A Comparison of Algorithms for Multi-Player Games

Nathan Sturtevant
UCLA, Computer Science Department
Los Angeles, CA 90024
nathanst@cs.ucla.edu

Abstract. The max^n algorithm (Luckhardt and Irani, 1986) for playing multi-player games is flexible, but there are only limited techniques for pruning max^n game trees. This paper presents other theoretical limitations of the max^n algorithm, namely that tie-breaking strategies are crucial to max^n, and that zero-window search is not possible in max^n game trees. We also present quantitative results derived from playing max^n and the paranoid algorithm (Sturtevant and Korf, 2000) against each other on various multi-player game domains, showing that paranoid widely outperforms max^n in Chinese Checkers, by a lesser amount in Hearts and that they are evenly matched in Spades. We also confirm the expected results for the asymptotic branching factor improvements of the paranoid algorithm over max^n.

1 Introduction and Overview

Artificial Intelligence researchers have been quite successful in the field of two-player games, with well-publicized work in Chess (Deep Blue, IBM) and Checkers [1], and are producing increasingly competitive programs in games such as Bridge [2].

If we wish to have similar success in multi-player games, there is much research yet to be done. Many of the issues surrounding two-player games may be solved, but there are many more unanswered questions in multi-player games. The most basic of these is the question of which algorithm should be used for playing multi-player games.

Unfortunately, the question is not as simple as just which algorithm to use. Every algorithm is associated with other techniques (such as alpha-beta pruning or transposition tables) by which it can be enhanced. There are too many different techniques to cover here, but we have chosen a few algorithms and techniques as a starting point.

This paper considers the max^n [3] and paranoid [4] algorithms. In this paper we introduce and discuss theoretical limitations with each algorithm, and then present our results from playing these two algorithms against each other in various domains. Our results indicate that the paranoid algorithm is worth considering in a multi-player game, however the additional depth of search offered by the paranoid algorithm may not always result in better play. These results address games solely from a perfect-information standpoint, leaving the question of imperfect information for future research.

One popular multi-player game we will not address here is poker. Billings, et. al

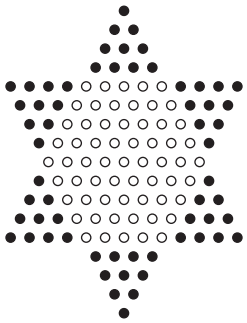

Figure 1: A Chinese Checkers board

[5] describe work being done on this domain. However, their focus is almost exclusively directed towards opponent modeling, with almost no search, while our focus is more on search.

2 Multi-Player Games: Hearts, Spades, and Chinese Checkers

To help make the concepts here more concrete, we chose two card games, Hearts and Spades, and the board game Chinese Checkers to highlight the various algorithms presented.

Hearts and Spades are both trick-based card games. Cards are dealt out to each player before the game begins[1]. The first player plays (*leads*) a card face-up on the table, and the other players follow in order, playing the same suit as lead if possible. When all players have played, the player who played the highest card in the suit that was led "wins" or "takes" the trick. He then places the played cards facedown in his discard pile, and leads the next trick. This continues until all cards have been played.

Hearts is usually played with four players, but there are variations for playing with two or more players. The goal of Hearts is to take as few points as possible. A player takes points when he takes a trick which contains point cards. Each card in the suit of hearts is worth one point, and the queen of spades is worth 13. At the end of the game, the sum of all scores is always 26, and each player can score between 0 and 26. If a player takes all 26 points, or "shoots the moon," he instead gets 0 points, and the other players all get 26 points each. These fundamental mechanics of the game are unchanged regardless of the number of players.

Spades can be played with 2-4 players. Before the game begins, each player predicts how many tricks they think they are going to take, and they then get a score based on how many tricks they actually do take. With 4 players, the players opposite each

[1] There are rules for trading cards between players, but they have little bearing on the work presented here.

other play as a team, collectively trying to make their bids, while in the 3-player version each player plays for themselves. More in-depth descriptions of these and many other multi-player games can be found in Hoyle et al [6].

Chinese Checkers is a perfect information game for 2-6 players. A Chinese Checkers board is shown in Figure 1. The goal of the game is to get 10 pegs or marbles from one's starting position to one's ending position as quickly as possible. These positions are always directly across from each other on the board. Pegs move by stepping to an adjacent position on the board or by jumping over adjacent pegs. One can jump over any player's pegs, or chain together several jumps, but pegs are not removed from the board after a jump.

2.1 Imperfect-Information Games

The paranoid and max^n algorithms, which we will cover next, are designed for perfect-information games such as Chinese Checkers. In order to use them with imperfect-information games such as Spades or Hearts, we must either modify the algorithms or modify the nature of the games we want to play. Both approaches have been used successfully in two-player games, but it remains to be seen how they can be applied successfully in multi-player games.

If, in a card game, we could see our opponents' cards, we would be able to use standard search algorithm to play the game. While in most games we don't know the exact cards our opponent holds, we do know the probability of our opponent holding any particular hand. Thus, we can create a hand that should be similar to what our opponent holds, and use a perfect-information algorithm to play against it.

The full expansion of this idea uses Monte-Carlo sampling. Instead of generating just a single hand, we generate a set of hands that are representative of the actual hand we expect our opponent to have. We then solve each of these hands using the standard minimax algorithm. When we have completed the analysis of each hand, we combine and analyze the results from each hand to produce our next play. As the play continues we update our models to reflect the plays made by our opponent. This is one of the techniques used to create a strong Bridge program. See [2] for a description of that work.

3 Multi-Player Game Algorithms

We review two multi-player game algorithms and their properties as a background for the theory and results found in later sections.

3.1 Max^n

The max^n algorithm [3] can be used to play games with any number of players. For two-player games, max^n simply computes the minimax value of a tree.

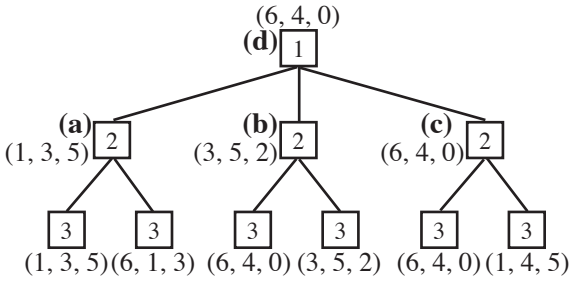

Figure 2: A 3-player max^n game tree

In a max^n tree with n players, the leaves of the tree are n-tuples, where the ith element in the tuple is the ith player's score. At the interior nodes in the game tree, the max^n value of a node where player i is to move is the child of that node for which the ith component is maximum. This can be seen in Figure 2. In this tree there are three players. At node (a), Player 2 is to move. Player 2 can get a score of 3 by moving to the left, and a score of 1 by moving to the right. So, Player 2 will choose the left branch, and the max^n value of node (a) is (1, 3, 5). Player 2 acts similarly at node (b) selecting the right branch, and at node (c) breaks the tie to the left, selecting the left branch. At node (d), Player 1 chooses the move at node (c), because 6 is greater than the 1 or 3 available at nodes (a) and (b).

One form of pruning, shallow pruning, is possible in a max^n tree. Shallow pruning refers to cases where a bound on a node is used to prune at the child of that node. To prune, we need at least a lower bound on each player's score, and an upper bound on the sum of all players scores. For a full discussion see [4] and [7]. This work shows that theoretically there are no asymptotic gains due to shallow pruning in max^n. This contrasts with the large gains available from using alpha-beta pruning with minimax. Another type of pruning, deep pruning, is not possible in max^n.

If a monotonic heuristic is present in a game, it can also be used to prune a max^n tree. The full details of how this occurs is contained in [4]. An example of a monotonic heuristic is the number of tricks taken in Spades. Once a trick has been taken, it cannot be lost. This guarantee can provide a lower bound on a player's score, and an upper bound on one's opponents scores.

In practice, almost no pruning occurs when using max^n to play Chinese Checkers. In Hearts, monotonic heuristic bounds on the scores can provide some pruning. In Spades, both monotonic heuristics and shallow pruning can be combined for more significant amounts of pruning.

3.2 Paranoid Algorithm

The lack of pruning in max^n has motivated research into other methods which might be

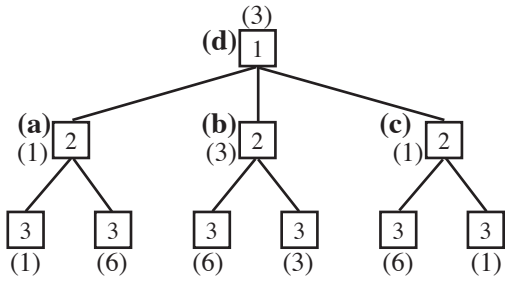

Figure 3: The paranoid version of the game tree in Fig. 2

able to search faster or deeper into a multi-player game tree. The paranoid algorithm does this by reducing a *n*-player game to a 2-player game. It assumes that a coalition of *n*-1 players have formed to play against the remaining player. We demonstrate this in Figure 3. In this tree Players 2 and 3 ignore their own scores, and simply try to minimize Player 1's score. At nodes (a) and (b), Player 2 makes the same choice as in Figure 1, but at node (c), Player 2 chooses the right branch. Then, at the root, (d), Player 1 chooses to move towards node (b) where he can get a score of 3.

If we search an *n*-player game tree with branching factor *b* to depth *d*, the paranoid algorithm will, in the best case, expand $b^{d(n-1)/n}$ nodes. [4] This is the general version of the best case bound for a two-player game, $b^{d/2}$ [8]. So, making the assumption that our opponents have formed a coalition against us should allow us to search deeper into a game tree, and therefore produce better play. Obviously as the number of players grows, the added efficiency of paranoid will drop. But, most multi-player games are played with 3-6 players, in which case paranoid can search 20-50% deeper.

4 Theoretical Properties of Max^n

Before we delve into the theoretical properties of multi-player algorithms, we return briefly to two-player games. For those familiar with game theory, one reason the mini-max algorithm is so powerful is because it calculates an equilibrium point and strategy for a given game tree. This strategy guarantees, among other things, some payoff *p*, regardless the strategy of the opponent. This statement is quite strong, and it allows us to, for the most part, ignore our opponents strategy. As we will see, we cannot make such strong statements about max^n.

4.1 Equilibrium Points in Max^n

It has been shown that, in a multi-player game, max^n computes an equilibrium point

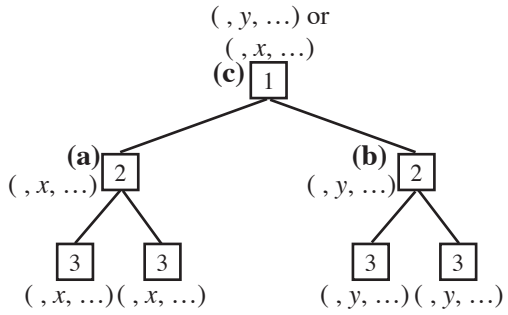

Figure 4: Tie breaking in a max^n game tree

[3]. But, the concept of an equilibrium point in a multi-player game is much weaker than that in a two-player game. In a multi-player game there are multiple equilibrium points that may have completely different equilibrium values and strategies. For those unfamiliar with game theory, it will suffice to understand that while there is a single minimax value for a two-player game tree, there are multiple possible max^n values for a multi-player game tree, which can all be different.

An example of this can be seen in Figure 2 at node (c). At this node Player 2 can make any choice, as either will lead to the same score for Player 2. But, if Player 2 chooses (1, 4, 5) as the max^n value of node (c), Player 1 will choose the result from node (b), (3, 5, 2) to be the max^n value of the tree. These are both valid, but different, equilibrium points in the tree. In a two-player game, since tie-breaking cannot affect the minimax value of the tree, ties are broken in favor of the left-most branch, allowing any additional ties to be pruned. However, in a multi-player game we cannot do this.

Lemma 1. In a multi-player game, changing the tie-breaking rule may arbitrarily affect the max^n value of the tree.

Proof: Figure 4 contains a generic max^n tree. We have represented Player 2's possible scores by x and y. The scores for the other players can obviously be arbitrarily affected by the way we break the ties for Player 2. By adjusting the tie breaking, we can also change whether Player 1 moves to the left or right from the root, and so can affect whether Player 2's score will be x or y. □

This result doesn't mean that max^n is a worthless algorithm. A game tree with no ties will have a single max^n value. In addition, each possible max^n value that results from a particular tie-breaking rule, will play reasonably, given that all players use that tie-breaking rule.

The choice of a tie-breaking rule amounts to a strategy for play. In Hearts, for instance, good players will often save the queen of spades to play on the player with the best score. Thus, we must consider our opponents strategy.

We illustrate the implications of lemma 1 in Figure 5. Each player holds 2 cards, as indicated, and three possible outcomes of the play are shown. The winning card of each trick is underlined. If cards are played from left to right in your hand by default,

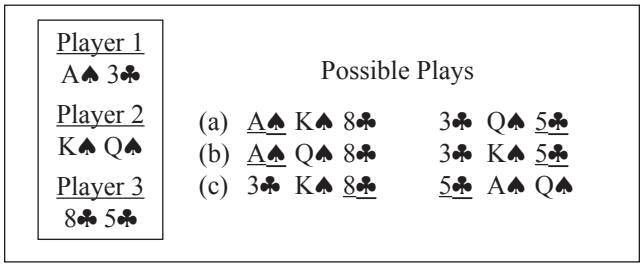

Figure 5: Tie breaking situation

Player 1 can lead the A♠, and Player 2 will not drop the Q♠, as in play (a). However, if Player 2 breaks ties differently, this could be a dangerous move, resulting in Player 1 taking the Q♠, as in (b). But, if Player 2 leads the 2♣, as in (c), Player 3 will be forced to take the Q♠.

The tie-breaking rule we have found most effective in such situations has been to assume that our opponents are going to try to minimize our score when they break ties. This obviously has a flavor of the paranoid algorithm, and it will cause us to try and avoid situations where one player can arbitrarily change our score. From our own experience, we believe that this is similar to what humans usually do when playing Hearts.

4.2 Zero-Window Search

Zero-window search [9] originates from two-player games. The idea behind zero-window search is to turn a game tree with a range of evaluations into a tree where every leaf terminates with a win or a loss. This is done by choosing some value v, and treating a terminal node as a win if its evaluation is $> v$, and as a loss if it is $\leq v$. Combining this approach with a binary search will suffice to find the minimax value of a game tree to any precision. This assumption results in highly optimized searches that can prune away most of the game tree. Zero-window search is one of the techniques that helps make partition search [10] efficient, and has played a large part in the success of that domain.

While there are limitations on pruning during the calculation of the $\max^n$ value of a tree, it is not immediately obvious that we cannot somehow prune more if we just try to calculate the bound on the $\max^n$ value of a tree, instead of the actual $\max^n$ value.

So, we could attempt to search a $\max^n$ game tree to determine whether the nth player will be able to get at least a score of v. Supposing at the root of a tree Player 1 gets v points. We can obviously stop at that point knowing that Player 1 will get $\geq v$ points. This is a bound derived from shallow $\max^n$ pruning, and at most it will reduce our search size from b^d to b^{d-1}. In addition, this will not combine well with a technique

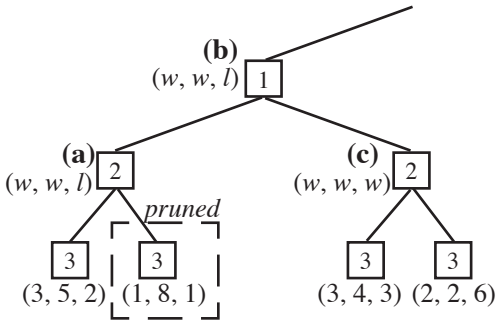

Figure 6: Finding bounds in a max^n game tree

like partition search. We would like to know if this idea can be applied to every node in the tree.

Suppose we want to consider all scores > 2 as a win. We illustrate this in Figure 6. Since Player 2 can get a score of 5 by moving to the left at node (a), Player 2 will prune the right child of (a), and return a max^n value of (w, w, l), where w represents a win and l represents a loss. However, at node (b), Player 1 would then infer that he could win by moving towards node (a). But, the exact max^n value of (a) is (1, 8, 1), and so in the real game tree, Player 1 should prefer node (c) over node (a).

So, the only bounds we can get on the max^n value of a tree are those that come from a search with shallow pruning, which, given no additional constraints, is already optimal.

[7] shows that "Every directional algorithm that computes the max^n value of a game tree with more than two players must evaluate every terminal node evaluated by shallow pruning under the same ordering." We can now expand this statement:

Theorem 1. Given no additional constraints, every directional algorithm that computes either the max^n value or a bound on the max^n value of a multi-player game tree must evaluate every terminal node evaluated by max^n with shallow pruning under the same ordering.

Proof: [7] has already shown that shallow pruning is optimal in its computation of the max^n value of a tree. If we replace backed-up max^n values in a game tree with just bounded max^n values, there may be ties between moves that were not ties in the original max^n tree, such as at node (a) in Figure 6. By definition, all possible moves when breaking ties must lead to equilibrium points for the sub-tree below the node. Since some of these moves may not have been equilibrium points in the original tree, any bound computed based on these moves may not be consistent with the possible max^n values for the original tree. □

5 Equilibrium Points in the Paranoid Algorithm

Table 1. The six possible ways to assign paranoid and max^n player types to a 3-player game

	Player 1	Player 2	Player 3
1	max^n	max^n	paranoid
2	max^n	paranoid	max^n
3	max^n	paranoid	paranoid
4	paranoid	max^n	max^n
5	paranoid	max^n	paranoid
6	paranoid	paranoid	max^n

While the paranoid algorithm may be a pessimistic approach to game playing, it is not entirely unreasonable, and it offers a theoretical guarantee not offered by max^n.

Equilibrium points calculated by the paranoid algorithm are theoretically equivalent to those calculated by minimax. This means that a paranoid game tree has a single paranoid value which is the guaranteed lower bound on one's score.

This may, however, be lower than the actual score achievable in the game, because it is unlikely that your opponents have truly formed a coalition against you. This leads to moves which are suboptimal, because of the unreasonable assumptions being made. In Hearts, for instance, paranoid may find there is no way to avoid taking a trick with the queen of spades. But, instead of forcing the opponents to work as hard as possible to make this happen, the paranoid algorithm may select a line of play that causes this to happen immediately. This is no different from the problem faced in two-player games. In [1], for instance, they discuss the difficultly Chinook faced with arbitrating between two lines of play that both lead to a draw, where on one path a slight mistake by the opposing player could lead to a win by Chinook, while the other leads clearly to a draw.

6 Experimental Results

Given these theoretical results, we would like to see what we can do in practice to play multi-player games well. To this end, we have run a number of experiments on various games to see how max^n and paranoid perform against each other. We have written a game engine that contains a number of algorithms and techniques such as the paranoid algorithm, max^n, zero-window search, transposition tables and monotonic heuristic pruning. New games can easily be defined and plugged in to the existing architecture without changing the underlying algorithms.

We first present the general outline of our experiments applicable to all the games, and then we will present the more specific details along with the results.

Our game engine supports an iterative deepening search process, but each particular game can choose the levels at which the iterations take place. Search can be bounded by time, nodes expanded, or search depth. In our experiments, we gave the algorithms a node limit for searching. When the cumulative total of all iterative searches expands more nodes than the limit, the search is terminated, and the result from the last completed iteration is returned.

We are using 3, 4, and 6-player games to compare 2 different algorithms. In a 3-player game, there are $2^3 = 8$ different ways we could assign the algorithm used for each player. However, we are not interested in games that contain exclusively max^n or exclusively paranoid players, leaving 6 ways to assign each player to an algorithm. These options are shown in Table 1. So, we ran most experiments 6 times, one time with each distribution in Table 1. For card games, that means that the same hand is played 6 times, once with each possible arrangement of cards. For Chinese Checkers, this varies who goes first, and what player type goes before and after you.

Similarly, in a 4-player game there are $2^4 = 16\text{-}2 = 14$ ways to assign player types, and in a 6-player game there are $2^6 = 64\text{-}2 = 62$ ways to assign player types.

For max^n, we implemented a tie-breaking rule that assumes our opponents will break ties to give the player at the root the worst possible score. Without this tie-breaking rule, max^n plays much worse.

6.1 Chinese Checkers

To simplify our experiments in Chinese Checkers, we used a slightly smaller board than is normally used. On the smaller board each player has 6 pieces instead of the normal 10 pieces. Despite this smaller board, a player will have, on average, about 25 possible moves (in the 3-player game), with over 50 moves available in some cases.

Besides reducing the branching factor, this smaller board also allowed us to create a lookup table of all possible combinations of a single players pieces on the board, and an exact evaluation of how many moves it would take to move from that state to the goal. The table is the solution to the single-agent problem of how to move your pieces across the board as quickly as possible. This makes a useful evaluation for the two-player version of Chinese Checkers. However, as additional players are added to the game, this information becomes less useful, as it doesn't take into account the positions of one's opponents on the board. It does have other uses, such as measuring the number of moves a player would need to win at the end of the game.

Because only one player can win the game, Chinese Checkers is a zero-sum, or constant-sum game. However, within the game, the heuristic evaluation is not constant-sum. Our heuristic evaluation was based on the distance of one's pieces from the goal. This means that we cannot use any simple techniques to prune the max^n tree. This combined with the large branching factor in the game makes max^n play Chinese Checkers rather poorly.

In our 3-player experiments we played 600 games between the max^n and paranoid

Table 2. Chinese Checkers statistics for max^n and paranoid

		Paranoid	Max^n
3-player 250k nodes	games won	60.6%	39.4%
	moves away	3.52	4.92
	search depth	4.9	3.1
4-player 250k nodes	games won	59.3%	40.7%
	moves away	4.23	4.73
	search depth	4.0	3.2
6-player 250k nodes	games won	58.2%	41.8%
	moves away	4.93	5.49
	search depth	4.6	3.85

algorithms. To avoid having the players repeat the same order of moves in every game, some ties at the root of the search tree were broken randomly. We searched the game tree iteratively, searching one level deeper in each successive iteration.

We report our first results at the top of Table 2. We played 600 games, 100 with each possible configuration of players. If the two algorithms played evenly, they would each win 50% of the games, however the paranoid algorithm won over 60% of the games it played.

Another way to evaluate the difference between the algorithms is to look at the state of the board at the end of the game and measure how many moves it would have taken for each player to finish the game from that state. When tabulating these results, we've removed the player who won the game, who was 0 moves away from winning. The paranoid player was, on average, 1.4 moves ahead of the max^n player.

Finally, we can see the effect the paranoid algorithm has on the search depth. The paranoid player could search ahead 4.9 moves on average, while the max^n player could only look ahead 3.1 moves. This matches the theoretical predictions made in section 3.2; Paranoid is able to look ahead about 50% farther than max^n.

We took the same measurements for the 4-player version of Chinese Checkers. With 4 players, there are 14 configurations of players on the board. We played 50 games with each configuration, for a total of 700 games. The results are in the middle of Table 2. Paranoid won 59.3% of the games, nearly the same percentage as in the 3-player game. In a 4-player game, paranoid should be able to search 33% farther than max^n, which these results confirm, with paranoid searching, on average, 4-ply into the tree, while max^n was able to search 3.2-ply on average. Finally, the paranoid players that didn't win were 4.23 moves away from winning at the end of the game, while the max^n

Table 3. 3-Player Chinese Checkers statistics for max^n and paranoid

		Paranoid	Max^n
250k nodes, fixed branching factor	games won	71.4%	28.6%
	moves away	2.47	4.4
	search depth	8.2	5.8
fixed depth search	games won	56.5%	43.5%
	moves away	3.81	4.24

players were 4.73 moves away. In the 4-player game some players share start and end sectors, meaning that a player can block another player's goal area, preventing them from winning the game. This gave max^n a chance to get closer to the goal state before the game ended.

In the 6-player game, we again see similar results. We played 20 rounds on each of 64 configurations, for 1280 total games. Paranoid won 58.2% of the games, on average 4.93 moves away from the goal state at the end of the game, while max^n was 5.49 moves away on average. In the 6-player game, we expect paranoid to search 20% deeper than max^n, and that is the case, with max^n searching 3.85 moves deep on average and paranoid searching 4.6 moves on average.

The max^n algorithm has an extremely limited search, often not even enough to look ahead from its first move to its second. This means that, although max^n can, in theory, use transposition tables in Chinese Checkers, it is unable to in practice because it cannot even search deep enough to cause a transposition to occur.

Because of this, we conducted another experiment with the 3-player games. In this experiment we again played 600 total games, limiting the branching factor for each algorithm, so that only the 6 best moves were considered at each branch. We chose to limit the branching factor to 6 moves because this will allow reasonable depth searches without an unreasonable limitation on the possible moves. If we limited the branching factor to just 2 moves, there wouldn't be enough variation in moves to distinguish the two algorithms.

The results from these experiments are found in Table 3. Under these conditions, we found that paranoid did even better than max^n, winning 71.4% of all the games even though max^n was able to search much deeper than in previous experiments. The paranoid algorithm could search 8.2 moves deep as opposed to 5.8 for max^n. At the end of the game, paranoid was, on average, only 2.47 moves away from finishing, as opposed to 4.4 for max^n.

Finally, we played the algorithms against each other with a fixed depth search. In this experiment, both algorithms were allowed to search 4-ply into the tree, regardless of node expansions. In these experiments the paranoid algorithm again was able to

Table 4. Games won in Spades and Hearts by max^n and paranoid

		Paranoid	Max^n
3-player Hearts, 250k Nodes	average score	8.1	8.9
	search depth	15.2	11.0
	vs. heuristic	5.6	5.6
4-player Hearts, 250k Nodes	average score	6.45	6.55
	search depth	14.3	11.2
	vs. heuristic	4.3	4.2
Spades, 250k Nodes	average score	5.67	5.67
	search depth	15.4	10.6
	vs. heuristic	6.06	6.12

outperform the max^n algorithm, albeit by lesser margins. Paranoid won 56.5% of the games played, and was 3.81 moves away at the end of the game, as opposed to 4.24 moves for max^n.

These results show that the paranoid algorithm is winning in Chinese Checkers both because it can search deeper, and because its analysis produces better play. We would expect similar results for similar boards games.

6.2 Perfect Information Card Games

For the card games Hearts and Spades we deal a single hand and then play that same hand six times in order to vary all combinations of players and cards. If max^n and paranoid play at equal strength, they will have equal scores after playing the hand 6 times. For both games we used a node limit of 250,000 nodes per play. These games were played openly allowing all players to see all cards.

For the 3-player games of Hearts and Spades we played 100 hands, 6 times each. In Hearts we also run experiments with the 4-player version of the game. For the 4-player game we also used 100 hands, played 14 times each for arrangement of players, for 1400 total games. Our search was iterative, as in Chinese Checkers. But, since points are only awarded when a trick is taken, we didn't search to depths which ended in the middle of a trick. We used a hand-crafted heuristic to determine the order that nodes were considered within the tree. This heuristic considered things like when to drop trump in Spades, and how to avoid taking the Queen of Spades in Hearts.

6.3 Hearts

The top of Table 4 contains the results for Hearts. Over these games, the paranoid player had an average score of 8.1 points, while the max^n player had an average score of 8.9 points. The standard deviation of the scores was 1.3, so these results are close, but paranoid has a definite advantage. There are about 26 points available in the game (in the 3-player version, one card is taken out of the deck randomly), so if the algorithms played with equal strength, they would have averaged about 8.5 points each. The paranoid algorithm could search depth 15.2 on average, while the max^n algorithm could only search to depth 11.0 on average. The paranoid algorithm is searching close to 50% farther in the tree than max^n, as expected for a 3-player game.

To compare both algorithms against a different standard, we also set up a different experiment that pitted each algorithm against a player that did no search, but just used the node-ordering function to pick the next move. Max^n and paranoid were allowed to search just 6-ply (2 tricks) into the game tree, but only took 5.6 points in an average game. This confirms that the search is doing useful computation and discovering interesting lines of play above and beyond what a simple heuristic can do.

In the 4-player game the algorithms are more closely matched. Paranoid did just slightly better, averaging 6.45 points per hand as opposed to 6.55 for max^n. The standard deviation per round was .67 points. Paranoid was able to search depth 14.3 on average, about 33% farther than max^n, which could search depth 11.2. In this case, it seems that the extra search depth allowed by paranoid is being offset by the (incorrect) assumption that our opponents have formed a coalition against us. Both players were again able to easily outperform the heuristic-based player, scoring, on average, just over 4 points per game.

6.4 Spades

In the actual game of Spades, players bid on how many tricks they are going to take, and they then play out the game, attempting to take exactly that number of tricks. We have experimented with the first phase of that process, attempting to ascertain how many tricks can be taken in a given hand, and we do this by playing out a game, trying to take as many tricks as possible.

The bottom of Table 4 contains the results. Over these games, both players had an average score of 5.67 points, which is what we expect for algorithms of equal strength, given the 17 points (tricks) in the 3-player game. However, the paranoid algorithm is searching 15.4-ply deep, on average, while max^n is only searching 10.6-ply deep. This means that paranoid can look ahead nearly 2 tricks more than max^n, about 50% deeper.

This is an interesting result because the paranoid algorithm is able to search deeper than max^n by a wider margin than in it can in Hearts, but it is still not able to outperform max^n. However, looking at the results from play against a heuristic player makes this more clear. When max^n and paranoid played a heuristic based player that does no search, they were barely able to outperform it. This indicates one of two things: Either

the process of estimating how many tricks can be taken in a three-player game is very simplistic, or neither algorithm is able to use its search to devise good strategies for play.

We expect that both are happening. In games like Bridge, a lot of work is spent coordinating between the partners in the game. With no partners in 3-player spades, there is little coordination to be done, and the tricks are generally won in a greedy fashion. Our own experience playing against the algorithms ourselves also lead us to this conclusion. Often times the only decision that can be made is how to break ties in giving points to the other players.

Note that these results are only for the bidding portion of the game. The scoring of the actual game play is somewhat different, with players trying to make their bid exactly. This changes the rules significantly, and may give one algorithm an edge.

7 Conclusion and Future Work

Our results show that the paranoid algorithm should be considered as a possible multi-player algorithm, as it is able to easily outperform the max^n algorithm in Checkers and slightly less so in Hearts. However, our current results from Hearts and Spades indicate that the max^n algorithm is more competitive than we expected in these domains.

As a general classification, we can conclude that in games where max^n is compelled to do a brute-force search, the paranoid algorithm is a better algorithm to use, while games in which the max^n algorithm can prune result in more comparable performance. Another factor that differs between Chinese Checkers and card games is that in card games your opponents can easily team up and work together. This is more difficult in Chinese Checkers, because if they block one piece you can simply move another, and the limited search depth prevents the formation of more complicated schemes. This makes the paranoid assumption more reasonable.

There are several areas into which we are directing our current research. First, we need to consider the development of additional algorithms or pruning techniques that will result in improved performance. We are also working on optimizing the techniques and algorithms we have already implemented. Next, and more importantly, the issue of imperfect information needs to be addressed. Paranoid and max^n played similarly in card games, which are always games of imperfect information. As there is no clear edge for either algorithm, the point that will differentiate these algorithms is how well they can be adapted for games of imperfect information. Finally, we need to concern ourselves not just with play against different algorithms, but with play against humans. We are in the beginning stages of attempting to write a Hearts program that is stronger than any existing program, and one that also plays competitively against humans.

Acknowledgements

We wish to thank Rich Korf for his guidance in developing this work, Haiyun Luo and Xiaoqiao Meng for the use of their extra CPU time to run many of our experiments, and the reviewers for their valuable comments and suggestions.

References

1. Schaeffer, J., Culberson, J., Treloar, N., Knight, B., Lu, P., Szafron, D. A world championship caliber checkers program, Artificial Intelligence, vol.53, (no.2-3), Feb. 1992.
2. Ginsberg, M., GIB: Imperfect Information in a Computationally Challenging Game, Journal of Artificial Intelligence Research, Volume 14, 2001, 303-358.
3. Luckhardt, C. Irani, K., An algorithmic solution of N-person games, Proceedings AAAI-86, Philadelphia, PA, 158-162.
4. Sturtevant, N., and Korf, R., On Pruning Techniques for Multi-Player Games, Proceedings AAAI-00, Austin, TX, 201-207.
5. Billings, D., Peña, L., Schaeffer, J., Szafron, D., Using Probabilistic Knowledge and Simulation to Play Poker, AAAI-99, 697-703.
6. Hoyle, E., and Frey, R.L., Morehead, A.L., and Mott-Smith, G, 1991, *The Authoritative Guide to the Official Rules of All Popular Games of Skill and Chance*, Doubleday.
7. Korf, R., Multiplayer Alpha-Beta Pruning. Artificial Intelligence, vol. 48 no. 1, 1991, 99-111.
8. Knuth, D., and Moore, R., An Analysis of Alpha-Beta Pruning, Artificial Intelligence, vol. 6 no. 4, 1975, 293-326.
9. Pearl, J, Asymptotic properties of minimax trees and game-searching procedures. Artificial Intelligence vol 14 no 2, 1980, 113-138.
10. Ginsberg, M, Partition search, AAAI-96, Cambridge, MA, 228-33.