the average score for player p is used instead of a single average payoff for the state.

The first question we address is: what computation is performed by UCT on a multi-player game tree? For the analysis we assume that (1) we are searching on a finite tree and that we can perform an (2) unlimited number of UCT samples. In this limit the UCT tree will grow to be the size of the full game tree and all leaf values will be exact.

Returning to Figure 1 we can look to see what UCT would compute on this tree. At node (a) Player 2 will always obtain a better payoff by taking the left branch to get (3, 7, 2). In the limit, for any value of C, the value at node (a) will converge to (3, 7, 2). At branch (b), however, both moves lead to the same payoff. Initially, they will both be explored once and have the same payoff. On each subsequent visit to branch (b), the move which has been explored least will be explored next. As a result, at the root of the sub-tree rooted at (b)UCT will return (6, 5, 1) half of the time and (2, 5, 5) the other half. The average payoff of the move towards node (b) will then be (4, 5, 3). The resulting strategy for the entire tree is as follows: the player at the root should move to the right towards node (b) to obtain an expected payoff of 4.

In this context, UCT is computing a strategy for a perfect-information extensive-form game tree, in which there always exists a pure-strategy equilibrium. Mixed equilibria, which randomly mix strategies, are not necessary for computing an equilibrium in extensive-form games. They would be needed, however, for simultaneous games, such as rock-paper-scissors.

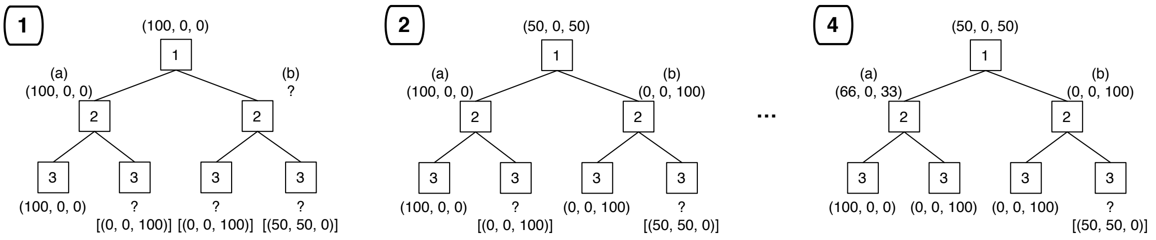

Figure 3: Mixed equilibrium in a multi-player game tree.

There are two pure-strategy equilibria in Figure 1. The first is for Player 1 to move towards node (a) and for player 2 to move towards the left-most child of the tree. In this equilibrium, Player 2 must move to the right branch of (b) as part of the strategy. The second equilibrium is the one we demonstrated, namely that Player 1 moves to node (b) at the root, and Player 2 moves to the left child of (b).

In addition to these two pure-strategy equilibria, there exists an infinite number of mixed-strategy equilibria. At node (b), Player 2 can choose either move. Assume Player 2 chooses the left child with probability p and the right child with probability $1 - p$, Player 1's strategy will be to move to node (a) as long as $p < 1/4$. If $p > 1/4$ Player 1's strategy will be to move to node (b). If $p = 1/4$, Player 1 will be ambivalent between the two choices.

The final strategy is mixed, in that Player 2 is expected to randomize at node (b). Playing a mixed strategy makes the most sense in a repeated game, where over time the payoffs will converge due to the mixing of strategies. But, in a one-shot game this makes less sense, since a player cannot actually receive the equilibrium value. Many games are repeated, however, although random elements in the game result in different game trees in each repetition (e.g., from the dealing of cards in a card game). In this context randomized strategies still make sense, as the payoffs will still average out over all the hands. Randomized strategies can also serve to make a player more unpredictable, which will make the player harder to model.

A mixed strategy is not required to play perfect-information extensive form games. When mixed strategies are traditionally used in game theory, it is expected that a variety of different values will be mixed by the strategy. In a multi-player game, however, UCT will, in the limit, only mix strategies when the player to move has identical payoffs for multiple moves.

Theorem 1 *UCT computes an equilibrium strategy, which may be mixed, in a multi-player game tree.*
Proof. We have demonstrated above that in some trees UCT will produce a mixed strategy. In the limit of infinite samples UCT returns a strategy that is in equilibrium in a finite tree because it selects the maximum possible value at each node in the UCT tree. This means that there is no other move that the player could choose at this node to improve unilaterally their payout. □

Although the value computed by UCT will be from a mixed strategy, the actually strategy played depends on several details of the implementation. For instance, if there is a tie in the tree, and ties are always broken the same way, the actual play will not be mixed, although an equilibrium will be selected that assumes mixing of strategies deeper in the tree. Additionally, the mixture of strategies in the tree will change depending on the exact values at the leaves.

We demonstrate this in Figure 3. In the first UCT step, the leftmost branch will be sampled and the tree will have a value of (100, 0, 0). In the second step, the right branch will be sampled resulting in an average value of (50, 0, 50) at the root. The third step, not shown, will sample the left branch again, bringing the average value down at node (a) to 50. Unless C is approximately 100, the left branch will then be sampled again in the fourth step. Thus, in the initial steps the left branch will be favoured over the right branch, although the average value will slowly converge on 50 from above, and the left branch will converge on 50 from below once the right child of node (b) is sampled. Thus, the actual mix of strategies used in practice is hard to predict, as in practice we can only do a finite number of node expansions on any given tree. But, in this example the left branch will be favoured over the right branch in the initial simulations on the tree.

The max^n algorithm does not mix strategies like UCT, although it is not hard to modify it to do so. But, the biggest strength of UCT is not its ability to imitate max^n, but its ability to infer an accurate evaluation of the current-state-based samples of possible endgame states.

Player 1	Player 2	Player 3
Type A	Type A	Type B
Type A	Type B	Type A
Type A	Type B	Type B
Type B	Type A	Type A
Type B	Type A	Type B
Type B	Type B	Type A

Figure 4: Six ways to arrange two player types in a 3-player game.

These theoretical results do not guarantee that UCT will be the best choice for any given game. But, they provide the assurance that UCT is performing a computation that will eventually converge on something which is not unreasonable.

3.1 UCT Enhancements

There are a variety of enhancements which have been used with UCT to improve performance. We review three enhancements here, and will evaluate their effectiveness in the experimental results. There is nothing in these enhancements which is inherently correlated to two-player search, and so they all have potential in multi-player games.

Enhanced playout strategies. Instead of playing out the game randomly, a more informed playout policy can be used. This policy might avoid obviously bad moves or use other domain-specific information to improve the quality of the playouts. Hand-crafted playout strategies have been shown to be highly effective in Go, but there is no guarantee that a strategy will improve performance (Gelly and Silver, 2007).

RAVE / History heuristics. The history heuristic (Schaeffer, 1989; Finnsson and Björnsson, 2008) was suggested in two-player games as a mechanism for improving move ordering. The idea is straightforward in that good moves played once in the tree are tried first later in the tree. Rapid Action Value Estimate (RAVE) (Gelly and Silver, 2007) is a similar mechanism for UCT which associates the values of the states after a move instead of just the ordering of the moves.

Initial state-value estimates. UCT normally does not use a heuristic evaluation function; only terminal values are backed up in the UCT tree. However, there are many times when an evaluation function may be available, either through learned or hand-coded knowledge. This evaluation function can be used to initialize values within the UCT tree to initial estimates. The knowledge can be assigned an equivalent value in simulations, with more simulations representing more confidence in the original estimates.

4. EXPERIMENTS

Given the theoretical results regarding UCT, we perform experiments with UCT to answer three main questions: First, does the strength of play from UCT vary significantly from previous approaches? Second, how do opponent modelling approaches interact with UCT? Finally, what is the effect of well-studied UCT enhancements in other domains.

Most experiments reported here are run between two different player types (UCT and an existing player) in either a 3-player or 4-player game. In order to reduce the variance of the experiments, they are repeated multiple times for each possible arrangement of player types in a game. For instance, in a 3-player game there are eight ways to arrange two players. Two of these arrangements contain all of a single type of player, so we remove these arrangements, resulting in the configurations shown in Figure 4. Playing duplicate matches in this way makes it easier to measure statistical significance.We perform experiments in Chinese Checkers, Hearts, and Spades.

4.1 Chinese Checkers

Our first experiments are in the game of Chinese Checkers. A Chinese Checkers board is shown in Figure 5. In this game the goal is to get your pieces across the board and into a symmetric position from the start state as

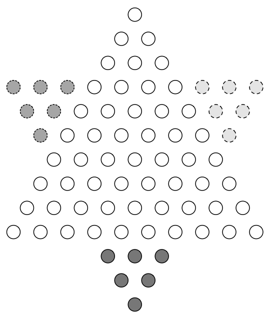

Figure 5: A 3-player Chinese Checkers board.

	UCT	Paranoid (diff)	Paranoid (dist)	Max^n (diff)	Max^n (dist)
UCT	-	8.0%	4.0%	3.7%	6.0%
Paranoid (diff)	92.0%	-	46.3%	25.0%	36.7%
Paranoid (dist)	96.0%	53.7%	-	46.3%‡	68.7%
Max^n (diff)	96.3%	75.0%	53.7%‡	-	56.3%†
Max^n (dist)	94.0%	63.3%	31.3%	43.7%†	-

Figure 6: Experiments in the game of Chinese Checkers.

quickly as possible. Chinese Checkers can be played with anywhere from two to six players. We experiment with the three-player version here. Chinese Checkers can be played on differently sized boards with different numbers of checkers. We use a slightly smaller board than is most commonly used, because this allows us to use a stronger evaluation function. The evaluation function for the opponent, a max^n player, is based on a lookup table containing the optimal distance from any state to the goal state given that no other players' pieces are on the board. A player using this evaluation function will play strong openings and perfect endgames. Because the evaluation function ignores the opponents' pieces, it is not quite as strong during the mid-game phase when the middle of the board can get congested.

The first experiment we present is a comparison across several algorithms and evaluation functions. The paranoid algorithm (Sturtevant and Korf, 2000) is a special case of max^n which assumes that all opponents are working against you. This reduces the game to a two-player game and allows deeper search. We evaluated paranoid and max^n with two different evaluation functions. In the first, players are just trying to minimize their own distance (dist) from the goal. In the second, players are trying to maximize the difference (diff) between their distance from the goal and other players distance from the goal. The results are in Figure 6.

In this experiment, all players were allowed 250,000 node expansions per move and 300 total games were played for each comparison. The UCT player used the 'farthest-first' playout rule, which we describe below. All results are statistically significant with 99% confidence except for those marked † which are only significant at 95%. The result marked ‡ does not reach statistical significance at 95% confidence.

UCT wins over 90% of the games against the other algorithms. Among the other algorithms, it is interesting to note that paranoid plays better with the *diff* evaluation function, while max^n plays better with the *dist* evaluation.

In Figure 7 we compare the strength of play as we successively double the number of nodes expanded. UCT needs a minimum of 25,000 node expansions to play well. With fewer expansions there is not enough data to form accurate estimates. For instance, UCT only wins 16.7% of games against paranoid when both algorithms are only allowed 1600 node expansions per move.

4.1.1 UCT Enhancements in Chinese Checkers

Prior to conducting the above experiments, we experimented with two versions of UCT. In both versions, we disallowed backwards moves; otherwise the game will not progress and the random sampling fails. The first version of UCT uses purely random playouts. The second version uses an epsilon-greedy playout policy with a 'farthest-first' playout policy. The move which advances a piece the farthest distance is selected with 95% probability and a random move is selected the remaining 5% of the time. We compared these policies by playing

Expansions	50k vs. 25k	100k vs 50k	200k vs 100k	400k vs 200k	800k vs 400k
Wins	60.7%	54.3%	55.2%	60.1%	51.8%

Figure 7: Relative strength as number of node expansions is doubled.

50 games in each of the configurations from Figure 4. (Thus there were 300 total games played with each player type playing 450 times.)

In the comparison we varied the limit on the number of nodes expanded from 400 to 50,000. In all cases the farthest-first strategy won at least 90% of the games played. There are two reasons for this. In comparisons with a fixed number of playouts, the farthest-first strategy also wins approximately 80% of the games. But, the farthest-first playouts are much shorter than random playouts, and thus given a fixed limit on node expansions will allow more samples. The best-move first strategy will finish a game in approximately 80 moves, while the pure-random strategy will take approximately 200 moves to finish a game, when starting at the root of the game tree.

We attempted to use the single-agent database to initialize values in the UCT tree, however the results were inconsistent. With 50,000 node expansions per move the initialization did not have a significant effect on the quality of play. However, initializing values from the database was able to improve the quality of play when the number of expansions was limited. For instance, a player with only 800 node expansions is able to beat a player with 6400 node expansions 60% of the time with no database initialization. But, when the player with 6400 expansions uses the database to initialize values, it wins over 80% of games. There are similar results when playing programs with an equally small number of expansions.

The history heuristic and RAVE both depend on the value of moves being correlated throughout a game tree. In a game like Go, much of the structure of the pieces on the board does not change from move to move. But, in Chinese Checkers, pieces are continually moving across the board, which seems to make it more difficult for these techniques to work effectively.

4.2 Hearts

Our second domain is the card game, Hearts. The goal of Hearts is to take as few points as possible. A game of Hearts is broken into many smaller hands, each of which is played independently. The game ends when one player reaches or exceeds 100 points. We experiment on the 4-player version of Hearts here, which is most common. Because every Hearts game is exactly 52 moves long, we used a simulation limit instead of a node limit when performing experiments.

Most card games have imperfect information; players cannot see their opponents' cards. Although it is an imperfect approximation, this can be handled through a different type of Monte-Carlo sampling. Hands can be generated for opponents which are consistent with play so far and solved independently. Then, the best move across all hands is chosen. We use this method to play imperfect information games, but it is much more expensive than solving the perfect-information games, and we have never seen a player that is strong in the perfect-information game do worse in the imperfect-information game with this method. Thus, we only show results from the perfect-information game.

One of the difficulties of Hearts is that there are two alternate ways to play the game. The normal goal is to take as few points as possible. But, if a player manages to take all the points, called 'shooting the moon', this player will get 0 points instead, and the other players will get 26 each. Thus, good players are willing to take a few points to keep other players from shooting the moon.

4.2.1 Hearts - Shooting the Moon

To measure the skill of different players in this aspect of the game, we created a library of games in which one player could possibly shoot the moon. These games were found by playing 10,000 games with two player types. In these games the ith player would try to take all the points, and the remaining players would try to avoid taking points, essentially helping this player shoot. Out of 40,000 possibilities (10,000 games times 4 possible hands in each game) we found 3,244 hands in which a player can shoot the moon if the opponents actively help. We

Algorithm Comparisons

	UCT	Self-trained	UCT-trained	Random play	Simple max^n
total	250	312	362	411	1377
perc.	7.70%	9.62%	11.16%	12.67%	42.45%

UCT Parameter Comparisons

C	0.0	0.2	0.4	0.6	0.8	1.0
total	444	310	285	292	315	332
perc.	13.69%	9.56%	8.79%	9.00%	9.71%	10.23%

C (self/others)	0.0 / 0.4	0.2 / 0.4	0.0 / 0.6	0.2 / 0.6	0.4 / 0.6
total	250	285	273	298	303
perc.	7.70%	8.79%	8.42%	9.19%	9.34%

Figure 8: Shooting the moon in Hearts.

then ran a set of different algorithms against this shooting player in the 3,244 hands. In the experiments, one player always tries to shoot the moon. The other players do not know that someone is explicitly trying to shoot the moon. So, we are measuring how well each algorithm can detect this and then prevent the other players from shooting. In Figure 8 we report how many times the opponent was able to shoot the moon against each search algorithm.

A simple max^n player, which does not explicitly attempt to stop the opponents from shooting, was only able to stop the opponent from shooting 1867 times, leaving 1377 times when the opponent shot. A random player was able to stop the opponent from shooting in all but 411 games. This player does not play hearts well, but as a result is able to disrupt the normal play of the game. Two players which learned to play hearts through self-play and play against UCT stopped the opponents from shooting in all but 312 and 362 cases respectively. The best algorithm in this comparison was UCT, which only had the opponent shoot 250 times when using 50,000 playouts.

We experimented with a wide variety of UCT parameters here, and we summarize the results in the bottom part of Figure 8. First, we varied C from 0.0 to 1.0 by increments of 0.2. In these experiments, values of 0.4 and 0.6 produced the best performance, stopping all but 285 and 292 situations respectively. We then tried a new approach, where we used one value of C for all nodes where the player at the root is to play, and used a different value at all other nodes. The intuition behind this approach is that a player should quickly find the best move for itself, but explore the opponents' responses more thoroughly. Using a value of 0.0 for the player at the root and 0.4 for other players produced the best behaviour, better than the results produced when either 0.0 or 0.4 was used for all players.

This experiment is interesting because UCT has to find the specific line of play an opponent might use to shoot the moon in order to stop it. Using a lower value of C increases the depth of the UCT tree, which helps ensure that a safe line of play is found. However, this does not necessarily guarantee that the best line of play is found. In some sense this can randomize play slightly without too large a penalty.

4.2.2 Hearts - Quality of Play

To test the quality of play, we played full games of Hearts. That is, a given configuration of players played multiple hands of Hearts until one player's score reached 100. The final score for an algorithm is the average score of all players using that algorithm at the end of the game. We also report the standard deviation of the final score, as well as the percentage of games in which a given player had the highest score at the end of the game. Experimental results are in Figure 9. All results are statistically significant with 99% confidence.

The current state-of-the-art players in Hearts use a learned evaluation function Sturtevant and White (2007). . We trained three players to play Hearts using methods similar to, but more efficient than those described in Sturtevant and White (2007). Whether we trained through self-play or against UCT players of varying strength, the learned players had essentially identical performance in the full game, scoring 20 points more on average than the UCT player. (Lower scores are better.) These results were stable both for the different learned players and against UCT with up to 50,000 playouts per turn; we report results against UCT with 5000 playouts. For comparison, an older, hand-tuned player (Sturtevant, 2003b) averaged 88.31 points per game against UCT, just better than the random player, which averaged 89.23 points a game. However, UCT's average score against the random player, 16.31, is much better than against the hand-tuned player, 51.77.

In the bottom part of Figure 9 we experiment with different numbers of playouts. Here we play UCT with k playouts against UCT with $2k$ playouts. The player with more playouts consistently averages 4 to 5 points better than the player with fewer playouts. This shows that UCT does improve its performance as the number of simulations increases.

In these experiments we used a value of $C = 0.4$. When compared to the alternating values of C used in the previous section, $C = 0.4$ provided the best performance, although both players always beat the learned players. The learned players tested here are not nearly as aggressive about attempting to shoot the moon as the player in the last section, which means that the ‘insurance’ paid to keep one’s opponent from shooting the moon is less likely to pay off in these experiments.

4.2.3 UCT Enhancements in Hearts

We experimented with each of the UCT enhancements in the game of Hearts, however these enhancements generally did not work well in Hearts.

First, we looked at playout rules. We tried several different playout strategies.The simplest strategy we tried was for players to play always the highest card they could that did not take the current trick and leading low cards, a strategy employed by novice players. UCT with this playout strategy, however, did much worse than the random playouts in UCT, although it is a much stronger player in the game. Since this did not work, we tried adding smaller rules to the playout; the only one that had a statistically significant effect was to play the queen of spades on another player whenever possible. This was the only playout rule used in the previous experiments.

In card games every card must be played exactly once in each the game. Whether a card is a good play or not depends entirely on the context, so we do not expect RAVE or the history heuristic to work well in Hearts. RAVE, in particular, assigns a value to a state based on the move, not on the context. As expected, we were not able to see any benefit from RAVE or the history heuristic. We then tried categorizing moves by their context. Moves were divided into three contexts, leading moves, following (in suit) moves, and sloughing (out of suit) moves. This division categorization should be more powerful, as leading low spades is almost always good, as is sloughing the queen of spades. But, even with this additional context, we were not able to see an advantage from RAVE-like approaches. Of course, this does not mean that a similar approach cannot be used, just that we have yet to find the correct approach. But, given the results in Hearts and Chinese Checkers, it seems that RAVE is particularly well suited for Go and not as well suited for other games.

Next, we attempted to pre-initialize the values of UCT nodes using the learned values from the learning players. The learning players only evaluate a game state at the end of a trick, so not every game state can be initialized with a UCT value. We tried initializing each game state with the learned value, and then played against the same player with no initialization. In Figure 10 we show the results as we vary the number of simulations. The results show that the pre-initialization helps when very few simulations are available for play. But, as the number of simulations increase, the initialized knowledge becomes less useful, and eventually hurts performance. The difference are statistically significant with 99% confidence for all results except with 1000 simulations.

	Learned	Hand-tuned	Random
UCT 5000	46.12 (30.6)	51.77 (27.2)	16.31 (13.7)
Opponent	67.30 (43.1)	88.31 (24.5)	89.23 (24.1)
loss perc.	83.9%	88.0%	100%

	$k = 100$	250	500	1000	2000	4000	8000
UCT k	79.46	77.69	77.18	76.49	76.25	76.47	76.59
(std dev)	(26.95)	(26.78)	(27.07)	(27.38)	(26.65)	(27.06)	(26.76)
loss perc.	66.4%	59.4%	58.7	57.1%	55.5%	58.3%	56.7%
UCT $2k$	67.07	69.92	71.40	71.52	72.04	71.91	72.17
(std dev)	(27.69)	(27.86)	(27.35)	(27.63)	(27.11)	(26.84)	(26.80)

Figure 9: Performance of UCT against various opponents.

	100	250	500	1000	5000
UCT with initialization	74.2	74.3	75.2	76.1	77.1
Plain UCT	77.2	77.0	77.1	76.0	74.4

Figure 10: Value initialization in Hearts.

4.3 Spades

Our third series of experiments is in the card game Spades. A game in Spades is divided into multiple hands which are played semi-independently. The first player to reach 300 points over all hands wins. In Spades players bid on the number of tricks which they expect to take. There is a large, immediate penalty for taking fewer tricks than bid. If, over a period of time, a player takes too many extra tricks (overtricks), there is also a large penalty. Thus, the best strategy in the game is to ensure that you make your bid, but then to avoid extra tricks after that.

Previous work in Spades (Sturtevant and Bowling, 2006; Sturtevant *et al.*, 2006) demonstrated that the selection of an opponent model is very important. A superior strategy may only be better if you have an accurate opponent model. We duplicate these experiments here to demonstrate that while UCT is strong, it does not avoid the need for opponent modelling. Then, we compare the prob-max^n algorithm to UCT.

Each player was given 7 cards, and the game was played with 3 players, so the total depth of the tree was 21ply. This means that the full game tree can be analyzed by prob-max^n. Again, each game was repeated multiple times according to the configurations in Figure 4. There are two player types in these experiments. The mOT player tries to make their bid and minimize overtricks (**m**inimize **O**ver**T**ricks). The MT player attempts to maximize the number of tricks taken (**M**aximize **T**ricks), irrespective of the bid. Player types are subscripted by the opponent model that is being used. So, mOT_{MT} is a player that tries to minimize their overtricks and believes that their opponents are trying to maximize the number of tricks they are taking. These types determine the utility given to a player at the leaf of the game tree. A player trying to maximize tricks will get higher utility for taking more tricks, while one trying to minimize overtricks will get lower utility for taking more tricks, as long as the bid is made.

Experimental results for Spades are in Figure 11. All experiments are run with UCT doing 10,000 samples per move with $C = 2.0$. There are no playout rules; all playouts beyond the UCT tree are purely random. Lines A and B show the performance of each player when playing against itself. mOT players average 245.91 points per game, clearly better than the MT player which only averages 202.71 points per game.

Lines C-F demonstrate what happens when a mOT player has correct (C-D) or incorrect (E-F) opponent models. The most interesting line here is line F. When mOT has the wrong opponent model and MT does as well, mOT has a lower average score than MT and only wins 43% of the games. As stated previously, these results have been observed before with max^n. These experiments serve to confirm these results and show that, in this situation, the mixed equilibrium computed by UCT is not inherently better than the equilibrium computed by max^n.

The discrepancy in performance can be partially resolved by the use of generic opponent models, which are learned automatically by soft-max^n. The learned, generic model only assume that the opponent is trying to make their bid, but nothing else. They can be used in UCT by modifying the utility of an outcome in the game. The UCT player still attempts to minimize overtricks, but the opponents utility is binary, only indicating whether or not they made their bid. Using the model, in lines G-J, the mOT_{gen} player beats every opponent except a mOT_{mOT} opponent.

Finally, in lines K-L we compare UCT using the mOT strategy and a generic opponent model to prob-max^n. In line K the prob-max^n player does not do any learning, while in line L it does. The results in line K are significant with 99% confidence, but in line L they are only significant with 95% confidence. At first glance, it seems that the UCT player is better than prob-max^n, but this is not entirely true.

When breaking down these results into per-player performance, we notice an interesting trend. The player which plays last (the third player) almost always scores worse than the other players. This is the case in lines A-J for the UCT player and in K-L for the prob-max^n player. There are two reasons why this is not surprising. (1) The player who moves last has extra constraints on their possible bids, meaning they may be forced to under- or over-bid. (2) The player moving last also has less control over the game.

We break-down the results by player position in Figure 12. Here, we can see more clearly what is happening. The

	Algorithm 1	Algorithm 2	Avg. Score Alg. 1	Avg. Score Alg. 2	Algorithm 1 win %
A	mOT_{mOT}	mOT_{mOT}	245.91	-	-
B	MT_{MT}	MT_{MT}	202.71	-	-
C	mOT_{MT}	MT_{mOT}	231.84	171.48	67%
D	mOT_{MT}	MT_{MT}	214.33	209.30	51.5%
E	mOT_{mOT}	MT_{mOT}	203.72	188.96	55%
F	mOT_{mOT}	MT_{MT}	179.19	212.76	43%
G	mOT_{gen}	mOT_{MT}	243.14	238.65	51%
H	mOT_{gen}	mOT_{mOT}	240.06	245.70	48.5%
I	mOT_{gen}	MT_{mOT}	235.41	181.54	64%
J	mOT_{gen}	MT_{MT}	215.72	207.17	51.5%
K	mOT_{gen}	prob-max^n	214.58	198.21	52.8%
L	mOT_{gen}	prob-max^n (learn)	212.60	202.67	53.2%

Figure 11: UCT performance in Spades.

	Algorithm	Player 1 Avg	Player 2 Avg	Player 3 Avg
K	mOT_{gen}	211.7	219.2	212.8
K	prob-max^n	227.2	220.3	147.1
L	mOT_{gen}	208.6	218.0	211.2
L	prob-max^n (learn)	227.7	228.1	152.3

Figure 12: Detailed comparison of UCT and prob-max^n.

prob-max^n player outplays UCT when playing as Player 1 with 99% confidence or Player 2 by a small margin, but loses badly as Player 3. As stated before, this is not a feature of prob-max^n, but of other algorithms as well. So, the novel aspect of this situation is that the UCT player manages to play not poorly against prob-max^n when in the third position. We do not have a systematic explanation of the effect, but have noticed that these types of inconsistencies are common in multi-player games. We are working on further analysis of this effect.

One may be tempted to think that the comparison here has been bogged down by opponent modelling issues. In fact, the opposite is true. This experiments show that opponent modelling *is* the issue in many multi-player domains, especially one with such sharp changes in the evaluation function, as is the case in Spades.

5. DISCUSSION AND ADDITIONAL EXPERIMENTS

We can now address the three questions stated at the beginning of Section 4. We start with a brief answer to all these questions. For question 1: in terms of performance, UCT is comparable to previous approaches. For question 2 we may state: UCT is better than the previous work in Hearts, but faces the same opponent modeling issues previously seen in Spades. UCT is much stronger in Chinese Checkers given a minimum amount of computational power. A program that must play very quickly should use max^n or paranoid, while given more thinking time, UCT is better. For question 3 it holds that standard UCT enhancements do not work well in Hearts, but custom playout rules improve performance in Chinese Checkers.

The preceding experimental results focus mostly on how each of the techniques works in each game. We now turn to the question of whether we can predict the performance of techniques based on the properties of the game being analyzed. Two properties seem to be important in determining whether playout rules and other enhancements will work well in a game. These are, (1) the branching factor of the game, and (2) the stability of positions in the game, which we will call *n-ply state variance*. We postulate that, while the multi-player nature of a game is important for playing well, it is not as important with regard to the UCT enhancements. We discuss both properties below.

First, we consider the branching factor of a game. While there were obvious playout rules for both Hearts and Chinese Checkers, the playout rules for Chinese Checkers were quite beneficial, while the rules in Hearts were detrimental. In a card game you are often following suit, and given additional suit reductions, such as only generating one move for adjacent cards, the branching factor is generally low. But, in Chinese Checkers there are many possible moves, and many of these moves are of low quality. Thus, we postulate that the number of possible moves during playout may be influencing the effectiveness of playouts.

We measured the average number of moves available to each player during playouts of Hearts and Chinese Checkers and show the results in Figure 13. The results are given as percentages; the total number of moves sampled in each game was over one million. The graph on the left shows that, in Hearts, 45% of the time there is only one legal move, and 70% of the time there are two or fewer legal moves. This contrasts with Chinese Checkers, which has a much broader distribution of available moves. Note that backwards moves were disallowed in Chinese Checkers, otherwise the branching factor would be even larger. The exact shape of the curve in Chinese Checkers also varies depending on the playout rule being used. The curve here is using the 'farthest-first' move heuristic.

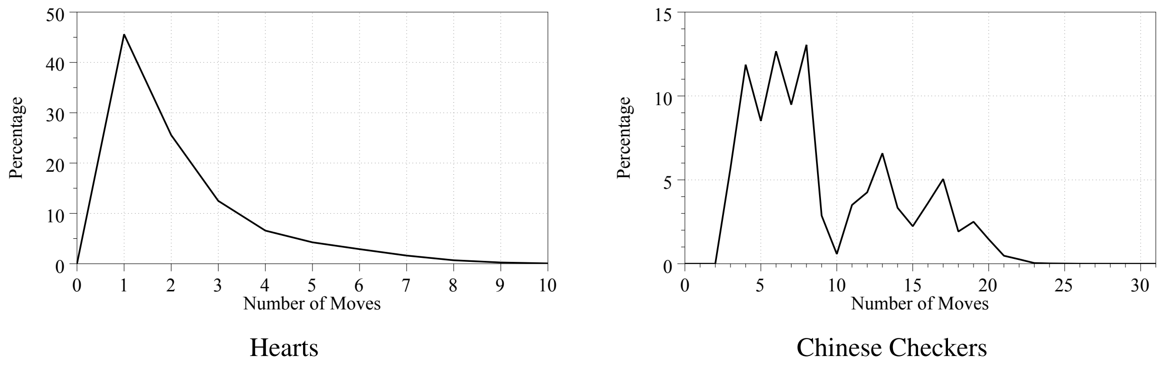

Figure 13: Distribution of legal move count in Hearts and Chinese Checkers.

If there is only one legal move to make, a playout policy is useless. With a low branching factor, there is a reasonable chance that good moves will be made with some frequency. But, games like Chinese Checkers and Go have large branching factors, with many irrelevant moves. In these games the probability of playing a good move during purely random playouts is quite small, and so it is not surprising that there is a large benefit to good playout policies. This is not to say that a good playout policy cannot be found in Hearts, just that it is of much less importance than in other domains with larger branching factors.

Second we consider the stability of a position in a game. This is a more difficult metric to quantify. If a game is being described in a logic-based language like GDL, used for the General Game Playing competition (Genesereth, Love, and Pell, 2005), it might be a measure of how many clauses change after any move has been applied. We could measure both the 1-ply variance, how much a single move changes the state of the game, as well as the n-ply variance, how much the application of n moves changes the state of the game. We believe that learning procedures which either initialize state values, such as RAVE, or learn playout policies, such as the history heuristic, will be effective in games where the n-ply variance is low. If most of the state stays the same it should be easier to generalize from one state to another.

In a game like Go only one piece is placed on the board at a time, so the 1-ply state variance is low; most of the state description remains unchanged from turn to turn. Over n moves in Go, there will generally be n changes in the state. But, for a constant n in a game with m total moves, the ratio of the changes to the state n/m will shrink as the game progresses. Thus, we may say that the n-ply state variance in Go is low.

In Chinese Checkers only one piece moves at a time, but the number of pieces is fixed, so the relative positions of the pieces are always changing. Unlike Go, after n-ply, the complete state of the game can change. This suggests, as our experimental evidence showed, that learning mechanisms will not work well in Chinese Checkers.

However, there is one exception to this general rule. It may be that there are other invariants which are not represented in the state of game which can be used for learning. For instance, the history heuristic would work well in Chinese Checkers if, instead of hashing the move, we hashed the distance a piece moved across the board. Almost any learning rule could quickly learn that moving a piece a long distance across the board is advantageous.

In a card game like Hearts, a player's cards remain mostly the same from turn to turn, but hands diverge as they are played out, and unique sets of cards remain to be played out. This suggests that the n-ply variance is also high in Hearts.

5.1 Experimental Validation

Given our analysis of these properties, we looked for a game which exhibited both a high branching factor and a low n-ply variance. We chose to experiment with the card game, Gin Rummy. Gin Rummy is normally played with 2 players. We modified the game to allow more than two players and experimented with both the two-player and three-player version of the game.

Gin Rummy is not a trick-taking game like Hearts or Spades. Instead, players try to match all the cards in their hands in either runs (of the same suit) or in 3- or 4-of-a-kind sets. Most hands end when a player has all the cards in their hand matched. Players alternate turns in the game. A turn consists of drawing one card from the deck or drawing one card from the discard pile, followed by discarding a new card to the discard pile. As players have 10 cards in their hand, the branching factor alternates between 2 and 11. This meets the criteria of having a high branching factor. Additionally, only one card changes in a player's hand at a time, meaning that the 1-ply variance is low. It is rare for a player to exchange all their cards in the course of a game, meaning the n-ply variance is also low.

There is a simple discarding rule for Gin Rummy, which is to discard any cards which do not have 'connectors' to other cards in your hand. For instance, if a player holds $10\diamondsuit$ $7\clubsuit$ $7\heartsuit$ $6\heartsuit$ this rule would discard the $10\diamondsuit$, as it cannot form a run or a set with the other cards. We first compared a random playout strategy to one using the simple discarding rule. In the two-player game the random playout strategy averaged 111 moves per random playout, while the simple discarding rule averaged 4.27 moves. In the three-player game we see a similar result, except that the discarding rule averaged 7.5 moves per game. This shows that there is a straightforward strategy which can vastly increase the quality of the playouts and confirms our intuition that games with larger branching factors have more room for better playout rules. In the two-player game the custom-playout strategy wins 88% of the games against the random playout strategy, given a 10,000 node expansions limit.

However we faced several issues when extending these results to the three-player version of the game. A first issue is that an evaluation function which makes sense in the two-player game produces strange behaviour in the multi-player game. A second, main issue is how to resolve a game when no player wins before the cards are exhausted. What we see is that rules can easily be 'exploited' by the computer players which, as a result, end up in playing very odd strategies. This probably explains why humans do not play the three-player game. Here we observe that the custom-playout strategy does not always win – it depends on the payoffs being used at the leaves of the tree. This also seems to involve issues of opponent modelling.

Yet, in both the two-player and three-player game we were able to use a form of the history heuristic to improve play over the baseline random player, and in some cases even over the custom-playout strategy. We computed a recency-weighted average of the value of a move in the UCT tree and then used Gibbs sampling with $\tau = 0.05$ as in (Finnsson and Björnsson, 2008) to bias the selection of moves during random playouts. This policy was able to beat the random policy, although the results depend on the evaluation function being used. A complete study of multi-player Gin Rummy is beyond the scope of this paper. It is sufficient to say that the potential for learning is present both in the two-player and multi-player game. However, one must be aware of the unique issues involved in multi-player games, as these may play an important role in any outcome.

We did not have any learning data for Gin Rummy and so did not test to see if we could learn data for pre-initializing node values in this game.

6. CONCLUSIONS AND FUTURE WORK

This paper provides a foundation for future work on UCT in multi-player games. It shows theoretically that UCT computes an mixed-strategy equilibrium, unlike the traditional computation performed by max$^{\text{n}}$. UCT is able to beat a player that uses a very strong evaluation function in the game of Chinese Checkers. In other domains, UCT plays on a par with existing programs in the game of Spades, and better than existing programs in Hearts.

These results are promising and suggest that UCT has the potential richly to benefit multi-player game-playing on a par programs. However, UCT in itself is not a panacea. UCT does not offer any solution to the problem of opponent modelling, and will need to be combined with opponent-modelling algorithms if we want to achieve expert-level play in these programs.

We also analyzed a number of UCT enhancements and suggested two metrics, branching factor and n-ply state variance which seem to be good predictors of whether several well-known enhancements will work well in UCT.

The card games we experimented with in this paper were all converted into perfect-information games for the experiments here. We are continuing to work on issues related to imperfect-information. For instance, UCT can be modified to handle some situations in imperfect-information games that cannot be solved by the straightforward approaches discussed here, but the exact application of these techniques is still a matter of future research.

7. REFERENCES

Billings, D. (2006). On the Importance of Embracing Risk in Tournaments. *ICGA Journal*, Vol. 29, No. 4, pp. 199–202.

Cazenave, T. (2008). Multi-player Go. *Computers and Games*, pp. 50–59.

Finnsson, H. and Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. *AAAI*, pp. 259–264.

Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in UCT. *ICML '07: Proceedings of the 24th international conference on Machine learning*, pp. 273–280, ACM, New York, NY, USA. ISBN 978–1–59593–793–3.

Genesereth, M. R., Love, N., and Pell, B. (2005). General Game Playing: Overview of the AAAI Competition. *AI Magazine*, Vol. 26, No. 2, pp. 62–72. http://games.stanford.edu/competition/misc/aaai.pdf.

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Proceedings of the 17th European Conference on Machine Learning*, pp. 282–293, Springer-Verlag, Heidelberg, Germany.

Luckhardt, C. and Irani, K. (1986). An Algorithmic Solution of N-Person Games. *AAAI-86*, Vol. 1, pp. 158–162.

Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212.

Sturtevant, N., Zinkevich, M., and Bowling, M. (2006). ProbMaxn: Opponent modeling in n-player games. *AAAI-2006*, pp. 1057–1063.

Sturtevant, N. R. (2003a). Last-Branch and Speculative Pruning Algorithms for Max^n. *IJCAI-03*, pp. 669–678.

Sturtevant, N. R. (2003b). *Multi-Player Games: Algorithms and Approaches.* Ph.D. thesis, Computer Science Department, UCLA.

Sturtevant, N. R. (2004). Current Challenges in Multi-player Game Search. *Proceedings of the 4th International Conference on Computers and Games* (eds. H. J. van den Herik, Y. Björnsson, and N. S. Netanyahu), Vol. 3846 of *Lecture Notes in Computer Science (LNCS)*, pp. 285–300, Springer-Verlag, Heidelberg, Germany.

Sturtevant, N. R. (2008). An Analysis of UCT in Multi-player Games. *Proceedings of the 6th International Conference on Computers and Games* (eds. H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands), Vol. 5131 of *Lecture Notes in Computer Science (LNCS)*, pp. 37–49, Springer-Verlag, Heidelberg, Germany.

Sturtevant, N. R. and Bowling, M. H. (2006). Robust Game Play Against Unknown Opponents. *AAMAS-2006.*

Sturtevant, N. R. and Korf, R. E. (2000). On Pruning Techniques for Multi-Player Games. *AAAI-2000.*

Sturtevant, N. R. and White, A. M. (2007). Feature Construction for Reinforcement Learning in Hearts. *Proceedings of the 5th International Conference on Computers and Games* (eds. H. J. van den Herik, P. Ciancarini, and M. H. M. Winands), Vol. 4630 of *Lecture Notes in Computer Science (LNCS)*, pp. 122–134, Springer-Verlag, Heidelberg, Germany.