An Analysis of UCT in Multi-Player Games

Nathan R. Sturtevant

Department of Computing Science, University of Alberta,
Edmonton, AB, Canada, T6G 2E8
nathanst@cs.ualberta.ca

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, when run in a multi-player game, 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.

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 [3]. 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. 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 get to 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 [6]. A second reason why multi-player games are difficult is because of opponent modeling. In two-player zero-sum games opponent modeling has never been shown to be necessary for high-quality play, while in multi-player games, opponent modeling is a necessity for robust play versus unknown opponents in some domains [9].

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 computes a mixed-strategy equilibrium in multi-player games and discuss the implications of this. Then, we analyze UCT's performance in a variety of domains, showing that it performs as well or better as the best previous approaches.

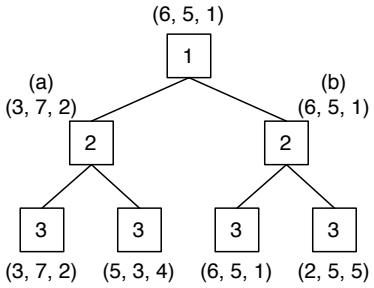

Fig. 1. A sample max^n tree

2 Background

The max^n algorithm [4] 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 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 Fig. 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 player's 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) to get 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 [8]. In practice it is 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 payoff, where the opponents' utility function is not 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 [1], you cannot guarantee that your opponents will play the best strategies possible. This might mean, for instance, that in Fig. 1 Player 2 has a different preference

at node (b). If Player 2 selects (2, 5, 5) at node (b), Player 1 should choose to move to the left from the root to get (3, 7, 2).

Two algorithms have been introduced that attempt to deal with imperfect opponent models. The soft-max^n algorithm [9] uses a partial ordering over game outcomes to analyze 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 [5] uses sets of opponent models and with probabilistic weights which are used for back-up at each node according to current opponent models. Both algorithms have learning mechanisms for updating opponent models during play. In the game of Spades, these approaches were able to mitigate the problems associated with an unknown opponent. We will discuss these results more in our experimental section.

2.1 UCT

UCT [3] is one of several recent Monte-Carlo-like algorithms proposed for game-playing. The algorithm plays games in two stages. In the first stage a tree is built and explored. The second stage begins when the end of the UCT tree is reached. At this point a leaf node 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/value returned by UCT converges on the same result as minimax.

3 Multi-Player UCT

The first question we address is what computation is performed by UCT on a multi-player game tree. For this analysis we assume that we are searching on a finite tree and that we can perform an 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 Fig. 1 we can look to see what UCT would compute on this tree. At node (a) Player 2 will always get 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) will

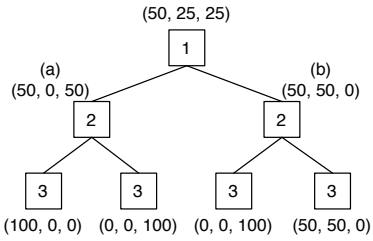

Fig. 2. Mixed equilibrium in a multi-player game tree

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 for the player at the root to move to the right towards node (b) to get an expected payoff of 4.

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.

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 unilaterally improve their payout. □

UCT will always compute an evenly mixed strategy at the leaves of the tree. But, there is no guarantee that in practice it will compute the expected mixed strategy at internal nodes in the tree. If ties are always broken from left to right, in Fig. 2 the value of node (a) for Player 1 will converge on 50 from above, while the value of node (b) for Player 1 will converge on 50 from below. As a result, depending on the number of samples and the value of C, UCT may not end up evenly mixing strategies higher up in the game tree. The max^n algorithm will never 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.

Table 1. 6 ways to arrange two player types in a 3-player game

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

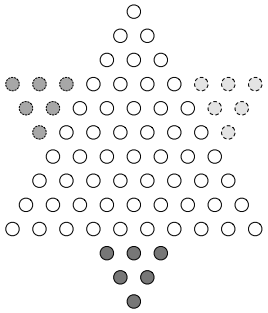

Fig. 3. A 3-player Chinese Checkers board

4 Experiments

Given the theoretical results presented in the last section, we run a variety of experiments to compare the performance of UCT to previous state-of-the-art programs in three domains. These experiments cover a variety of game types and give insight into the practical performance of UCT in multi-player games.

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 Table 1. Playing duplicate matches in this way makes it easier to measure statistical significance.

4.1 Chinese Checkers

Our first experiments are in the game of Chinese Checkers. A Chinese Checkers board is shown in Fig. 3. In this game the goal is to get your pieces across the board and into a symmetric position from the start state as quickly as possible. Chinese Checkers can be played with anywhere from two to six players. We

Table 2. Experiments in the game of Chinese Checkers

UCT (Random)	100	500	2500	10000	50000
Average Distance	9.23	4.21	2.08	1.95	2.29
Wins	1%	13%	43%	44%	34%
UCT (e-greedy)	100	500	2500	10000	50000
Average Distance	4.12	2.36	1.63‡	1.43†	0.90
Wins	14%	33%	42%	50%	73%

experiment with the three-player version here. Chinese Checkers can be played on different 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. This 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.

We used two versions of UCT for these experiments. For both versions, we disallowed backwards moves; otherwise the game will not progress and the randomly sampling fails. The first version of UCT uses purely random playouts. The second version uses an epsilon-greedy playout policy that takes the best greedy move (the one that moves a piece the farthest distance) with 95% probability, and plays randomly 5% of the time. We compared these policies by playing 100 games in each of the configurations from Table 1. (Thus there were 600 total games played with each player type playing 900 times.)

When comparing these policies, given 500 samples each, the epsilon-greedy policy won 81% of the games. When the game ended, the epsilon-greedy player was, on average, 0.94 moves away from the goal state. In contrast, the random-playout player was 2.84 moves away from the goal state. The player that wins is averaged in as 0 moves away from the goal state. These distances were computed using the $\max^n$ player's evaluation function.

A comparison of the two UCT players against the $\max^n$ player is found in Table 2. In these experiments C was fixed at 4.0, and we varied the number of samples that UCT was allowed, recording the percentage of wins against the $\max^n$ player as well as the average distance from the goal at the end of the game. The player with random playouts never won more than 44% of the games played. The epsilon-greedy player, however, performed on a par with the $\max^n$ player when given 10,000 samples, and beat it by a large margin when given 50,000 samples. In both of these situations, the UCT player was closer to the goal at the end of the game. All results are statistically significant with 99% confidence except for those marked † which are only significant at 95%. The results marked ‡ are not statistically significant.

It is interesting to note that the epsilon-greedy player's performance monotonically increases as we increase the number of playouts, but the random-playout player does not exhibit this same tendency. In particular, the player with 50k simulations plays markedly worse than the player with 10k simulations. It seems that this occurs because there are so many bad moves that can be made during random playout. Thus, the random playouts are not reflective of how the game will actually be played, and may lead towards positions which are likely to be won in random play, but not in actual play.

In these experiments the UCT player has a computation advantage over the max^n player, which only looks ahead 4-ply, although the max^n player has a significant amount of information available in the evaluation function. To make this more concrete, at the start of the game there are 17,340 nodes in the 4-ply tree given pruning in max^n, but there are 61 million nodes in the 7-ply tree. By the fourth move of the game there are 240 million nodes in the 7-ply tree. This is important because the only significant gains in performance in a three-player game come with an additional 3-ply of search, when the first player in the search can see an additional[1] move of his own moves ahead. On a 2.4Ghz Intel Core 2 Duo, we can expand 250k nodes per second, so a 7-ply search takes about 1000 seconds. Our UCT implementation plays about 900 games per second, so it takes about a minute to do 50k samples. Thus, while the UCT computation is more expensive than the max^n computation, the max^n player will not see an improvement in playing strength unless it is allowed significantly more time to search.

If we were looking to refine the player here, the database lookups used by the max^n player could be used as an endgame database by the UCT player and might further increase its strength or speed. Regardless, we have shown that UCT is quite strong in the game of Chinese Checkers, when given sufficient time for play.

4.2 Spades

Our next experiments are 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 [9, 5] 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 modeling. Then, we compare the prob-max^n algorithm to UCT.

[1] For brevity, we use 'he' and 'his' whenever 'he or she' and 'his or her' are meant.

These experiments were played open-handed so that all players could see other players' cards. In real play, we can sample many possible opponent hands and solve them individually to find the best move, as has been done in Bridge [2]. But, in our experiments, open-handed results have always been highly correlated with a sampling approach, so we only perform the open-handed experiments, which are much cheaper.

Each player was given 7 cards, and the game was played with 3 players, so the total depth of the tree was 21 ply. 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 Table 1. 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.

Experimental results for Spades are in Table 3. 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 just assume that the opponent is trying to make their bid, but nothing else. 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. The player who moves last has extra constraints on their possible

Table 3. UCT performance in Spades

	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%

Table 4. Detailed comparison of UCT and prob-max^n

	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

bids, meaning they may be forced to under- or over-bid. The player moving last also has less control over the game.

We break-down the results by player position in Table 4. Here, we can see what is happening more clearly. The 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 avoid playing poorly against prob-max^n when in the third position. We do not have a systematic explanation of this effect, but have noticed that these types of inconsistencies are common in multi-player games. We are working on further analysis.

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

4.3 Hearts - Shooting the Moon

Our third domain is the card game, Hearts. The goal of Hearts is to take as few points as possible. Like Spades, a game of Hearts is made up of multiple hands; a game ends when a player's score reaches or exceeds 100 points, and the player with the lowest score wins. We experiment on the 4-player version of Hearts here, which is most common. We play these games with all the cards face up, for the same reasons as in Spades.

One of the difficulties of Hearts is that there are two conflicting 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.

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. Out of 40,000 possibilities (10,000 games times 4 possible hands in each game) we found 3,244 hands in which a player might be able to shoot the moon. We then ran a set of different algorithms against this shooting player in the 3,244 hands. In Table 5 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 samples.

We experimented with a wide variety of UCT parameters here, and we summarize the results in the bottom part of Table 5. 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 behavior, 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

Table 5. Shooting the moon in Hearts

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

	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%

	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%

line of play is found. In some sense this can randomize play slightly without too large a penalty.

4.4 Hearts - Quality of Play

To test the quality of play, we played repeated 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 Table 6. All results are statistically significant with 99% confidence.

The current state-of-the-art players in Hearts use a learned evaluation function [10]. We trained three players to play Hearts using methods similar to, but more efficient than those described in [10]. 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.) The results were stable both for the different learned players and against UCT with up to 50,000 samples per turn; we report results against UCT with 5000 samples. For comparison, an older, hand-tuned, player [7] 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 Table 6 we experiment with different numbers of playouts. Here we played UCT with k playouts against UCT with $2k$ playouts. The player with more playouts consistently averages 4-5 points better than the player with fewer playouts. This shows that UCT does improve its performance as the number of simulations increases.

Table 6. Performance of UCT against various opponents

	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)

In all these experiments, UCT is doing purely random playouts. We experimented with various playout policies, but there were no simple policies which increased the strength of play. For instance, a simple policy for play is to play always the highest card that will not win the trick. As a game-playing policy, this is stronger than random play, but makes a poorer playout module than random.

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. Here is to remark that both players always beat the learned players. The learned players tested are not nearly as aggressive about attempting to shoot the moon as the player in the last section. This means that the 'insurance' paid to keep one's opponent from shooting the moon is less likely to payoff in these experiments.

5 A Summary of Findings and Future Work

This paper provides a foundation for future work on UCT in multi-player games. It shows theoretically that UCT computes a mixed-strategy equilibrium, unlike the traditional computation performed by $\max^n$. UCT, given a strong playout policy and sufficient simulations, 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 slightly better than existing programs in Hearts.

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

Additionally, 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 simple approaches discussed here, but the exact application of these techniques is still a matter of future research.

References

1. D. Billings. On the importance of embracing risk in tournaments. *ICGA Journal*, 29(4):199–202, 2006.
2. M. Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Articial Intelligence Research*, 14, 2001.
3. L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, pages 282–293. Springer-Verlag, 2006.
4. C. Luckhardt and K. Irani. An algorithmic solution of N-person games. In *AAAI-86*, volume 1, pages 158–162, 1986.
5. N. Sturtevant, M. Zinkevich, and M. Bowling. Probmaxn: Opponent modeling in n-player games. In *AAAI-2006*, pages 1057–1063, 2006.
6. N.R. Sturtevant. Last-branch and speculative pruning algorithms for $\max^n$. In *IJCAI-03*, pages 669–678, 2003.
7. N.R. Sturtevant. *Multi-Player Games: Algorithms and Approaches*. PhD thesis, Computer Science Department, UCLA, 2003.
8. N.R. Sturtevant. Current challenges in multi-player game search. In *Computers and Games (CG 2004)*, pages 285–300, 2006.
9. N.R. Sturtevant and M.H. Bowling. Robust game play against unknown opponents. In *AAMAS-2006*, 2006.
10. N.R. Sturtevant and A.M. White. Feature construction for reinforcement learning in hearts. In *Computers and Games (CG 2006)*, pages 122–134, 2007.