Prob-Max^n: Playing N-Player Games with Opponent Models

Nathan Sturtevant and **Martin Zinkevich** and **Michael Bowling**
Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada T6G 2E8
{nathanst, maz, bowling}@cs.ualberta.ca

Abstract

Much of the work on opponent modeling for game tree search has been unsuccessful. In two-player, zero-sum games, the gains from opponent modeling are often outweighed by the cost of modeling. Opponent modeling solutions simply cannot search as deep as the highly optimized minimax search with alpha-beta pruning. Recent work has begun to look at the need for opponent modeling in n-player or general-sum games. We introduce a probabilistic approach to opponent modeling in n-player games called prob-max^n, which can robustly adapt to unknown opponents. We implement prob-max^n in the game of Spades, showing that prob-max^n is highly effective in practice, beating out the max^n and soft-max^n algorithms when faced with unknown opponents.

Introduction and Background

Researchers have often observed deficiencies in the minimax algorithm and its approach to game playing. Russell and Norvig (1995), for instance, gave a prominent example of where minimax play can be flawed through slight errors in the value of leaf positions. Others have shown that minimax search can be pathological, returning less accurate results as search depth increases (Beal 1982; Nau 1982). While new algorithms have been designed for better analysis of games (Russell & Wefald 1991; Baum & Smith 1997) or for opponent modeling (Carmel & Markovitch 1996) these approaches have not been widely used in practice. There are a variety of reasons for this, but the primary one seems to be that minimax with alpha-beta pruning is simple to implement and adequate for most analysis.

In this paper we turn the research focus from two-player, zero-sum games to n-player, general-sum games. Much less research has gone into this area, but problems in this domain are much more suitable for incorporating additional information such as opponent models. We extend the results in our previous work (Sturtevant & Bowling 2006), which showed that opponent modeling is needed for n-player games by introducing prob-max^n. Prob-max^n is a search algorithm in the tradition of max^n but makes use of probabilistic models of the opponents in the search. We also show how the probabalistic models can form the basis for learning models during play, through Bayesian inference. In the game of Spades we demonstrate that prob-max^n is superior to existing approaches.

Opponent Modeling Algorithms

Early work in opponent modeling focused on the problem of recursive modeling (Korf 1989; Iida *et al.* 1993a; 1993b). While this early work is interesting, it has not made its way into use by current game-playing programs. Carmel and Markovitch (1996), for instance, look at the performance of a checkers program using opponent modeling. But, CHINOOK, which is considered the best program in this domain, does not use explicit opponent modeling. Instead, it relies on other techniques to achieve high performance. Donkers and colleagues (2001) take a more probabilistic approach to opponent modeling which is somewhat similar to the approach we take in this paper. We will address these differences after we have presented our new work.

We believe that one reason these approaches haven't found success in practice is because they have been applied to two-player, zero-sum games. From a practical and theoretical point of view these games are much easier than general-sum games, and thus there is much less of a need to model one's opponent. We demonstrate a domain where, even given a perfect evaluation function (we search to the end of the game tree), we need to take into account a model of our opponent.

Motivating Example: Spades

Spades is a card game for two or more players. For this research, we consider the three-player version of the game, where there are no partnerships. The majority of the rules in Spades are not relevant for this work, and there are any number of other games, such as Oh Hell, which have similar properties to Spades. We will only cover the most relevant rules of the game here.

Each game of Spades is broken up into a number of hands, which are played as independent units. Hands are further broken up into tricks. Before a hand begins each player must predict, in the form of a bid, how many tricks they expect to take in the following hand. Scores are determined according to whether players make their bids or not. If a player doesn't take as many tricks as they bid, they get a score of $-10\times$*bid*. If they take at least as many tricks as they bid they get $10\times$ *bid*. The caveat is that the number of tricks taken over a player's bid (overtricks) are also tallied, and when, over the course of a game, a player takes 10 overtricks, they lose 100 points. Thus, the goal of the game is to make your bid without taking too many overtricks.

Spades is an imperfect information game because players are not allowed to see their opponents cards. One common approach to playing imperfect-information games is to use Monte-Carlo sampling to generate perfect-information hands which can then be analyzed. While there are some drawbacks to this approach, it has been used successfully in domains like Bridge (Ginsberg 2001). Because this approach works well, we focus our new work on the perfect-information game and all experiments in this paper are played with open hands. meaning that players can see each other's cards.

Importance of Modeling

To help motivate this paper we present some previous results from the game of Spades without explaining the full details of how the experiments were set up and run. These details will be duplicated for our current experiments and are covered in the experimental results section of this paper. The trends shown here motivate the practical need for this line of research. Specifically, we consider two different "player types", defined by their utility function over game outcomes. The first player type, called mOT, tries to minimize overtricks. The second player type, called MT, tries to simply maximize tricks. When doing game tree search, we must have a model of our opponents. In two-player zero-sum games we normally assume that our opponent is identical to ourselves. Recent experiments (Sturtevant & Bowling 2006) have shown that this approach is not robust in n-player games.

Consider what happens when these two player types compete, where they both have correct opponent models. That is, the mOT players knows which opponents are maximizing tricks, and the MT players knows which opponents are minimizing overtricks. In this case it is not surprising that an mOT player wins nearly 75% of the games against MT players. What is surprising is that, if each player instead assumes their opponents have the same strategy that they do, an mOT player then only wins 44% of the games.

These results are not due to uncertainty in heuristic evaluation: all game trees are searched exhaustively. Instead, there is a fundamental issue of opponent modeling. In 3-player Spades we cannot blindly assume that our opponents employ our same utility function, without potentially facing disastrous results. This is in distinct contrast to the very successful use of this principle in two-player, zero-sum games.

Multi-Player Game-Tree Search

The first game-tree search algorithm proposed for n-player games was max^n.

Max^n

Max^n (Luckhardt & Irani 1986) is the generalization of minimax to any number of players, while in a two-player, zero-sum game it will return the same result as minimax. The values at the leaves of a max^n tree (*max^n values*) are n-tuples, where the ith value in the tuple corresponds to the score or utility of a particular outcome for player i. The max^n value of a node where player i is to move is the value of the child node for which the ith component is maximal. In the case of a tie, any outcome may be selected.

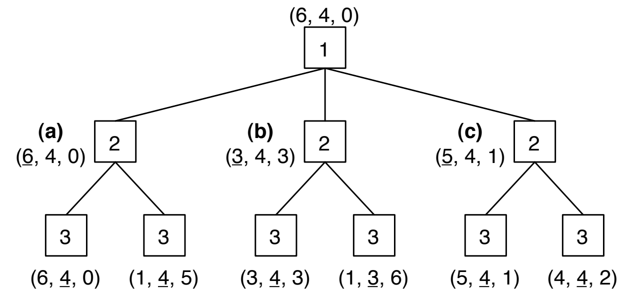

Figure 1: An example max^n tree.

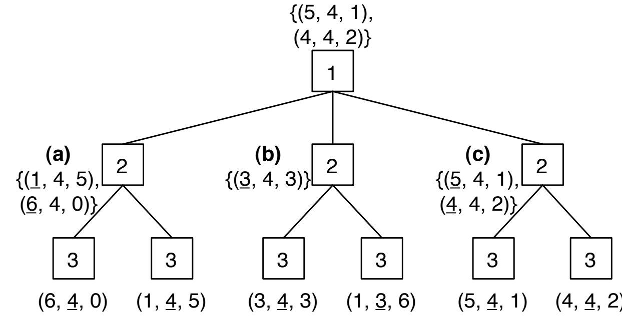

Figure 2: An example soft-max^n tree.

Figure 1 demonstrates the max^n algorithm. Each node in the tree is a square, inside of which is the player to move at that node. At node (a) Player 2 can choose between two outcomes, (6, 4, 0) and (1, 4, 5). Because Player 2 gets 4 from either choice we arbitrarily break the tie to the left and return the value (6, 4, 0). At node (b) Player 2 will choose (3, 4, 3) to get 4, instead of (1, 3, 6) to get 3. Player 2 also has a tie at node (c), and chooses the value (5, 4, 1). At the root of the tree Player 1 chooses the left branch to get (6, 4, 0), the final max^n value of the tree.

If all players use max^n to search a game tree, and all leaf values are known, the resulting strategies will be in equilibrium, meaning that no player can do better by changing their strategy. But, this analysis doesn't provide a worst case guarantee. A player, for instance, may be able to change their strategy in a way that decreases another player's score without causing their own score to decrease. In fact, mistaken analysis at even a single node of a max^n tree can arbitrarily effect the payoff of the resulting strategy (Sturtevant 2004).

Soft-Max^n

The soft-max^n algorithm (Sturtevant & Bowling 2006) addresses many of the shortcomings of max^n. At the simplest level it avoids trying to predict how ties will be broken. When a tie is encountered in a soft-max^n tree, instead of choosing a single value to return, a set of values (a *max^n set*) is returned instead. This set of values represents the possible outcomes that could be chosen if one were to play down a particular branch of a tree.

We use the same tree from Figure 1 to demonstrate soft-max^n in Figure 2. The max^n value at node (b) is computed in the same manner as in max^n. But, at nodes (a) and (c) we form max^n sets containing both possible outcomes at those nodes, because Player 2 is indifferent between the outcomes. This allows Player 1 to make a more informed decision at the root of the tree. If, for instance, Player 1 just needs 3 points to win, moving towards (c) will guarantee a win. If Player 1 needs 6 points to win, Player 1 can choose to move towards node (a), the only possible move that will lead to a win.

This simple explanation of soft-max^n omits some important details. In practice, the utilities for a game should also be modified for a soft-max^n search. If we are not certain that an opponent prefers one outcome to another, we should not guess or arbitrarily predict how that opponent will act, but instead consider the specific outcomes to be ties. More precisely, soft-max^n can be implemented given a partial-ordering function for values in the game tree. Whenever the children of a node do not have a distinct maximal value due to the partial ordering, a max^n set will be backed up instead of a single value.

Performance

Soft-max^n's performance in Spades are reported in the experimental results section. The summary of these results is that soft-max^n provides a reasonable gain in winning percentage over using plain max^n. The main message to be understood from these results is that mistaken assumptions regarding how one's opponents are going to play can have a strong adverse effect on performance in practice. It is much safer to use a generic opponent model than to make overly strong assumptions about an opponent.

There are a few drawbacks to soft-max^n which we address in this paper. First, the number of outcomes in any soft-max^n set can grow, at least in theory, to the size of the number of leaves in the game tree. This may not be a drawback in some domains, such as Spades, because the number of unique leaf-values in the game tree is asymptotically smaller than the size of the game tree, but it is always a potential issue.

A related, and more important, drawback is that soft-max^n does not clearly specify how the player at the top of the tree should decide between the moves available. There is no associated information with the returned values that specifies how often they occur in the game or how likely we think we are to receive any of those possible outcomes when playing on a given branch of the tree.

Finally, while an inference method for learning soft-max^n opponent models through play has been proposed (Sturtevant & Bowling 2006), this inference mechanism is brittle. It requires that our opponents play exactly according to one of our models. If this is not the case we will be forced to use the fully generic opponent model. Thus, to improve upon soft-max^n we propose a new algorithm, prob-max^n.

Prob-Max^n

Prob-max^n is similar to soft-max^n in that we want to return information from multiple children of a node, instead of just from the single maximal child. In essence we would just like to add probabilities to a soft-max^n tree. However, instead of adding probabilities to each outcome within a soft-max^n set, we are going to maintain utilities of models. The number of models used will likely be much smaller than the number of outcomes possible in the game.

First, for each player i, we have some set of N opponent models $m_{i,1} \dots m_{i,N}$. A model for an opponent consists of a utility function over outcomes. Like the vector of utilities in max^n, we will maintain a utility matrix u, such that $u[i,j]$ is the utility for player i under model $m_{i,j}$. At terminal nodes, $u[i,j]$ is determined using the utility function of $m_{i,j}$. Consider an internal node in a game tree where the set of children is C. We will use a new update rule to compute the utility of this node. At each node in the game tree, we will determine the probability, $\mathrm{probChoice}[c]$, that the player to move at that node selects any given choice $c \in C$. Recursively, we determine the utility of each choice such that $\mathrm{utilityOfChoices}[c][i,j]$ is the utility for player i under model $m_{i,j}$ given that choice c is made. Then we compute $u[i,j]$ of the current node to be:

$$u[i,j] = \sum_{c \in C} \mathrm{probChoice}[c]\ \mathrm{utilityOfChoices}[c][i,j] \quad (1)$$

In other words, this is the expected utility. It is simply a weighted sum of the utility matrices of the children. What is left is to define $\mathrm{probChoice}[c]$. Suppose that i_{current} is the player to move at a given node in the game tree. Then, like max^n, we find the optimal choice(s) for the current player i_{current}. However, each of player i_{current}'s models $m_{i,1} \dots m_{i,N}$ has its own preference with regards to the optimal choices. To combine the models, we consider our global belief, $\mathrm{probModel}[i,j]$, that player i is playing with model j, for each $m_{i,j}$ (so $\sum_{j=1}^{N} \mathrm{probModel}[i,j] = 1$). We assume each model is ϵ-greedy, in the sense that it will assign ϵ probability uniformly over all choices, and $1-\epsilon$ probability uniformly over the optimal choices for $m_{i,j}$. This allows us to anticipate possible deviations from our model. If $B \subseteq C$ (the "best" choices) are the choices $c \in C$ that maximize $u[a][i,j]$, then $\mathrm{probModelsChoice}[c,j] = \frac{1-\epsilon}{|B|} + \frac{\epsilon}{|C|}$ if $c \in B$ and $\mathrm{probModelsChoice}[c,j] = \frac{\epsilon}{|C|}$ if $c \notin C$. Finally, we combine the probabilities of the models' choices:

$$\mathrm{probChoice}[c] = \sum_{j=1}^{N} \mathrm{probModelsChoice}[c,j]\ \mathrm{probModel}[i_{\mathrm{current}}, j] \quad (2)$$

The above procedure is not only used for opponent decision nodes, but is also used for the player's own decision nodes. In this case, $\mathrm{probModel}[i,j]$ used in the above calculation actually comes from the recursive belief of how

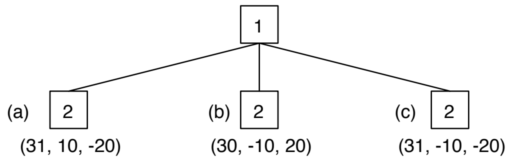

Figure 3: Prob-max^n example tree.

	Model: MT	Model: mOT
Player 1	31	29
Player 2	10	10
Player 3	-20	-20

Figure 4: Prob-max^n value of node (a) from Figure 3.

the other players model the prob-max^n player. We do this to avoid assuming that the opponents have a perfect model of the decisions the prob-max^n player will make during the game. On the other hand, when the prob-max^n player actually makes a decision at the root of the tree, it does know its own decision rule, and so should take advantage of this knowledge when making a decision. In order to make decisions with this extra information, we must maintain additional information in the search, u_{true}, which is our belief about our own expected utility at any node in the tree. u_{true} is easily computed from its children. At opponent decision nodes, we combine the children's utilities based on $\mathrm{probChoice}[c]$. The u_{true} value at the root player's decision nodes is the maximal u_{true} value from among the children of that node. At the root of the tree, prob-max^n makes the move which leads to the largest u_{true}. Although, u_{true} entirely determines prob-max^n's action, u_{true} is computed based on $\mathrm{probChoice}$ computations throughout the tree, which are determined by the propagating $u[i, j]$ matrices.

Example

We demonstrate the computation done by prob-max^n in a small example shown in Figure 3. The values shown at the leaves are the payoffs for a hand of Spades, where one point is awarded for each overtrick[1].

In Figure 4 we show how the value at node (a) is represented during back-up by prob-max^n. The first step at the leaves of the tree is to convert the payoffs from the game into utilities. For this example we have two models for each player, a maximizing tricks model (MT) and a minimizing overtricks model (mOT). For the MT model the utility is just the payoff in the game, while the mOT model subtracts the number of overtricks from a player's score. Thus when, at node (a) in Figure 3 Player 1 takes one overtrick, the MT model has utility of 31 while the mOT model has utility 29, as shown in Figure 4.

[1]Overtricks are usually tallied this way because a player's score *mod* 10 will then be the number of overtricks they have taken.

	Choice (a)	Choice (b)	Choice (c)
Payoff	(31, 10, -20) [bid+1]	(30, -10, 20) [bid]	(31, -10, -20) [bid+1]
MT Utility	31	30	31
MT Weight	$\frac{(1-\epsilon)}{2} + \frac{\epsilon}{3}$	$\frac{\epsilon}{3}$	$\frac{(1-\epsilon)}{2} + \frac{\epsilon}{3}$
mOT Utility	29	30	29
mOT Weight	$\epsilon/3$	$(1-\epsilon)+\epsilon/3$	$\epsilon/3$

Figure 5: Calculating weights for choices in prob-max^n.

	Model: MT	Model: mOT
Player 1	30.55	29.45
Player 2	-4.5	-4.5
Player 3	-2	-2

Figure 6: Final prob-max^n value of root node in Figure 3

```
// PROB-MAX^N  computes the Utility Matrix for an
// internal or external node.
PROB-MAX^N (node, Models)
  if TERMINAL(node)
    Return Models.EVALUATE(node)
  set i_current=node.GETCURRENTPLAYER()
  UtilityMatrix choices[]
  for each c in node.GETCHILDREN()
    choices[i]=PROB-MAX^N (s, Models)
  Return COMBINE(choices, i_current, Models)
```

Table 1: Pseudo-code for prob-max^n. *node* is the node in the game tree to be evaluated. *node*.GETCHILDREN() returns the children of a node. *node*.GETCURRENTPLAYER() returns the player to act. *Models* contains the set of models. *Models*.EVALUATE(*node*) returns a utility matrix.

Given a table of values for each possible move, we calculate the probability that Player 1 makes each move given each model. This computation is shown in Figure 5. For all players, the minimum probability of making any move is $\epsilon/3$. For the MT player outcomes (a) and (c) have the same utility, so the remaining weight $(1 - \epsilon)$ is distributed evenly between these outcomes. For the mOT, choice (b) has the best utility, so we expect mOT to choose this choice with additional weight $1 - \epsilon$.

Supposing that $\epsilon = 0.30$, then the MT model would choose branches (a) through (c) with probability 0.45, 0.10 and 0.45 respectively. Similarly, mOT would choose these moves with probability 0.1, 0.8 and 0.1. If $\mathrm{probModel}$(MT) $= \mathrm{probModel}$(mOT) $= 0.5$, then we expect Player 1 to choose outcome (a) and (c) with probability 0.275 and outcome (b) with probability 0.45.

The final value returned by prob-max^n for this example can be computed by multiplying the utility of each outcome under each model by the probability that the outcome would be selected. So, the utility for Player 1 using model MT is $0.275 * 31 + 0.45 * 30 + 0.275 * 31 = 30.55$. All values for this example are shown in Figure 6. See Table 1 and Table 2 for pseudo-code that implements prob-max^n.

```
Global double probModel[,]
Global double epsilon = 0.1
Global int you=1

// COMBINE combines the utility matrices.
COMBINE(choices, i_current, Models)
  // answer and probChoice initialized to zero.
  UtilityMatrix answer
  double probChoice[1 ... |choices|]
  for m in 1 ... Models.N:
     probChoice +=
       probModel[i_current, m]
       ×GETPROBCHOICE(choices, i_current, m)
  answer = Σ_{c=1}^{|choices|} probChoice[c]choices[m]
  // If we are to play, the true utility is
  // the maximum true utility.
  if (i_current==you)
     answer_true = max_{c∈1...|choices|} choices[c]_true
  Return answer

// choices[c][i,j] is the utility of the jth model
// of agent i if choice c is taken.
// GETPROBCHOICE returns the probabilities of the
//choices associated with the model.
GETPROBCHOICE(choices, i_current, model)
  // argmax... returns the set of all choices
  // that maximize the utility of model.
  set B = argmax_{c∈1...|choices|} choices[c][i_current, model]
  for i in 1... |choices|
     if i ∈ B
        weights[i]=( (1-ε)/|B| + ε/|choices| )
     else
        weights[i]=( ε/|choices| )
  Return weights
```

Table 2: Pseudo-code for COMBINE. $\mathrm{probModel}[i,j]$ is the probability of the jth model of player i, and is initialized elsewhere. *Models*.N is the number of models. *you* is the index of the searching player.

Theoretical Underpinnings

One way to interpret prob-max^n is as a belief about the opponents. If for each agent i we have N_i models $m_{i,1} \dots m_{i,N_i}$, and $N = N_1 + N_2 + N_3$ models total, then we believe that our opponents believe that they are playing the following game among N (instead of 3) *mini-players*. Standing behind (or inside the head of) each real player i in the real game of spades, there are N_i mini-players. Any situation where player i would move in spades, an N_i-sided die is rolled and j pips show up. Then, the ith player plays whatever mini-player $m_{i,j}$ recommends. Note that the mini-players for each player are distinct; no mini-player can play for two players)

We consider mini-players that with probability ϵ act at random and with probability $1 - \epsilon$ choose an action that maximizes some utility function $u_{i,j}$ over outcomes. Thus, what makes $m_{i,1}$ and $m_{i,2}$ distinct is that they are trying to achieve different outcomes (*e.g.*, one might be trying to maximize tricks, the other might be trying to minimize overtricks). Moreover, we assume that not only does each mini-player believe the game evolves in this fashion, but they believe others believe that the game also evolves in this fashion.

Theorem 1 *The prob-max^n algorithm computes the probability that each player will take each action correctly given the assumptions described above.*

Proof: For each node in the game tree, we compute the utility matrix, consisting of an expected utility of each mini-player. This expected utility is what that particular mini-player expects to get given that node is reached. We compute this utility matrix by traversing the tree bottom up, like max^n. However, instead of taking the branch that maximizes utility for the nth player, we have a more complicated update rule.

What we do is attempt to predict the probability $p(a)$ that player i makes each move $a \in A$. We do that by first finding, given the die had j pips, the probability that player i makes a move a, which we denote $p(a|j)$. We know that model $m_{i,j}$ will almost maximize utility. Since we have utility matrices for every child of the node, we can determine which choices maximize the utility of $m_{i,j}$. If s of t total choices are optimal, for each optimal choice a^*, $m_{i,j}$ will play it with a probability $p(a^*|j) = \frac{1-\epsilon}{s} + \frac{\epsilon}{t}$, and each sub-optimal choice a', $m_{i,j}$ will play it with a probability $p(a'|j) = \frac{\epsilon}{t}$.

Thus, if $p(j) = 1/N_i$, the probability that j pips come up on the die, then the probability that player i plays action a is $\sum_{j=1}^{N_i} p(a|j)p(j)$. Given these $p(a)$, we can compute the expected utility of every model given the node is reached. If $u_{i',j}(a)$ is the utility of $m_{i',j}$ if action a is chosen, then the utility of this node for model $m_{i',j}$ is $u_{i,j}(this) = \sum_{a \in A} p(a) u_{i',j}(a)$. This is exactly what our algorithm computed in the previous section. ■

Given this belief, our algorithm is attempting to maximize some true utility u_{true}. The true utility is updated based upon the distribution over actions that was described before for other agents, but for ourselves, the true utility is simply the maximum true utility of all children.

Theorem 2 *The algorithm in the previous section maximizes the true utility.*

Proof: Our belief about other agents can be described as a *behavior*, a distribution over actions at every point in the game. Thus, the expected utility for every node of the opponent is a weighted sum of the utility of all the children, i.e. if A is the set of actions, then $u_{true}(this) = \sum_{a \in A} p(a) u_{true}(a)$. When we ourselves move, we choose the action with highest utility, so $u_{true}(this) = \max_{a \in A} u_{true}(a)$. ■

Discussion

Given a complete description of prob-max^n we can now describe the relation of prob-max^n to PrOM (Donkers, Uiterwijk, & van den Herik 2001). Both algorithms allow multiple opponent models and assign a probability to each model.

One difference between PrOM and prob-$\max^n$ is that prob-$\max^n$ is designed for games with more than two players while PrOM is for two-player games. More importantly, PrOM and prob-$\max^n$ handle recursive modeling differently. In PrOM, opponent models are minimax agents, while in prob-$\max^n$ opponent models use epsilon-greedy move selection with a common recursive probabilistic model.

Learning in Prob-$\mathbf{Max^n}$

Until this point, we have assumed that $\mathrm{probModel}[i, j]$ was fixed. This is the same as assuming a multinomial prior over the models for each player. Alternatively, we don't have to pick one particular multinomial for the opponents, but could define a prior over multinomials. Dirichlet priors are one class of priors. By using a Dirichlet prior, if we observe a player playing like one of the models, we will expect the player to play like that model in the future.

The most well-known use of Dirichlet priors is the bucket of words technique in document classification (*i.e.* naïve Bayes). One assumes that documents of a particular type are formed by randomly generating words from some fixed but unknown multinomial distribution over words. Its popularity stems from the fact that the posterior Dirichlet can be determined by simply counting how many times a word occurred in documents of a certain concept. This is used to predict the probability that a new document was generated from a particular case.

In our case, determining the posterior belief is more difficult, because instead of observing a sequence of words, we observe choices that could have been generated by any of the models. Thus, for each choice, the model that generated that choice is a *latent variable*. In order to exactly calculate the posterior, we would have to iterate over all the exponentially many possible assignments to the latent variables. However, we used a Markov chain Monte Carlo Method (MCMC), which is a fast approximation technique for inference in the presence of many latent variables (Neal 1993).

Experimental Results

We evaluate prob-$\max^n$ in the game of Spades, replicating the experimental setup of our previous work (Sturtevant & Bowling 2006). In particular we played a total of 600 games of Spades, which end after a player reaches 300 points. These games consisted of only 100 *unique* sequences of deals, where the sequence was repeated for all possible ways that two player types can be assigned to the three seats at the table (see Table 3). The situation where all of the players were of an identical type was ignored, leaving six permutations for six hundred games. Each hand consisted of seven cards being dealt to each player from a 52 card deck and all cards were public information. Prob-$\max^n$ can produce its first move for such a hand in less than one second.

For each algorithm of interest, four experiments of 600 games, as described above, were performed. Each experiment consisted of the candidate algorithm (soft-$\max^n$ or prob-$\max^n$) paired against a $\max^n$ opponent with a particular utility function and model of its opponents' utilities (*viz.*, MT_{MT}, MT_{mOT}, mOT_{MT}, and mOT_{mOT}, where the subscript refers to the player's model of its opponents.) Half of the games then involved two candidate algorithms at the table with one $\max^n$ player, and the other half involved two $\max^n$ players and one candidate algorithm. We report average scores for each player type and their win rate, which if the players were equal would be exactly 50% since half the players are of a particular type.

	Seat 1	Seat 2	Seat 3
1	A	A	B
2	A	B	A
3	A	B	B
4	B	A	A
5	B	A	B
6	B	B	A

Table 3: The six ways to arrange two player types, A and B, in a three-player game.

Players A v. B	Player A			
	Score	%Win	%Gain	%Loss
mOT_g v. MT_{mOT}	241.7	68.5	15.0	6.8
mOT_g v. MT_{MT}	218.2	53.5	9.5	5.5
mOT_g v. mOT_{MT}	242.2	54.8	4.8	8.0
mOT_g v. mOT_{mOT}	230.6	46.0	8.8	4.0

Table 4: Performance of soft-$\max^n$.

We first examine the performance of soft-$\max^n$ with the results presented in Table 4. Each row shows the outcome of an experiment against one of the $\max^n$ opponent types. In addition to showing the average score and winning rate for the player type, the table also shows the "%gain", which shows the algorithm's improvement in winning rate over using standard $\max^n$ with the wrong model. Additionally, "%loss" is the amount that could be gained by playing $\max^n$ with the correct opponent model. As is clear in the table, soft-$\max^n$ does provide a degree of robustness to incorrect models. It also shows that further gains are possible. Note that all of the %gain and %loss values are statistically significant at the 95% confidence level.

We now examine the performance of prob-$\max^n$ with the results presented in Table 5. The columns have the same meaning as in Table 4 except now "%gain" shows the improvement in winning rate of prob-$\max^n$ over soft-$\max^n$. These results show an improvement over soft-$\max^n$ against every single $\max^n$ opponent type. The improvements in most cases are as dramatic as soft-$\max^n$'s original improvements over incorrect models. In the case of MT_{mOT} the improvement is not statistically significant, but soft-$\max^n$'s performance against this opponent was already very strong.

The performance against mOT_{mOT} is now so strong that not only is prob-$\max^n$ winning more games, it is actually performing better than the $\max^n$ player with the perfect model, *i.e.*the same player. Although this seems counterintuitive, the result illustrates the importance of second-level recursive reasoning. In the perfect model case, which involves mOT_{mOT} in self-play, all player's models are perfect at all levels of recursion. In the siutation of prob-$\max^n$ against this opponent, the $\max^n$ player correctly believes its

Players A v. B	Player A			
	Score	%Win	%Gain	%Loss
mOT_p v. MT_{mOT}	248.0	71.0	2.5 ∘	4.3
mOT_p v. MT_{MT}	232.2	59.8	5.3	0.2 ∘
mOT_p v. mOT_{MT}	252.2	62.7	7.9	-0.1 ∘
mOT_p v. mOT_{mOT}	244.8	53.0	7.0	-3.0

Table 5: Performance of prob-max^n. "∘" denotes statistically insignicant results. All other gains and losses are significant at the 95% confidence level.

opponent is minimizing overtricks. However, it incorrectly believes that its opponent's model of itself is equally correct. Instead prob-max^n's model is a probabilistic one. One might think that prob-max^n's first level modeling error would be worse than max^n's second level error. We can conclude from these results, though, that prob-max^n's robustness to modeling errors shields it from mistakes that max^n's deterministic beliefs cannot. We do not report the results here, but we have run experiments with prob-max^n against opponents for which prob-max^n does not have opponent models, and prob-max^n is still able to play robustly and win a majority of games.

Learning Performance

We also applied prob-max^n with Bayesian inference to the same set of experiments described above. The learning results were interesting. At the end of the match we examined the posterior model over the the max^n opponents' utility distribution. The inference correctly skewed the distribution in favor of the player's actual type for 98.8% of the MT opponents. For mOT_{mOT}, it correctly skewed the distribution 81.8% of the time. And for mOT_{MT} opponents, it correctly skewed the distribution 67% of the time. Clearly, mOT type players are more difficult to identify, particularly when they have an incorrect belief about the prob-max^n player.

Note that less than 3% of the opponents were inferred to have distributions far (posterior's mean distribution assigining less than 30% to the correct type) from their true type. So inference more often than not assigns the correct model, and rarely puts too much probability on an incorrect model.

Although the inference results are quite successful the actual effect on prob-max^n's play was minimal. The results showed slight improvements against some opponents but none of the results were statistically significant improvements or losses. We suspect this is because prob-max^n's performance is already so strong against these opponents, there is little opportunity left for learning to improve play.

Conclusions

In this paper we introduced the prob-max^n algorithm for incorporating models of opponents in n-player games. We have shown that the algorithm outperforms soft-max^n against a variety of opponents. In addition we described how Bayesian inference could be use to identify an opponent's model through play. We show it can successfully identify a player's type in the course of a single game, although it did not lead to significant gains. We believe, though, that the probabalistic modeling framework of prob-max^n, coupled with inference, can lead to very strong players for multi-player games.

Acknowledgments

This work was supported by the Alberta Ingenuity Center for Machine Learning (AICML) and the Informatics Circle of Research Excellence (iCORE).

References

Baum, E. B., and Smith, W. D. 1997. A bayesian approach to relevance in game playing. *Artificial Intelligence* 97(1-2):195–242.

Beal, D. F. 1982. Benefits of minimax search. In Clarke, M. R. B., ed., *Advances in Computer Chess*, volume 3, 17–24. Oxford, UK: Pergamon Press.

Carmel, D., and Markovitch, S. 1996. Incorporating opponent models into adversary search. In *AAAI-96*, 120–125.

Donkers, H. H. L. M.; Uiterwijk, J. W. H. M.; and van den Herik, H. J. 2001. Probabilistic opponent-model search. *Inf. Sci.* 135(3-4):123–149.

Ginsberg, M. L. 2001. GIB: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research* 14:303–358.

Iida, H.; Uiterwijk, J. W. H. M.; van den Herik, H. J.; and Herschberg, I. S. 1993a. Potential applications of opponent-model search. part 1, the domain of applicability. *ICCA Journal* 16(4):201–208.

Iida, H.; Uiterwijk, J. W. H. M.; van den Herik, H. J.; and Herschberg, I. S. 1993b. Potential applications of opponent-model search. part 2, risks and strategies. *ICCA Journal* 17(1):10–14.

Korf, R. E. 1989. Generalized game trees. In *IJCAI-89*, 328–333.

Luckhardt, C., and Irani, K. 1986. An algorithmic solution of N-person games. In *AAAI-86*, volume 1, 158–162.

Nau, D. S. 1982. An investigation of the causes of pathology in games. *AIJ* 19(3):257–278.

Neal, R. 1993. Probabilistic inference using markov chain monte carlo methods. Technical Report CRG-TR-93-1, University of Toronto.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall.

Russell, S., and Wefald, E. 1991. *Do the right thing: studies in limited rationality*. Cambridge, MA, USA: MIT Press.

Sturtevant, N. R., and Bowling, M. 2006. Robust game play against unknown opponents. *Fifth International Joint Conference on Autonomous Agents and Multi-Agent Systems*.

Sturtevant, N. 2004. Current challenges in multi-player game search. In *Proceedings, Computers and Games*.